Kubernetes notes

## Kubernetes Installation

Kubernated uses many components. Majority of them being the K8 master nodes, K8 worker nodes, KubeCtl (command line manager for K8) and kube admin (UI based kube admin I suppose).

For the development system, we can deploy minikube – this will give us a single node setup of K8 master and worker.

### Installation commands for Minikube in Ubuntu/Linux.

sudo apt install curl \*\* if you dont have curl installed already.

Minikube needs some VM. Install oracle vm

sudo apt-get install virtualbox

(some popup will be shown askinf to set some machine owner key. press ctl+page down and enter a password and reenter as asked. Reboot the system. continue to boot-ignore any ask for the MOK)

sudo apt-get install virtualbox—ext–pack (this didnt get installed for me)

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube

minikube start

This will provide an update on successful startup of the minikube.

## Basic Components:

### Node -

A physical or virtual machine. Nodes can be of 2 types- master node and worker node.

### Master node -

Does not do any actual job of running apps, but it manages how the kube cluster is managed. In a prod setup, there will be minimum 2 master nodes in the cluster.

The machine resources for a master is hence allocated less.

### Worker node -

The one doing actual job of running application in the so called pods. Worker nodes are normally allocated plenty of machine resources as possible.

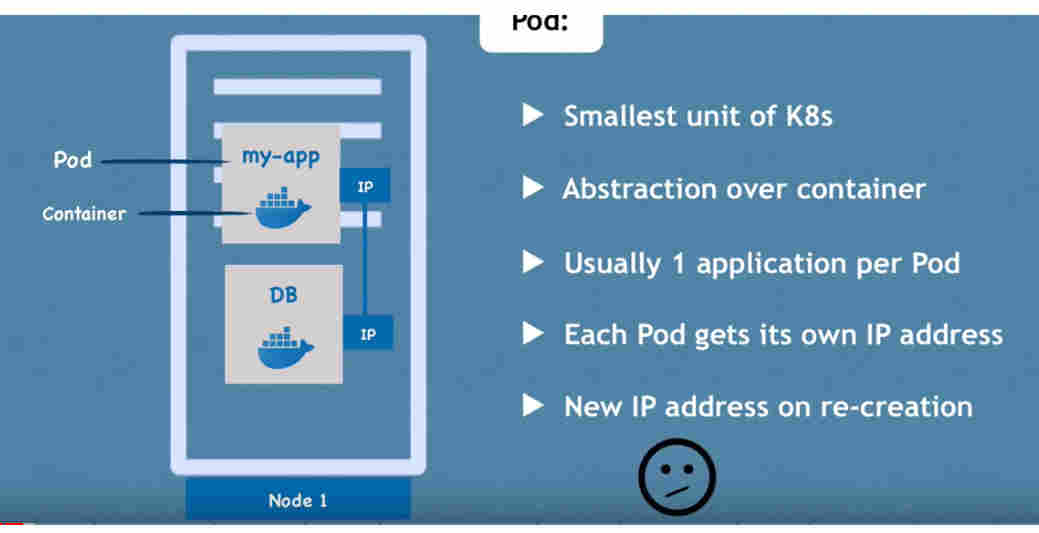
## Kubernetes components

### 1. Pod

Smallest K8s unit. It is an abstraction on the container(eg docker) so that container is not direcly linked to the node in any way. You can always change the container within a pod if you wish to.

Pod is NORMALLY meant to run ONE app container(even though technically we can run multiple containers within a pod- done for some dependent or side apps for the main app).

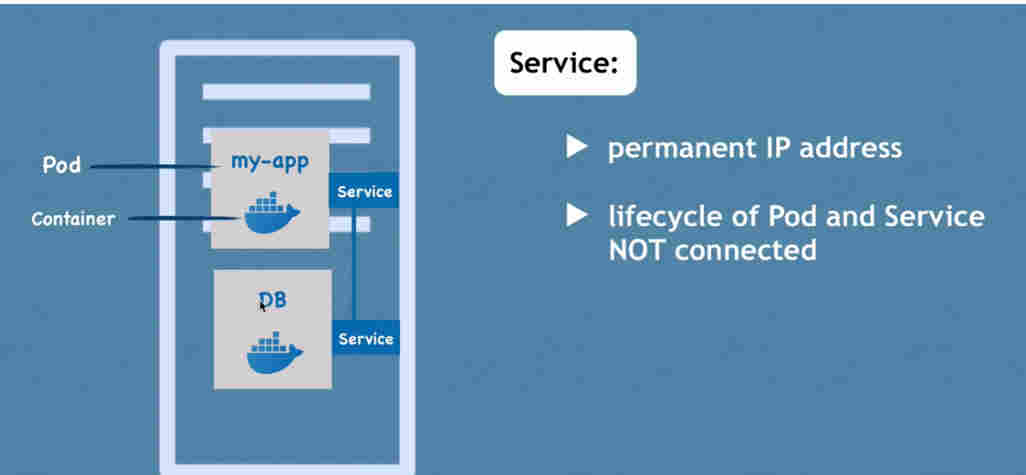
Each pod gets its own IP address for communicating each other. Pods are ephemeral (die easily when node out of resource etc etc). When a new pod is created for a dead pod, it gets new IP address.



### 2. Service

It is a static/permanent IP address that can be attached to a pod. When creating the service, specify it to be Internal or External.

The service works as Load-Balancer too by defalut.



**External Service** – this enables the pod service to be accessible from outside the node.

**Internal Service** – this service can be accessed only from within the **node**.

### 3. Ingress

It enables the service IP address:port to be changed to the domain name or something.

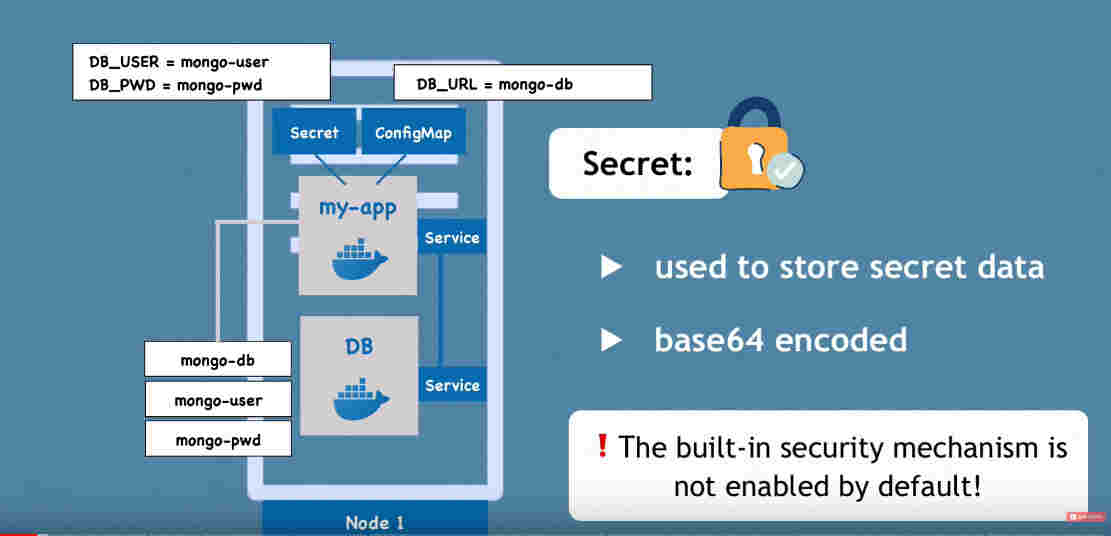
So an external request will come to the Ingress first, and then will go the the Service.

### 4. Secret and ConfigMap

Config map is used to store the global variables to be used in a Pod. This normally saves the db-url etc.

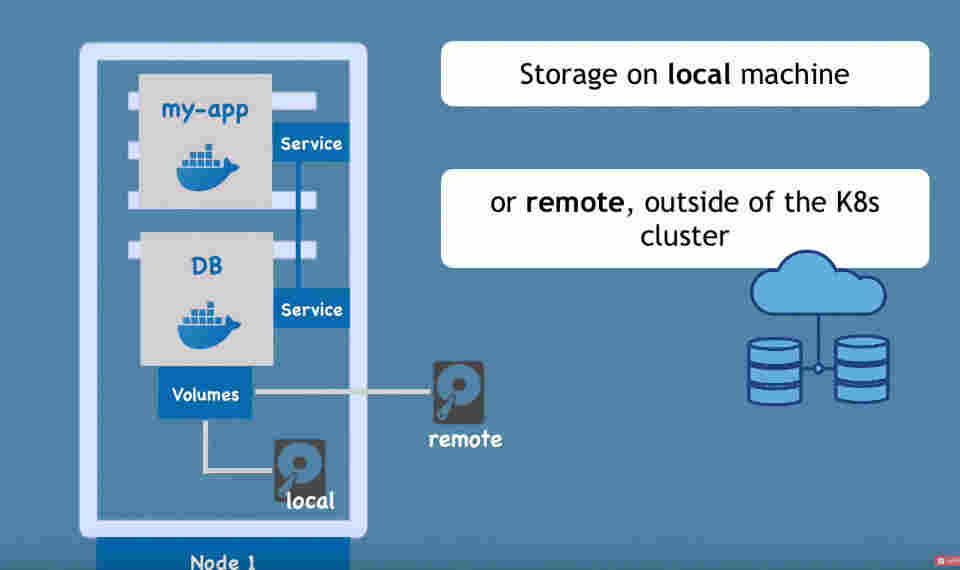
Secret is used to store any credentials like the username and password etc.

config and ConfigMap are linked to the pod.



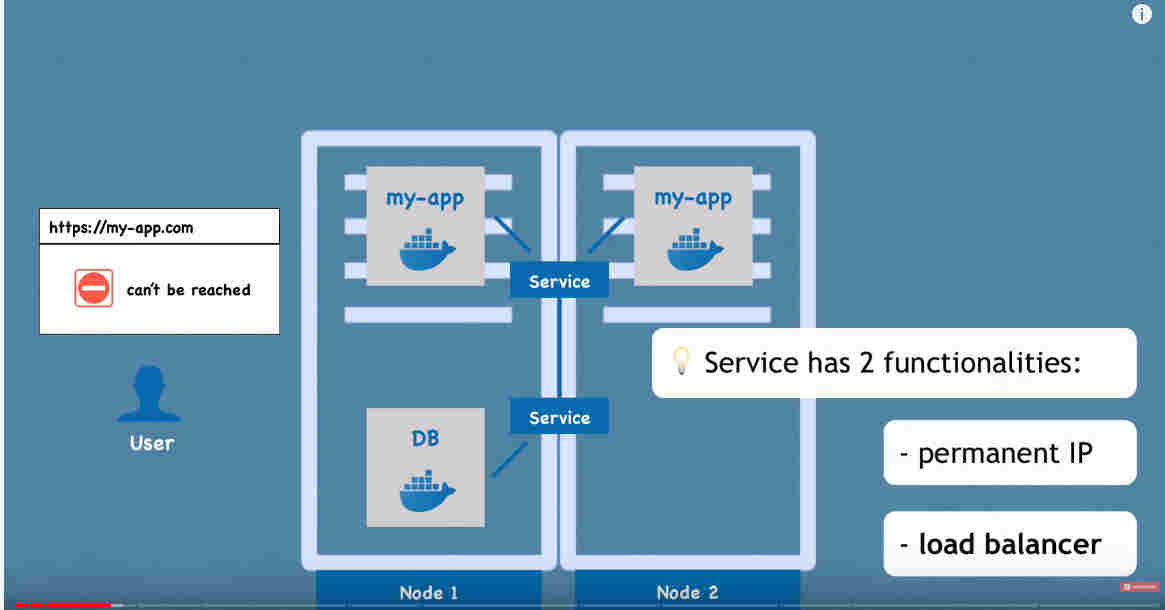
### 5. Volumes

It is used to connect a storage (internal in node or an external/cloud )to the pot.



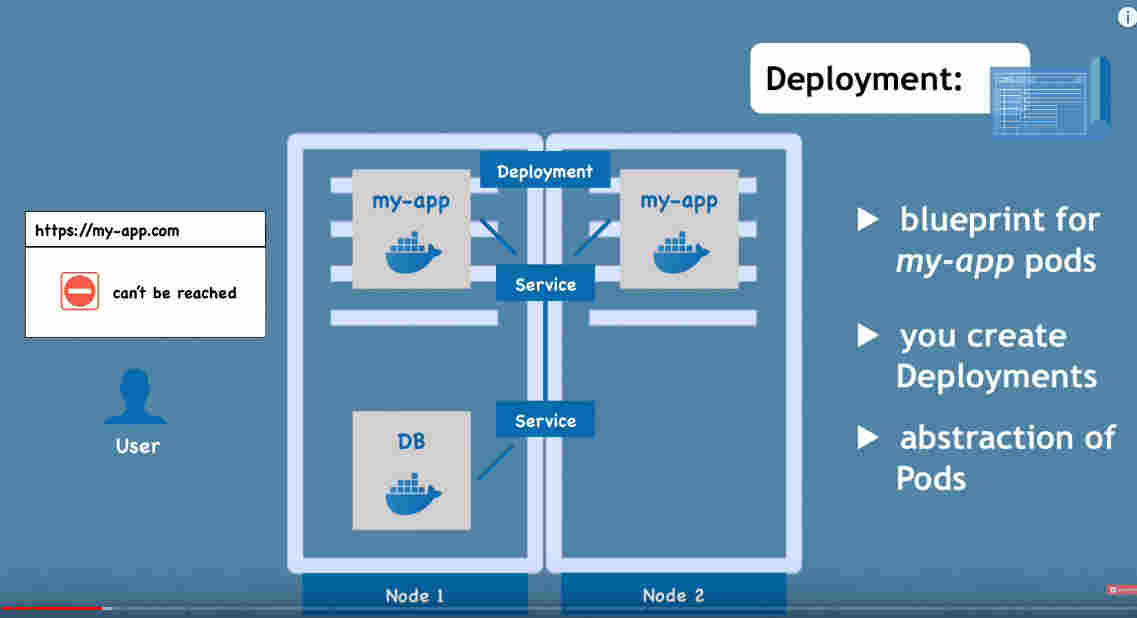
### 6. Replication

In Practice, the pods are always replicated onto multiple nodes. This is done using deployment specification component of K8s. In short, we don’t work with pod, but with deployments.



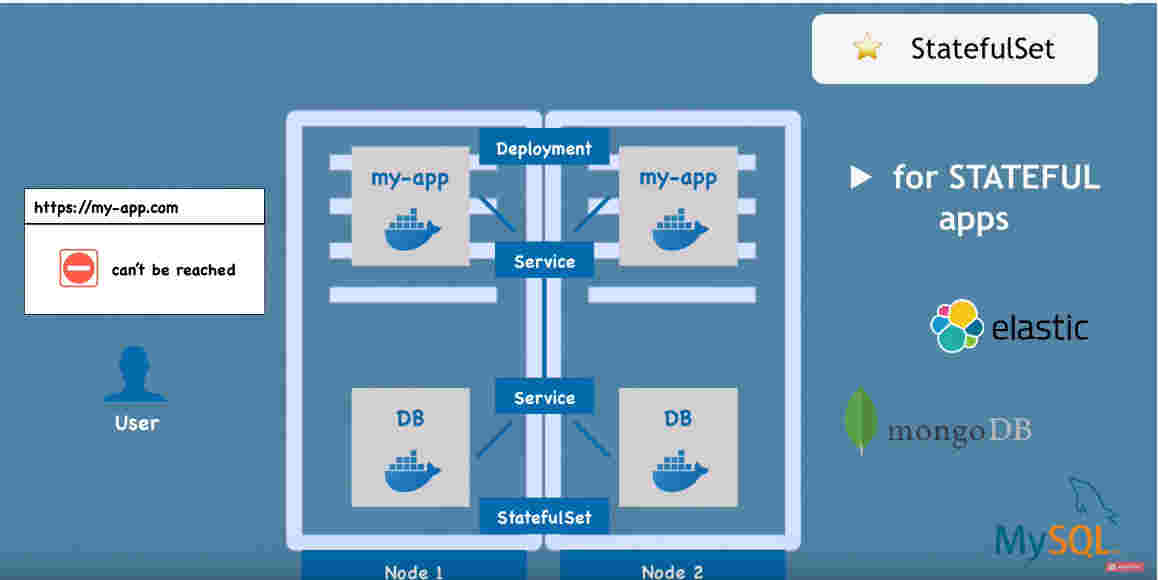
### 7. Deployments

This is a blueprint of all pod specification for our application. This will tell how many pod replicas to be created for each of the pods used(have to specify explicitly) in the application. Any abstraction below the deployment is automatically managed by K8s.

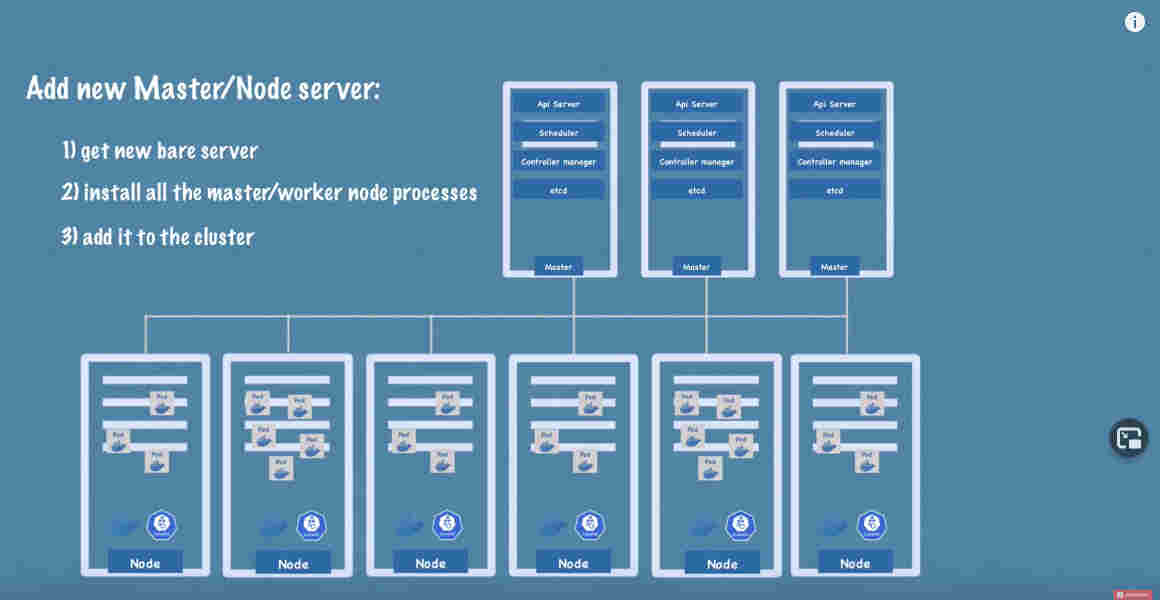


### 8. StatefulSet

This is used for the deployment of the stateful services like the database. Stateful services do not use the regular Deployment K8s components. But configuring this is very tricky. Its a good practice to host GB outside k8s cluster and keep stateless apps in K8s.

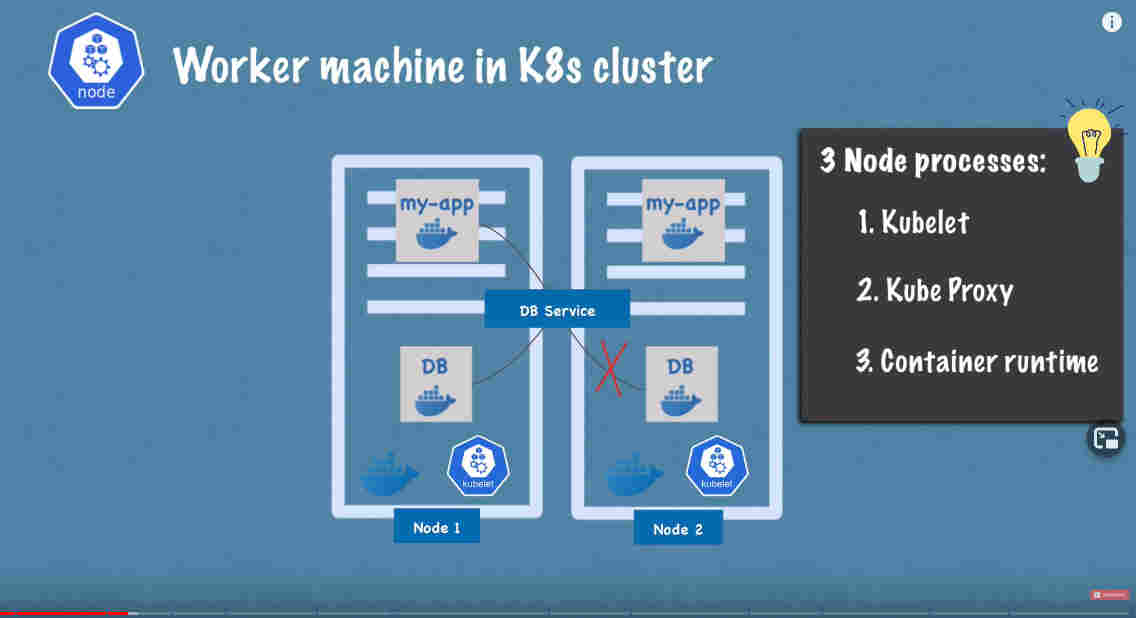


## K8s Architecture



### Worker Node in K8s

A worker node should have 3 processed running so that the pods can be scheduled and managed– kubelet, kubeproxy, container runtime.



### Container runtime –

Any container runtime(like docker) should be running on a node.

### Kubelet

Interface between the container runtime and the NODE. Kubelet will take pod config and startup the pod with a container inside and assign the resources from node to pod.

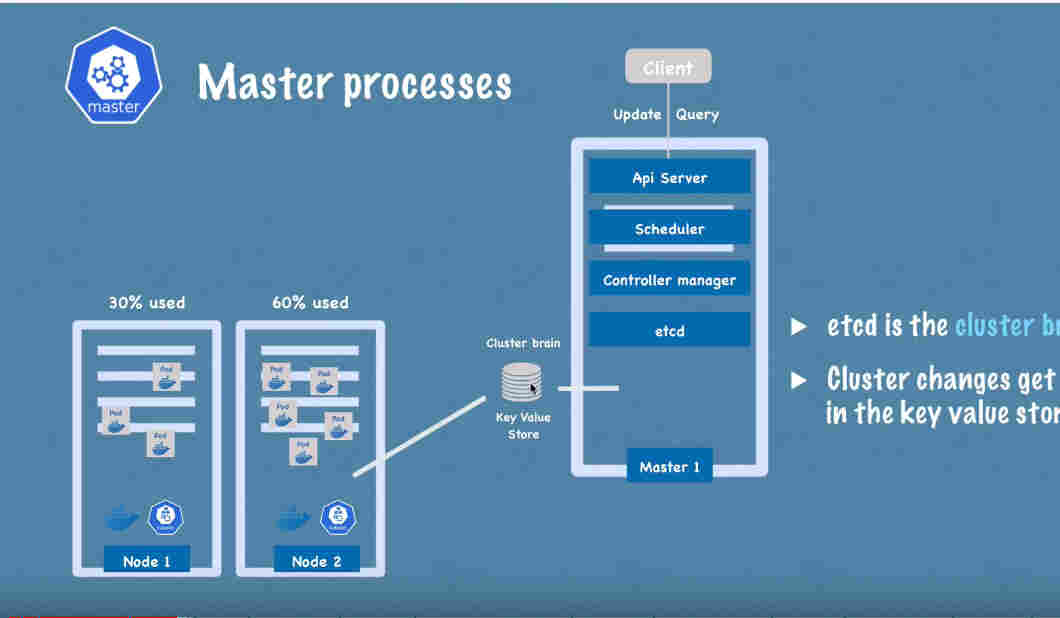
### Kubeproxy

This will forwards the requests coming to the services to the actual pods. This does intelligent routing(requests as contained within a node wherever possible) to reduce network overhead.

### Master node in K8s

Master nodes control the K8s cluster.

4 processes will be running – API server, scheduler, controller manager, etcd



### API Manager

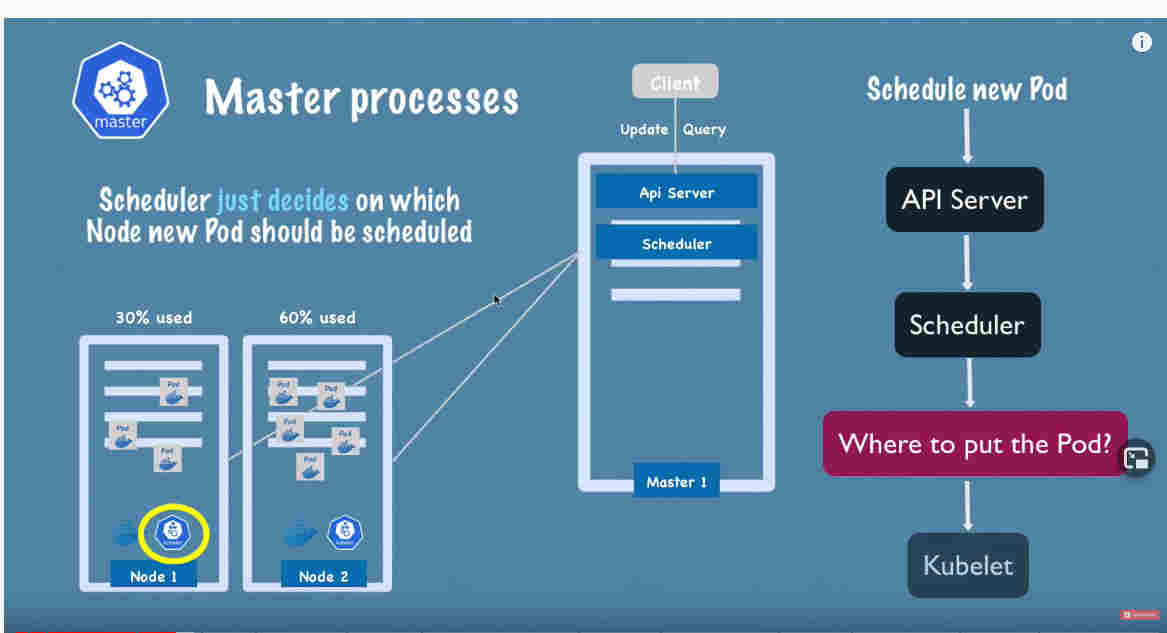
This is the single entry point/cluster gateway to the K8s cluster(for calls like deployment, pod scheduling, health check query etc etc)

Authenticate any query or update request and forward the requests to other processes.

### Scheduler

Takes care of the scheduling of the pods.

Looks at the pod scheduling request resource needs, cross check the resource availability of the resources on all worker nodes(lookup data in etcd) and then decides and forward the request to corresponding kubelet.



### Controller Manager

Monitor Detects the pod state changes like crash.

Inform the scheduler to take action. Scheduler follow the steps just like a new pod scheduling.

Controller Manager – > Scheduler – > Kubelet

### etcd

This is a key value store of whole cluster info, but not the application data.

All changes to cluster (new pod creation, pod dying etc)are recorded in etcd.

## KubeCtl Commands

Official documentation -

https://kubernetes.io/docs/reference/kubectl/overview/

### Install the Kubectl

sudo snap install kubectl (use –classic if throwing error)

### Get the nodes info

kubectl get nodes

minikube kubectl get nodes (this works too if kubectl is not installed on system)

### Get the pod info

kubectl get pod #pods also will work

### view cluster services

kubectl get services

- there is a default service called ‘kubernetes’.

### Creating a deployment (and hence the pod )

In the below example, ***nginx-depl*** is the deployment name.

kubectl create deployment nginx-depl –image=nginx

### Get Replica sets

The replica set configuration is done within the Deployment.

kubectl get replicaset

### Debugging/ Checking pod logs

kubectl logs <<pod name >>

### Describing the pod info

This gives a details pod log info.

kubectl describe pod <<pod name>>

### View the terminal of the a pod

-it stands for Interactive terminal.

Kubectl exec -it [podname] – bin/bash

### Delete a deployment

Kubectl delete deployment [deployment-name]

### Deployment using yaml files

kubectl apply -f [filename.yaml] # -f indicates file

### Get all info of the cluster

kubectl get allocate

### K8s Abstraction Layers



## K8s configuration files

Any config file start with API version and Kind.

API version specify the version.

Kind will tell what kind of config file it is -Deployment, Service etc

after this info the config file has 3 parts

### Metadata

This is info like name and label etc

### Spec

This is the actual specification for the configuration. The parameters will change based on the Kind.

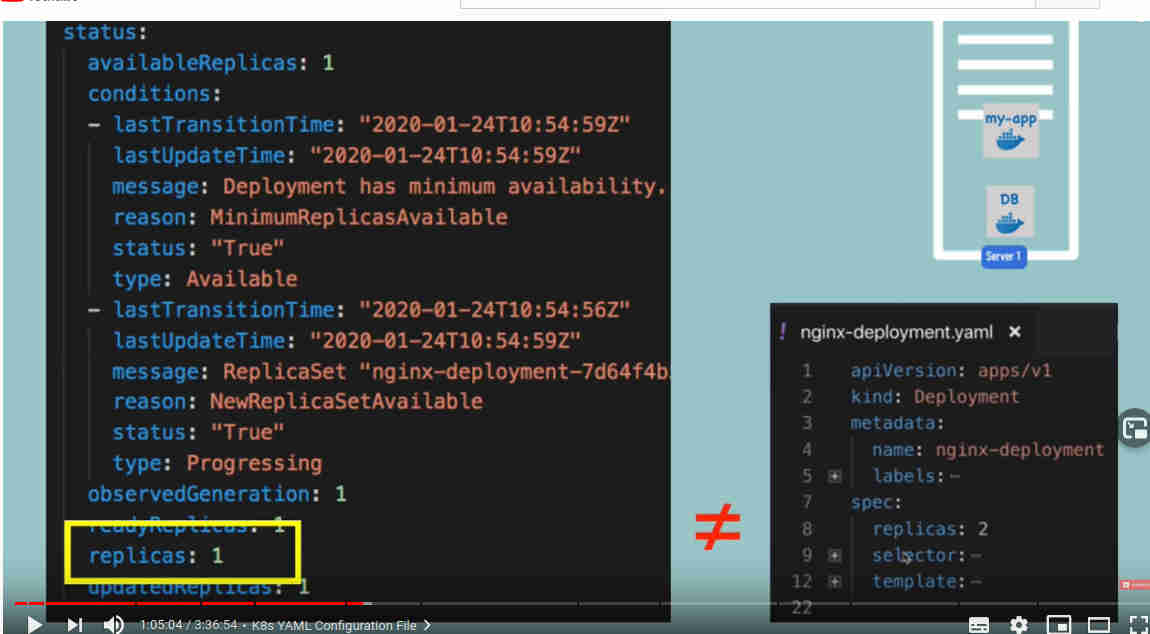
### Status

This is a special section which is NOT done by user but by K8s.

K8s will compare the desired and the actual state continuously and take necessary action.

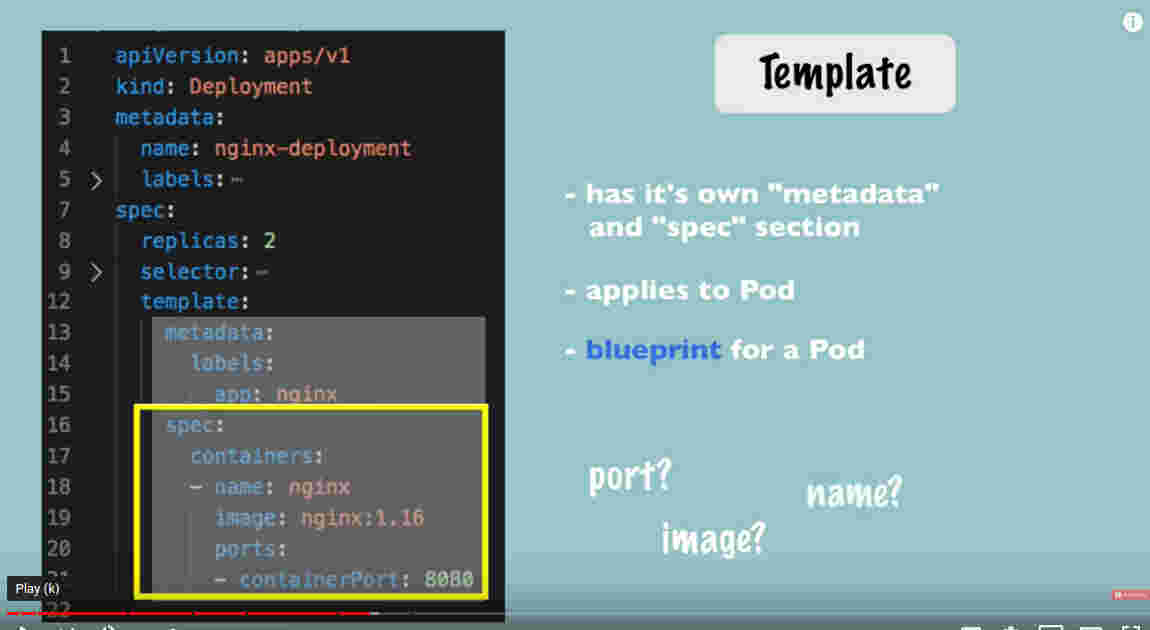
We can view the status section by using get command with out param as yaml

kubectl get deployment [deployment-name] -o yaml



### Spec- Template

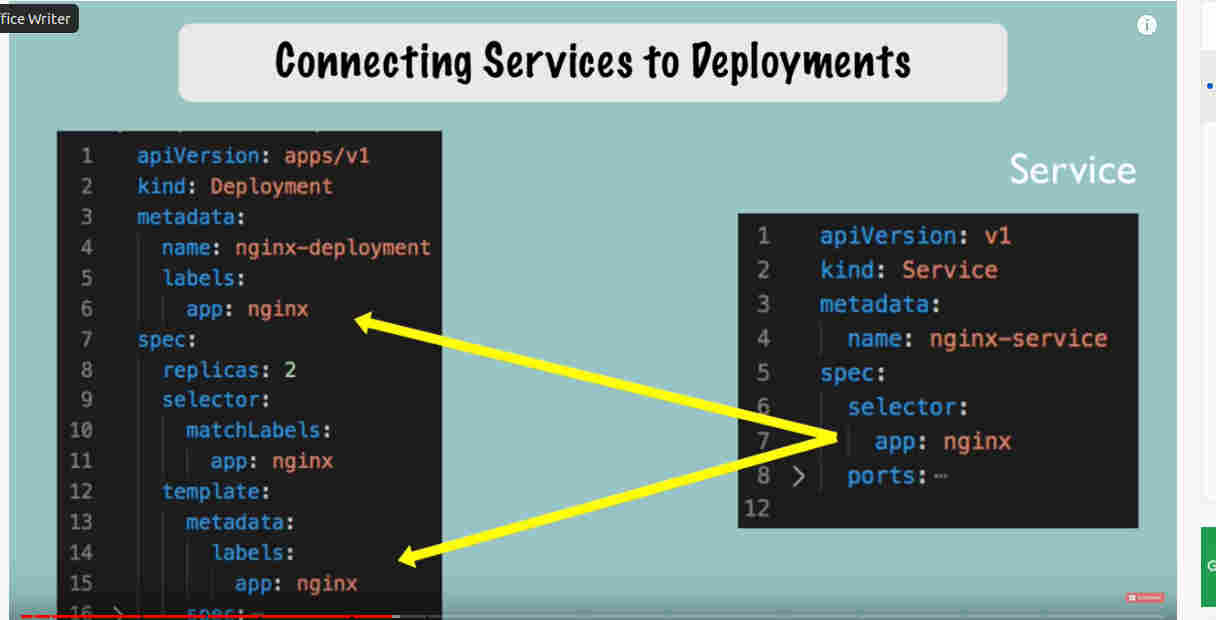
In case of deployments, Spec section will have ***template*** which is used for configuring a pod. Template has its own spec under it.



The labels specific to the deployment and the pod comes from the below sections. This is important to connect the deployment and the corresponding pods.

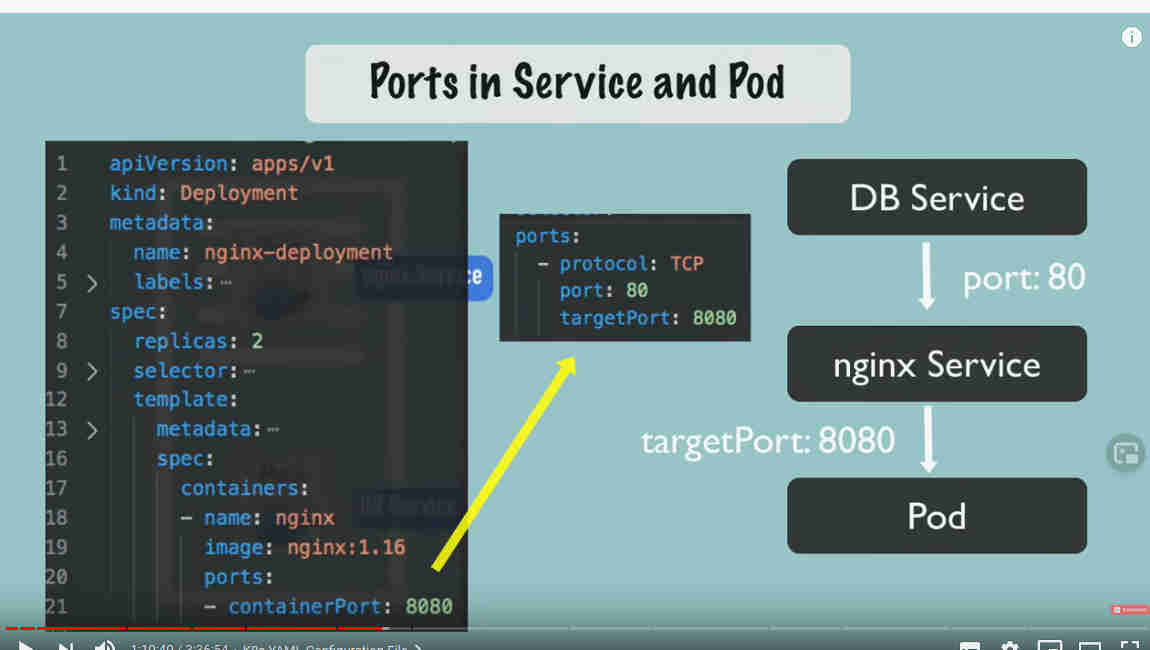


The Service and the deployment/pods are also linked based on the labels.

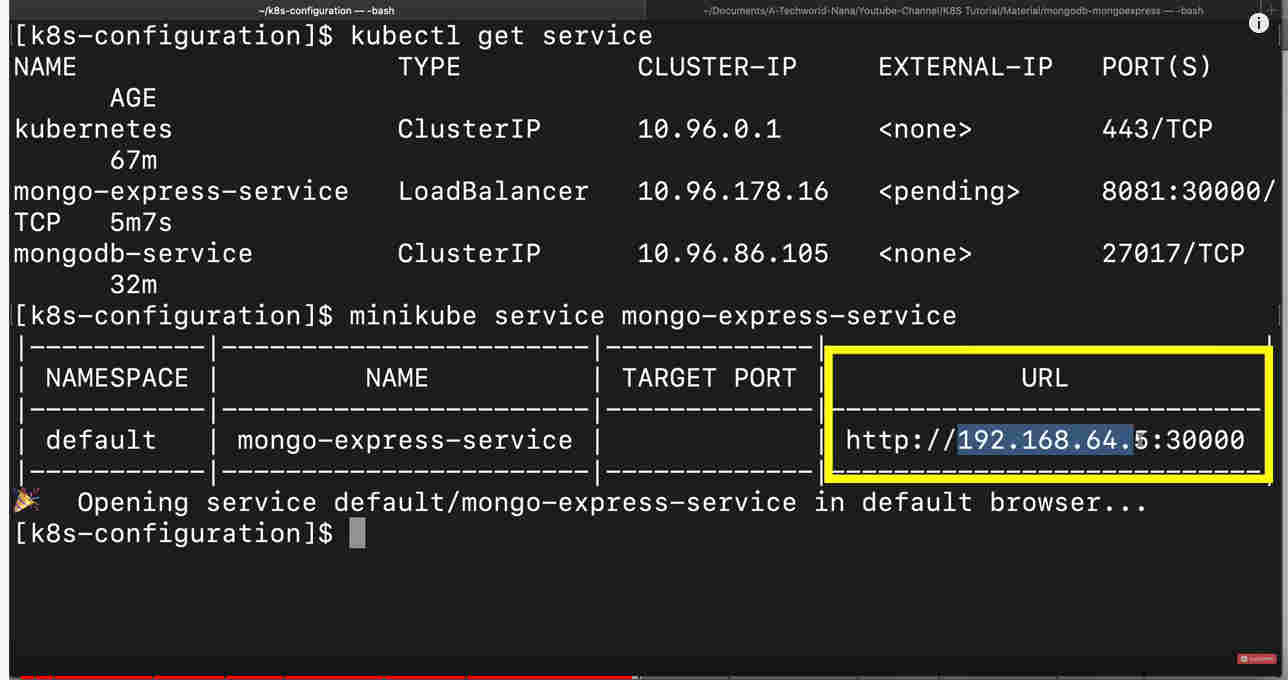


### Linking the Service port and pod ports

Target port in service config and the container port in pod spec should match. In the below example, service expose port 80 and route the incoming request to 8080.



### External Service Port Details



For Internal Service, a Cluster-IP(IP to be used within the K8s Cluster) will be given. This is internally assigned, we need not do anything.

For External Service (configured by Type=LoadBalancer), a Cluster-IP and External-IP will be assigned by K8s. In the ports section we can see 8081:30000. 8081 is the pod port, and 30000 is the external port or node port.

In PROD setup, the external IP for accessing an external Service will be displayed. Minikube show this as <pending>.

Run the minikube command( minikube service [service-name]) to set the external-IP as in the screenshot above, then it will give the details as above.

## Kubernetes Namespaces

Organise resources in Namespaces

sort of virtual group within the K8s cluster.

### Default K8s Namespaces

They are 1)Kube-system, 2)kube-node-lease, 3)kube-public, 4)default . Minikube had one additional default name space 5)kubernetes-dashboard

### Kube-system -default K8s Namespace

This should NOT be changed. This contains the components for the kubectl and master, managing processes etc.

### kube-public -default K8s Namespace

This contains publicly accessible data.

This has a configMap that contains Cluster Info which can be read without authentication.

Eg: output of the command - > kubectl cluster-info

### Kube-node-lease -default K8s Namespace

Holds the info about the heartbeat of the nodes. This basically translates the info on the availability of all the nodes in the cluster at a time.

### Default-default K8s Namespace

This is the default namespace where the resources are created unless we create a new namespace.

### User created namespaces

Can be done using ***kubectl create namespace [namespace-name]*** or using config file. Example below

*apiVersion: v1*

*kind: ConfigMap*

*metadata:*

*name: mongodb-configmap*

*namespace: my-namespace*

### Need for Namespaces

Keep the components organized.

Gives better overview of what is where and why.

Logical grouping of resources like database , ELK-components, monitoring-components etc.

For different business and technical teams so that components are not overwritten.

Same cluster can be used for DEV, SIT,UAT with namespaces.

Limit the user access to resources and namespaces so that other namespaces are not damaged.

### Deciding the namespaces

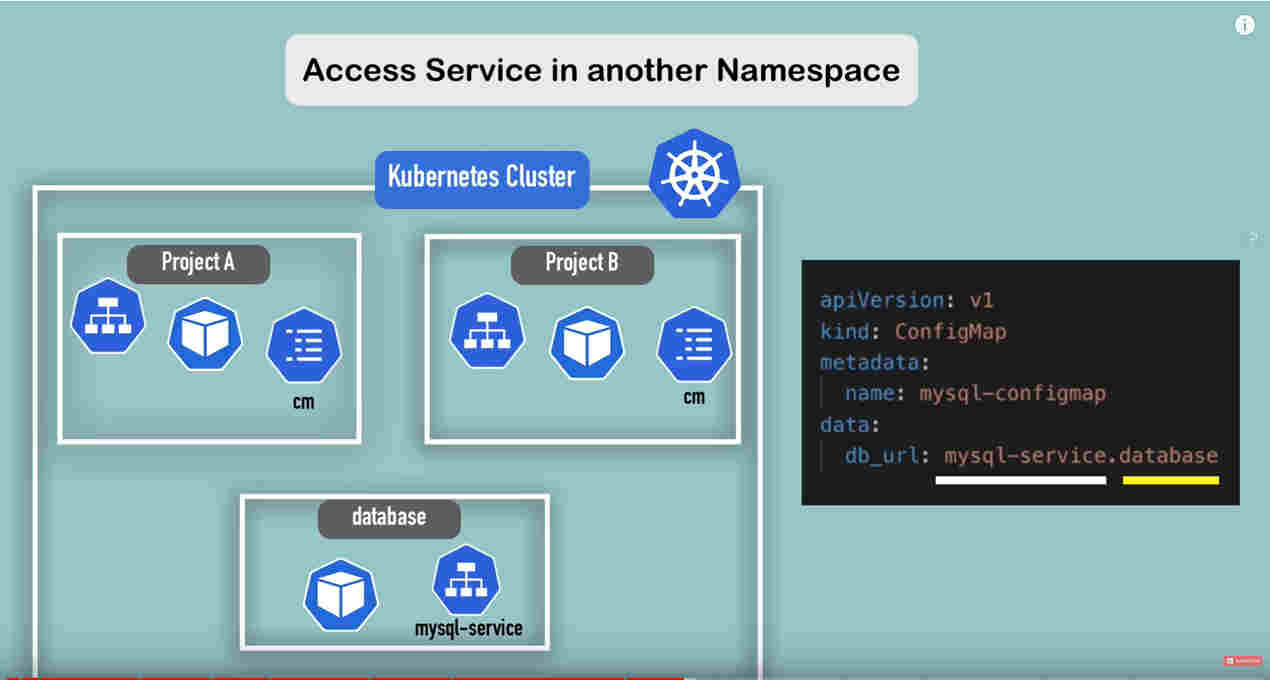
Many resources are not accessible from other namespaces.

ConfigMap,Secret can be accessed only from the current namespace.

Volume and node cannot be namespaced. They are always global.

***Kubectl api-resources - -namespaced=true/false*** gives the corresponding resources.

### Service Access from other namespace

In the example above , the configmap from one namespace is accessing service from ‘database’ namespace. We have to specify the namespace after the service name.

### Adding resources within a namespace

1) using kubectl flag to specify the namespace

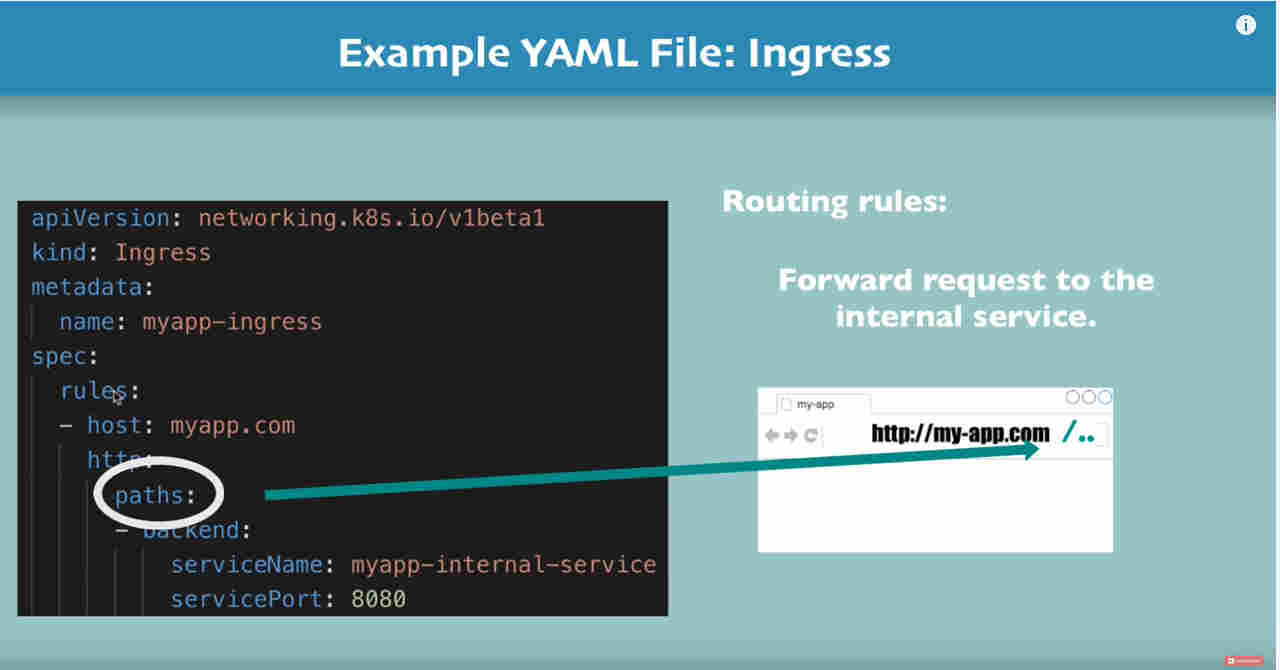
***kubectl apply -f [yaml-file-name] - -namespace=[namespace-name]***

2) Specify the namespace in the yaml in metadata section.

## K8s Ingress S**etup**

### **Yaml config**

External Service will expose only the IP and Port outside. Ingress will convert it to dns.

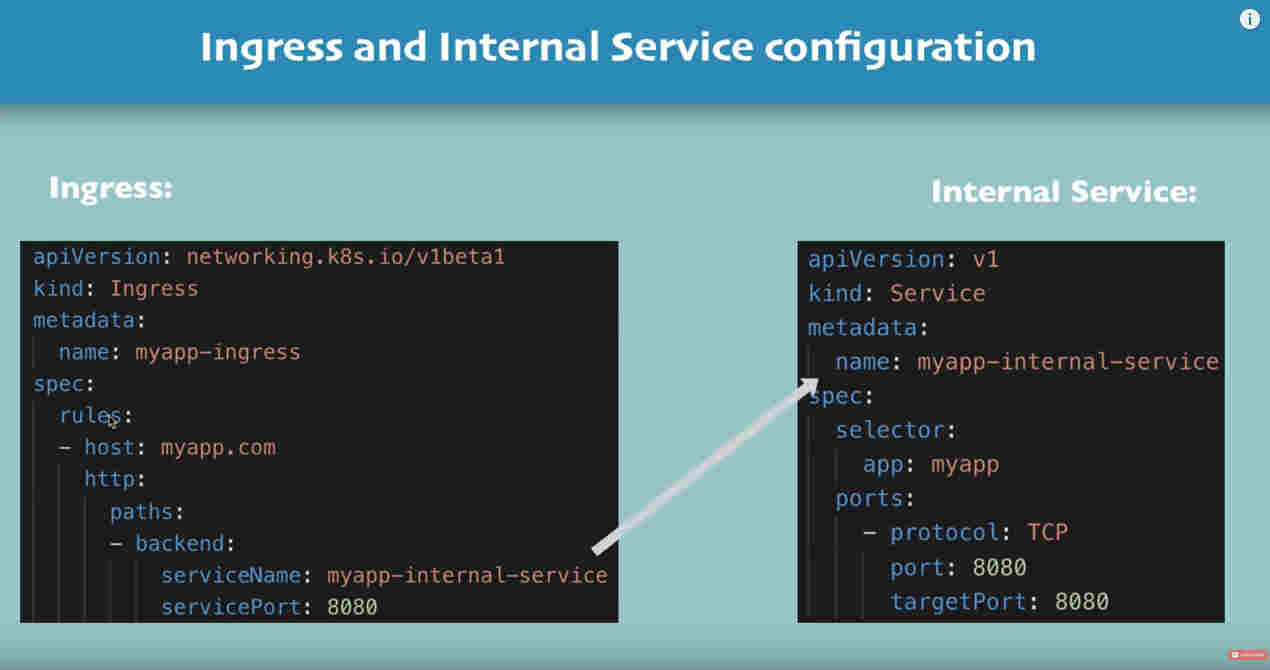


In sample above, look at kind, and Spec.

Host – specify the hostname.

Http – connection from the ingress to the service (NOT from browser to ingress component).

### Ingres service linking

The service name should be mapped properly.

Backend.ServicePort in ingress should be same as Service.Port(NOT target port-this will be container port)

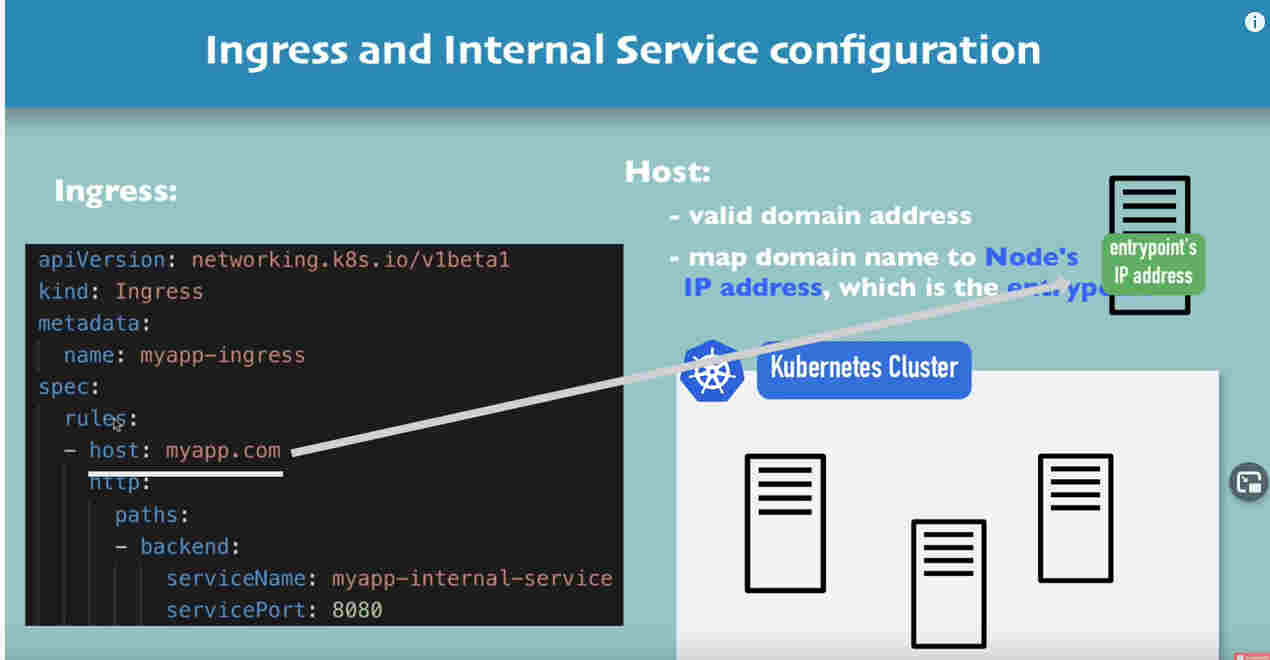
### Deciding the hostname setup

This should be a valid domain name.

The hostname should be mapped to the IP address of the **entrypoint Node** of the K8s cluster.

The entrypoint for a k8s cluster can be set as a node within the cluster or a server outside the cluster. So the dns mapping to be done accordingly.

Check the diagram below



### Ingres controller

Ingress config is only a specification for the rules.

The ingress rules are evaluated by the Ingres implementation which is Ingress Controller pod in the node.

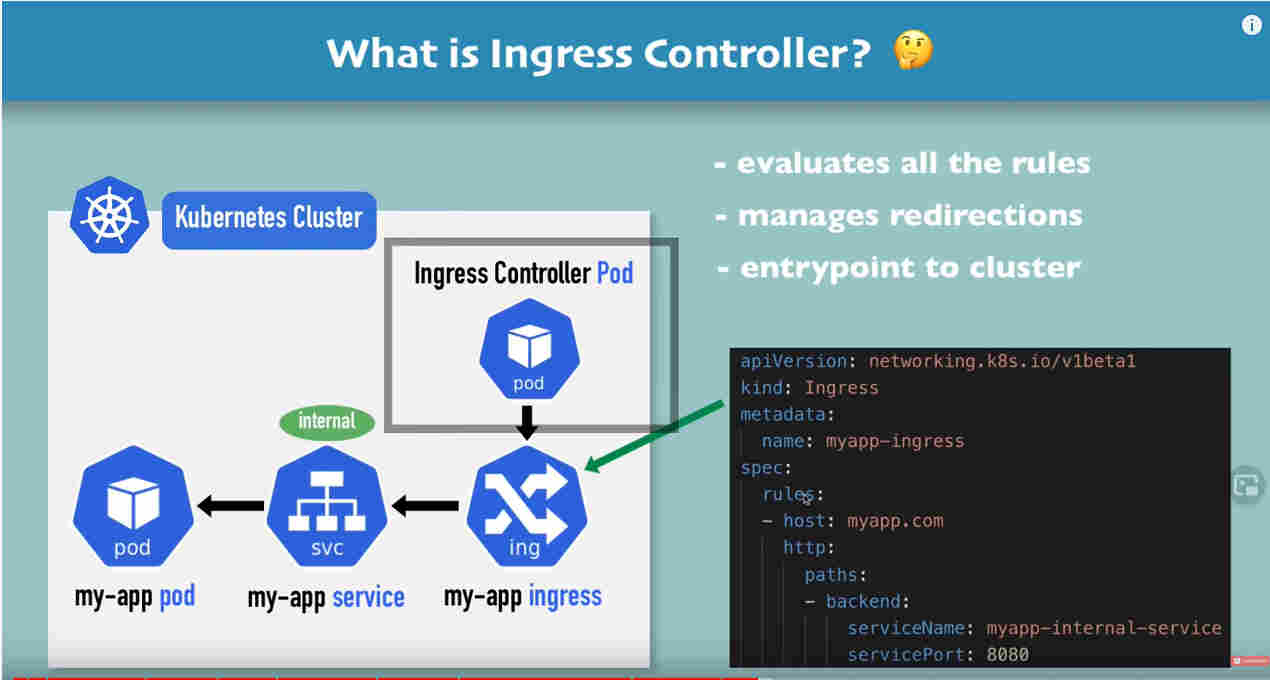
This does the rule evaluation and routing/redirection.

Act as entrypoint to cluster.

Several 3rd party ingress controllers available.

Ingress has a default backend which is used in case there is no matching rules available.

Ingress controller can route to different hosts(when using subdomains) and route to different paths(url paths)



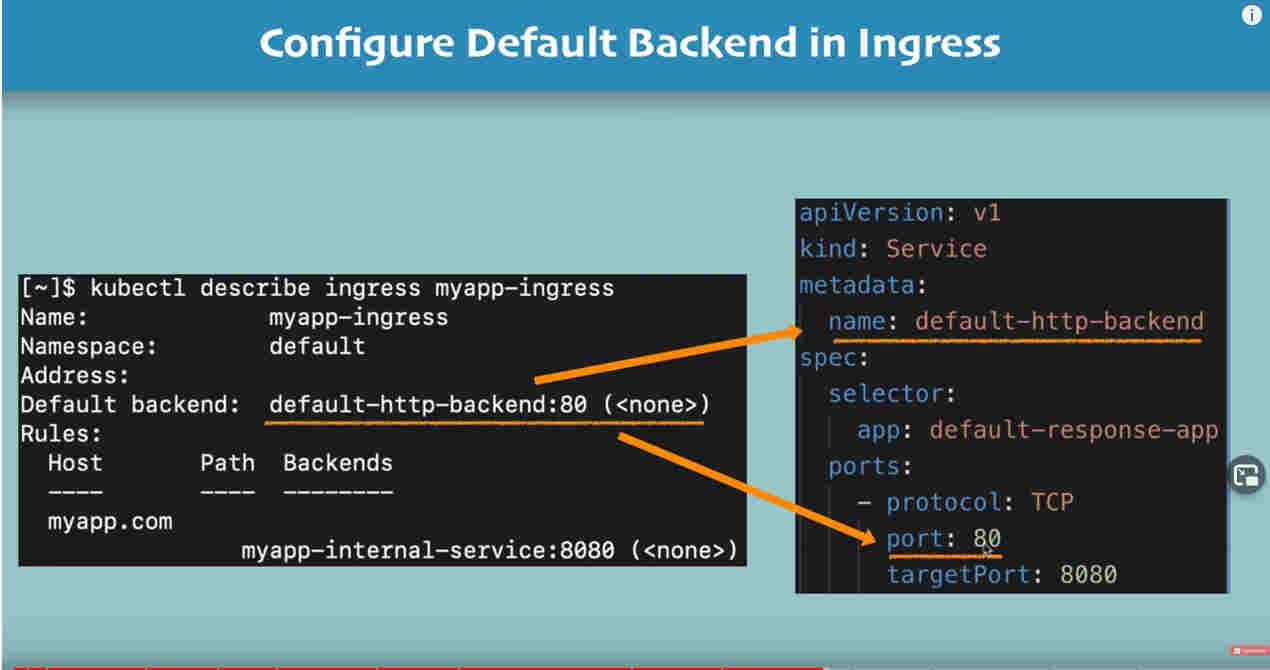
### **Creating** Ingress for the kubenetes-dashboard(to access it with domain name)

The ingress config was not working

### Configuring Default Ingress backend

There will be defalut backend for ingress. We can map this default to a different one which can give meaningful response.

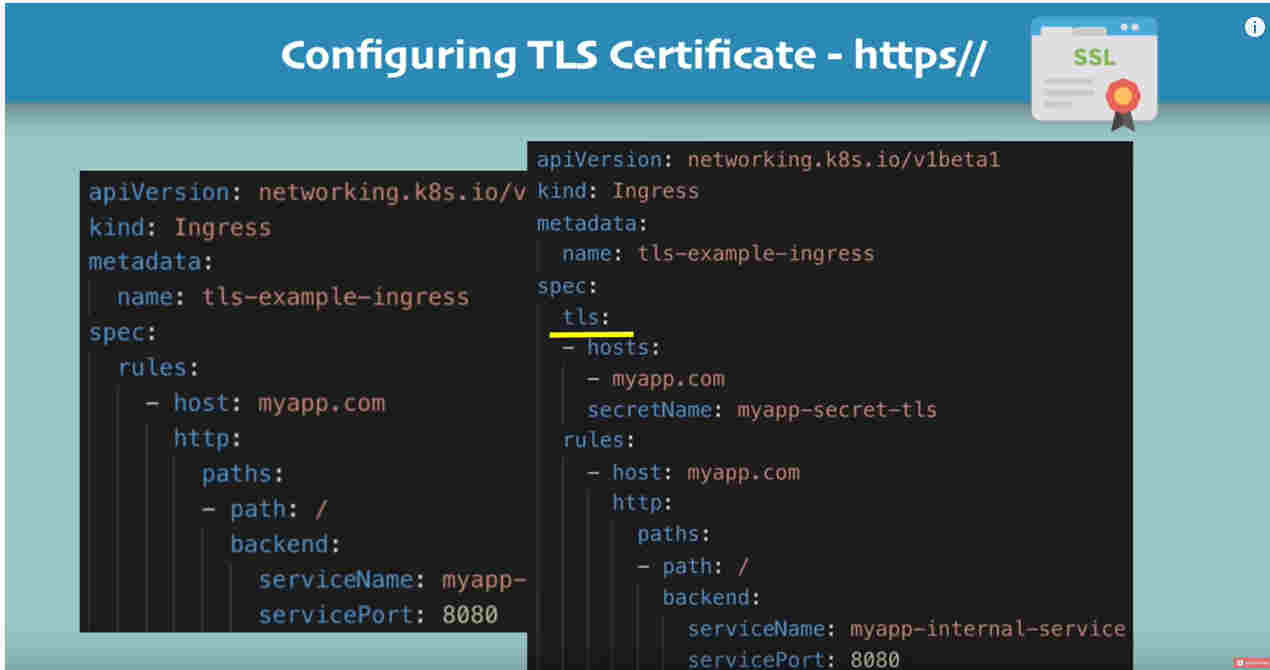
A new service to be configured that will map to default backend.

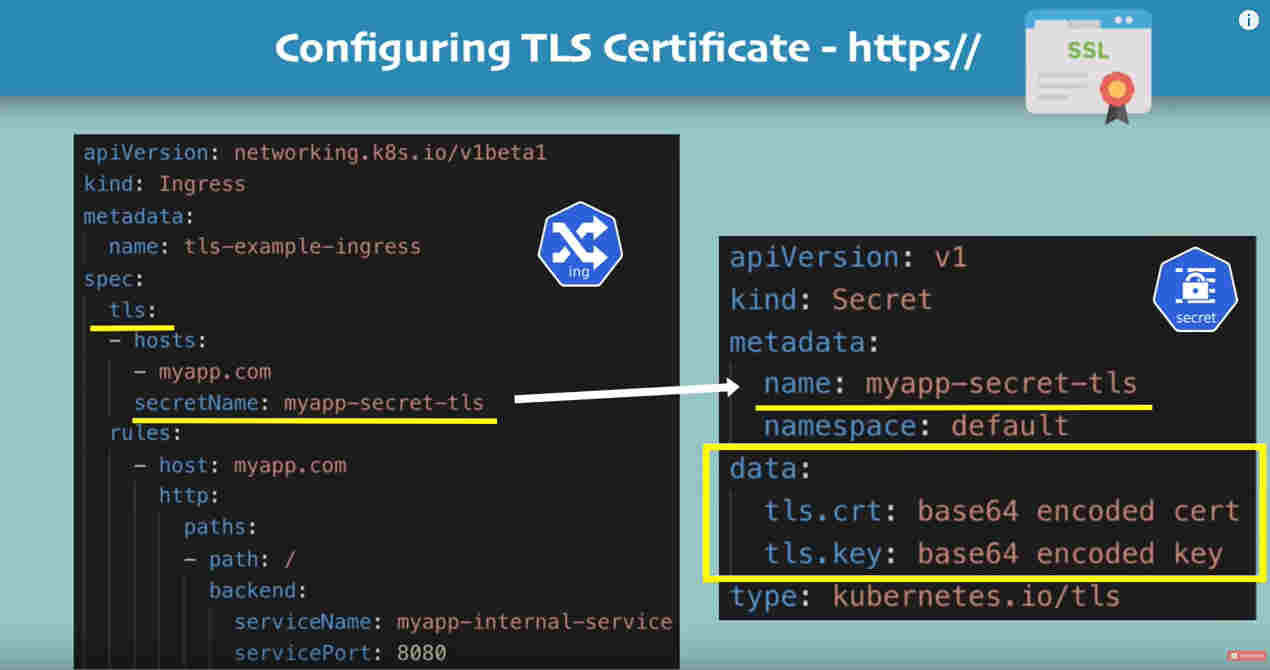


### Configuring TLS

Specify the TLS section(4 lines) just above the rules and specify the secretName which is configured as a secret. The certificate file content and the key should be base64 encoded. **Type** of the **Secret** should be ***kubernetes.io/tls.***

The secret should be in same namespace as the Ingress component.





## Helm

Package manager for K8s.

Package yaml files and distribute in public and private repos.

We can search helm repos using commandline (***helm search <keyword>***) or in **helm Hub**.

Hem has a templating service to ease the config file creation when working with so many files.

### Helm charts

Bundle of yaml files for a specific implementation(k8s config for elastic search – Configmaps,Secrets,Services,Deployments, etc)

sample helm charts – MongoDB, ElasticSearch, mySQL,Prometheus.

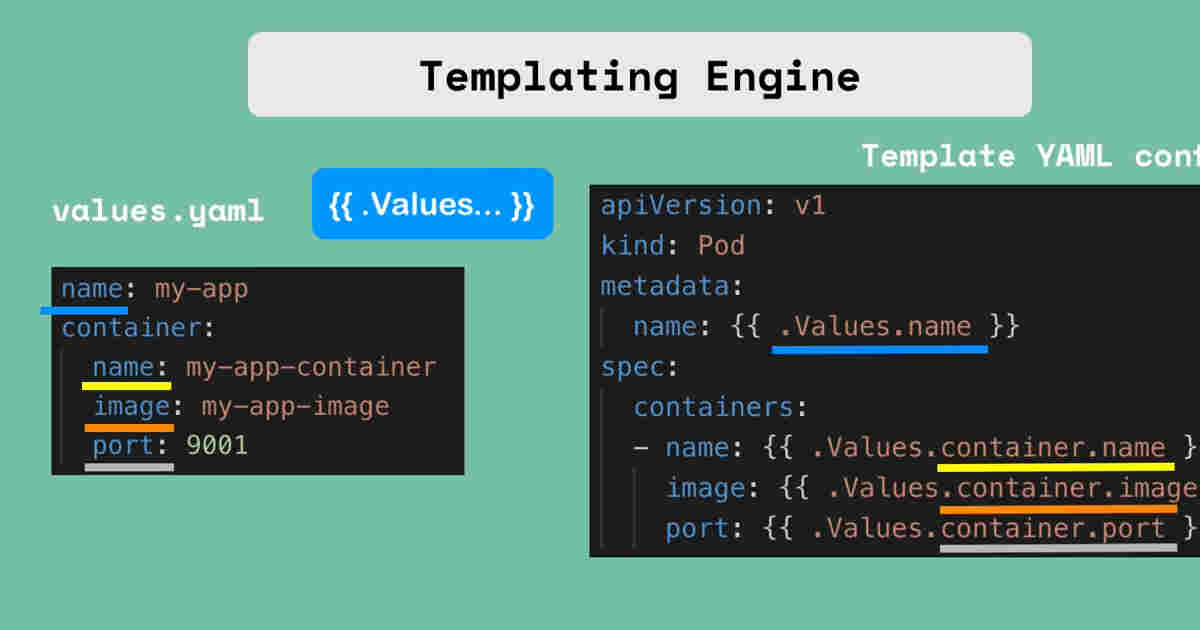
### Helm templating service –

common blueprint yaml for all services if we have many services to be deployed. Dynamic values can be replaced by placeholders. The placeholders values are kept in a values.yaml file.

This means overall one config file and many values files specific to each service(dynamic values will be very less , so the values file will be small in general.)

Templating helps in case of the CI/CD implementation.

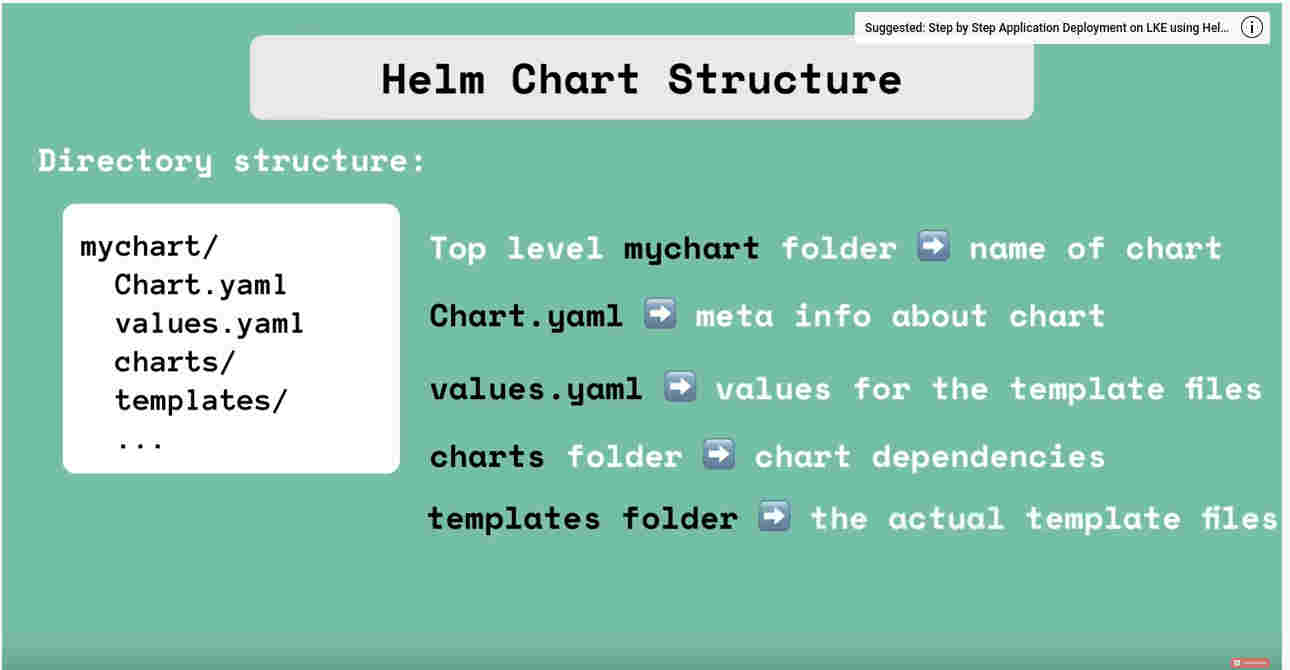
It will help in maintaining one helm Chart for an app that can be deployed to Staging, Uat, prod just by updating the values file.



### Helm Chart Structure

Main folder – name of the chart.

Charts folder – any other dependent charts.

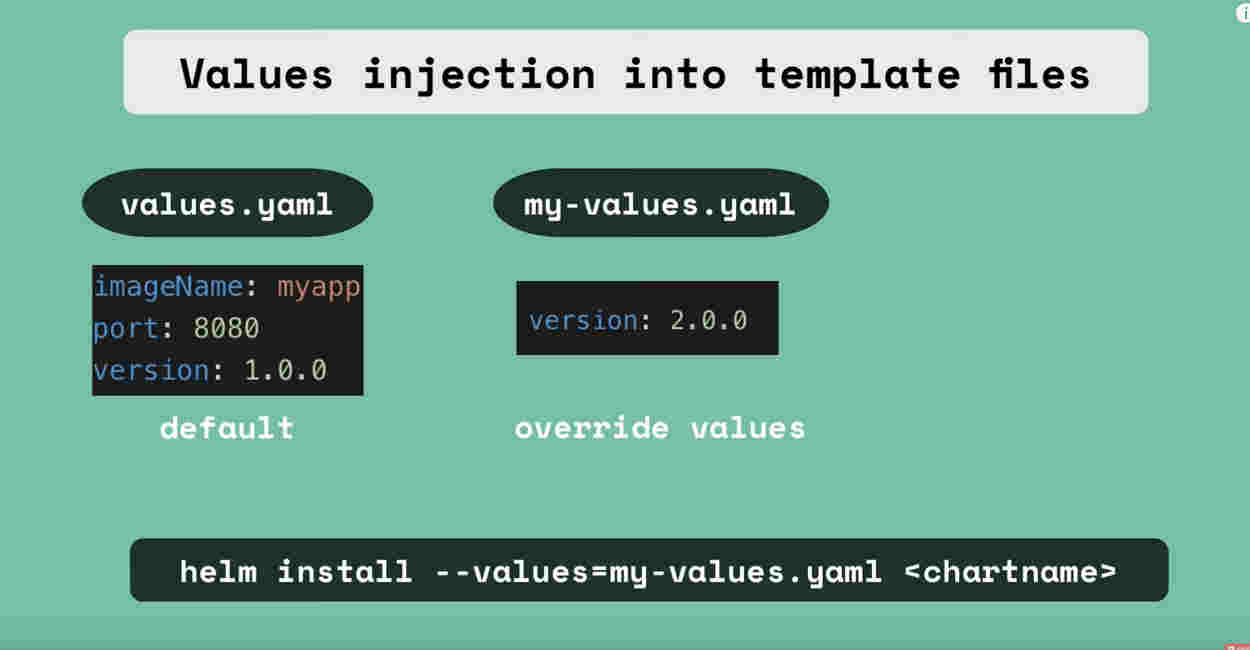


### Installation using the helm chart

Values from the override values file will replace default values from the package file.

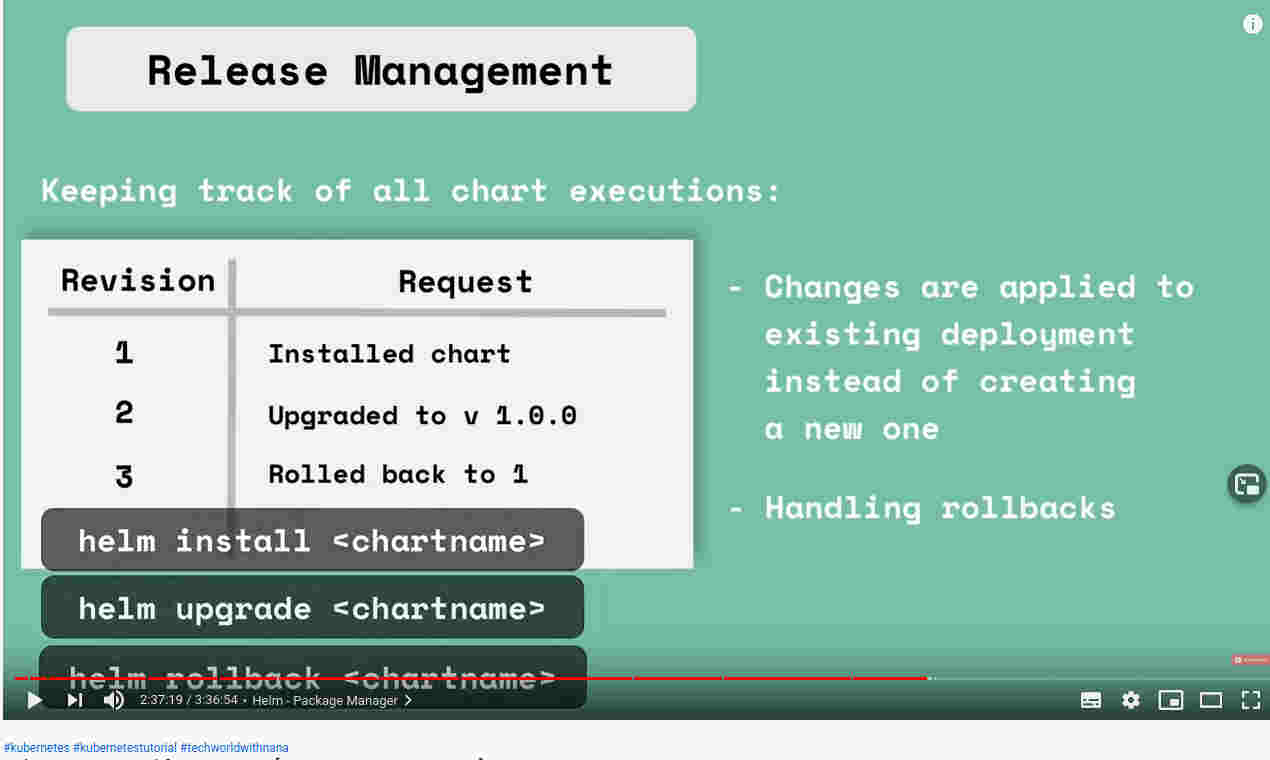
Any new values in override file will be taken as well.

Helm has Helm Client (CLI) and Helm Server(TILLER-Running on K8s Cluster). TILLER is obsolete after Helm V3.



### **Helm** Release management

It maintains a history of all the config updates.



## Persistent Volume Component.

There is no data persistence in K8s out of the box.

The storage should be able to handle the following .

- Should not depend on pod lifecycle.

- should be available on all nodes

- Should survive even if the whole cluster crashes.

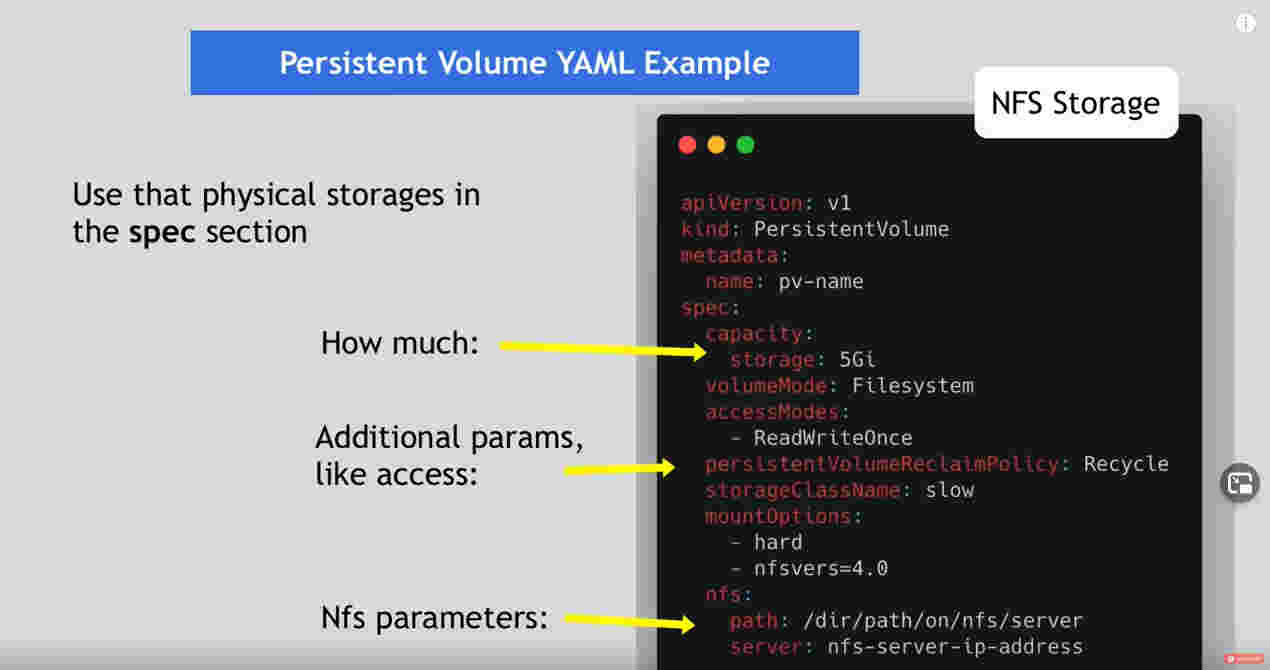
Since should be accessible to whole cluster, Persistent volume is not namespaced.

Created using the config yaml files.

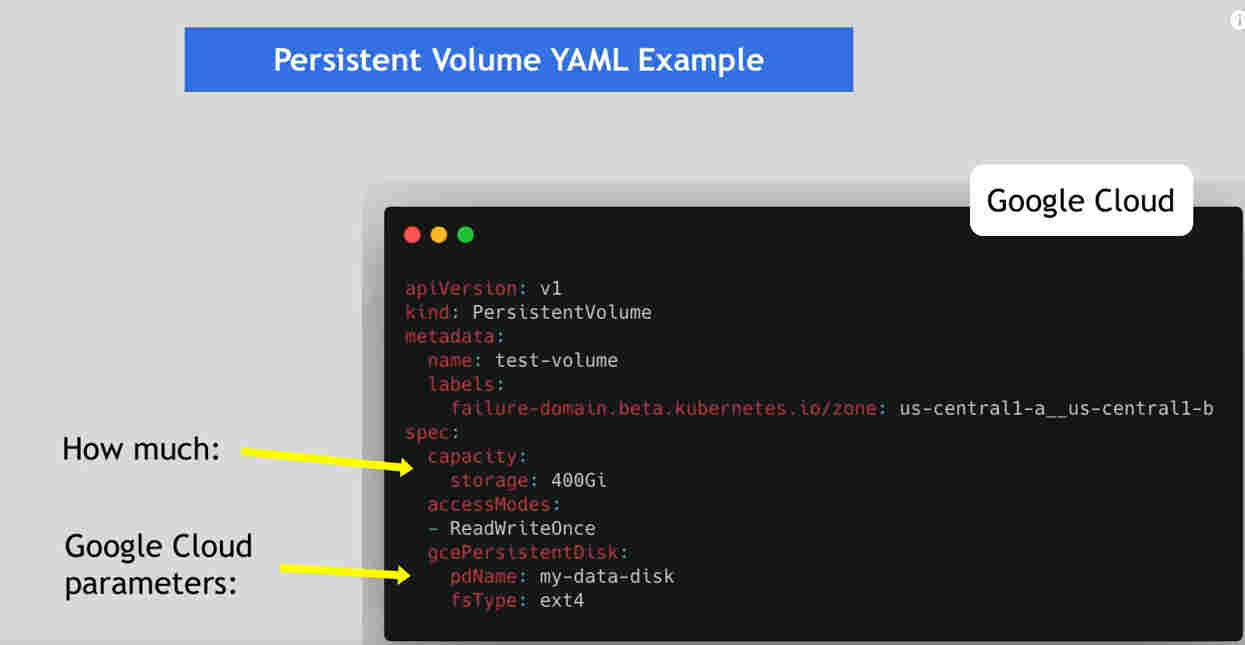
Can be configured using on-the-node disc , cloud, NFS etc.

Persistent Volume is setup by K8s Admin and the Claim Component is configured by the Developer.

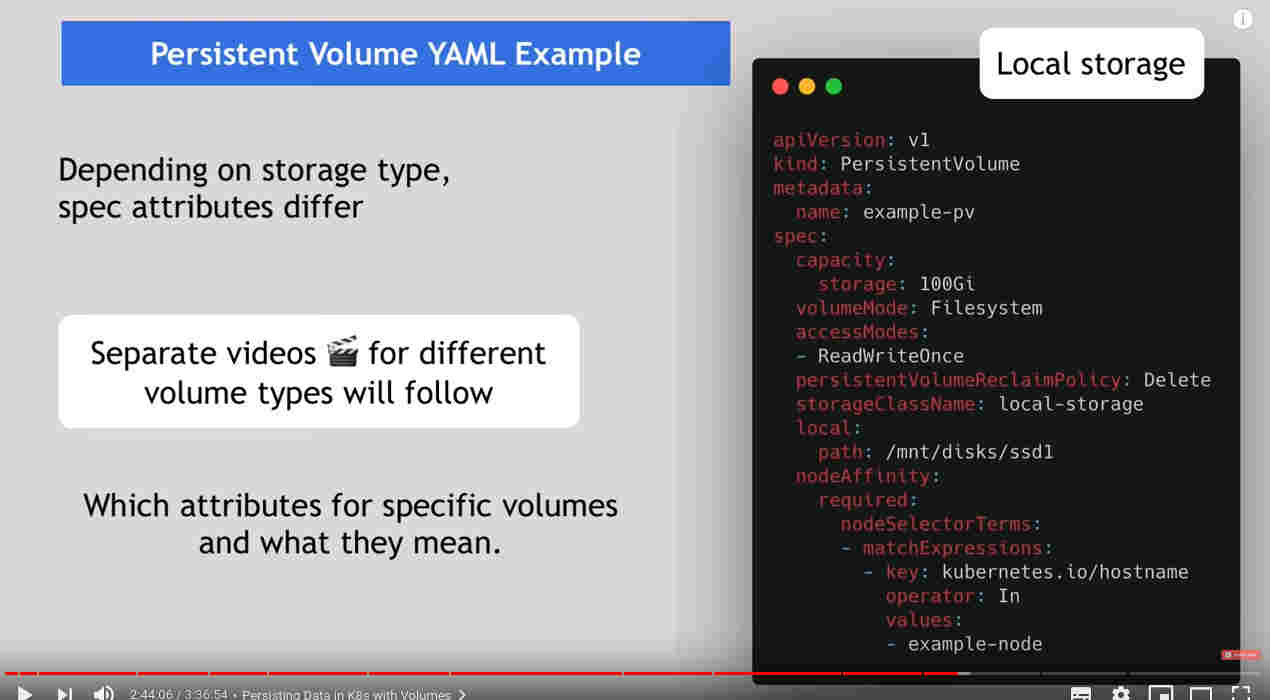
### NFS - Persistent Storage Setup.



### Google Cloud - Persistent Storage Setup.



### Local Storage - Persistent Storage Setup.



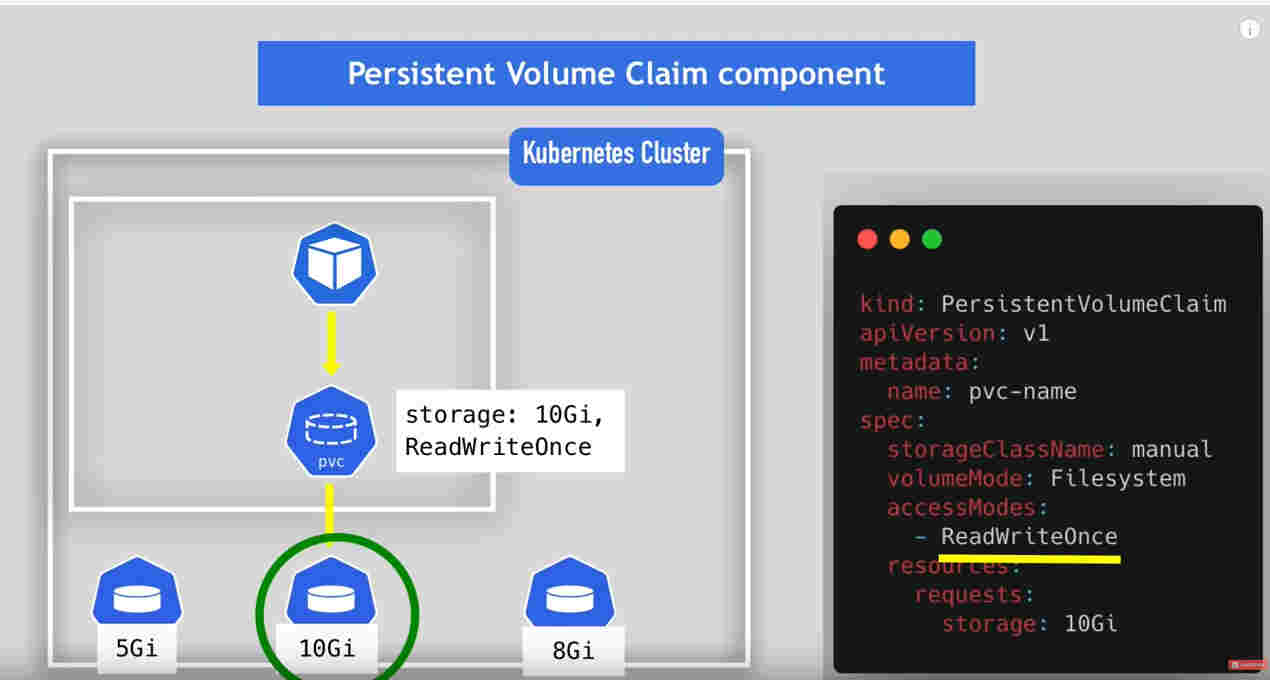
## Persistent Volume Claim Component.

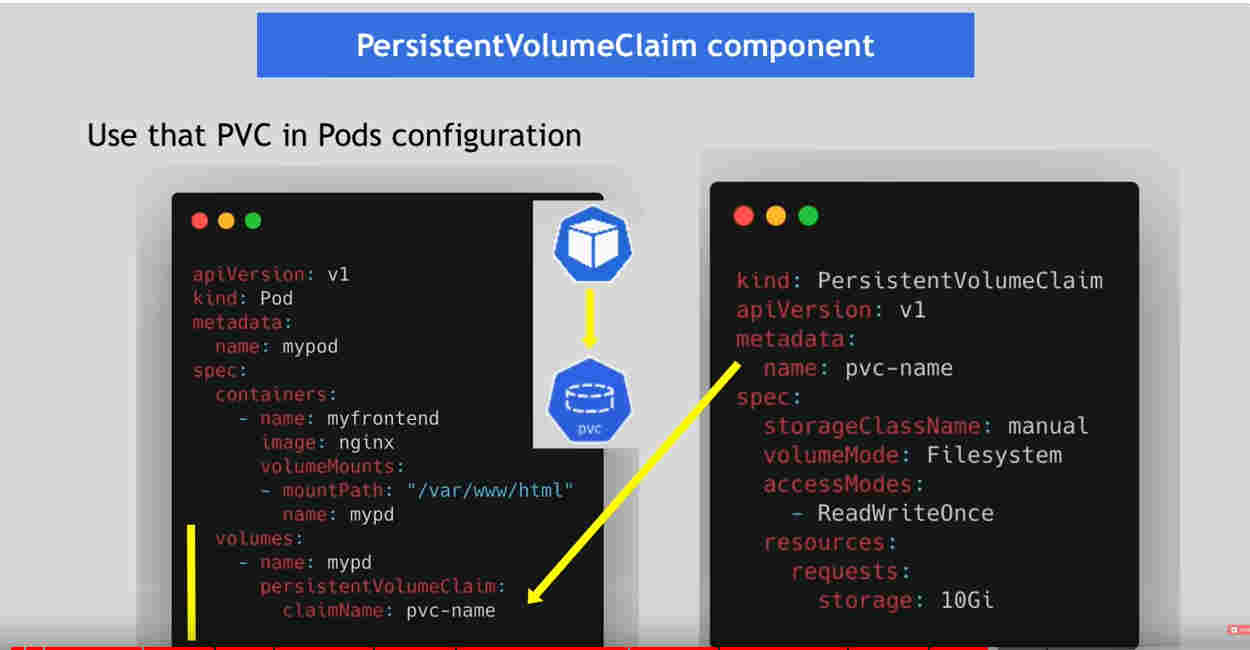
With persistent volume component, we only configure a Persistent volume. To use it , an application that needs it should claim it using the Persistent volume claim component.

This is also done using config file.

The volumes available that match the Claim criteria for the appln.

The application which needs to use a claim component should specify that in the Pod config(Deployment Config file) – Pod and the Clain should be in the same Namespace.

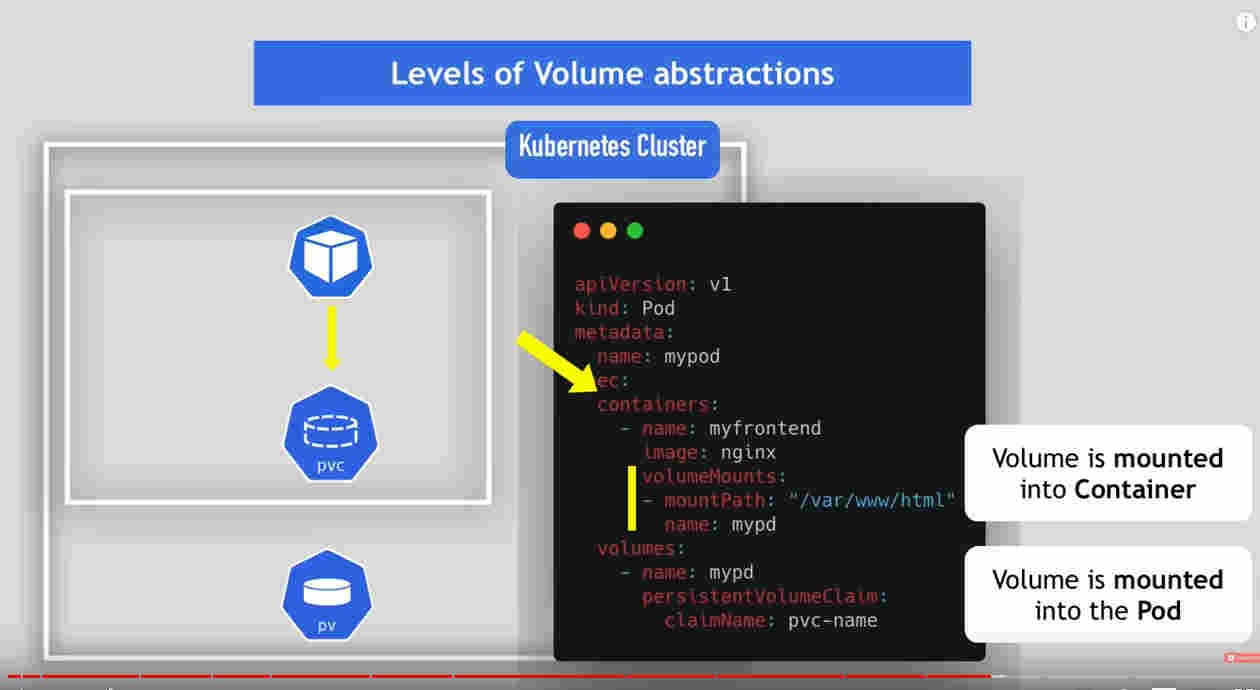




### Persistent Volume abstraction

The Pod claims the volume matching the Claim → Volume is mounted into the pod → The Volume is mounted into the Container.

If there are multiple container in a Pod, **we can mount them onto one or More Containers.**



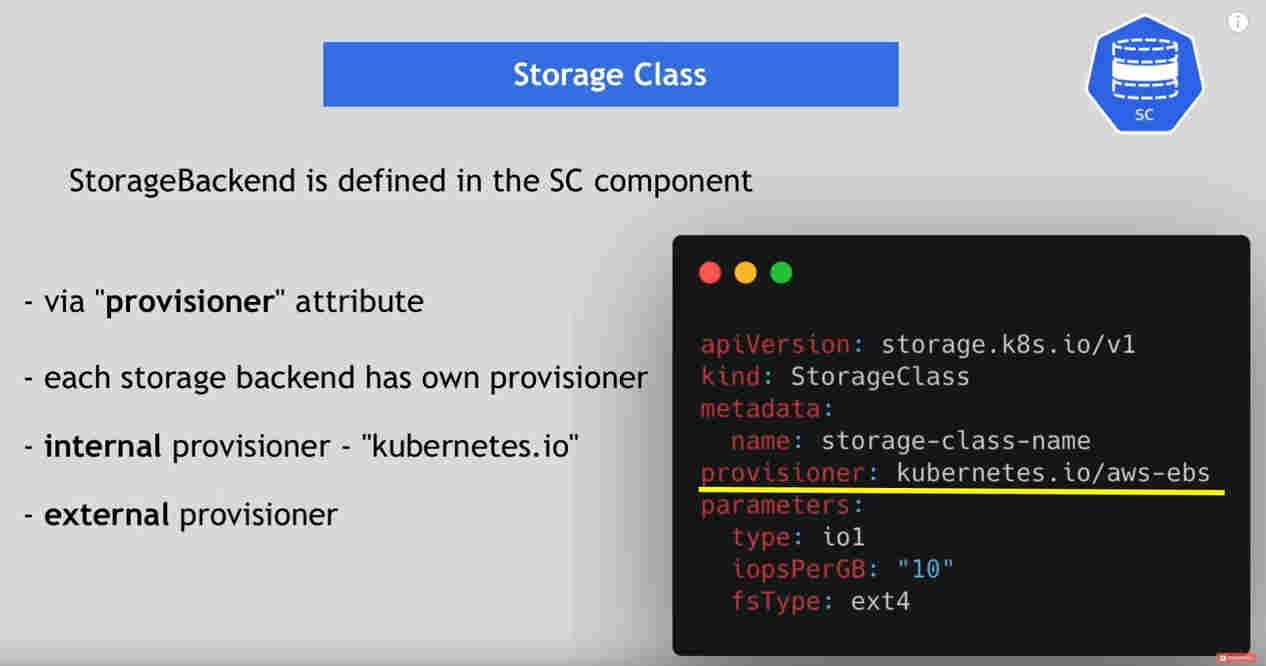
## Storage Class Component.

This is to reduce the redundancy when there are lot of PV and PVC to be implemented.

Storage Class provisions a PV dynamically when a PVC claims a storage.

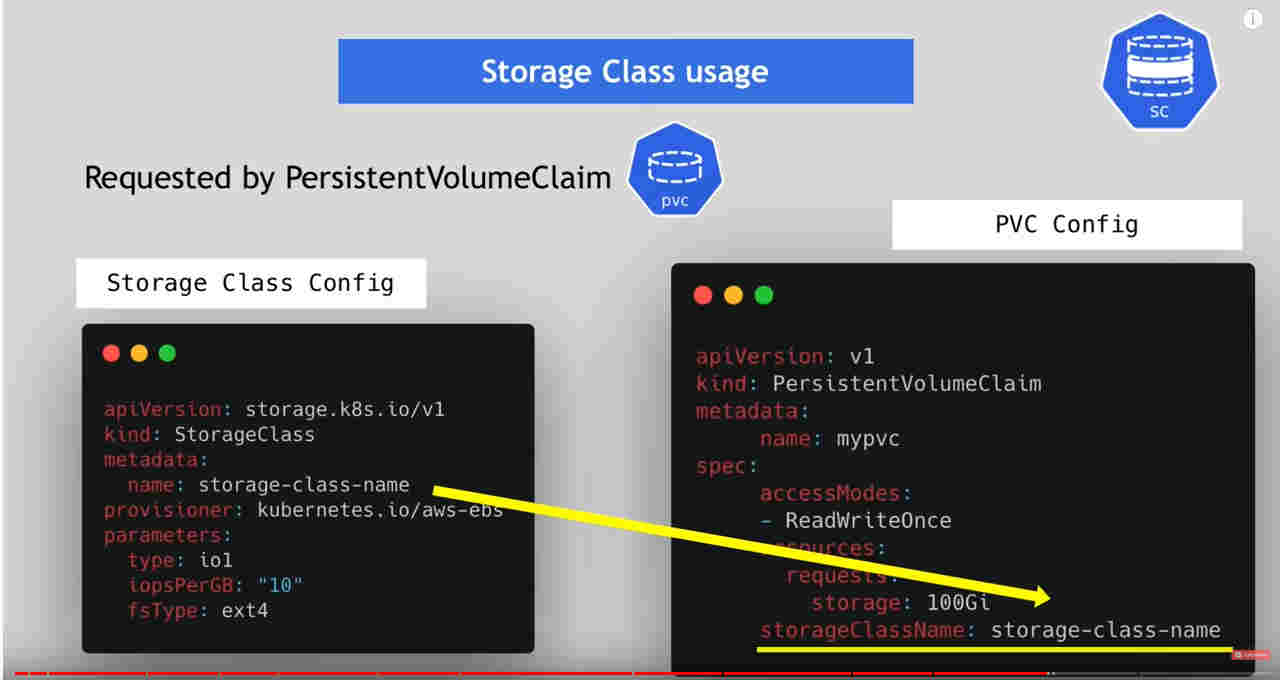
Created using config file.

Actual backend to be used is specified here.



### Using the Storage Class

Storage Class is sued by PVC just like a PV.



## Kubernetes Service – in Detail.

Several types of Services

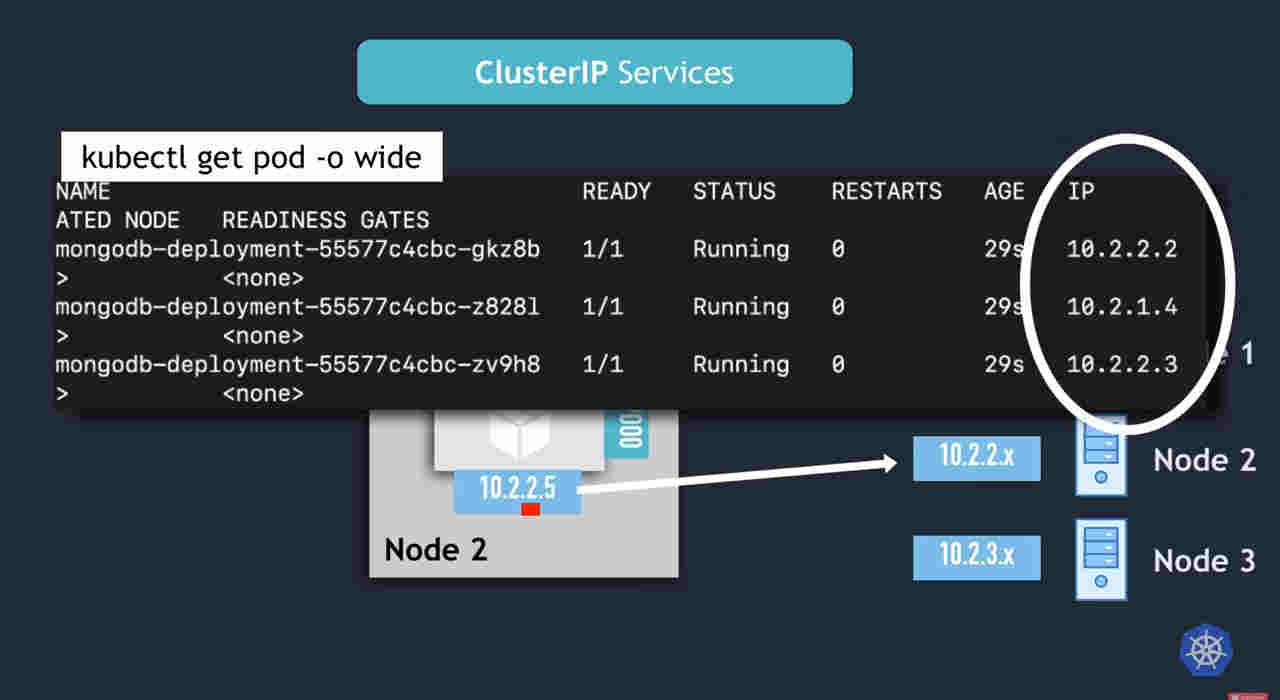
### Cluster IP Service

Default type if no type specified.

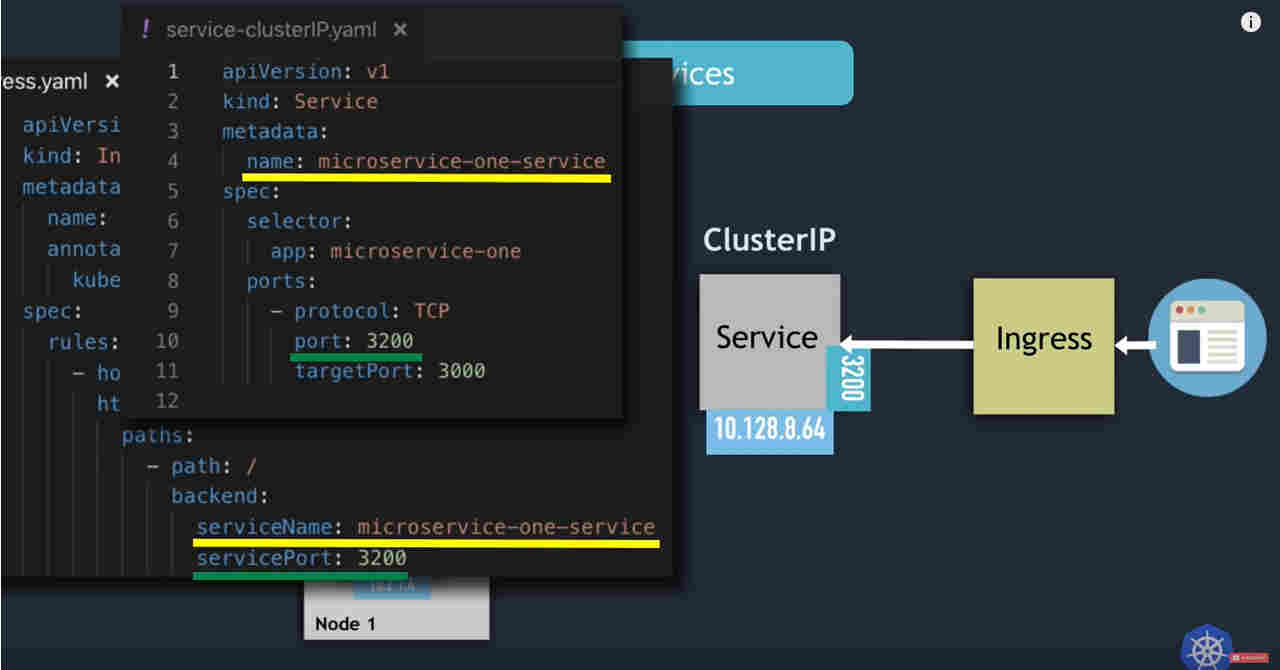
This works as an LB.

Can be accessed only from within the Cluster.

Each node in Cluster gets a range of IP that can be provided to the Pods.

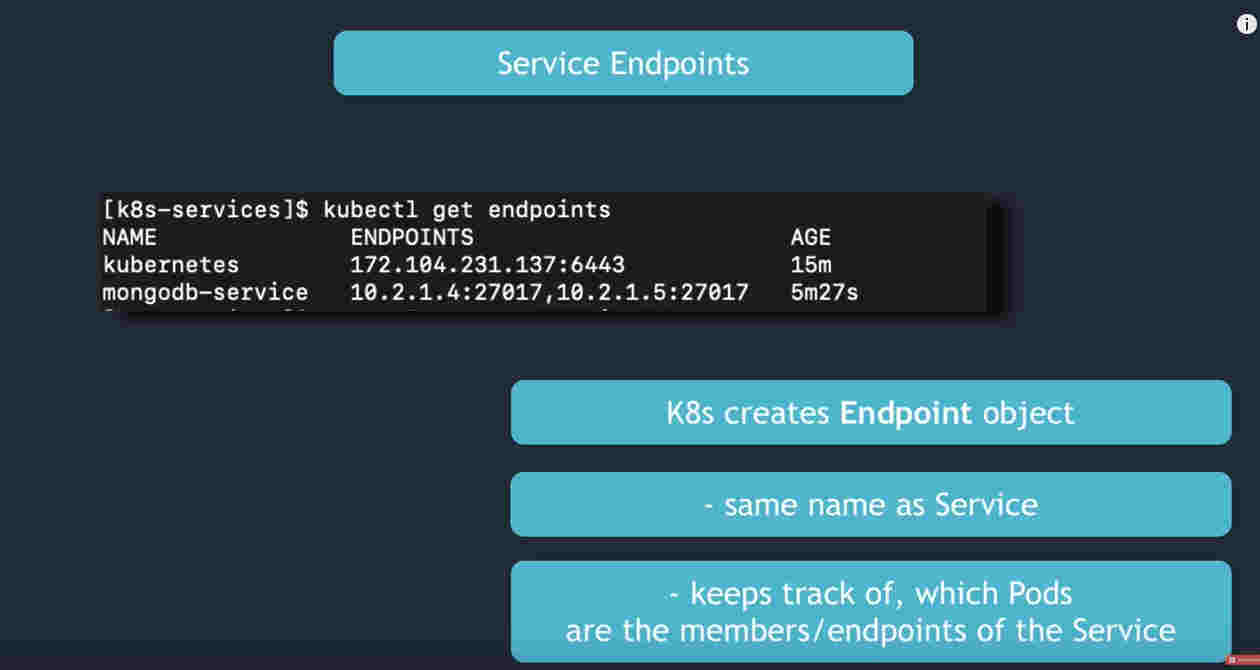


Relationship between Ingress and service and port bindings.



### Endpoint Objects

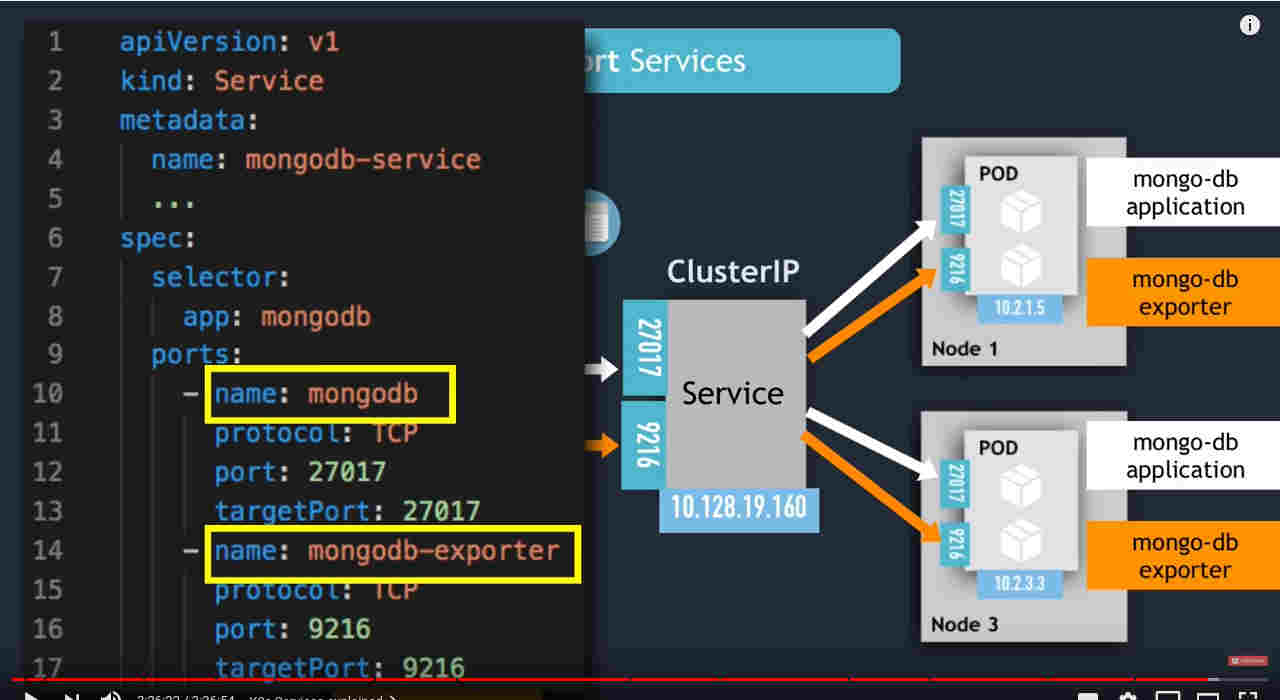
All the Service endpoint details are saved as Endpoint Objects with name as Service name.



### Multi Port Services

Services with one name but can connect to 2 or more ports of the same target application.

Need(as in screenshot) to explicitly name each port binding.

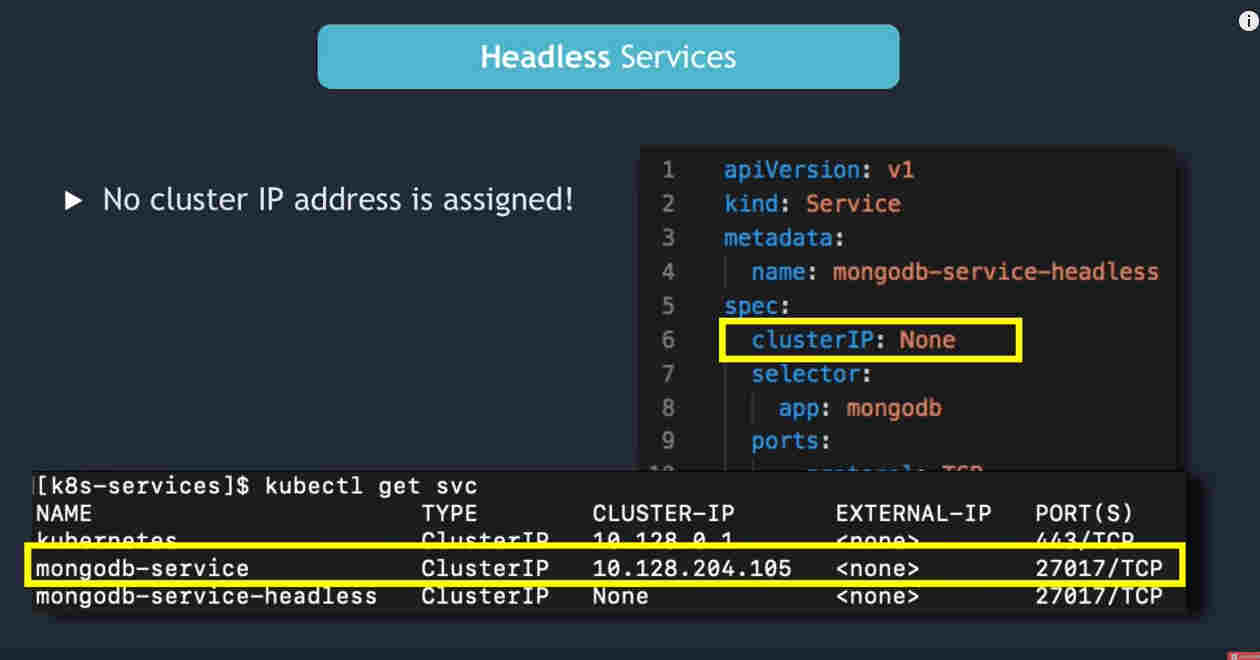


### Headless Service

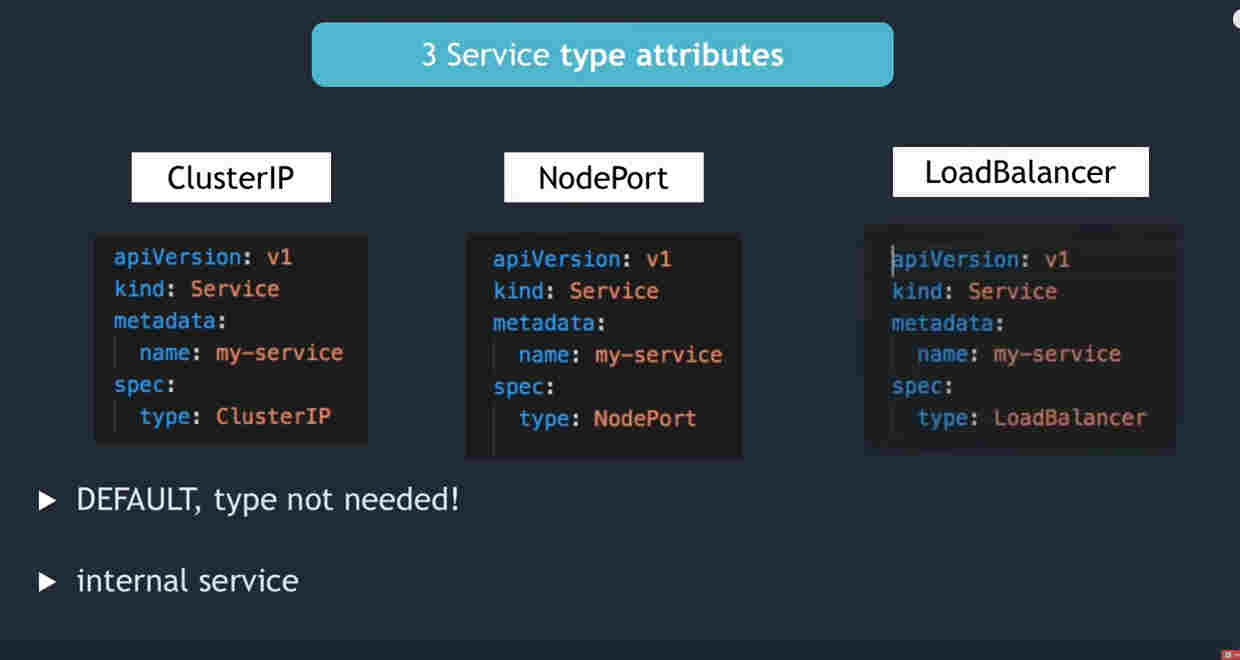
Used when the service has to connect to a specific individual pod.

Normally used in case of Stateful applications where there will be a master and slave instances.

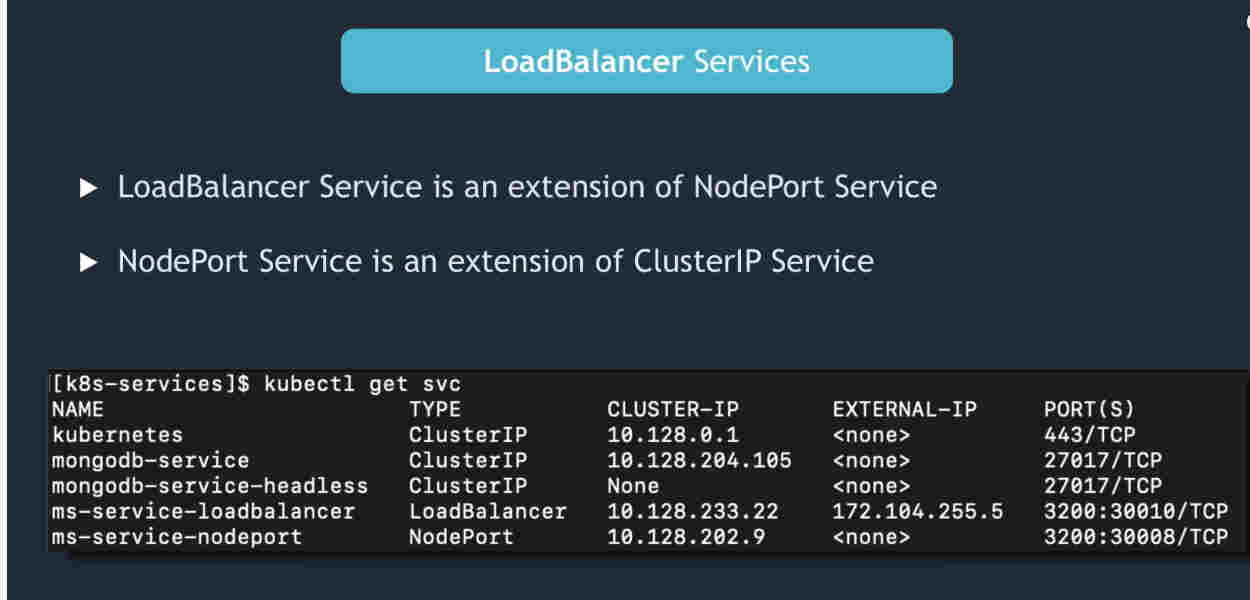
Specify the ClusterIP as None. When the Clients make a DNS lookup for the service, they get all the pod IPs rather than Service IP(cluster IP).



### Service TYPE attribute



Look at the Ips and Ports exposed in last 3 entries below.



### Nodeport Service

This service can be **directly accessed with the ClusterIP** without need of Ingress.

There is a nodeport to be configured.

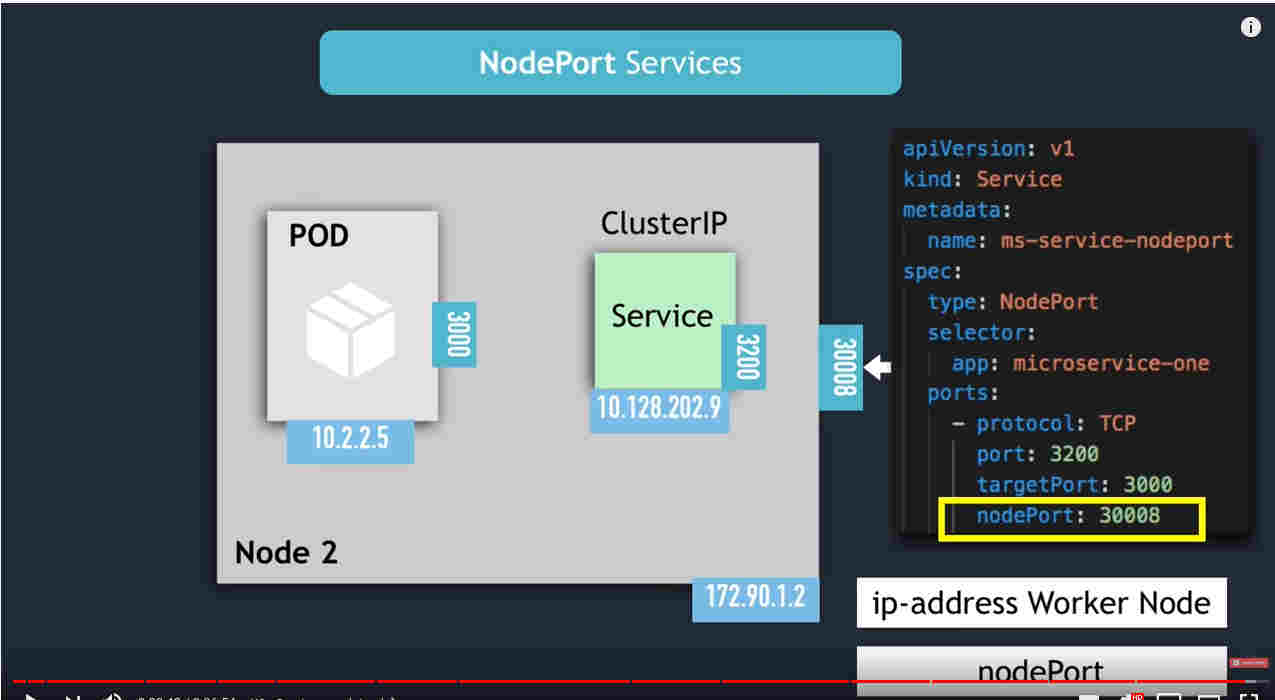
The port exposed by service is in the range of 30000 – 32767.

The **NodePort** configured will be automatically routed to the **Port** configured. ie Nodeport 3008 to port 3200.

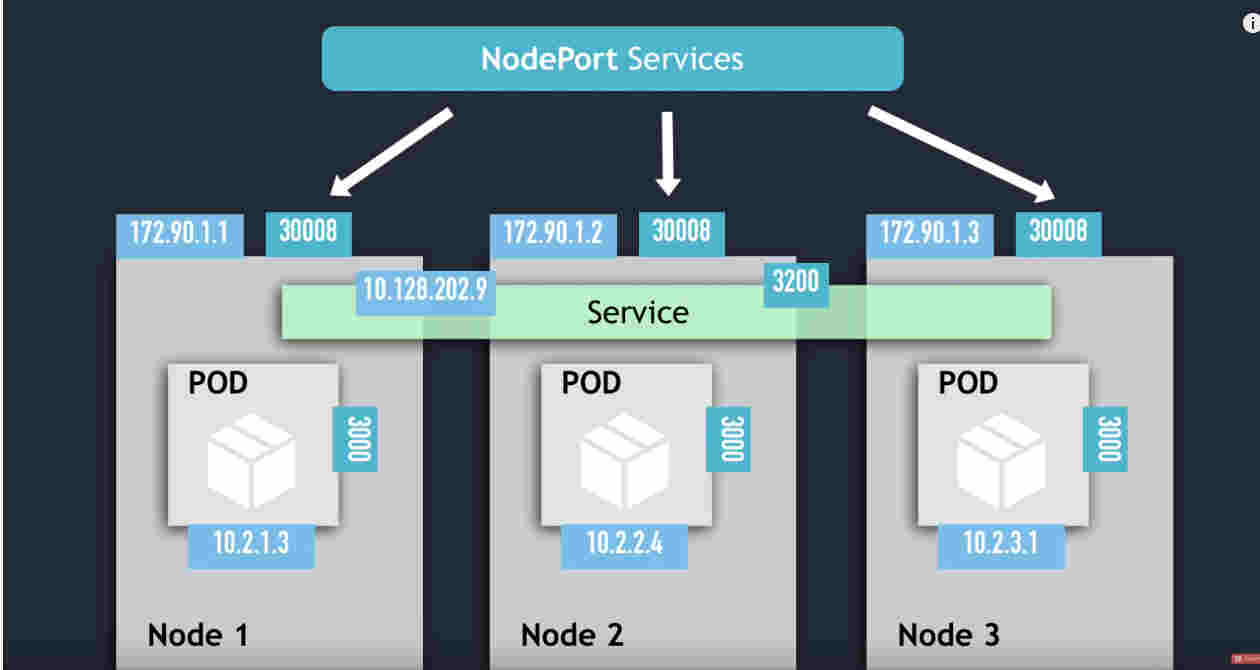
Not very secure since the node port is directly opened to outside.

Any external request will directly come to the Node.

***This is not Normally used in Production.***



Look at the IPS at Pod, node etc



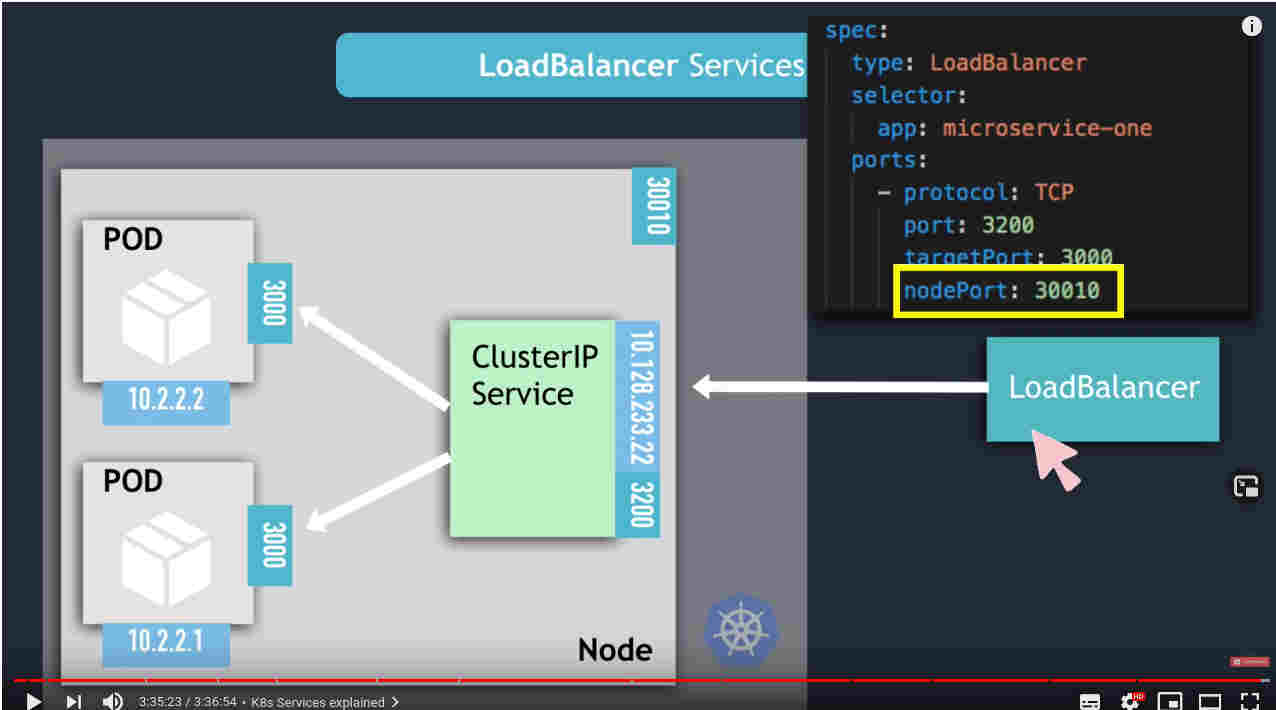
### Load balancer Service

Service is made available outside through the Cloud provider’s LB.

**Type is LoadBalancer**

Nodeport also to be configured.

Any external hit will come the the Cloud LB, then to the ClusterIP service.



## K8s Dashboard access

Documentation - https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

### Dashboard installation

The K8s dashboard can be installed by below config yaml.

*kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0/aio/deploy/recommended.yaml*

The yaml above can be edited as per our needs. I am editing now to make the default ClusterIP to LoadBalance type dashboard-service.

Kubectl -n kubernetes-dashboard edit svc kubernetes-dashboard

[in the editor, press I for insert mode. After making changes , , press esc, then :wq to save the file and exit]

Now, the Service will be running in LoadBalancer mode. Get service will give us the external IP to be used. In Minikube run command – minikube n kubernetes-dashboard service kubernetes-dashboard .

This will open the dashboard UI.

### Service Account Creation

Create a serviceaccount in the **Default** namespace.

kubectl create serviceaccount ***dashboard*** -n default

Now create a role-binding(named ‘**dashboard-admin** ’) for the user created. The Role (‘cluster-admin’) is assigned to the service account dashboard in the default ns.

kubectl create clusterrolebinding dashboard-admin -n default --clusterrole=cluster-admin - -serviceaccount=default:***dashboard***

Get the join token for the above user. Run the below command which will generate a token. Highlighted the user below.

kubectl get secret $(kubectl get serviceaccount ***dashboard*** -o jsonpath="{.secrets[0].name}") -o jsonpath="{.data.token}" | base64 –decode

use the token in the page opened few steps earlier.