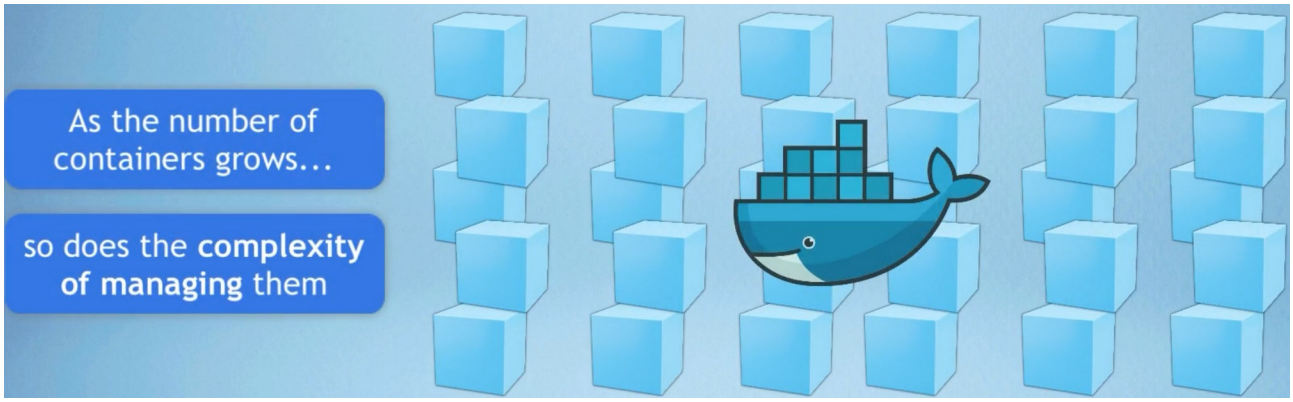


And the docker compose will be very over welling , we need something more powerful that automates deployment , scaling of the containers .

Topics to be covered:
1. Introduction to k8s
2. Main k8s components -> pods , node, sVc , secret, deploy , Sts, ing. etc.
3. Minikube & kubectl - Local setup
4. Main kubectl commands -> interacting with k8s cluster.
5. YAML configuration file -> configure k8s components
6. Organising components using **namespaces**
7. Configuration connectivity **services** .
8. Make app available outside the cluster **Ingress**
9. Persistence of data in k8s using **volumes**
10. ConfigMap and secrets as volumes type
11. Deploy stateful app like databases using S**tateFulSet**
12. Package manager tool for k8s - **helm**
13. Extending the k8s apis - **operator**

Also learn how to deploy microservices applications in k8s
Production and security best practices in k8s.
Authorisation in k8s- role based access control (RBAC)

Lets start :)

K8S is a open source container orchestration framework or tool .
Developed by google
Helps u manage application which are made of 100’s or 1000’s of containers.
It helps u manage they in different container environment eg: physical , virtual , hybrid or cloud environment .

What problem does k8s solves ?

Tread from monolithic to microservices , increased use of containers .

What features does container orchestration offers ?

1. High availability or no downtime.
2. Scalability or high performance => which high load in the serves scale up vica versa!
3. Disaster recovery - back up and restore.

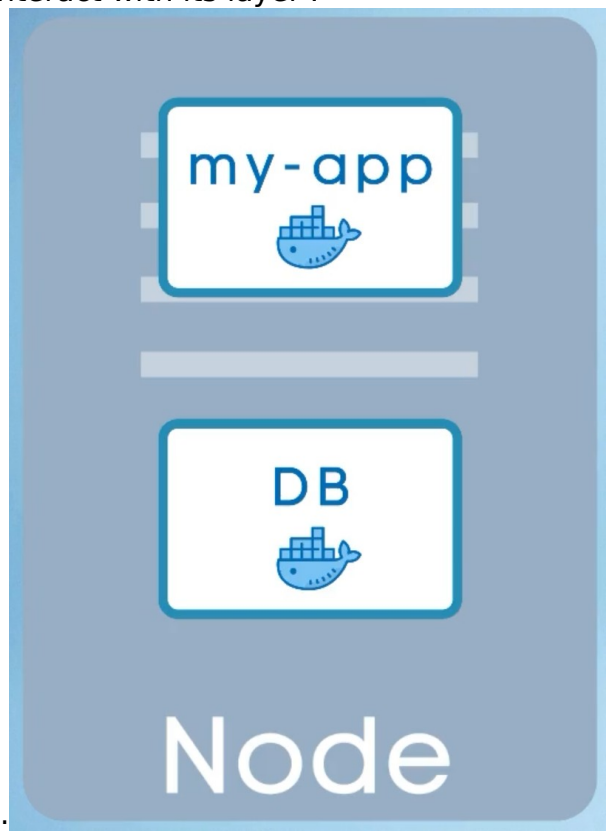
Main k8s components :

Pod , service , configMap, Deployment , Ingress , Secret , StatefulSet .

1. Node and pod !

Node = virtual or physical environment .
Pod is a smallest unit in kubernetes .
Abstraction over the container so that u don’t need to interact with the container technology and only k8s will interact with its layer .

Usually 1 application per pod .



Here u have one server and two container pods running on it with a abstraction layer on top on them .

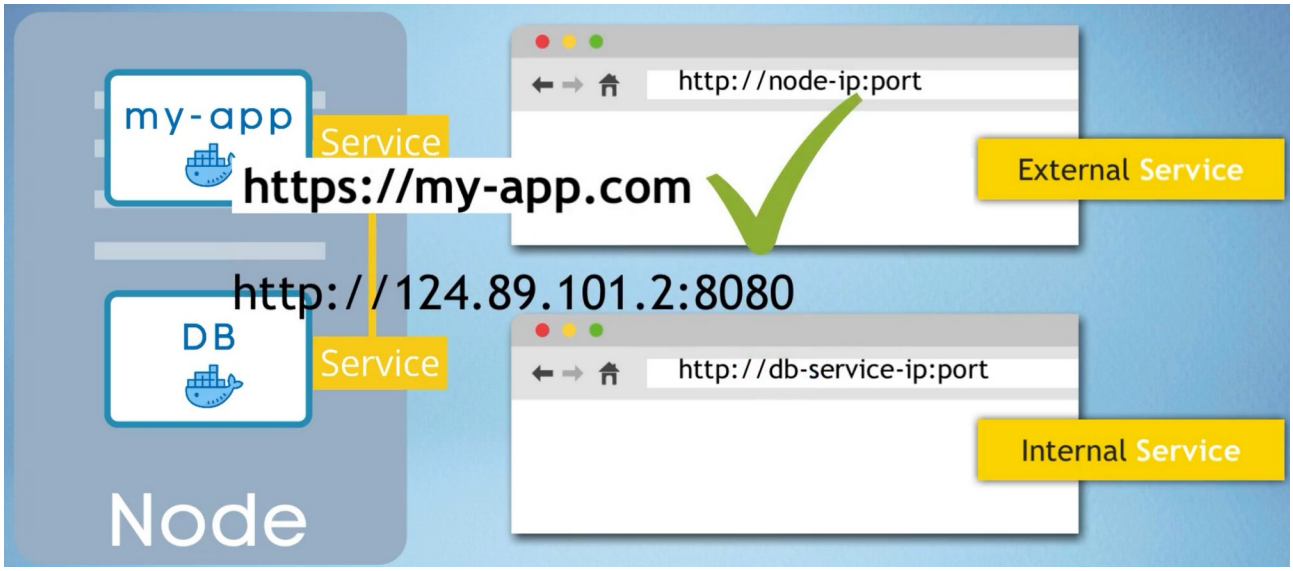
How these pods contact with each other -> each pods get a ip address.
And each pod can communicate with each other using that internal ip address its not the public ip address.

Pods are ephemeral!
Means they can die easily because of some error like app crashed inside or the server node were pods are run out of the resources and a new one will be created on its place with a new IP address assigned to it .

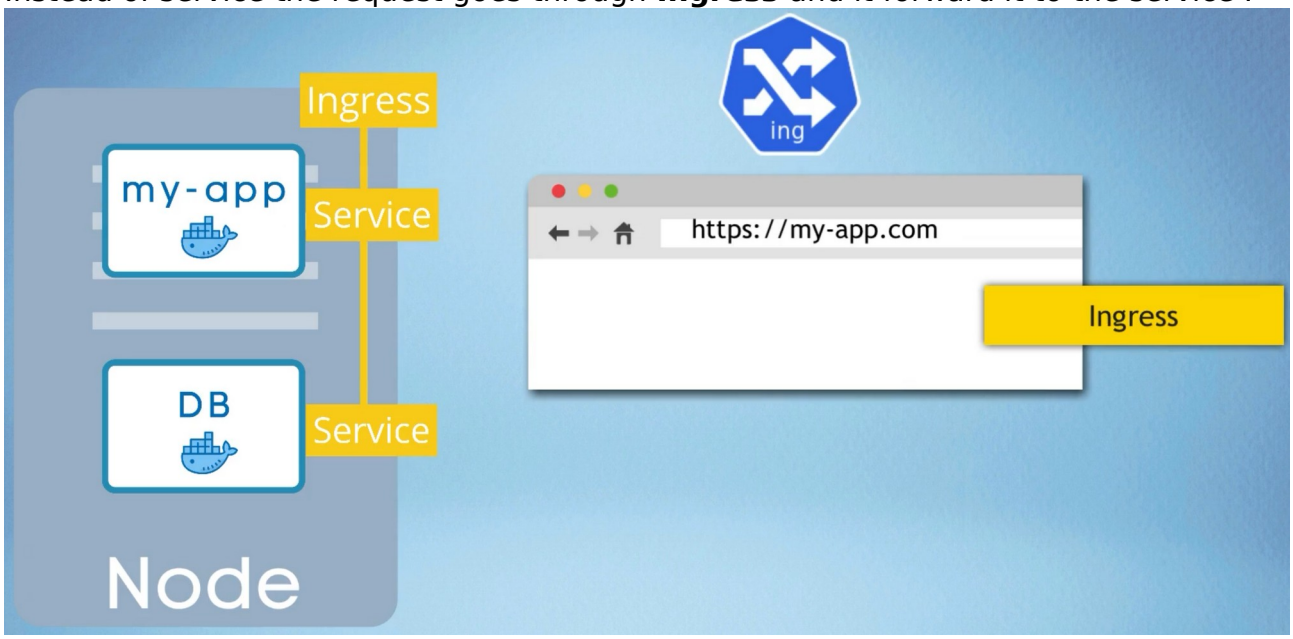
Now its very inconvenience to replace the ip address so because of that another component of k8s service is used .

Service : it is a permanent ip address , life cycle of service and pods are not connected .

Now u want your application to be accessible in the browser so for that u need a external service .



Instead of service the request goes through **Ingress** and it forward it to the service .



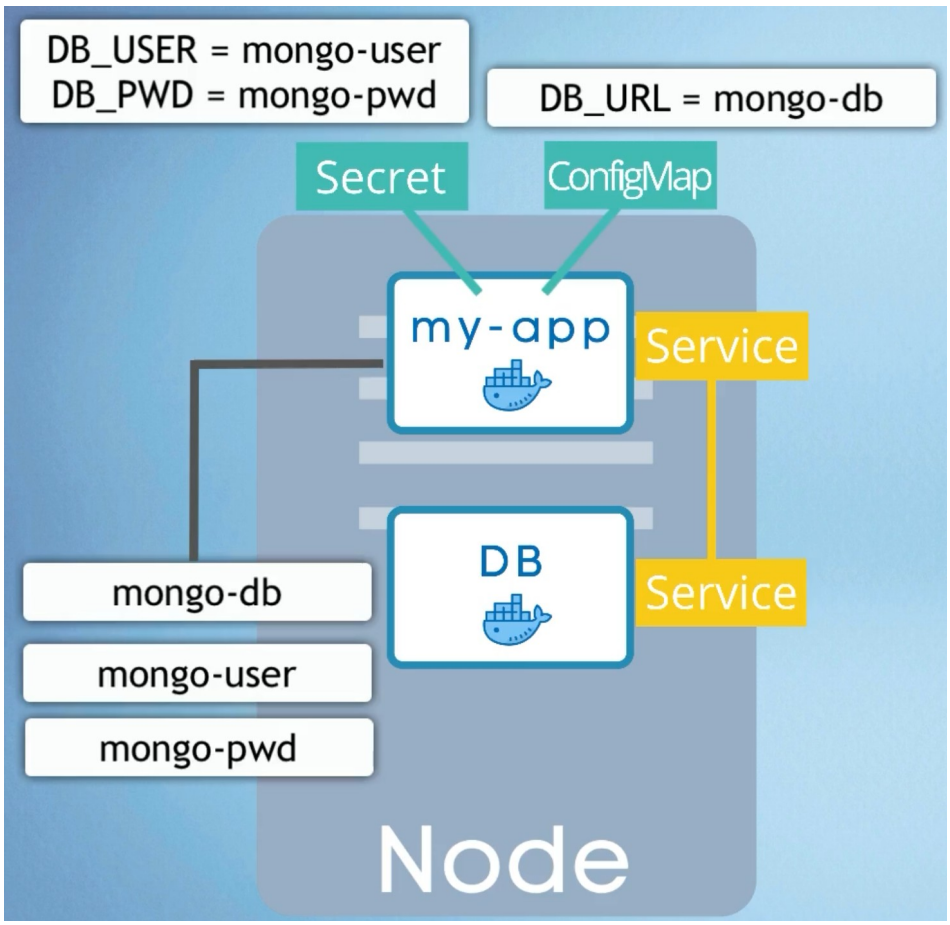
configMap and secret :

My app will talk to eg: mongo-db service but where do u configure all these -> it is usually database URL is in the build image of the application . So for example the endpoint of the service will change to oracle-db etc. so u would have to adjust that url in that application so u will have to rebuild the application with new version -> and u will have to push it to the repo -> pull that new image in your pod and restart every thing so it is little bit tedious to just change the URL of the database .

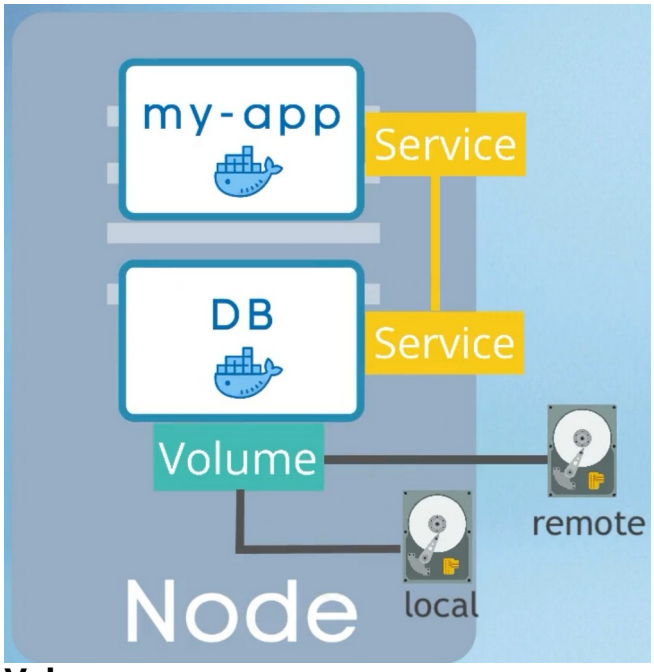
So for that. ConfigMap helps -> will contain external configuration of your application like URL of database and in k8s u just connect it to the pod .and that pod will get the data that the configMap contain , now u just change the information in the configMap and u don’t need to build another image again.

configMap is for the non - confidential data only.

So for the database user name and password u have another component in k8s **secret .**
We use 3rd party tool and deploy it to k8s those tools will encrypt the secret data k8s don’t provide encrypting itself it used third party tools .


DB_USER = mongo-user
DB_PWD = mongo-pwd
DB_URL = mongo-db
Secret
ConfigMap
my-app
Service
DB
Service
mongo-db
mongo-user
mongo-pwd
Node

U can use configMap or secret inside your application pod using environment variable or as a properties file.



Volumes :

If the data base get restarted the data will be gone but using volumes u can persist the data => it attaches the physical storage in the hard drive to your pod it can be either in the local machine the same server node or it can be on the remote storage out side the k8's cluster .

K8s explicitly does not provide persistence of data .

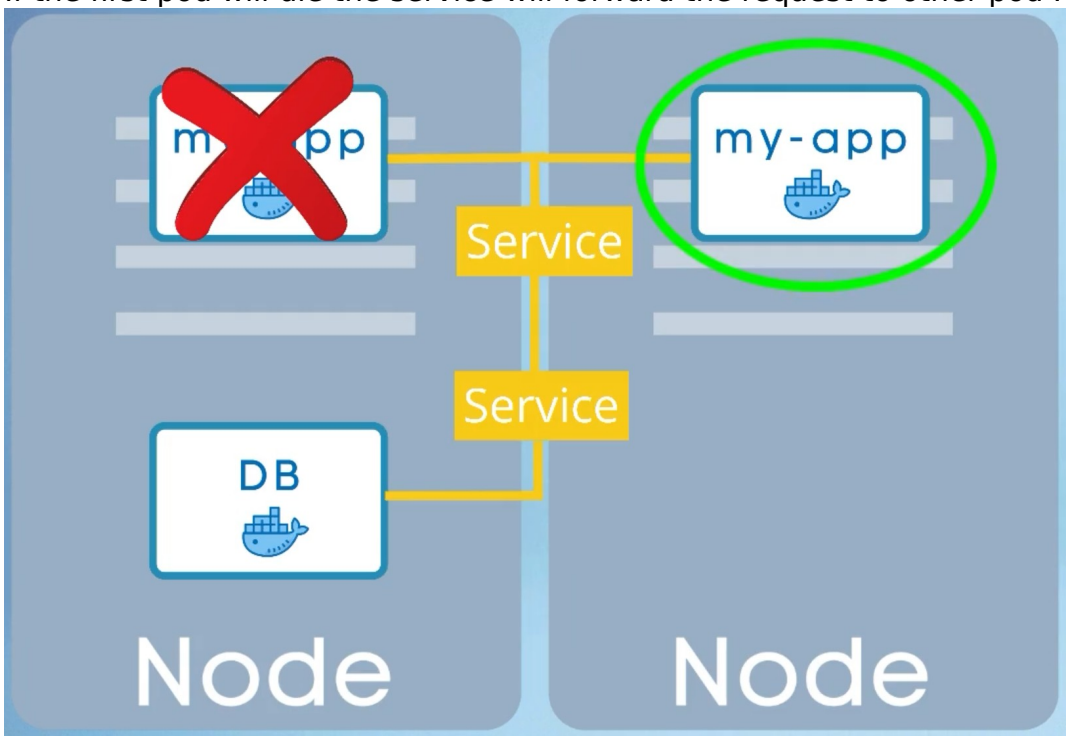
Deployment and stateFulSets :

Till now every thing is good user can access our application through the browser.
What happens if my application pod dies and user will have a downtime and can not reach the application which is very bad thing if it happen in production .
So this is the advantage of distribution system and container so instead of relying on one applicant node we are replicating everything in on multiple server nodes

Service is like a permanent ip address with a dns name and service is also a load balancer .

U will not actually create a second pod rather than u will define a blueprint defining how many replicas u want to have . And that component is called **Deployment** .
In practice u will not be creating pod u will be creating deployments because there u can scale up or down no of replicates of pods u need . **Pods were the layer of abstraction over the containers and deployment is another abstraction on top of pods** which makes it more convince to interact with pods and do some other configuration on them .

If the first pod will die the service will forward the request to other pod .



But what about replication Database -> we done replication DB using deployment .
Reason of the is DB has a state meaning if we have a clone of DB they will have to access the same shared data storage and there u will need a mechanism recording which DB pod is currently writing or reading to that DB in order to avoid data inconsistency .

And that mechanism is offered by k8s component -> **stateFulSet**
Eg: MYSQL , ELASTIC SEARCH etc all the stateful application should be creating using statefulset and not deployment .

Stateful set is just like deployments used to scale up or down database or replicate them and it makes sure no database inconsistency are offered .

Deploying DB application using statefulset in k8s cluster can be tedious so it more difficult so better practice is to host database outside the k8s cluster and just have a stateless application inside of the k8s cluster .

DaemonSet:

We want to collect logs from our cluster from each pod .
So we need a application which will run on each node and collect all the logs that pods are generating there
When we add node we need to adjust the replica no. in the deployment each time and when we delete a node we again need to adjust the replica no.
When deployment we cant also ensure all the pods are distributed equally among all the nodes.
So for that we have another k8s component which manages all these thing for u .

Daemon set automatically calculates how many replica u need based on the nodes and pods u have in the k8s cluster .
When add the node it will add the pods when delete node it will automatically remove pods (garbage collected)

Now u will not have to define the replicate count the deployment .
Automatically scale up and scale down .

Also guarantee one replica per pod .

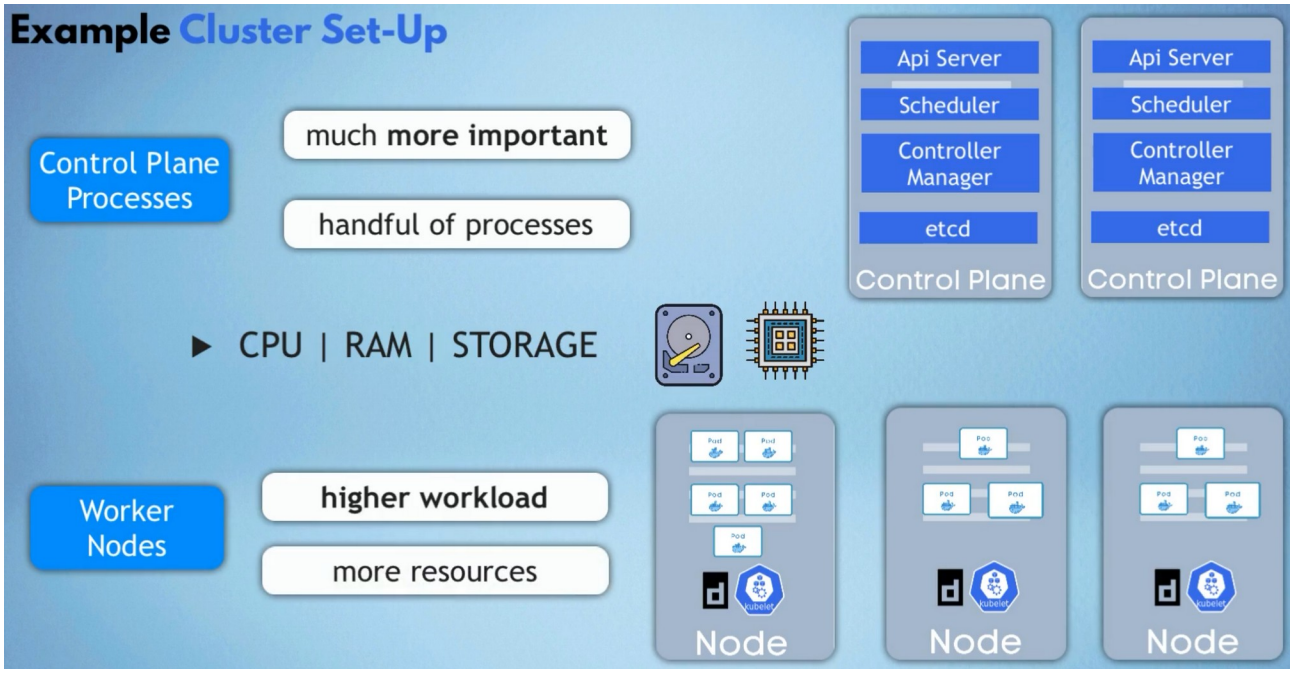
Wrap up the main components of k8s we have covered :

1. Pod : abstraction of container
2. Service : communication
3. Ingress: route traffic into cluster .
4. ConfigMap: external configuration
5. Secret: secret external configuration
6. Volumes : data persistence
7. Deployment : pod blueprint replication
8. Statefulset : DB pod blueprint replication
9. demonSet: automatic replica collect logs .

Kubernetes Architecture :

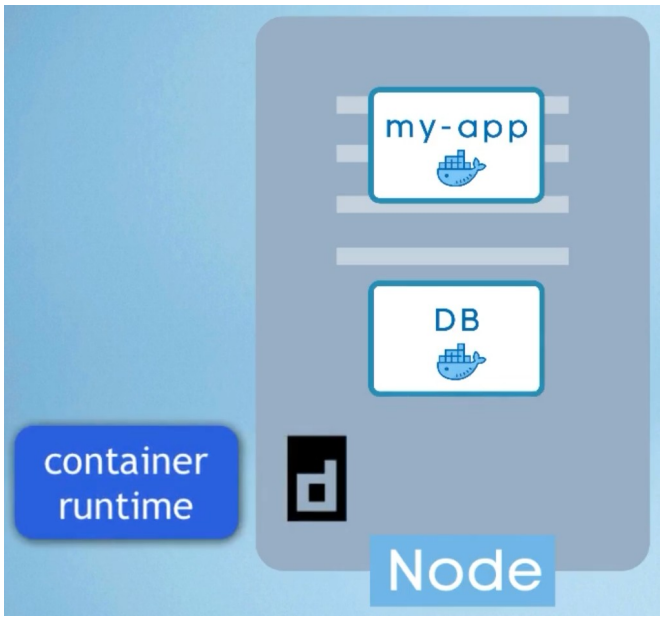
2-Types of nodes or machines

1. Control plane



2. Worker node

What is the difference roles it have inside the k8s cluster how k8s does what it does how the cluster is self managed and self healing and automated how u as a operator on the k8s cluster has the must lesser work to done to mange all this .



Node processes :
Nodes are the cluster servers that actually do the work there for some time called the worker node
Each node have multiple pods in it .
3 processes must be installed in each node .

1process is the contender run time eg: docker container , container D , crio-0 the most popular contender run time used today in most of the k8s cluster is container -D which is much more light weight than docker .

U can run docker images in any container run time there for u will see the docker icon in pods .

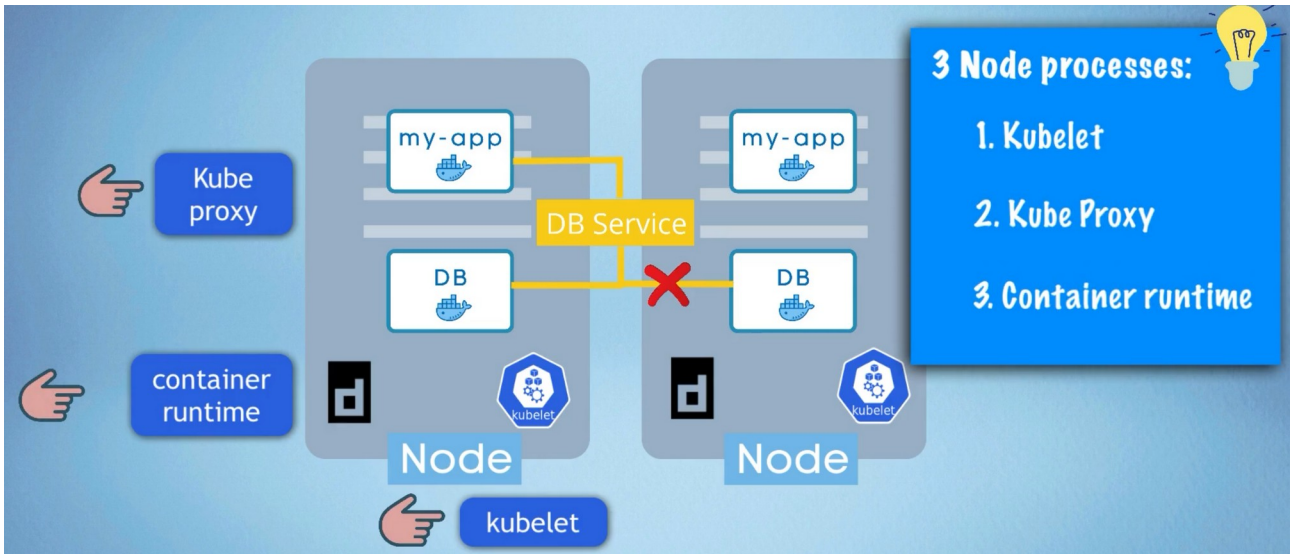
Container runtime need to be installed in every node .

2-process Kubelet is the process of k8s which interact with both container and node . Kubelet will take the configuration starts the pod with a. container inside and assigning resources from that node to that container .

Usually k8s clusters are made up of multiple nodes which also must have container run time and kubelet services inside each node and u can have 100s of those node which would run other pods , container and replicates of existing pods t**hese nodes communicate with each other using services.**

Which is a load balancer which catches the request directed to the pod of application . And forward it to the respective pods .
3- process Kube proxy responsible for forwarding the requests from the services to pods .

So container run time , kubelet and kubeproxy processes need to be installed each node



So , how do u interact with these cluster ?
How to know scheduled pod?
Monitor?
Reschedule/ restart ?
Join new node? Who do it join the cluster and join pods to it .
Ans is _—> all of this process are managed by **control plane nodes** .

C**ontrol plane processes :**
4 - process run on every control plan node that control the cluster state and worker node as well.

1process : api server -> when as a user u want to deploy new application in k8s cluster u interact with Apl server through UI or CLI . Api server is a like a cluster gateway also act as a gatekeeper for authentication .

So when every u want to schedule new pods deploy new applications >create new services or new other components u have to contact to the api server on the control plan node and api server than validates your request and if every thing if fine then it

will forward your request to other processes . In order to schedule the pod of create new component that u requested .

Also if u want to query the states of your cluster health etc . U make request to the api server and it gives back u the response .
Which is good for security as u have only one entry point to the cluster .

2process : scheduler -:> scheduler will schedule which pod to deploy on which worked node or next component will be scheduled to which worker node.

Process that actually does the scheduling that starts that pod with a container is the kubelet , so kubelet get the request from the schedular and start a pod on the node .

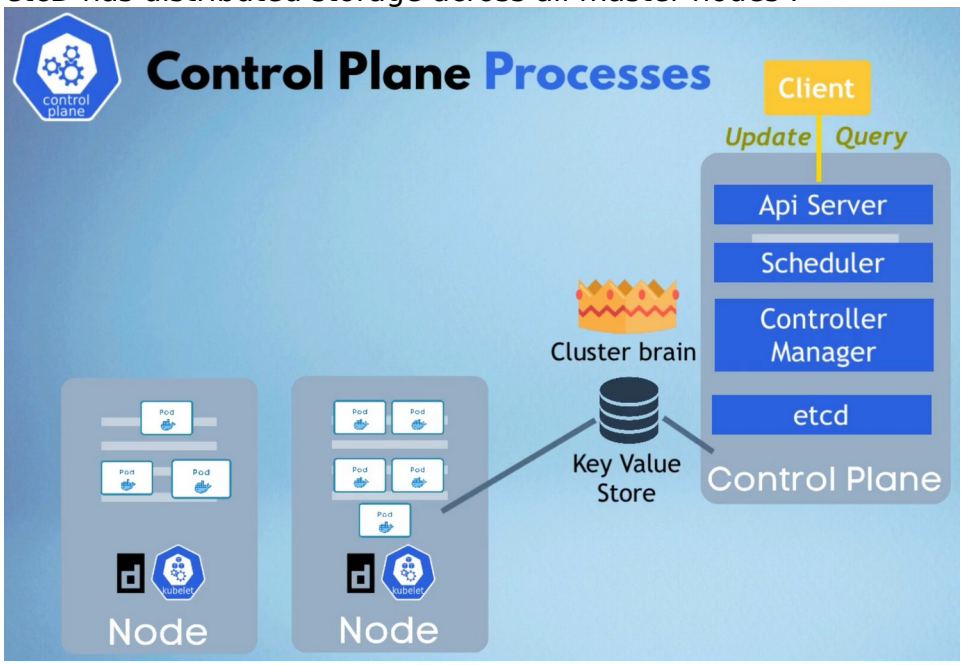
3process : controller manager -> when pods die on a node then we have to reschedule those pods again . Controller manger will detect cluster state changes .
And try to recover the cluster state as soon as possible .
Controller manger request scheduler and scheduler decides based on the resource left calculation which worked node to schedule the pod .

4process: etcD -> etcD (key-value storage) is a cluster brain .
Which record very change in the cluster eg: new pod scheduled when a pod dies all of these charges get updated in the key value store .

All of the other process controller manger and scheduler works because of etcD data .

How does the schedular do know which resources are available on each worker node .?
How does controller manger know which cluster state changes ?
Is cluster health oK query request to the api ?

All all of these information is stored in the etcD .
What is not stored in the actual etcD is the actual application data .
In practice k8s cluster is make up of multiple control planes , api server is load balanced .
etcD has distributed storage across all master nodes .



Cluster setup :
In a very small cluster u would probably have 2 control plane node and 3 worker node .
Console plane process are more important but they have lower work load and have handful of processes to run there they required lower resources like cpu , ram , storage :

On other side worked nodes has higher workload therefore required more resources .

As the complexity increased u can add more control plane and worker nodes on your cluster to make the application requirement full fill .

Minikube and kubeCtl :

What is Minikube ?

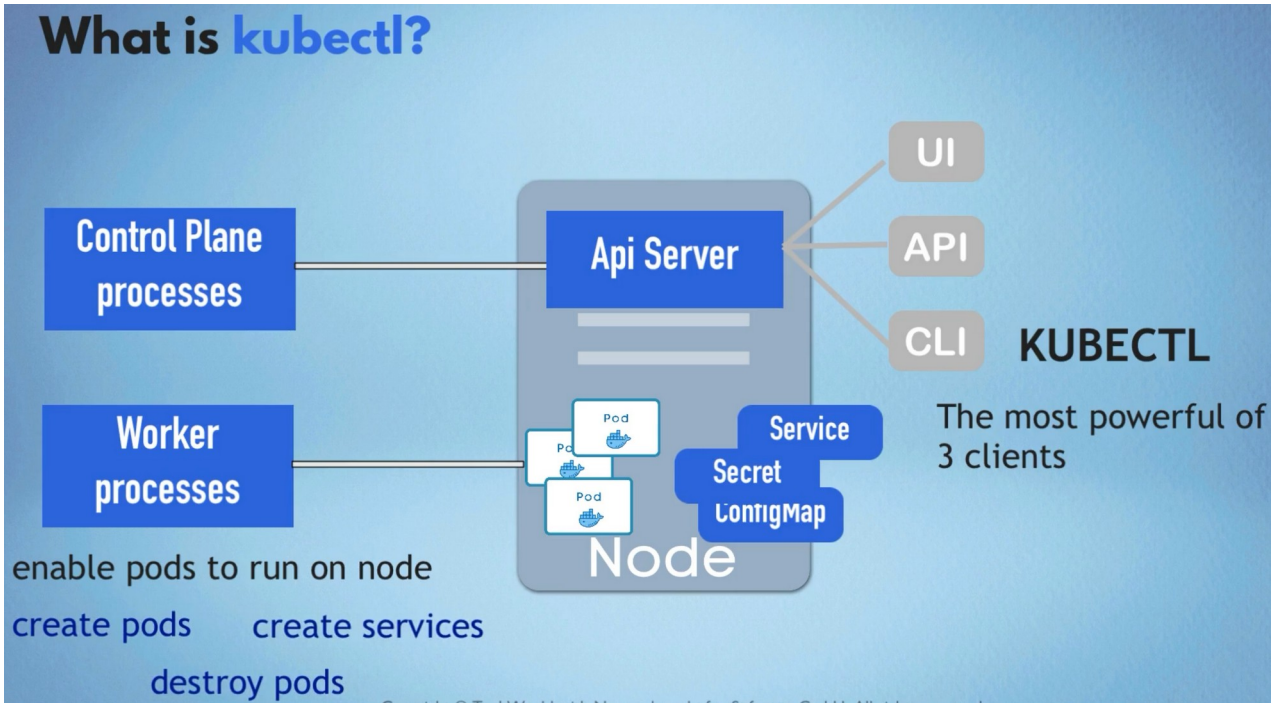
So in a production cluster generally u will have multiple control plane node and worker nodes .
Separate virtual and physical machines which each represent a node .

So if u will have to setup all this manually it should be very difficult as u will lack resources cpu , memory , ram etc .

So for that u have a open source tools called Minikube ->
Minikube is a one node cluster where the control plane and worker process run on the one node.

And that node will have docker control run time preinstalled .

What is kubeCTL ?



So now u have this virtual node on your local machine that represent Minikube u need a way to interact with that cluster like creating pods and other k8s components on Minikube and u can do it via kubeCTL which is a command line tool for k8s .

Minikube run both control plan and worker processes .
One control plan process ie : API server is the main entry point for k8s cluster (enables interaction with cluster) with different clients like UI , api , command line tool which is kubeCTL and kubeCTL is most powerful of all the clients .
Then worked nodes will enable pod to run on node -> create pods -> destroy pods or create services etc .

kubeCTL can interact with any type of k8s cluster eg: Minikube cluster , cloud cluster etc.

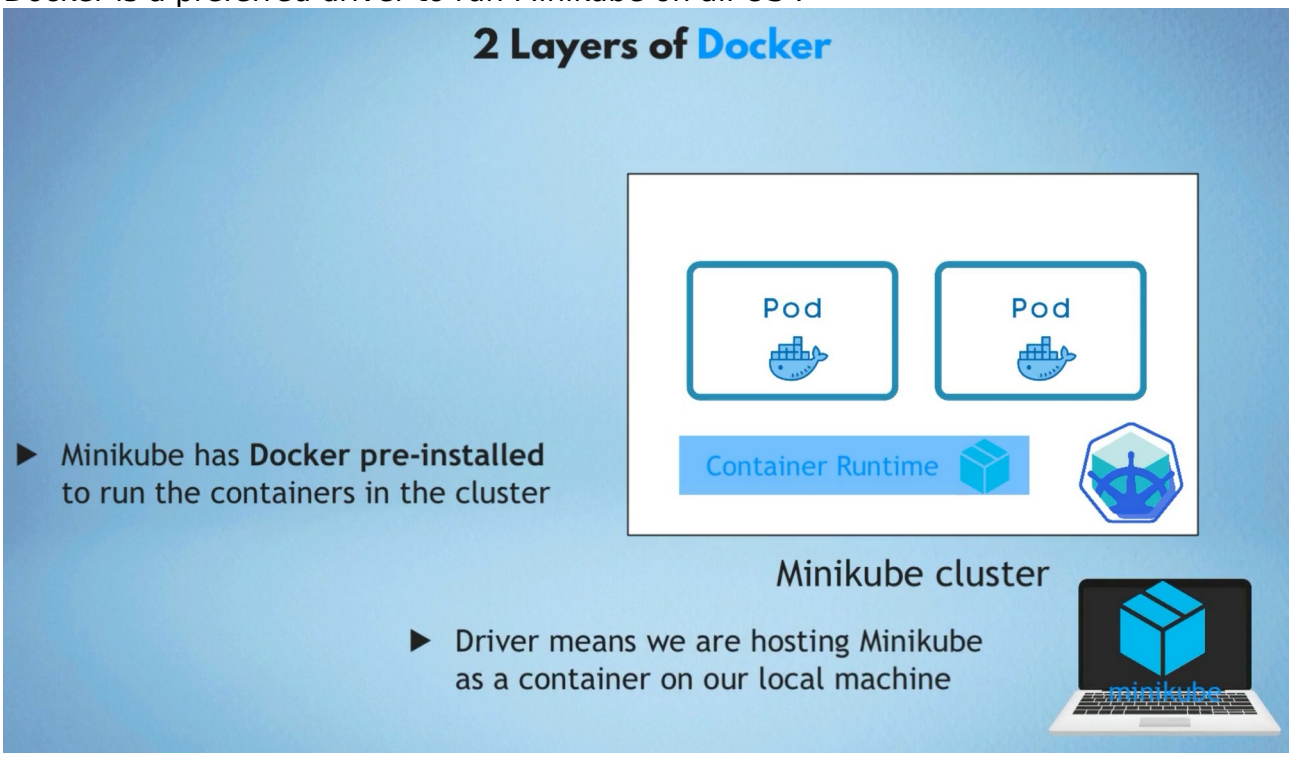
Installation and creation of Minikube cluster :

Visit official documentation of Minikube .

U can run Minikube either a container or a virtual machine on your laptop.
So we need a virtual machine or container tool installed on our laptop eg: docker , container -D , cri_O , vm ware , oracle virtual box etc .

For Mac OS - m1 - air ->
1. brew install minikube
2. Start your cluster -> minikube start —driver docker
3. Open the driver page to see the list of supported container or virtual machine tools .

4. Docker is a preferred driver to run Minikube on all OS .



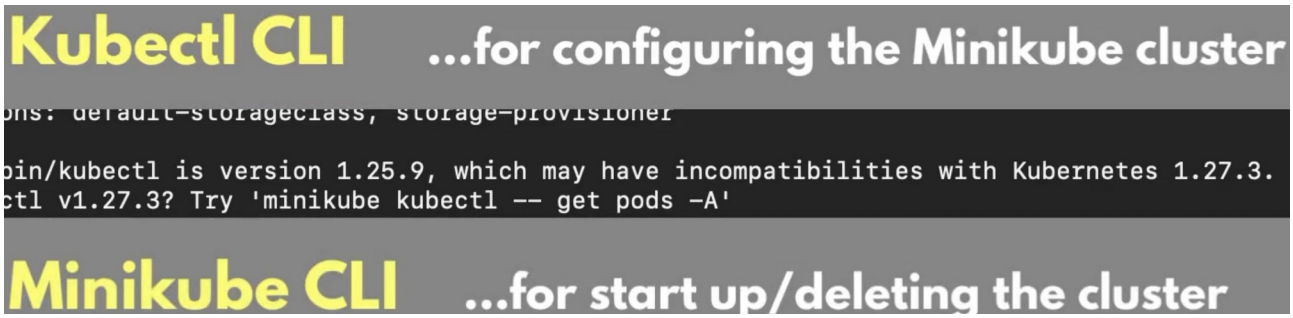
We have two layers 1. Minikube running as a docker container and docker inside Minikube to run our application container .

5. When docker desktop installed open it .

6. To check the status of your cluster -> Minikube status .

7. Minikube has kubeCTL as dependency -> now u can interact with Minikube using kubeCTL

8. Kubectl get nodes -> get status of node .



Main kubectl command :

Create and debug pods in Minikube cluster :

1. Kubectl get nodes
2. Kubectl get pod
3. Kubectl get services
4. Kubectl create -h // to create components -h is help
5. U will not see pods in above commands because pods are the smallest units in k8s cluster .

Usually in practice u are not working with pods directly . There is a abstraction layer over the pods ie : deployment .
So this is what we will create and it will create pod under neath .

Creating a nginx deployment ->

: kubectl create deployment nginx-depl —image=nginx // it will download the latest image of nginx from docker hub .

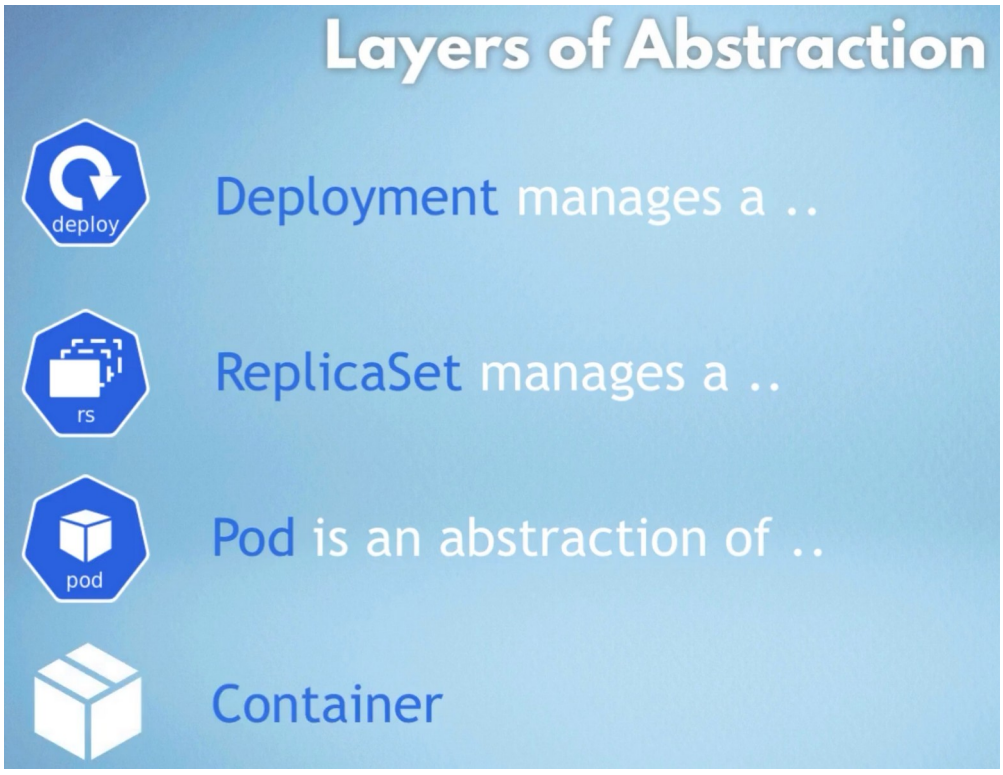
```
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl create deployment nginx-depl --image=nginx
deployment.apps/nginx-depl created
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get deployment
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
nginx-depl   1/1     1            1           59s
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
nginx-depl-85c9d7c5f4-c87zm   1/1     Running   0          2m32s
```

Between deployment and pod there is another layer called replicaset

: kubectl get replicaset

```
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get replicaset
NAME                    DESIRED   CURRENT   READY   AGE
nginx-depl-85c9d7c5f4   1         1         1       4m21s
```

Deployment manages a -> replicaset manages a -> pod is a abstraction layer to container _>container .
Everthing below deployment is managed by k8s .



: kubectl edit deployment nginx-depl // autogenerated file with default values.

```
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl edit deployment nginx-depl
Edit cancelled, no changes made.
```

Debugging pods :

Kubectl logs <podName>

```
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl create deployment mongo-deployment --image=mongo

deployment.apps/mongo-deployment created
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
mongo-deployment-5dfbb89958-26dvq   0/1     ContainerCreating   0          23s
nginx-depl-85c9d7c5f4-c87zm         1/1     Running             0          16m
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl describe pod  mongo-deployment-5dfbb89958-26dvq
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
mongo-deployment-5dfbb89958-26dvq   1/1     Running   0          2m20s
nginx-depl-85c9d7c5f4-c87zm         1/1     Running   0          17m
```

after some time container inside the pod will run automatically .

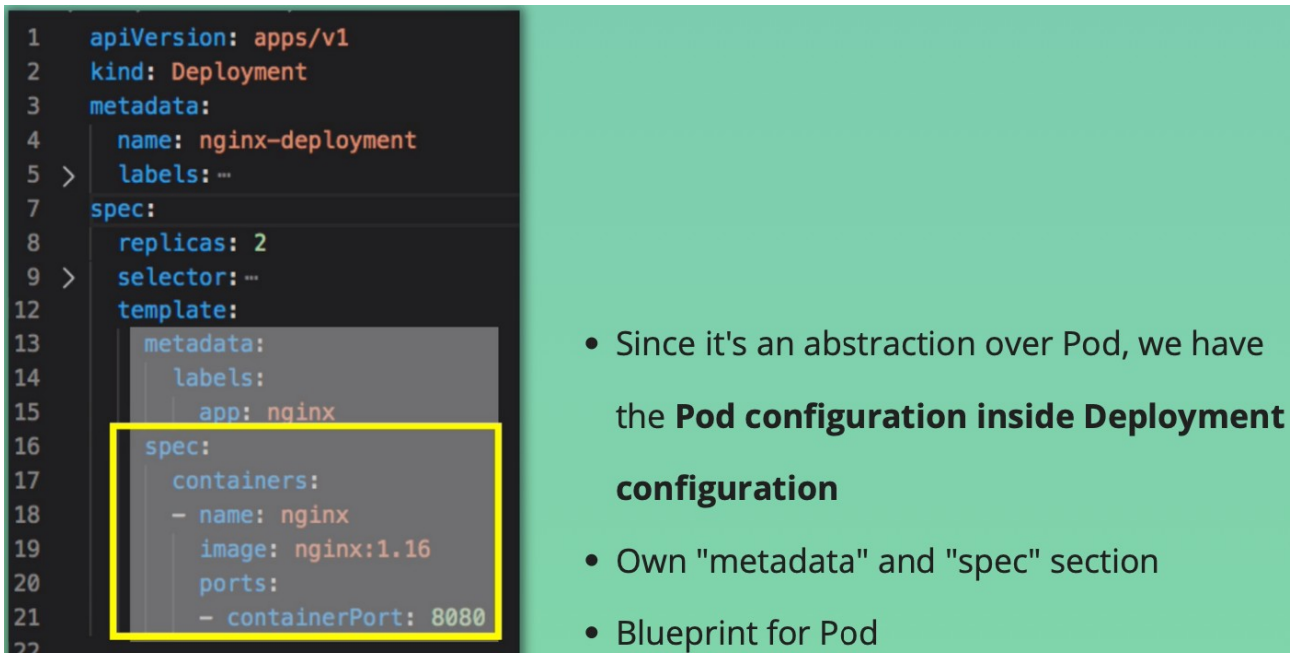
Terminal of mongo DB application container ->

Kubectl exec -It(interactive terminal) pod_name — bin/bash

```
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl exec -it mongo-deployment-5dfbb89958-26dvq -- bi
n/bash
root@mongo-deployment-5dfbb89958-26dvq:/# ls
bin   data  docker-entrypoint-initdb.d  home        lib    mnt  proc  run   srv  tmp  var
boot  dev   etc                        js-yaml.js  media  opt  root  sbin  sys  usr
root@mongo-deployment-5dfbb89958-26dvq:/# exit
exit
```

Delete deployment ,apply configuration file . :

```
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get deployment
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
mongo-deployment   1/1     1            1           9m2s
nginx-depl         1/1     1            1           24m
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
mongo-deployment-5dfbb89958-26dvq   1/1     Running   0          9m16s
nginx-depl-85c9d7c5f4-c87zm         1/1     Running   0          24m
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl delete deployment mongo-deployment
deployment.apps "mongo-deployment" deleted
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
nginx-depl-85c9d7c5f4-c87zm         1/1     Running   0          25m
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get replicaset
NAME                          DESIRED   CURRENT   READY   AGE
nginx-depl-85c9d7c5f4         1         1         1       26m
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl delete deployment nginx-depl
deployment.apps "nginx-depl" deleted
(base) vishaldwivedi@vishals-MacBook-Air-2 ~ % kubectl get replicaset
No resources found in default namespace.
```



All the crud operation happens at the deployment level every thing under neath just followed automatically .

U will usually work with k8s configuration files as -> there can be many option in the command line that u can put which is tedious .

```
% kubectl create deployment name image option1 option2
```

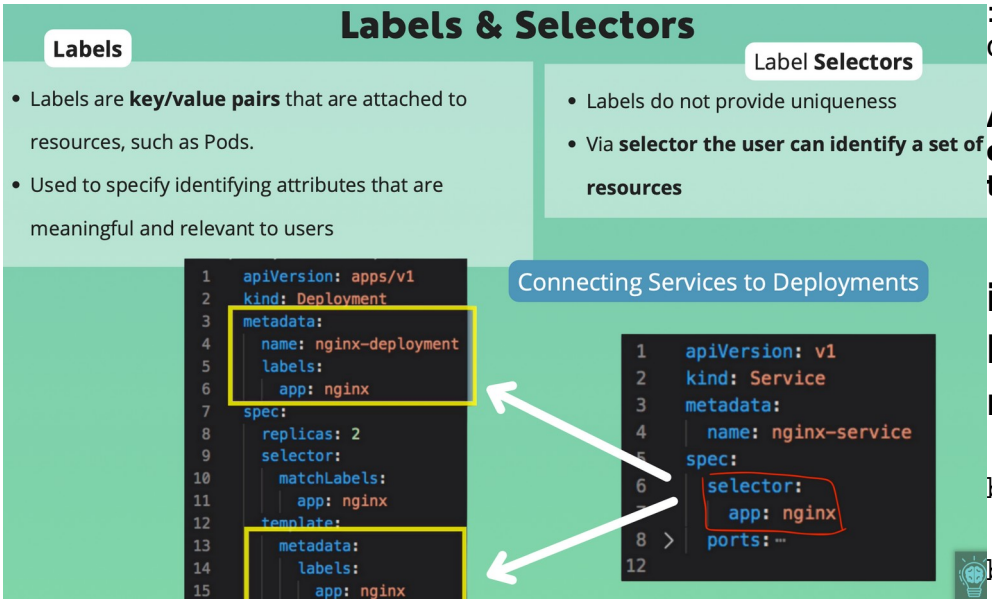
Kubectl apply will take the configuration file as a parameter and does what every u have written there .

Creating configuration file .=>

Eg: Kubectl apply -f nginx-deployment.yaml

So first creating this file :

: touch nginx-deployment.yaml



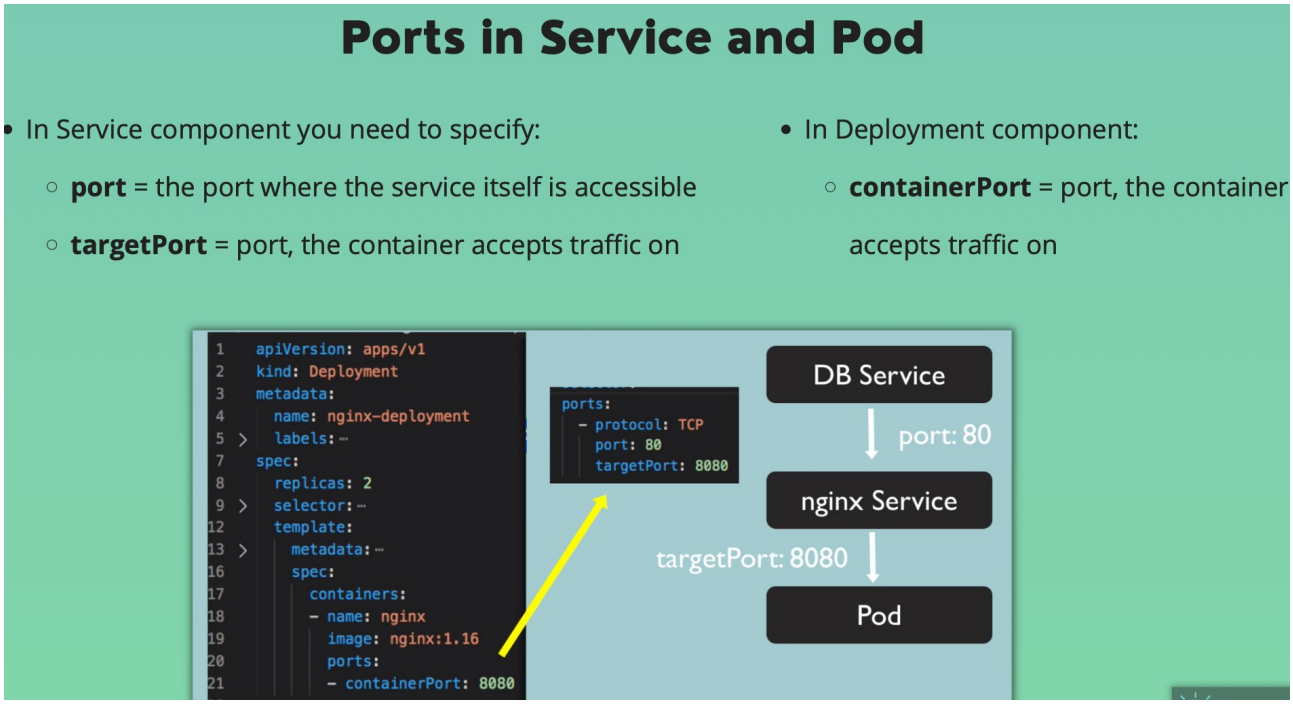
:vim nginx-deployment.yaml

All the commands used till now ->

install hyperhit and minikube

```
brew update
```

```
brew install
```




```
hyperkit
```

```
brew install minikube
```

```
kubectl
```

nginx-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.25
        ports:
        - containerPort: 8080
```

nginx-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080

```

minikube

create minikube cluster

```
minikube start --vm-driver=hyperkit

kubectl get nodes

minikube status

kubectl version
```

delete cluster and restart in debug mode

```
minikube delete

minikube start --vm-driver=hyperkit --v=7 --alsologtostderr

minikube status
```

kubectl commands

```
kubectl get nodes

kubectl get pod

kubectl get services

kubectl create deployment nginx-depl --image=nginx

kubectl get deployment
```

```
kubectl get replicaset

kubectl edit deployment nginx-depl
```

debugging

```
kubectl logs {pod-name}

kubectl exec -it {pod-name} -- bin/bash
```

create mongo deployment

```
kubectl create deployment mongo-depl --image=mongo

kubectl logs mongo-depl-{pod-name}

kubectl describe pod mongo-depl-{pod-name}
```

delete deplyoment

```
kubectl delete deployment mongo-depl

kubectl delete deployment nginx-depl
```

create or edit config file

```
vim nginx-deployment.yaml

kubectl apply -f nginx-deployment.yaml

kubectl get pod

kubectl get deployment
```

delete with config

```
kubectl delete -f nginx-deployment.yaml
```

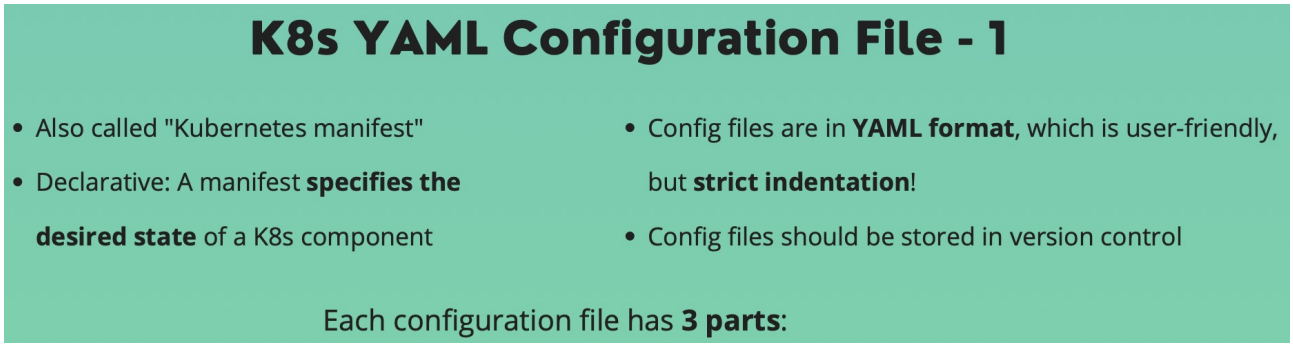
#Metrics

`kubectl top` The kubectl top command returns current CPU and memory usage for a cluster’s pods or nodes, or for a particular pod or node if specified.

Which kubectl apply : u can both create and update a component .

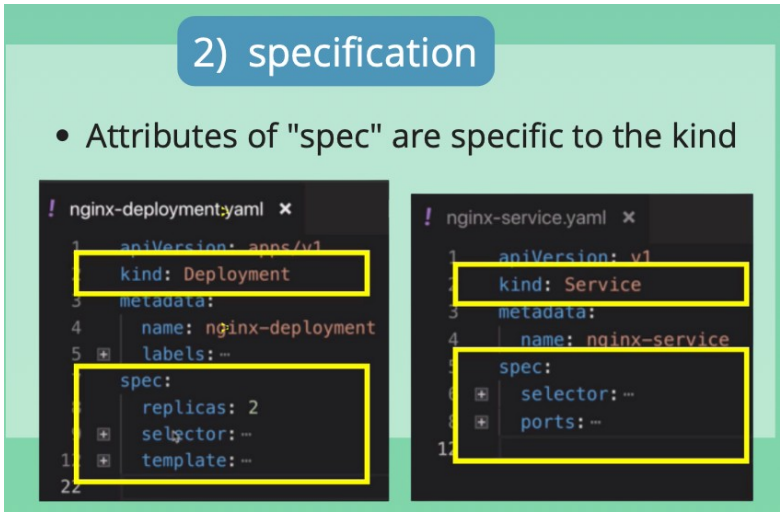
YAML configuration file :

3-parts of k8s config-file .



demo-test-deployment.yaml 418 B

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
  labels:
    app: my-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-image
          env:
            - name: SOME_ENV
              value: $SOME_ENV
          ports:
            - containerPort: 8080
```



1. Metadata -> name of the component .
2. specification -> replica ,selector ,port
3. Status -> it will be automatically generated and added by k8s , the way it works is k8s will always compare what is the desired state and what is the actual state .

Where does k8s get that status data to automatically add in status part in yaml file >?
Info comes from etcD - it hold the current status of k8s components .

:: yaml validator to can check your yaml file here is it is correctly written as yaml is very strict in its indentations . And store these configuration file on with your code .

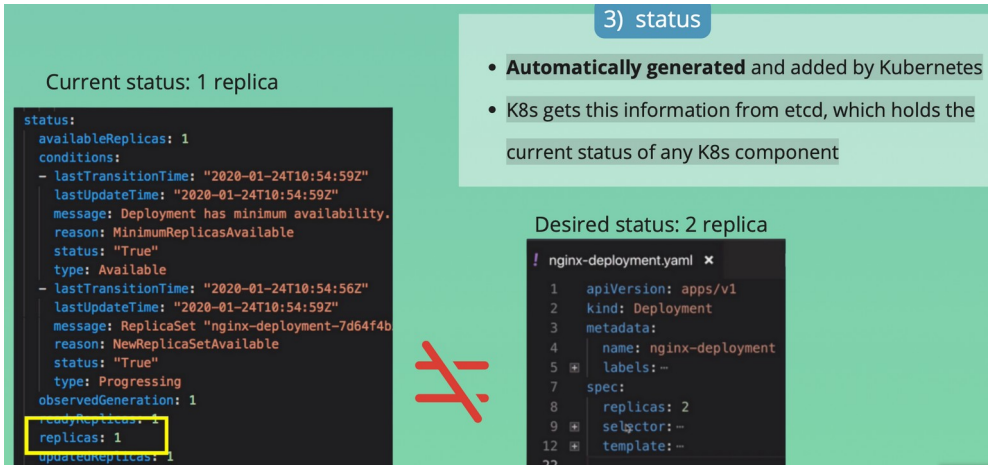
Blueprint for pods (template) :
Deployments mange the pods that are below them
So how does it happen where does this whole thing defined in the configuration ?

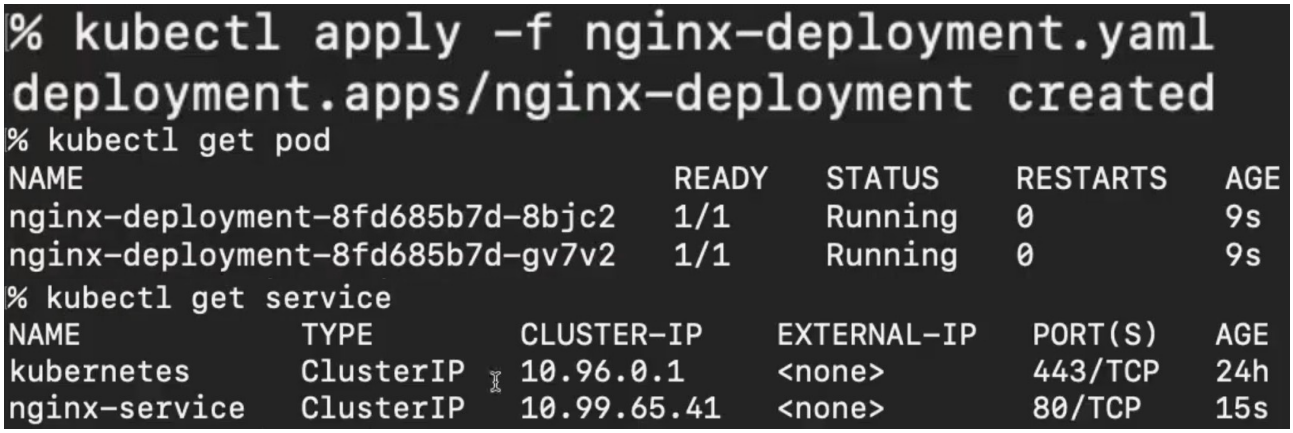
Since it’s an abstraction over pod , we have the pod configuration inside deployment configuration , template has its own metadata and spec section its a bullring for pod.

So this configuration apply to a pod means pod should have its own configuration .

Connection components (labels and selectors and ports) :

In metadata part u give components of deployment a key- value pair .
Pod get the label through the template blueprint
This label is matched by a selector .
Selector will connect the server to the deployment or pod .




% kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
% kubectl get pod
NAME READY STATUS RESTARTS AGE
nginx-deployment-8fd685b7d-8bjc2 1/1 Running 0 9s
nginx-deployment-8fd685b7d-gv7v2 1/1 Running 0 9s
% kubectl get service
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 24h
nginx-service ClusterIP 10.99.65.41 <none> 80/TCP 15s

```
% kubectl apply -f nginx-service.yaml
service/nginx-service created
% kubectl describe service nginx-service
Name:              nginx-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=nginx
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.99.65.41
IPs:               10.99.65.41
Port:              <unset>  80/TCP
TargetPort:        8080/TCP
Endpoints:         10.244.0.15:8080,10.244.0.16:8080
Session Affinity:  None
Events:            <none>
```

Via above command u can see the updated status of the yaml file

```
% kubectl get pod -o wide
NAME                                READY   STATUS    RESTARTS   AGE    IP           NODE       NOMINA
ESS GATES
nginx-deployment-8fd685b7d-8bjc2    1/1     Running   0          100s   10.244.0.15  minikube   <none>
nginx-deployment-8fd685b7d-gv7v2    1/1     Running   0          100s   10.244.0.16  minikube   <none>
% kubectl get deployment nginx-deployment -o yaml > nginx-deployment-result.yaml
```

Project :
Web application requisition to its DB .
: first we will create a mongoDB pod
In order to talk to that pod we gonna need a service a internal service means no external requests from the internet are allowed to the pod .
Only component inside the same cluster can talk to it.

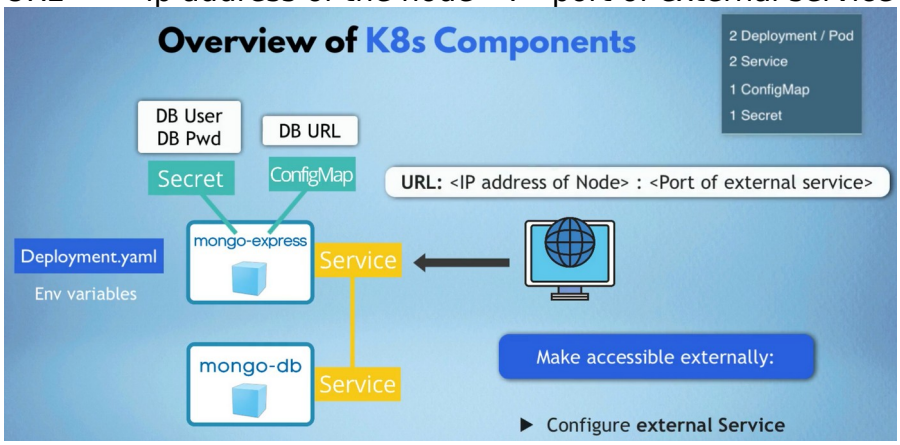
: second we will create a mongo Express deployment
One we need DB URL of mongo DB so that mongoExpress can connect to it .
Second one is credentials username and password so that to authentically to DB .

And we can pass these credential and DB url information to the mongoExpress through mongoExpress deployment.yaml file(config_file) through ENV_variables.

Create a configMap that contains DB_URL .
Create a secret that contains Credentials.

And we gonna reference both of these inside the config_file .

So we need mongo express to be accessible through browser for that we need to create a external service. That would allow external request to talk to the pod .
URL =><ip address of the node> : <port of external service>



Browser request flow through k8s components :

Request come from the browser -> mongo express external service -> mogo_express pod -> internal service of mogo_DB(configMap= database URL) -> mongoDB pod and authenticate the request using the credentials .

1. Create a mondo DB deployment config_file
So to docker mongo images -> to see what are the external configuration or ports etc information about the image .

So this is a configuration file that will be checked in a repo -> so u don't want to show credential secrets in configuration file itself .

So we will create a secret from where we wanna take the values .

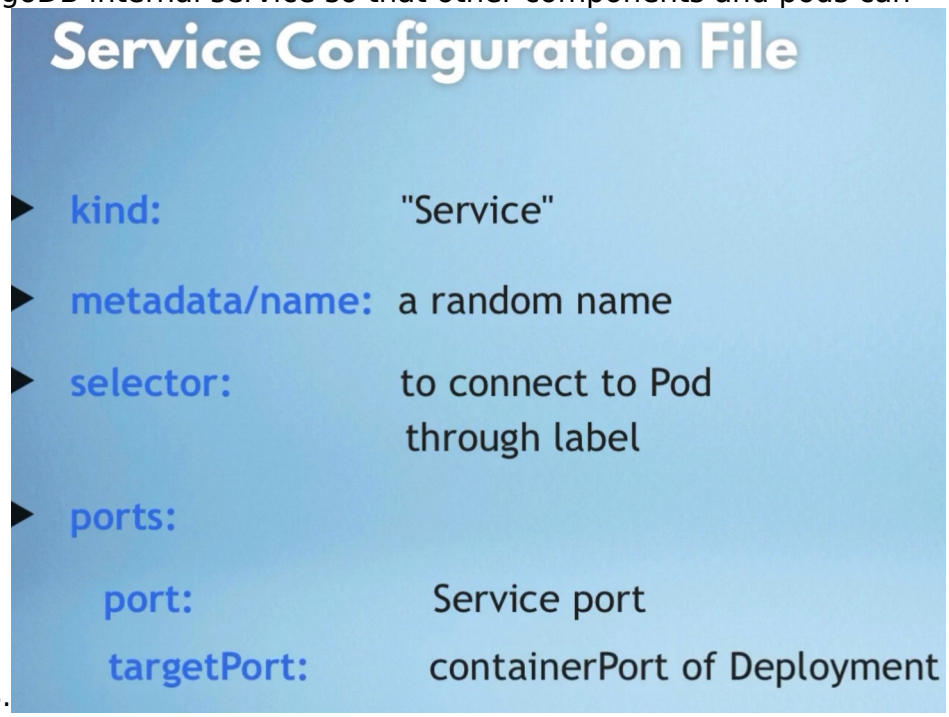
Create secret
Generate -> base 64 for more security .
: echo -n 'username' | base64

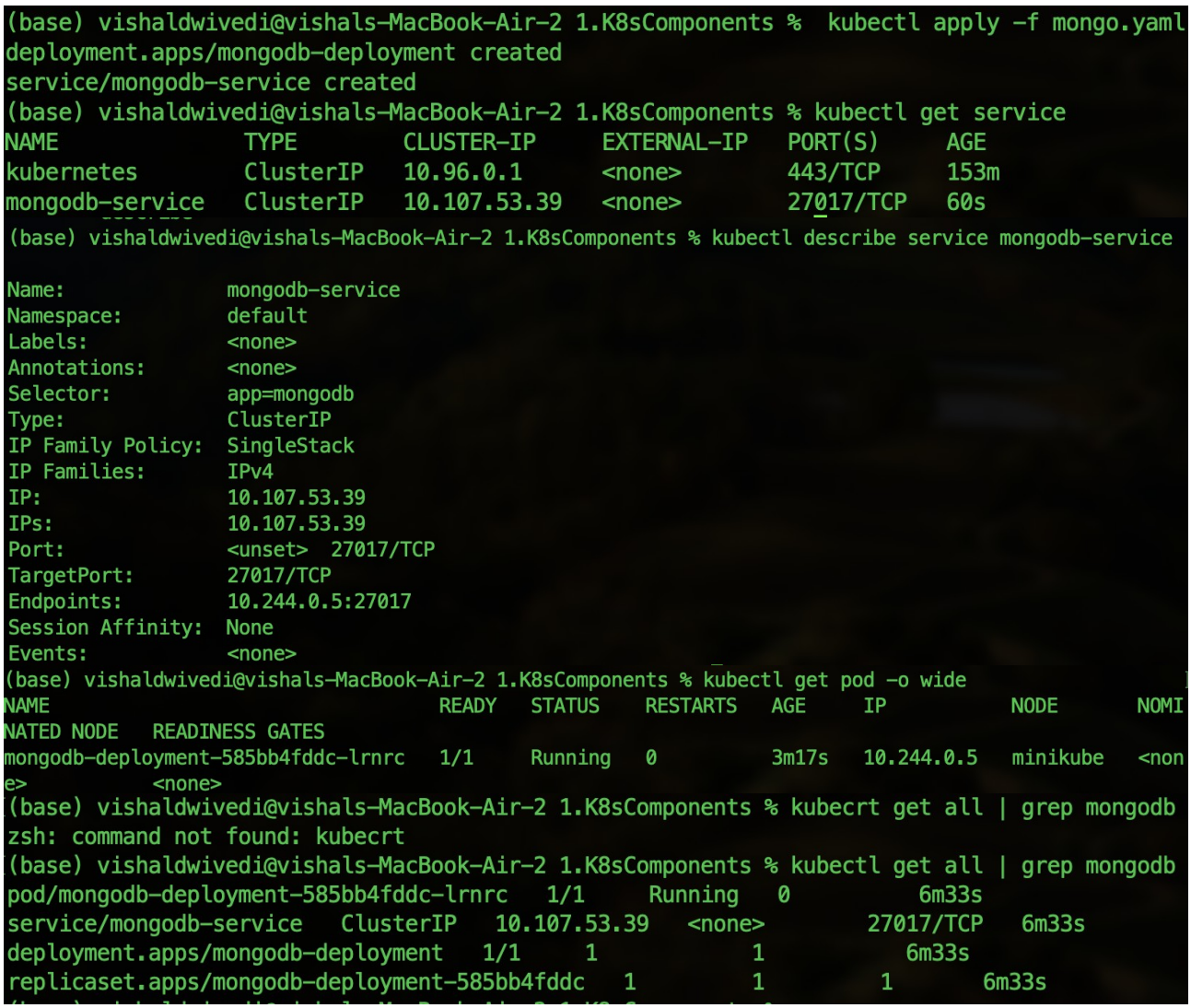
: echo -n ‘password’ | base64

```
(base) vishaldwivedi@vishals-MacBook-Air-2 1.K8sComponents % kubectl apply -f mongo-secret.yaml
secret/mongodb-secret created
(base) vishaldwivedi@vishals-MacBook-Air-2 1.K8sComponents % kubectl get secret
NAME             TYPE     DATA   AGE
mongodb-secret   Opaque   2      39s
```

Only when we apply secret then u can use that on your configuration file before apply ing configuring file .

Next is create in mongoDB internal service so that other components and pods can talk to that mongoDB.

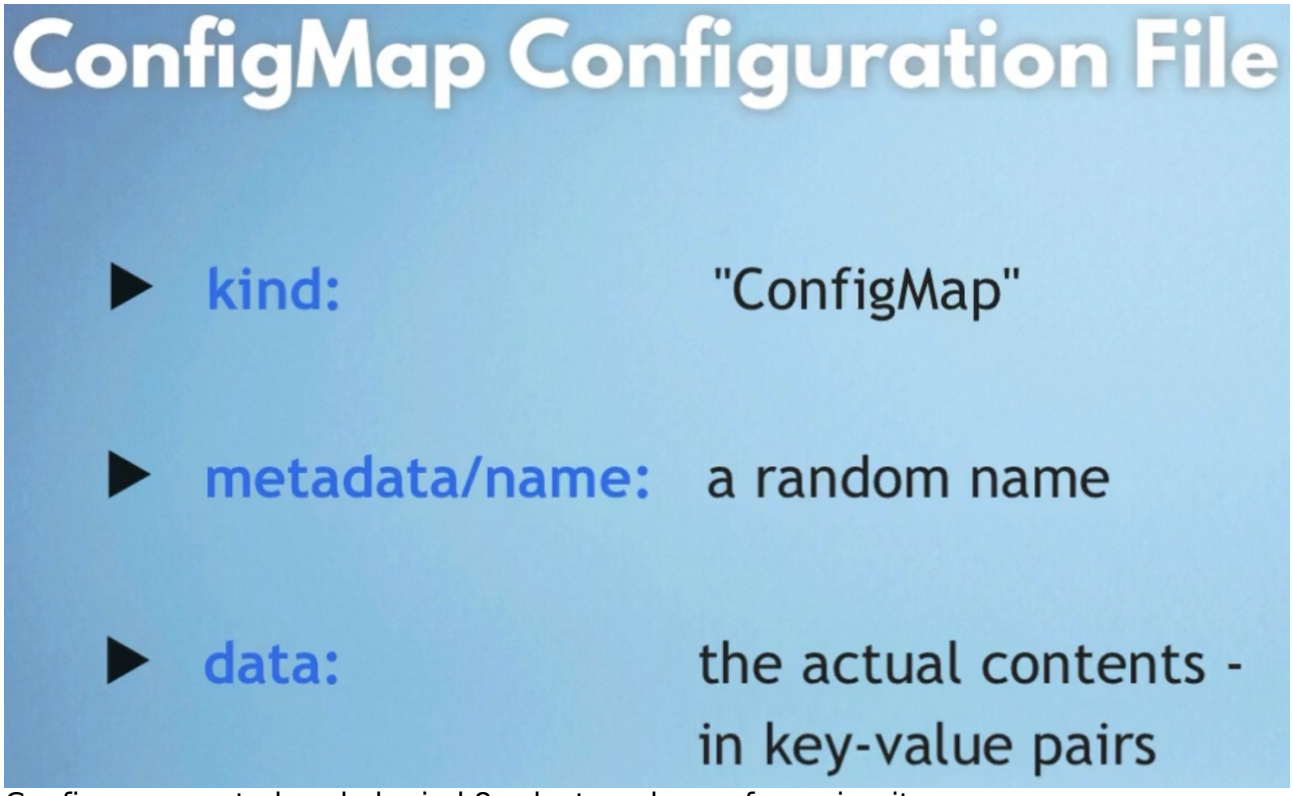




Next we will create a mongoExpress deployment and its external service. Also a external configuration where we will put Database url for mongoDB (configMap)

We need to tell it which database to connect to internal service mongoDB address .

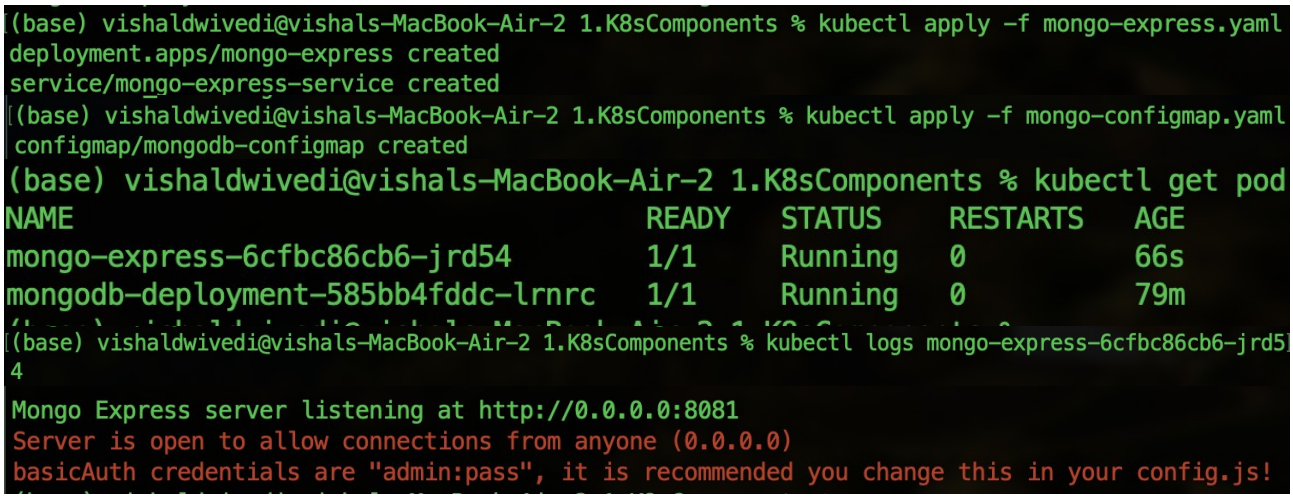
Which credentials to authenticate >



Configmap must already be in k8s cluster when referencing it .

```
- name: ME_CONFIG_MONGODB_ADMINUSERNAME
  valueFrom:
    secretKeyRef:
      name: mongodb-secret
      key: mongo-root-username
- name: ME_CONFIG_MONGODB_ADMINPASSWORD
  valueFrom:
    secretKeyRef:
      name: mongodb-secret
      key: mongo-root-password
- name: ME_CONFIG_MONGODB_SERVER
  valueFrom:
    configMapKeyRef:
      name: mongodb-configmap
      key: database_url
```

So these are the three env variable that mongoExpress need to authenticate with mongoDB .

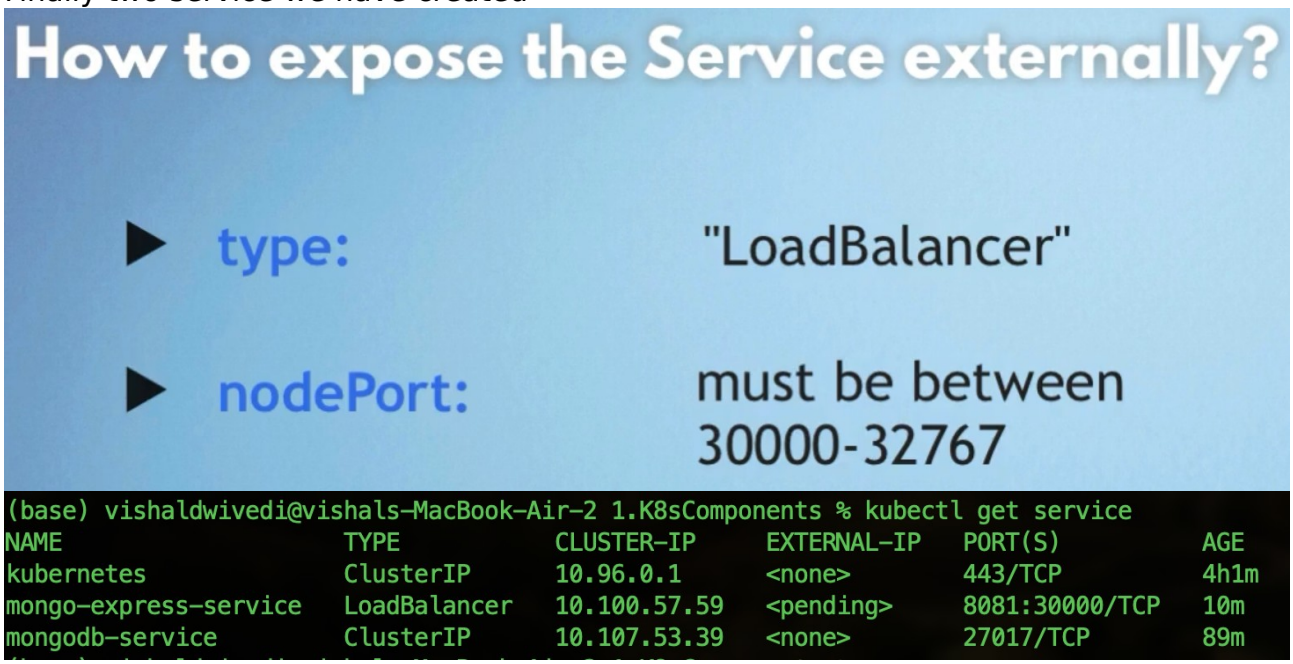


Yes!! Mongo - Express server has started . And data base is connected .

Now final step is to access mongo express from the browser .
So for that we need external service for mongo express .

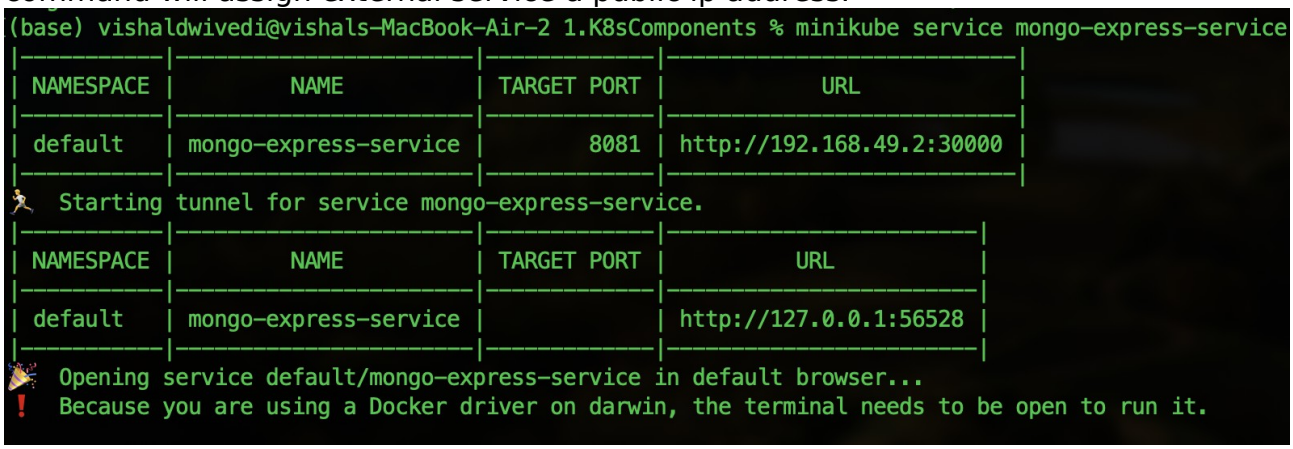
Load balancer here accepts external requests assigning a service a external ip address .
Node port -> port where this external ip address will be opened .
Port that u need to put into the browsers to access this service

Finally two service we have created ->

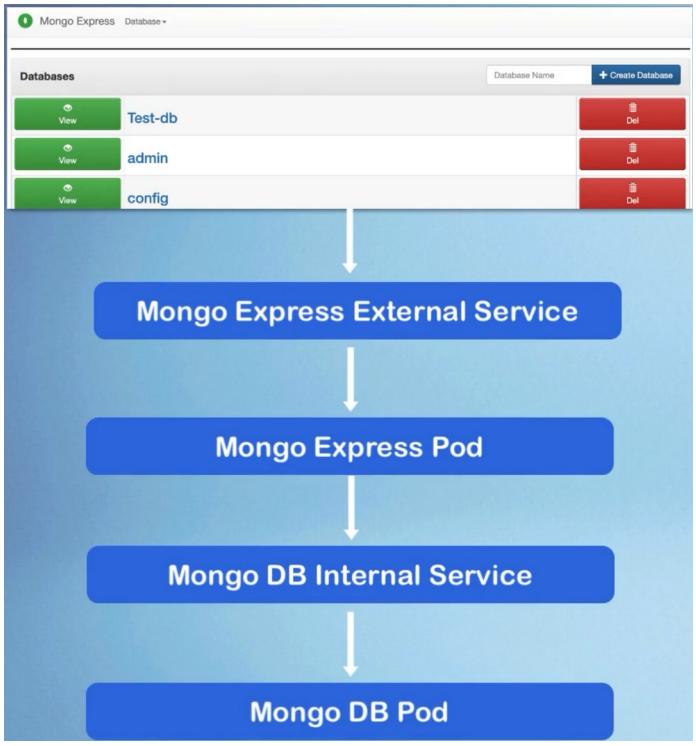


Cluster ip will get service its internal ip address only reachable with in the cluster.
Loadbalancer will also give internal ip address to the service but also give service a external ip address where the external request will be coming from .
In clusterIp u have ports for only internal ip address where as we can see for loadbalancer service we have both 8081 port for internal and port 30000 for external ip address.

Currently we can see that for load balancer external service we have pending ip address ->
In minikube we have to do command : minikube service <serviceName> // this command will assign external service a public ip address.



give a URL to external service in minikube -> minikube service mongo-express-service


Mongo Express
Database
Databases
Database Name
Create Database
View
Test-db
Del
View
admin
Del
View
config
Del
Mongo Express External Service
Mongo Express Pod
Mongo DB Internal Service
Mongo DB Pod