What is Kubernetes? (Java T Point)

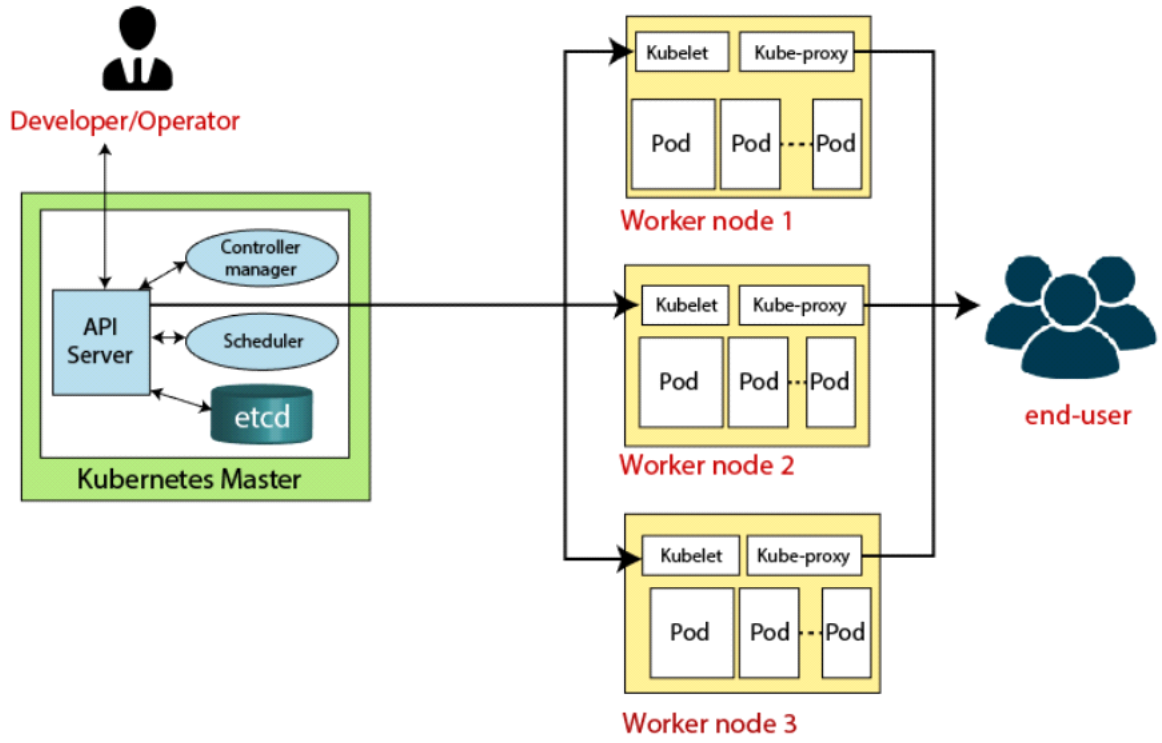
Sunday, April 24, 2022 10:56 PM

- Kubernetes is an extensible, portable, and open-source platform designed by Google in 2014.
- It is mainly used to automate the deployment, scaling, and operations of the container-based applications across the cluster of nodes.
- It is also designed for managing the services of containerized apps.

Key Objects of Kubernetes

Pod: It is the smallest and simplest basic unit of the Kubernetes application. This object indicates the processes which are running in the cluster.
Node: A node is nothing but a single host, which is used to run the virtual or physical machines.
Service: A service in a Kubernetes is a logical set of pods, which works together. With the help of services, users can easily manage load balancing configurations.
ReplicaSet: A ReplicaSet in the Kubernetes is used to identify the particular number of pod replicas are running at a given time.
Namespace: Kubernetes supports various virtual clusters, which are known as namespaces. It is a way of dividing the cluster resources between two or more users.
Pod: It is a deployment unit in Kubernetes with a single Internet protocol address.
Automatic Bin Packing: Kubernetes helps the user to declare the maximum and minimum resources of computers for their containers.
Service Discovery and load balancing: Kubernetes assigns the IP addresses and a Name of DNS for a set of containers, and also balances the load across them.
Automated rollouts and rollbacks: Using the rollouts, Kubernetes distributes the changes and updates to an application or its configuration. If any problem occurs in the system, then this technique rollbacks those changes for you immediately.
Persistent Storage: Kubernetes provides an essential feature called 'persistent storage' for storing the data, which cannot be lost after the pod is killed or rescheduled.
Self-Healing: This feature plays an important role in the concept of Kubernetes. Those containers which are failed during the execution process, Kubernetes restarts them automatically. And, those containers which do not reply to the user-defined health check, it stops them from working automatically.

Kubernetes Architecture



Master Node or Kubernetes Control Plane:

- The master node in a Kubernetes architecture is used to manage the states of a cluster.
- It is actually an entry point for all types of administrative tasks. In the Kubernetes cluster, more than one master node is present for checking the fault tolerance.

Four different components which exist in the Master node or Kubernetes Control plane:

- API Server
  - The Kubernetes API server receives the REST commands which are sent by the user. After receiving, it validates the REST requests, process, and then executes them. After the execution of REST commands, the resulting state of a cluster is saved in 'etcd' as a distributed key-value store.
- Scheduler
  - The scheduler in a master node schedules the tasks to the worker nodes. And, for every worker node, it is used to store the resource usage information.
    In other words, it is a process that is responsible for assigning pods to the available worker nodes.
- Controller Manager
  - The Controller manager is also known as a controller. It is a daemon that executes in the non-terminating control loops. The controllers in a master node perform a task and manage the state of the cluster. In the Kubernetes, the controller manager executes the various types of controllers for handling the nodes, endpoints, etc.
- ETCD
  - It is an open-source, simple, distributed key-value storage which is used to store the cluster data. It is a part of a master node which is written in a GO programming language.

Worker/Slave node

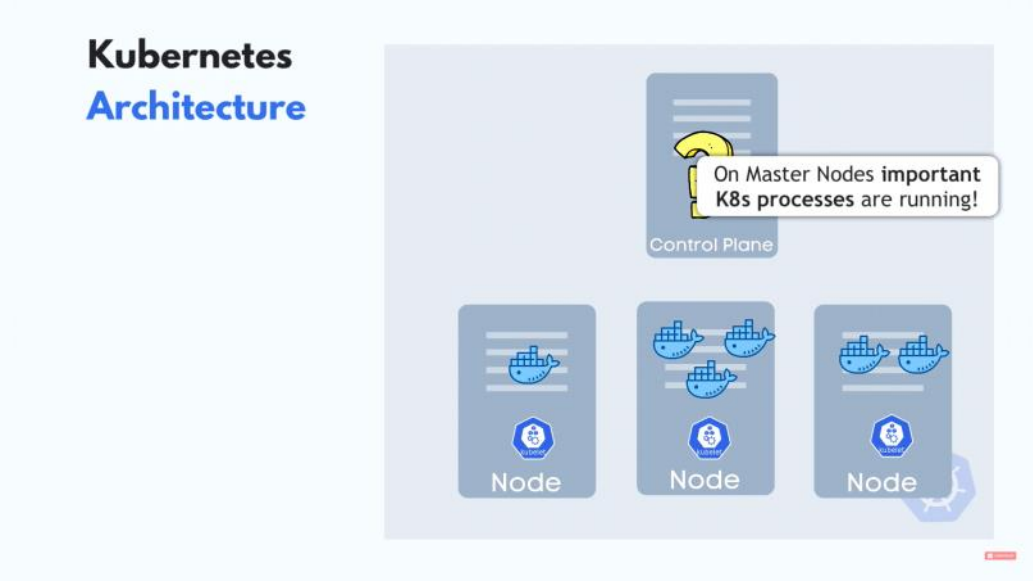
- The Worker node in a Kubernetes is also known as minions. A worker node is a physical machine that executes the applications using pods. It contains all the essential services which allow a user to assign the resources to the scheduled containers.
- **Kubelet**
  - This component is an agent service that executes on each worker node in a cluster. It ensures that the pods and their containers are running smoothly. Every kubelet in each worker node communicates with the master node. It also starts, stops, and maintains the containers which are organized into pods directly by the master node.

- **Kube-proxy**
  - It is a proxy service of Kubernetes, which is executed simply on each worker node in the cluster. The main aim of this component is request forwarding. Each node interacts with the Kubernetes services through Kube-proxy.
- **Pods**
  - A pod is a combination of one or more containers which logically execute together on nodes. One worker node can easily execute multiple pods.

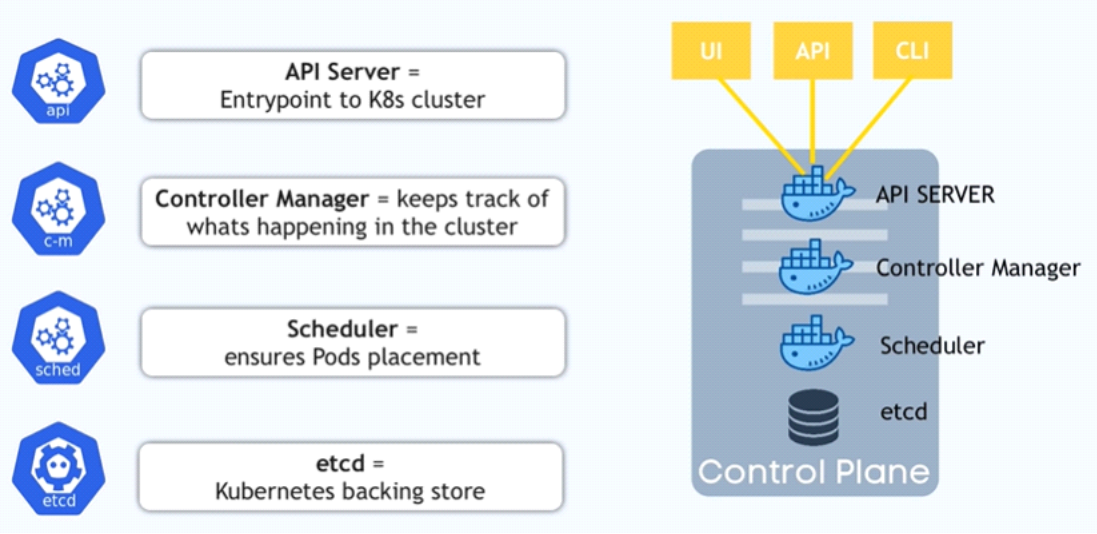
Kubernetes (Nana Video)

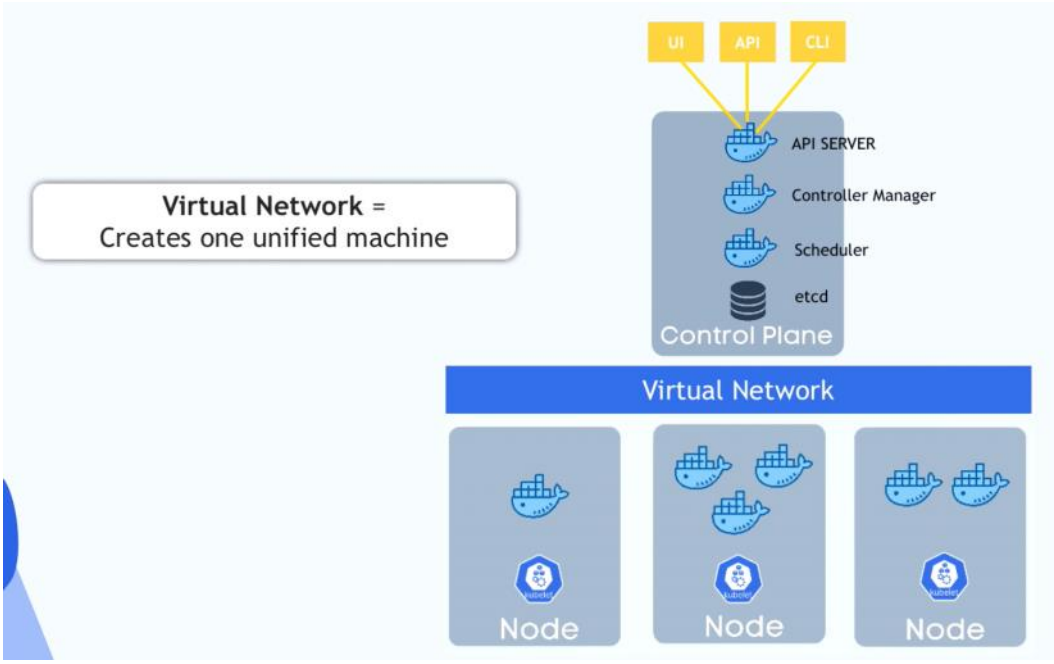
Monday, April 25, 2022 1:58 PM

- Kubernetes is a open source container orchestration tool, which is developed by google and it helps manage containerized applications may be it containerize by the docker or any other tool. And it can work in any different deployment environments.
- **What problems do Kubernetes solve?**
    - In general at the stage of monolithic services there is no concept called containers, and when coming to trending technology called microservices. Where in microservices we may have different small and individual functionality which requires different environment. So instead of establishing independent environment for all the functions. We are establishing a container where it provides environment for the functionality, and containers are not resource hungry. And as it is not resource hungry, many containers can work in parallel and there is a demand for a proper way of managing those hundreds of containers. And as there is increased in usage of containers and trend moves from monolith to microservices, the concept of Kubernetes emerged.
- **What features do orchestration tools offer?**
    - High availability or no downtime
    - Scalability or high performance
    - Disaster recovery - backup and restore
- Kubelet establishes the communication between nodes and perform some tasks like running application processes.
- On worker nodes there are different number of containers on each and kubelet is present on all worker nodes , worker nodes are actually run our containers.
- On master nodes we run several Kubernetes processes that are absolutely necessary to run the application.



- One of the important process that runs on master node is API Server , API Server is the entry point to the Kubernetes cluster
- All the UI , External API and CLI interact with containers using master node's API server.
- Another process running on the Master node is control manager. This keep a track of what's happening in the cluster, if any slave node issue must be resolved or to assign any container to any slave node etc..
- Another process running on the master node is scheduler , it ensures where the new PODS to be placed according to the resources available.
- ETCD is the another process, it contains all the information regarding status of the container or node , contains configuration data

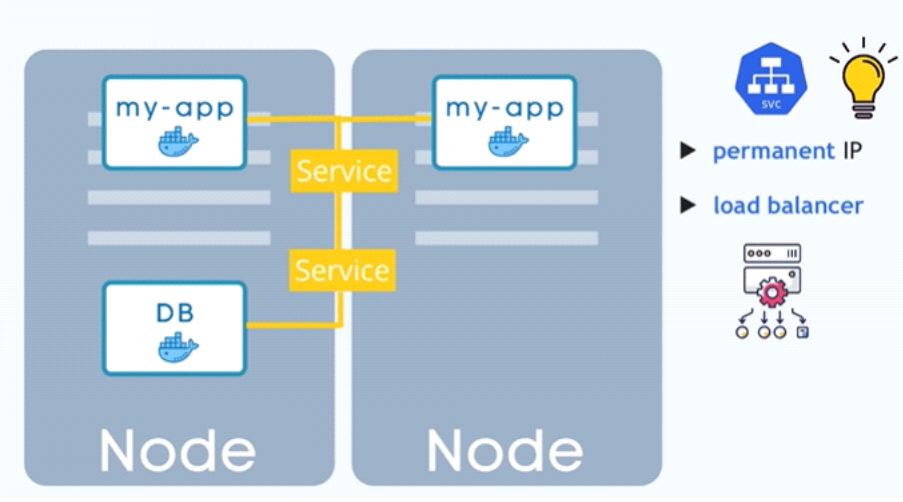




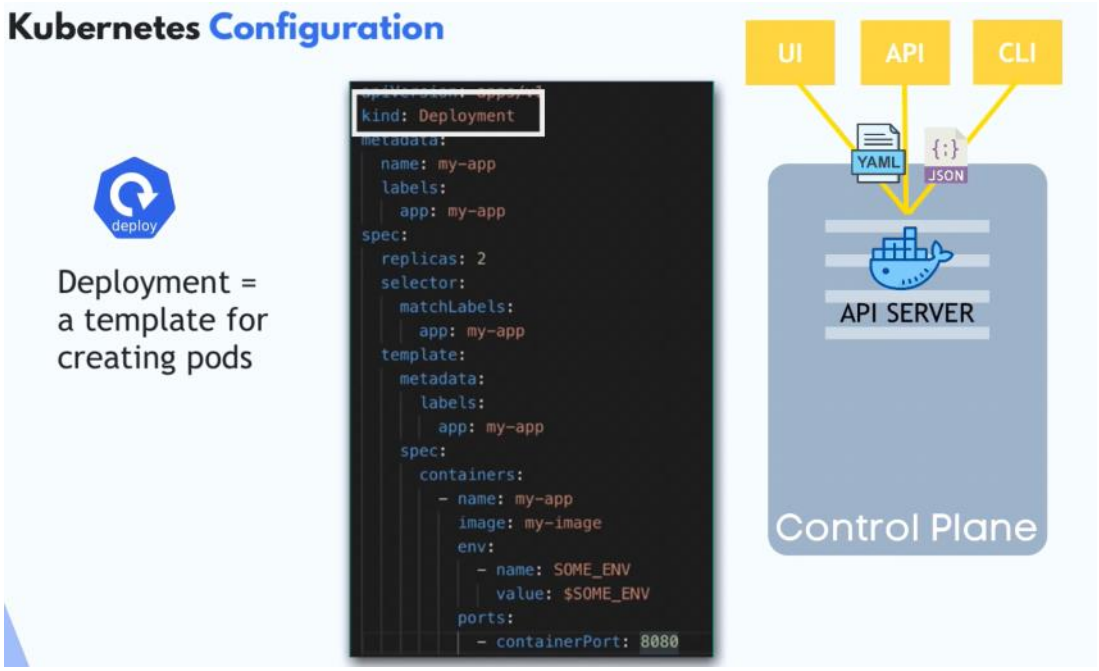
- Here virtual network is used to merge all the nodes into one unified machine , to club all the resources and provide output to the Master node.
- Difference between worker and master nodes
- **Master nodes:**
  - Also called as control plane nodes
  - It only perform the master processes
  - These contain very less resources comparitively to worker nodes
  - But these are very important , if all the master nodes are dead then we could not communicate with the cluster. So we must have a backup master node , if one node fails we can use the other and perform communication with the slave nodes.
- **Worker nodes**
  - Slave nodes
  - These run the main application , hence it takes higher workload
  - Consume more resources.
- Node is a virtual or physical machine and pod is a smallest unit in Kubernetes , pod is abstraction over container. We only interact with Kubernetes layer we could not manipulate docker container directly. Usually we run 1 application per pod
- Each node contains multiple pods and each pod has it's own IP address.
- As pods are ephemeral , if the resources are out then a new pod will be allocated with a new IP address, and the new POD is same as the old but IP address is different , so the changing of IP address is not so convenient , because we need to change repeatedly whenever the POD restarts. SO a new concept called SERVICE is introduced. Instead of IP a Service is allocated for

each pod. Service is a permanent IP address, Lifecycle of service and ip of pod are different . And hence, if pod dissolves and restarts the , IP changes but not the service.

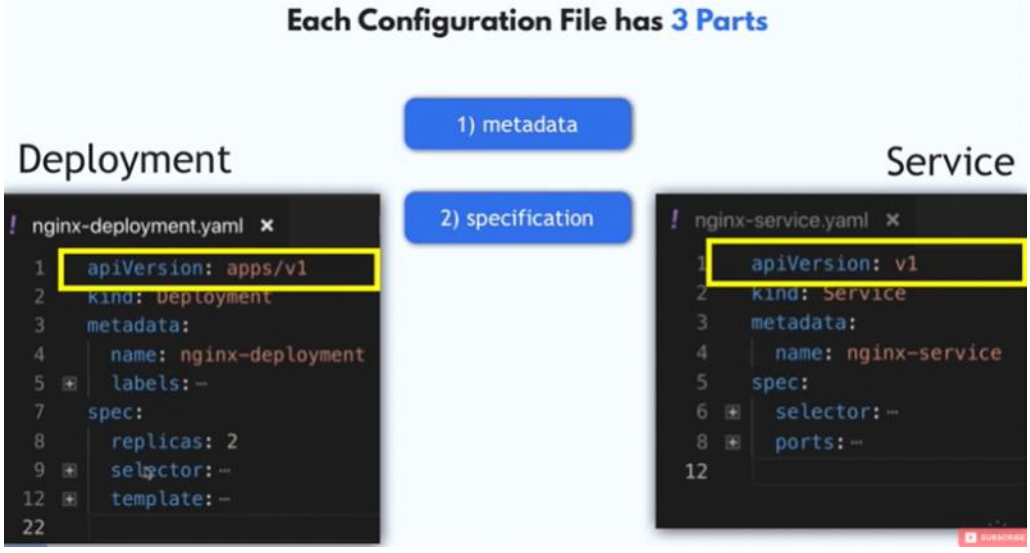
- Suppose for any pod if we want to access is through a web browser then external service is used for that container and if we need to hide it from the browser and use it in internally we use internal service. We can specify the type of service on creation.
- Internal service is the default type.
- URL service of external service is not practical because it will be IP and port number and as we prefer domain name system as the usage we have a concept of ingress assigned to the pod similar to service and the IP address. Ingress may be assigned or not assigned for every page . Generally it will be arranged only for the front page.
- Pods communicate with each other using service.
- Usually when we are building the application , and keeping all the necessary links in the main script files which are part of the image then , if we need to change a small variation in the script we need to rebuild the image . So Kubernetes offers a service called configmap which contain details about the URL's and those URL's are used internally by the image. Here image and configmap are two separate things , config map is external to the image. In simple words , configmap is external configuration of your application.
- Similar to configmap we have another component called secret, which is used to store all the usernames and passwords or any confidential information , they will be in encrypted format . This is used to store the secret data.
- Both configmap and secret are connected to pods via links. Pods get the links of configmap and secret . So pod can access information which is inside configmap and secret.
- If database pod get's restarted then all the information inside the pod get's disappeared. So to persist the data of the pod for a long time we use volume component. That volume can be attached to the local machine or a remote server like cloud database. So if current pod dissolves then the restarted pod will be retrieve data from local or remote server.
- If application pod of the node dies then the application will be down , so to prevent that we keep the application container on multiple nodes and as we know they have the same service id , it mean if one application pod crashes then the other prevent the application crash and run the site. This is the same concept that we use in load balancing.



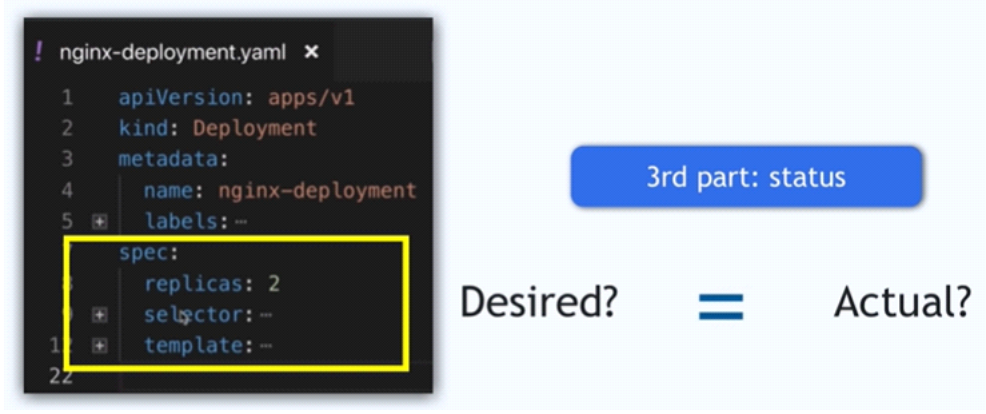
- Here while we create the application pod we can tell number of replica's we need to create using blueprints. This is majorly done in deployment phase where the number of replicas can be increased or decreased based on uprise or downtime.
- We can have the same issue while we use the database pods.
- Stateful set component is used for replication of the database and the main motive is to avoid inconsistencies between the database pods across the replica's
- So in this way even if my app application and database are down we can prevent downtime using this replica creations of application and database , so we will always have nodes to run the application and server and prevent downtime.
- Deployment (for application) and StatefulSet (for database) are used for replication purpose.



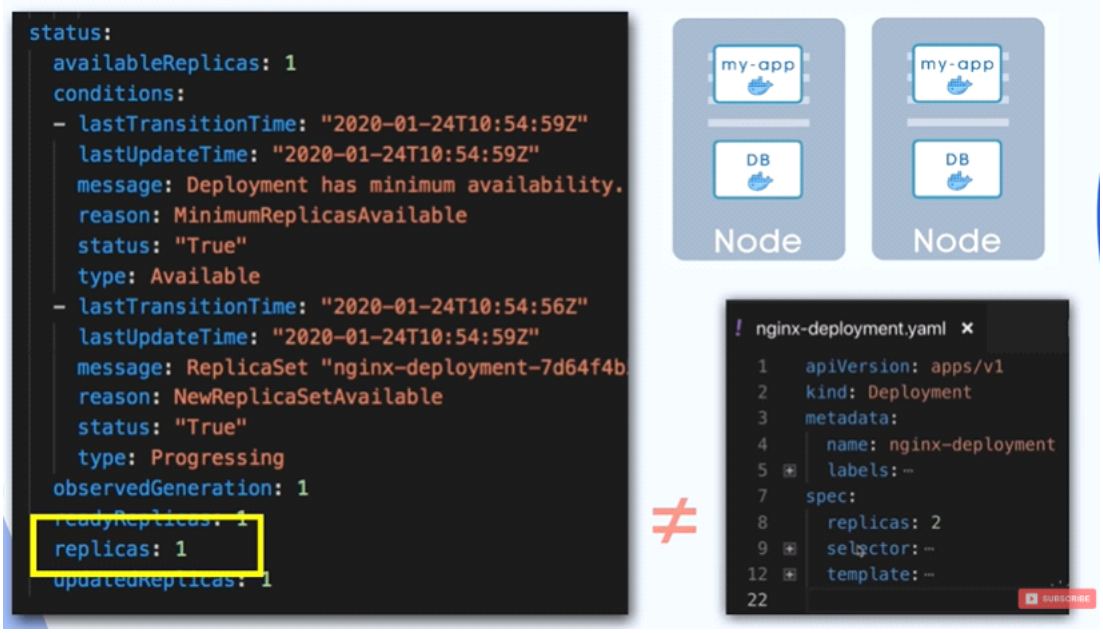
API server receive the yaml and json configuration files through UI or API or CLI, the type of template , name of the pod , number of replicas and for which we are creating replica and for which image it is going to get connected and some environmental variables are being declared on the Configuration file which is passed to the API Server. Deployment kind is used to create a template for creating pods.



- The specification section have it's own arguments according to the kind of the configuration file.



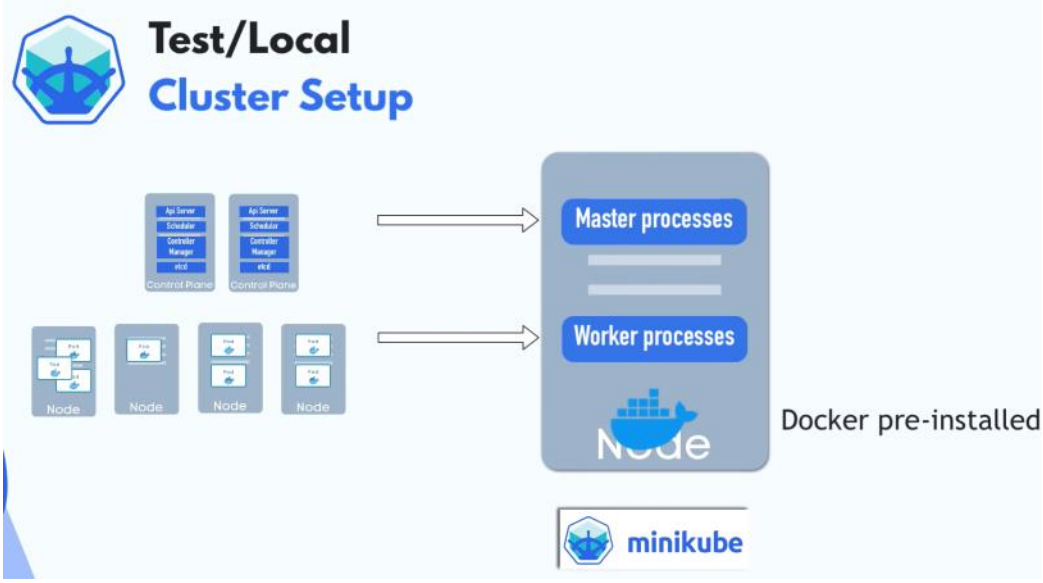
- The third part of the configuration file is status , where we declare the status in specification and if it is not same in the Kubernetes side then kubernetes try to match the status automatically and run the configuration.



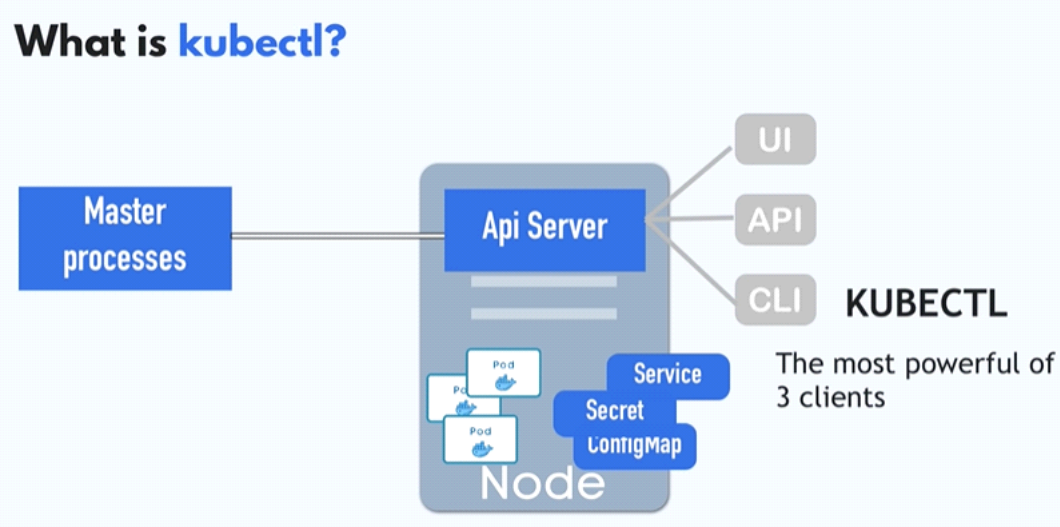
- The left side image is the status report from the side of kubernates and if the status of same parameter is not matched with what user is expected then kubernates automatically try to resolve it. Now at the kubernates side , we are having replicas as 1 in the kubernates side and at the user side replicas is 2. so kubernates create the other replica by itself and match the user to the kubernates side.
- This status report is generated with the help of etcd which is present in the master node.
- Yaml configuration files are human friendly it follows strict indentation. We can use code editor parsers for yaml files.

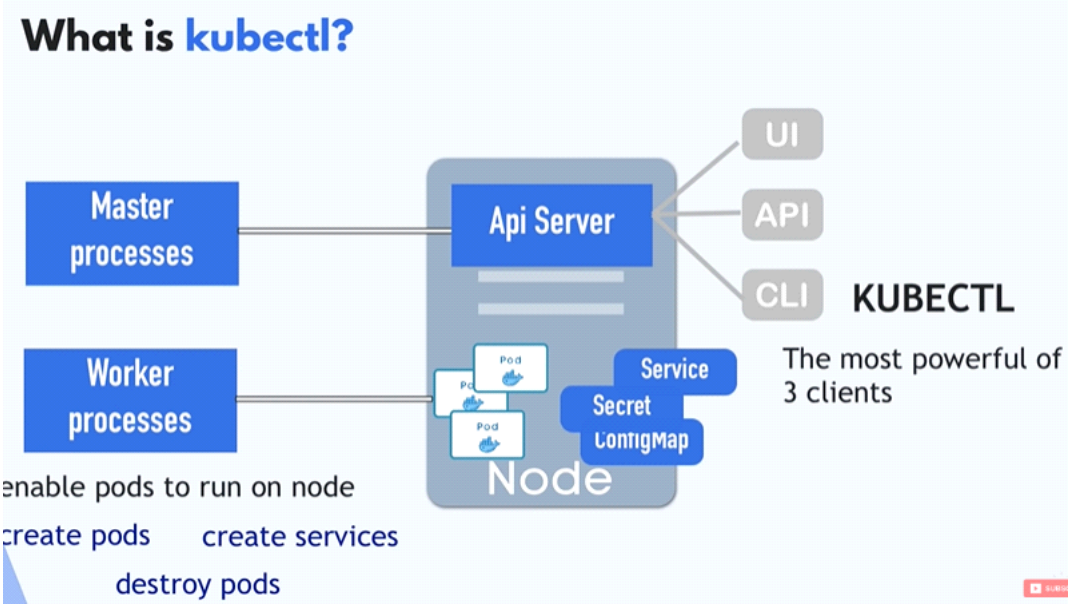
Minikube :

When we try to establish a production cluster we would have multiple masters and multiple worker nodes there we would have many nodes it means we would have separate virtual or physical machines. All these processes work on individual nodes of their own , but we could not set them up into our local machine , because we must setup multiple nodes. But when coming to minikube all the master and worker process will be kept under same node , it mean master and node process run on one machine.

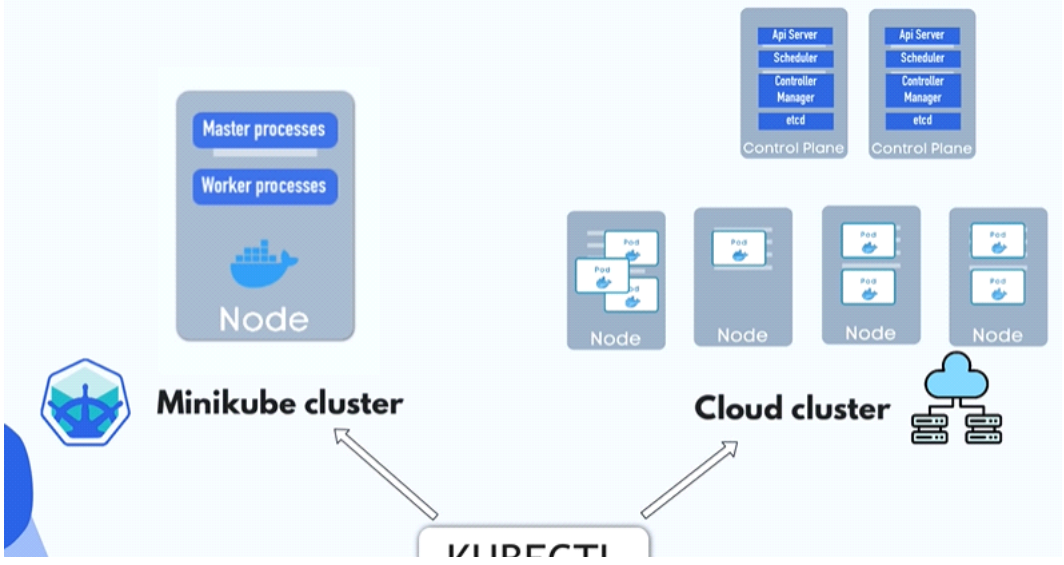


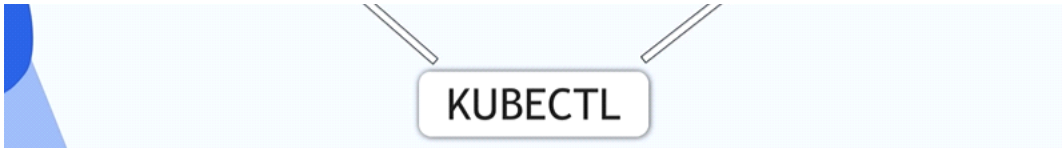
- All the master and worker processes kept under same node and to run those containers we have docker pre installed in the node. And the node which contain all the master and worker process is called minikube . Minikube have docker pre installed.



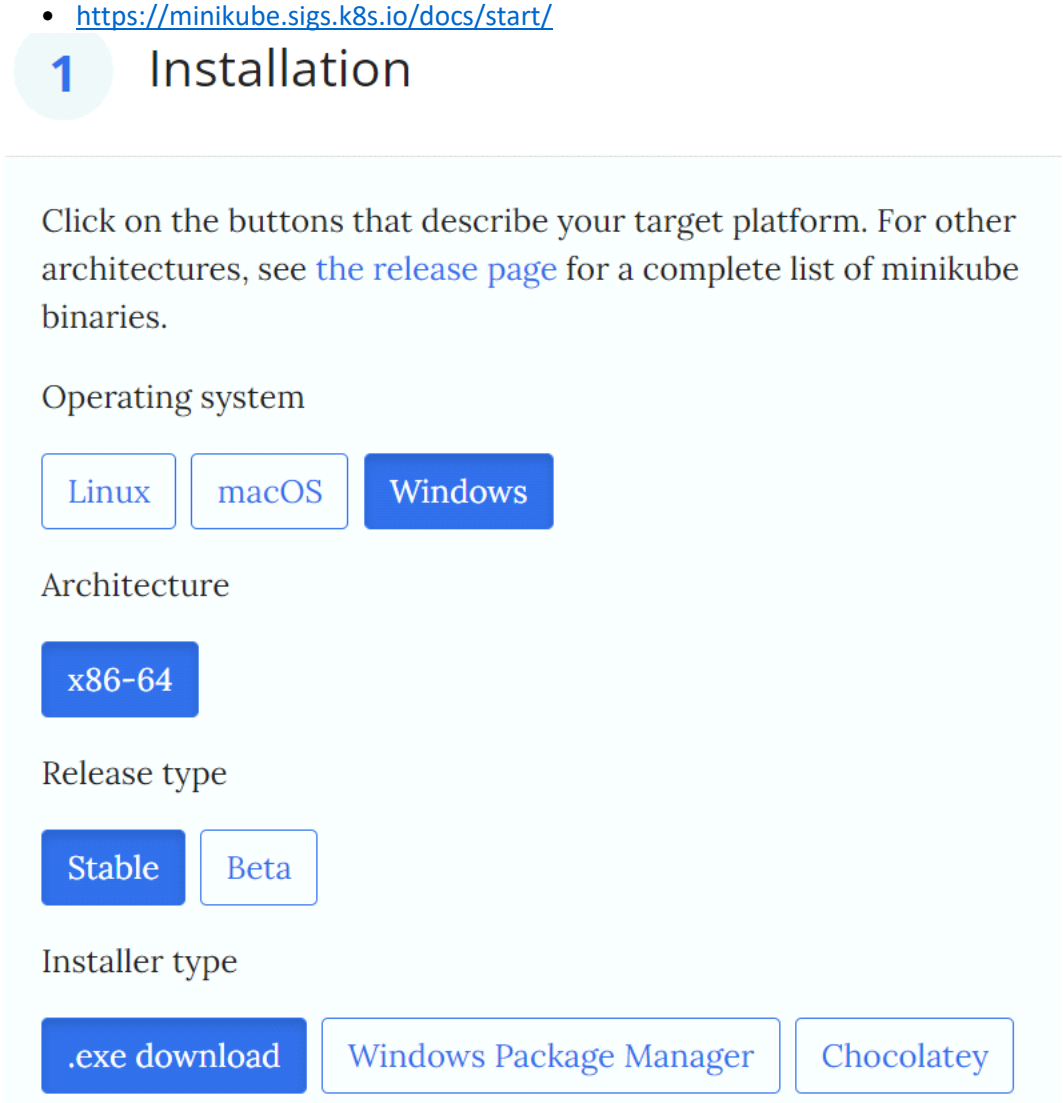


- Here we are having a node called Minikube where we are having multiple pods and services , IP , secret , configmap. So now we need some support to create those inside the node. For that we use kubectls. In general, the main communication between external to the node happens through the API server. We use UI , API , CLI which uses kubectl to contact with the API Server and to manipulate the internal pods and components of the node. API server is part of master processes and the instructions that API server got will be passed into worker process, as we know worker processes are the one which run the process or container. Worker processes can enable pods to run on node , create pods , create services, destroy pods.
- KUBECTL is used to interact with the Minikube cluster or any cloud cluster

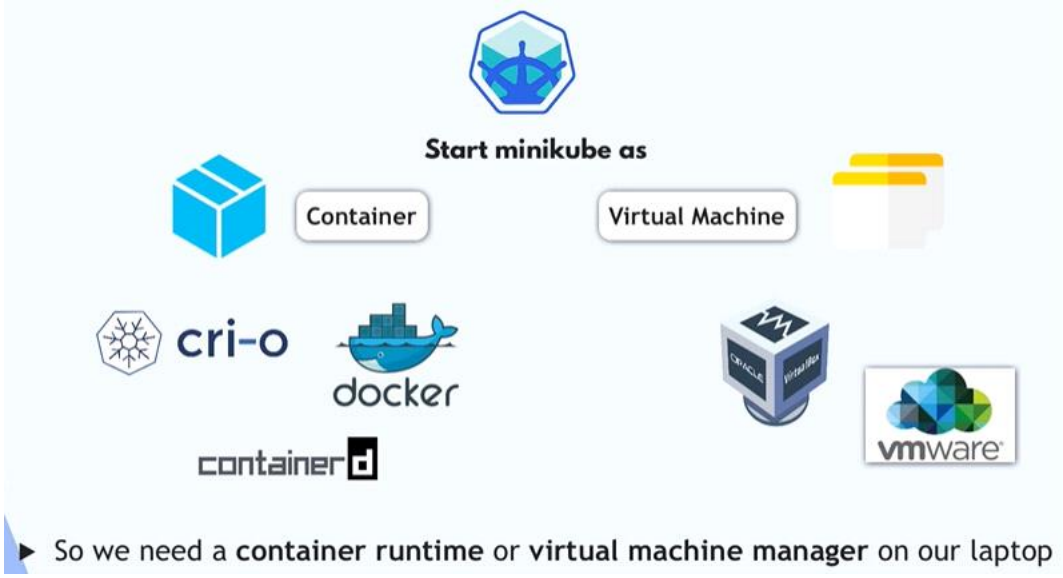




- Minikube can run as a container or a virtual machine.
- Follow this page for minikube installation.
- https://minikube.sigs.k8s.io/docs/start/



We can choose installer type as windows package manager or chocolatey also, the installation steps will be changed respectively.



2 Start your cluster

From a terminal with administrator access (but not logged in as root), run:

```
minikube start
```

If minikube fails to start, see the drivers page for help setting up a compatible container or virtual-machine manager.

Redirect to the drivers page

Linux

- Docker - container-based (preferred)
- KVM2 - VM-based (preferred)
- VirtualBox - VM
- None - bare-metal
- Podman - container (experimental)
- SSH - remote ssh

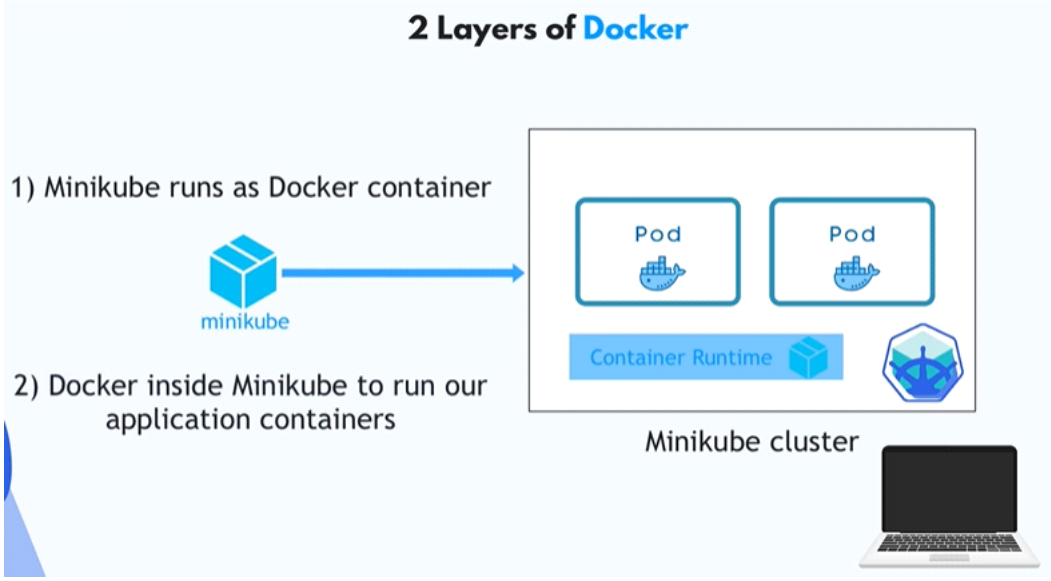
macOS

- Docker - VM + Container (preferred)
- Hyperkit - VM
- VirtualBox - VM
- Parallels - VM
- VMware Fusion - VM
- SSH - remote ssh

Windows

- Hyper-V - VM (preferred)
- Docker - VM + Container (preferred)
- VirtualBox - VM
- VMware Workstation - VM
- SSH - remote ssh

To run or start the Minikube we can run it as container , virtual machine. So we need to install the driver required to run the Minikube, the docker is the preferred type to run the Minikube.



Minikube it's self a driver , and it is pre installed with the docker. It is a docker container , and inside of it several pods of container runs. Here Minikube runs as docker container, and inside we have docker which is pre installed , it is used to run our application containers.

```
[\W]$ minikube start --driver docker
😄  minikube v1.23.2 on Darwin 10.14.6
✨  Using the docker driver based on user configuration
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
💾  Downloading Kubernetes v1.22.2 preload ...
    > gcr.io/k8s-minikube/kicbase: 355.40 MiB / 355.40 MiB  100.00% 1.70 MiB p/
    > preloaded-images-k8s-v13-v1...: 511.84 MiB / 511.84 MiB  100.00% 2.37 MiB
🔥  Creating docker container (CPUs=2, Memory=1985MB) ...
🐳  Preparing Kubernetes v1.22.2 on Docker 20.10.8 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
[\W]$
```

Whenever we want to start a Minikube based on a driver we need to use the following command . To execute that command we need to install Minikube and docker to install on the system and use the following command , so that a Kubernetes cluster starts.

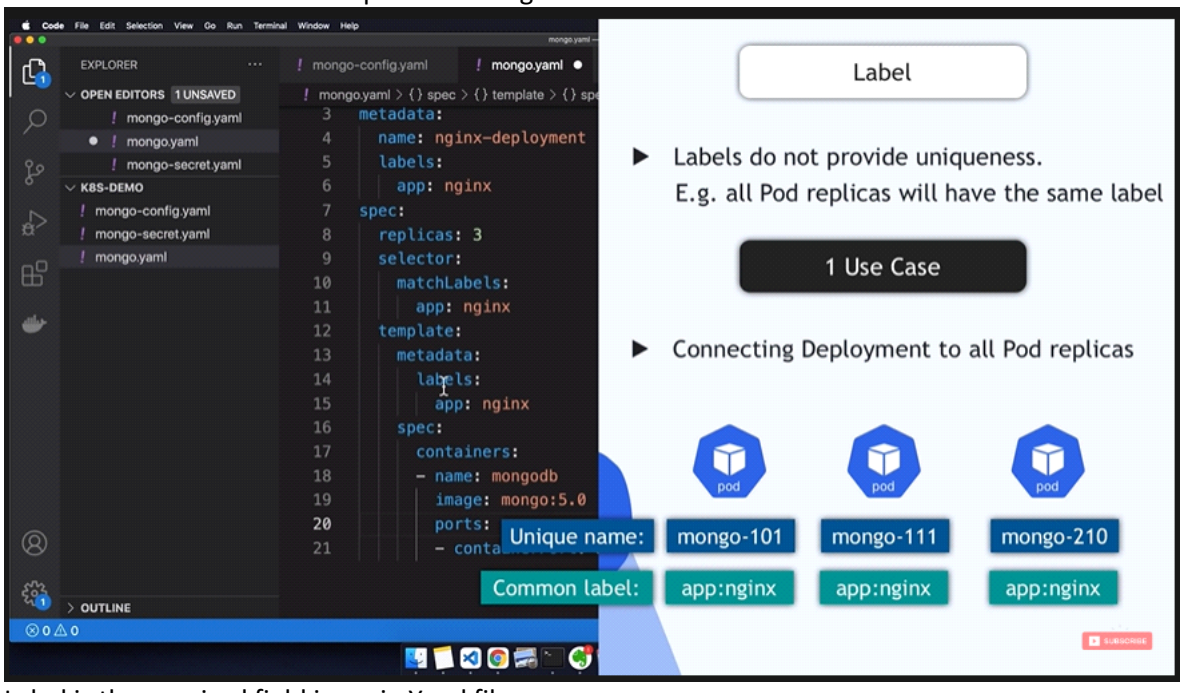
```
[\W]$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

[\W]$
```

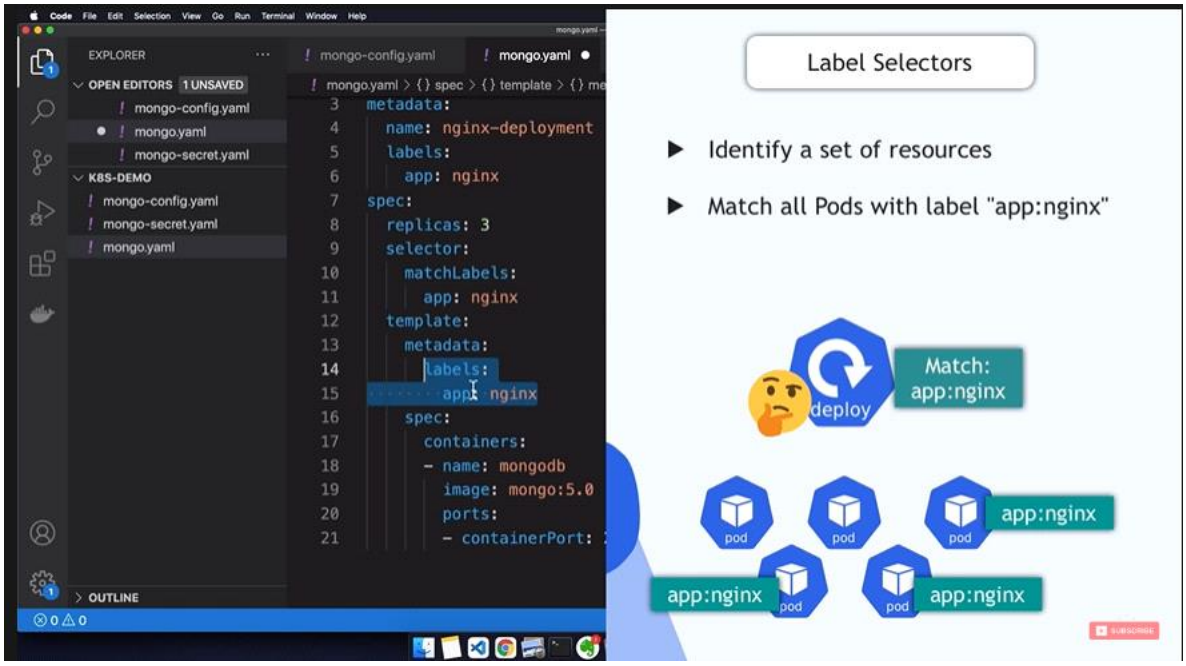
And to know the status of Minikube cluster , use the following command to know the status of Minikube cluster. To interact with Minikube we use Kubectl, and Kubectl will be installed as a dependency while we install Minikube.

```
[\W]$ kubectl get node
NAME       STATUS   ROLES                  AGE    VERSI
minikube   Ready    control-plane,master   117s   v1.22
[\W]$
[\W]$
[\W]$
[\W]$
[\W]$
[\W]$
[\W]$
[\W]$
[\W]$
[\W]$
```

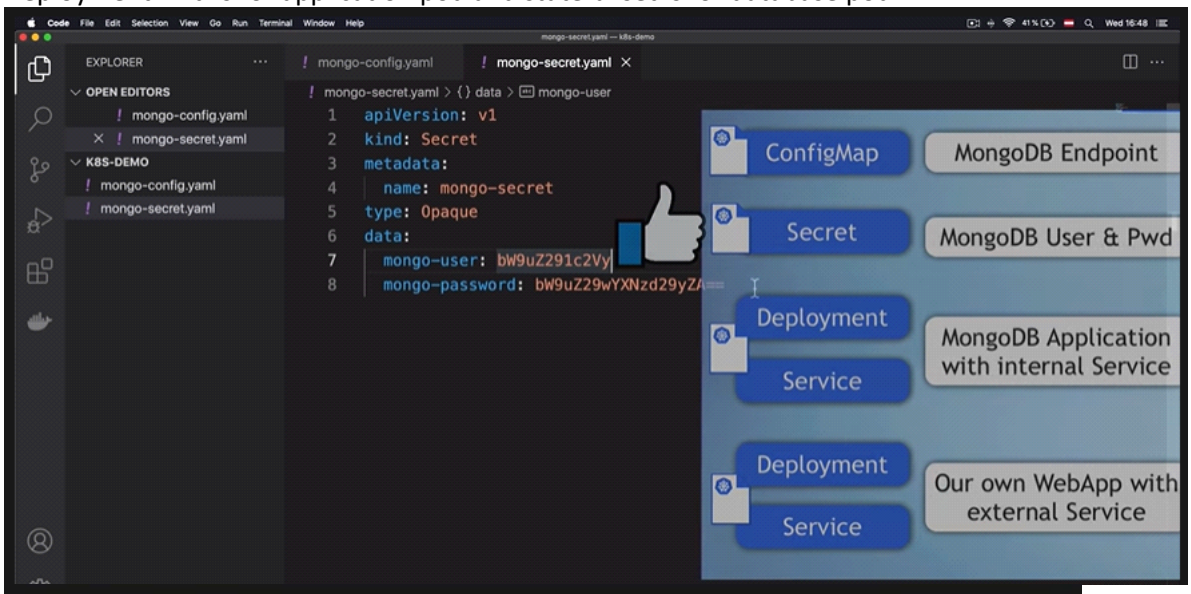
Here the following command is used to get all the nodes under Kubectl. We are going to interact with Minikube cluster with the Kubectl. Kubectl is used for configuring the Minikube cluster and Minikube cli is use d for startup and deleting the cluster.

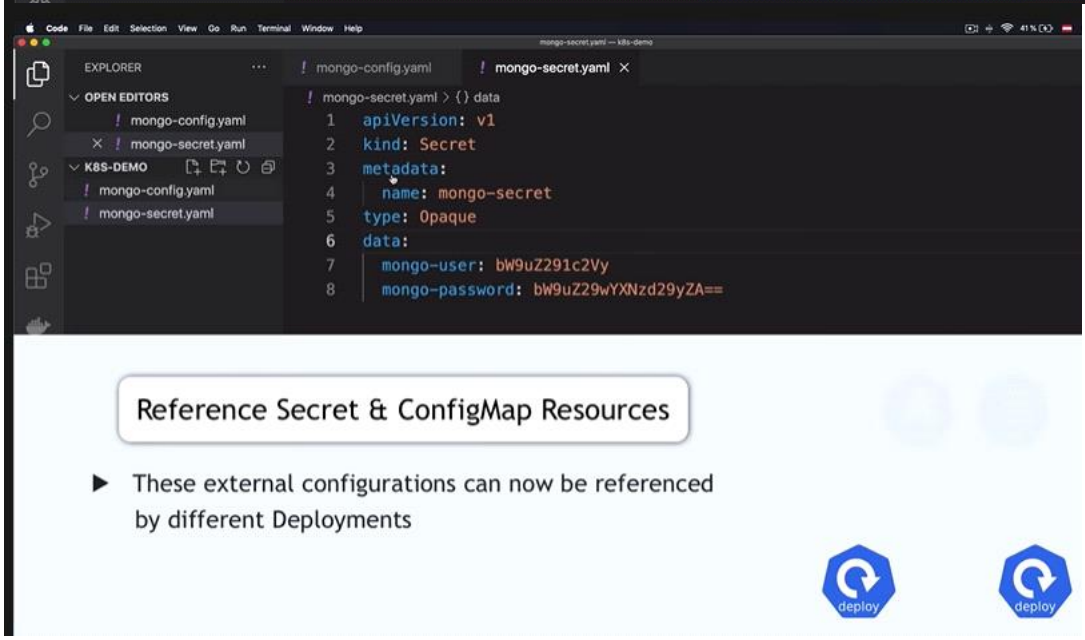


Label is the required field in main Yaml file.

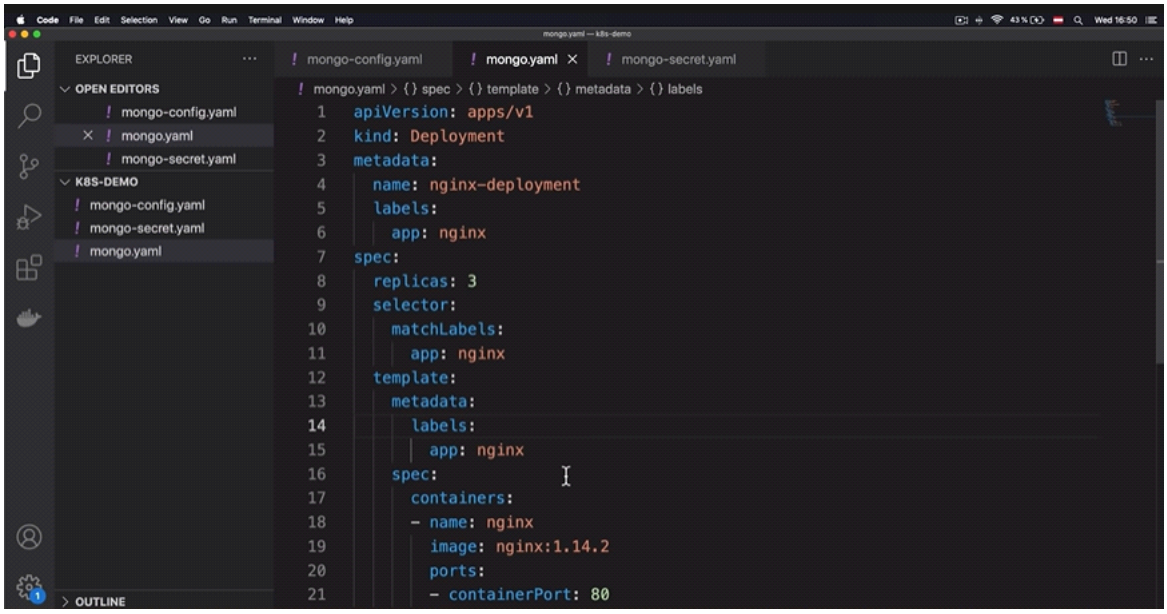


Replicas is how many pods we want to create.
Deployment kind is for application pod and stateful set is for database pod.



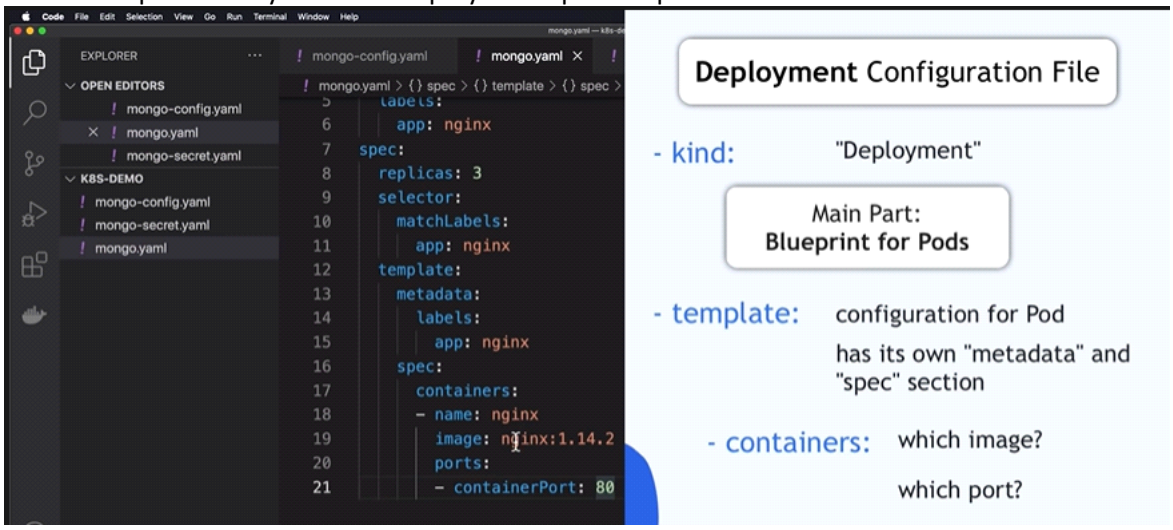


- In mongo.yaml we declare the deployment and service in 1 file because they belong together.



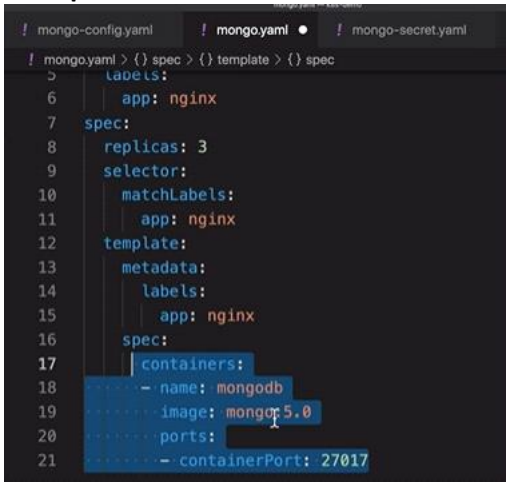
mongo.yaml will be complex than the mongo-config and mongo-secret, the config and secret file's can be reference here.

- The spec in this yaml file is deployment specific specification



Mongo.yaml file is the deployment configuration file , where we defined type as the deployment in kind. In mongo.yaml file we declare the blueprint for pods. In template we declare the configuration of the pod , it will have it's own meta data and deployment or pod specific specification. And in the spec inside the template we declare the container details like which image and port for the pod.

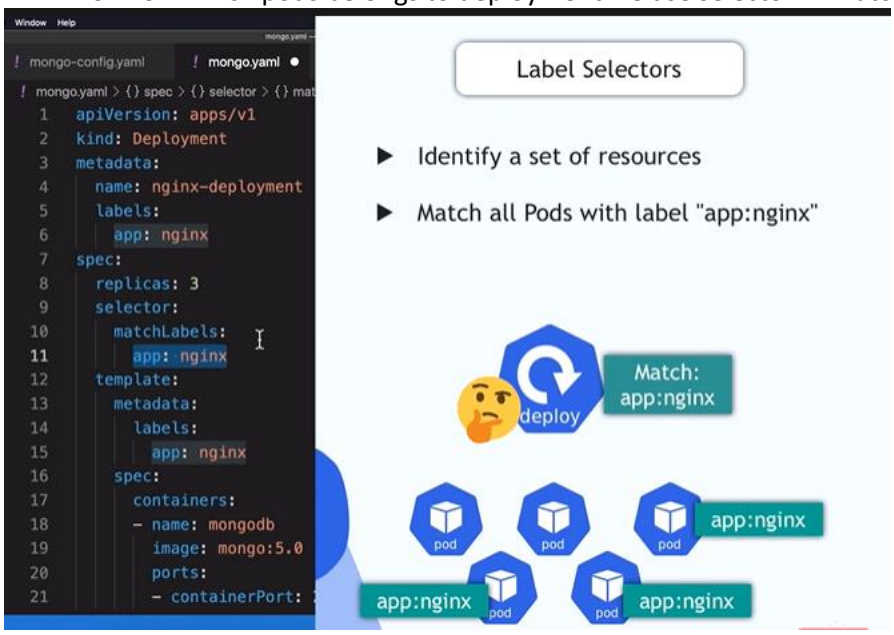
Example:



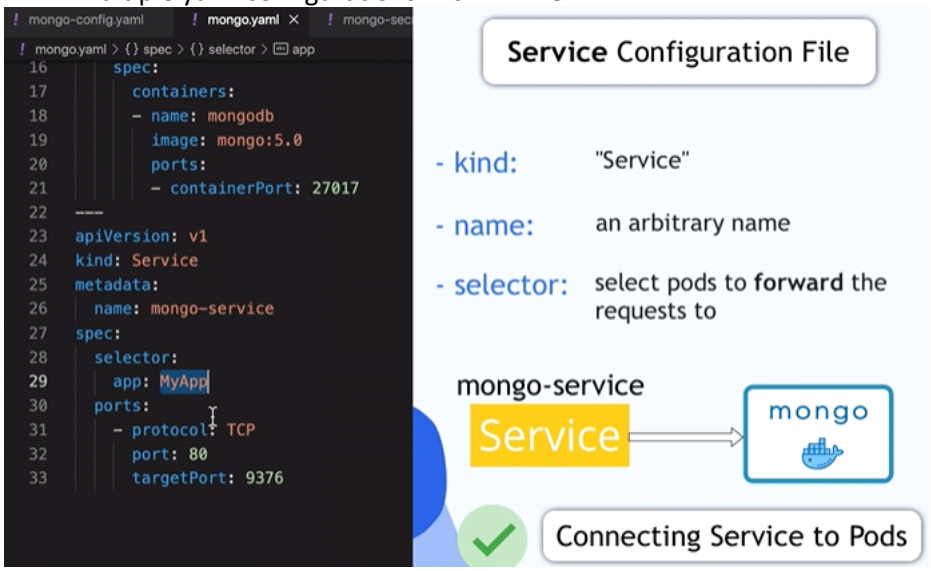
- We can have labels in the yaml file, labels are key/value pairs that are attached to K8 resources.
- Label do not provide the uniqueness, all pod replicas will have the same label , it is used in connecting deployment to all pod replicas. If a pod is not working , then other pods will take

that work. And each pod will have it's own unique name. But they share a common variable. We can identify all replicas of the application with the same label name.

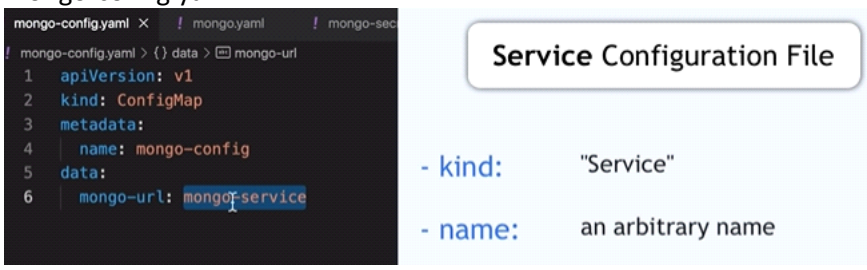
- To know which pods belongs to deployment we use selector -> match labels for it.



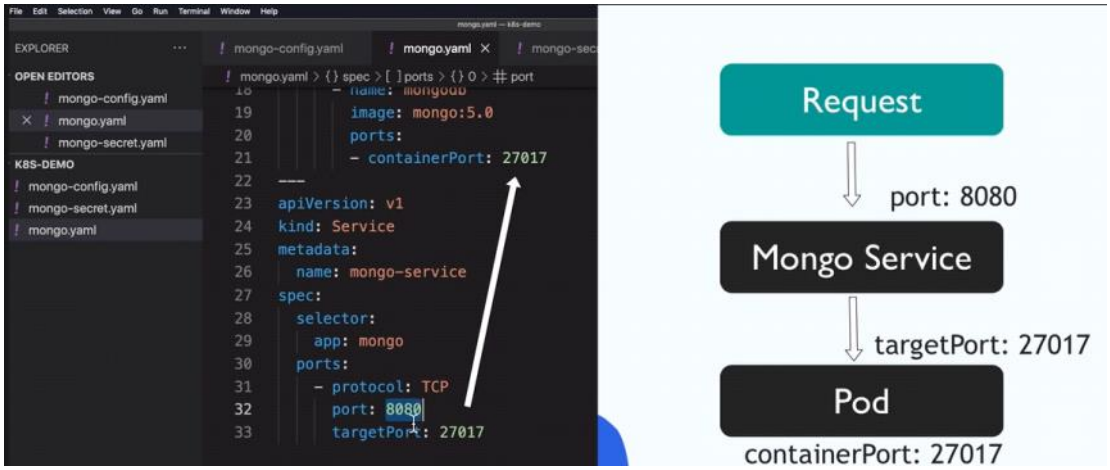
- Here we can see, we are having many pods, and we need to do deployment for the pods which have matchlabels as nginx under specification of that pod . And we defined the type of process to run under kind of the main yaml file as deployment.
- Here in the above example we will add pods which have label name as nginx.
- In this example we are going to declare two types of configurations they are deployment and service , we separate those configurations by "---" three dashed line . As we can declare multiple yaml configurations within 1 file



The services and deployment which we declare at the .yaml file will be used by the other yaml files , in this example mongo-service will be declared at mongo.yaml and will being used as a service in mongo-config.yaml



The service and pods which are declared in mongo.yaml will be connected through the selection app name.



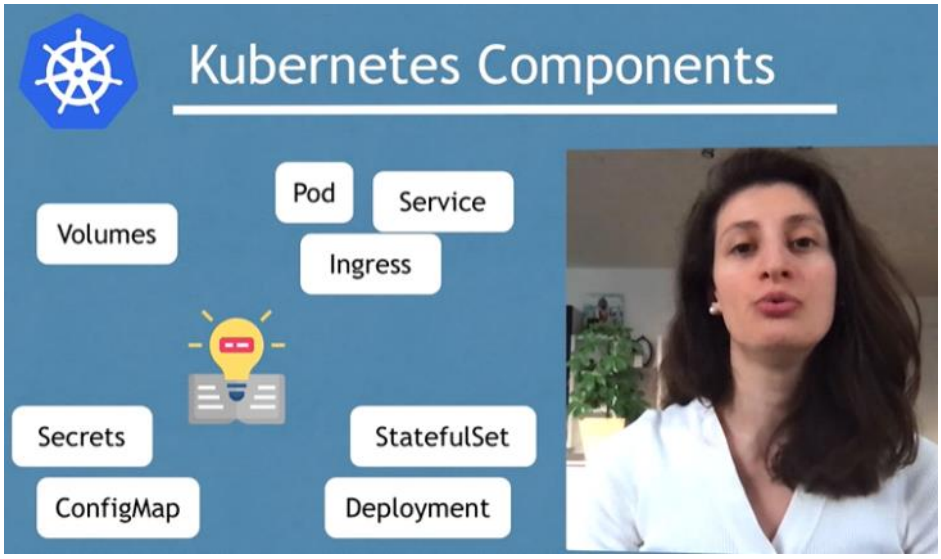
- Here as we know we are creating a pod and a service in the mongo.yaml file , where we need to post a request from mongo to the pod through the mongo service which is created by us in mongo.yaml. Here the service will be host at 8080 as declared and the mongo service will push the request to the pod which is located in port 27017 and that address is located at mongo service.
- Deployment must be done only for stateless applications
- Stateful set is used
-

- Ī

Kubernetes (4hrs video)

26 April 2022 16:14

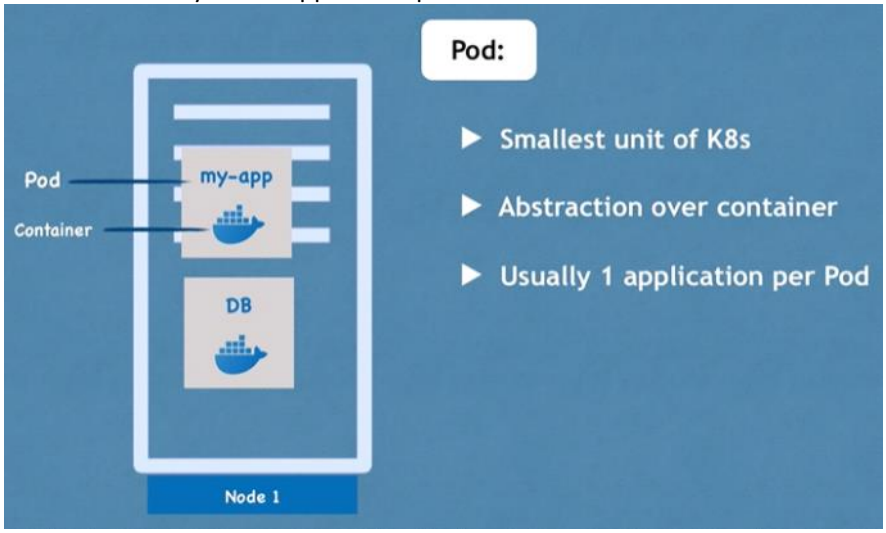
- Kubernetes is open source container orchestration tool , it is developed by google.
- Kubernetes manage the containerized applications , may be they are containerized by the docker or any other software or framework, It can work in different deployment environments like it will work in either physical , cloud and virtual machines.
- The trend from monolith to microservices, make containers more demanded because containers are the one which provide the environment to run the microservices. We could not have a different physical or virtual machine to run a microservice, because it is cost sensitive. And to manage all those hundreds of containers we could use some scripts, but maintaining an manipulating using those scripts are so difficult . So we need a software which helps us to manage those containers and this process of managing is called orchestration.
- These orchestration tools offer
    - High availability or no downtime
    - Scalability or high performance
    - Disaster recovery - backup and restore.
- Kubernetes manages hundreds of containers , so it can even manage the servers and the nodes which are require for request processing . So we will have no downtime of the application. And hence we will have fast processing of requests , so we would have high performance. And Kubernetes will prevent loss of data if any node crashes whether it may be a application node or database node, it replaces the node with the other node with previous data initialized.



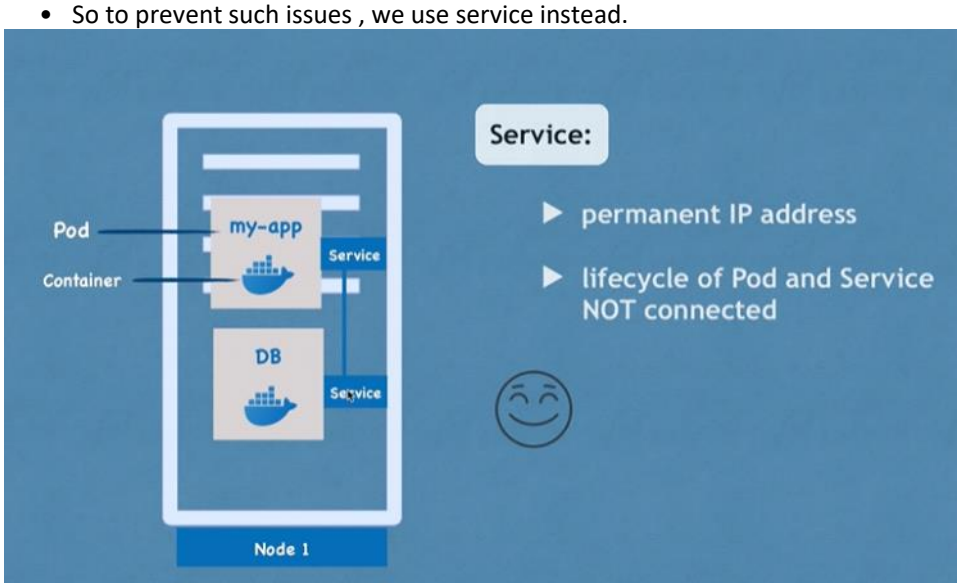
Worker nodes

POD:

- Smallest unit of Kubernetes , pod is a abstract layer over a container . We only interact with the Kubernetes layer. But not directly with the docker container.
- POD usually runs 1 application per POD.

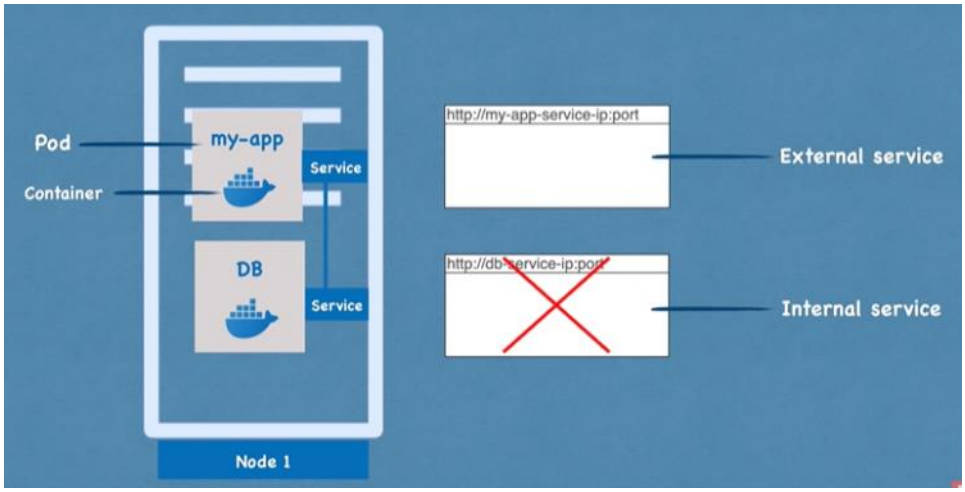


- Here we can see Node 1 is a server and it contain multiple pods and each pod contains usually one container where here we have two pods, one pod consists application image and other contains database image.
- Here each pod has it's own IP address
- But this IP address technique have it's own defects , like if a pod which have an IP address have no resources to run the process then the particular pod dissolves and other pod is created for the same functionality but with different IP address , but this is a tough technique because , we need to change all the other pod's output addresses which are communicating the data to the earlier node as we have a new address need to be assigned.
- So to prevent such issues , we use service instead.

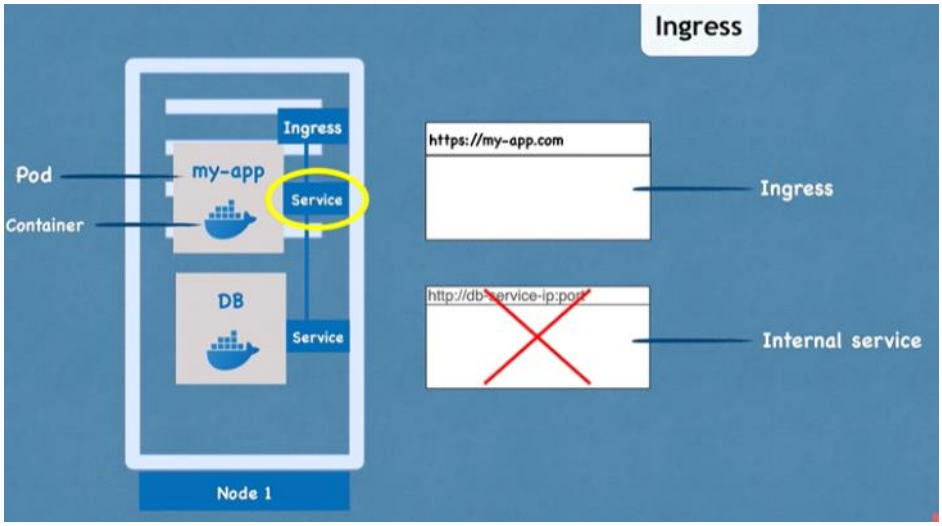


Service

- Here there is another concept called service , where we have a pod and IP address and a service is assigned to it. But the service is like a permanent IP address where when POD dissolves the Service address won't dissolve but IP does, so whenever a new pod is created for the same purpose as the previous the service address doesn't change , so the communication process between those pods will not be affected , and there is no need to change any communication details.

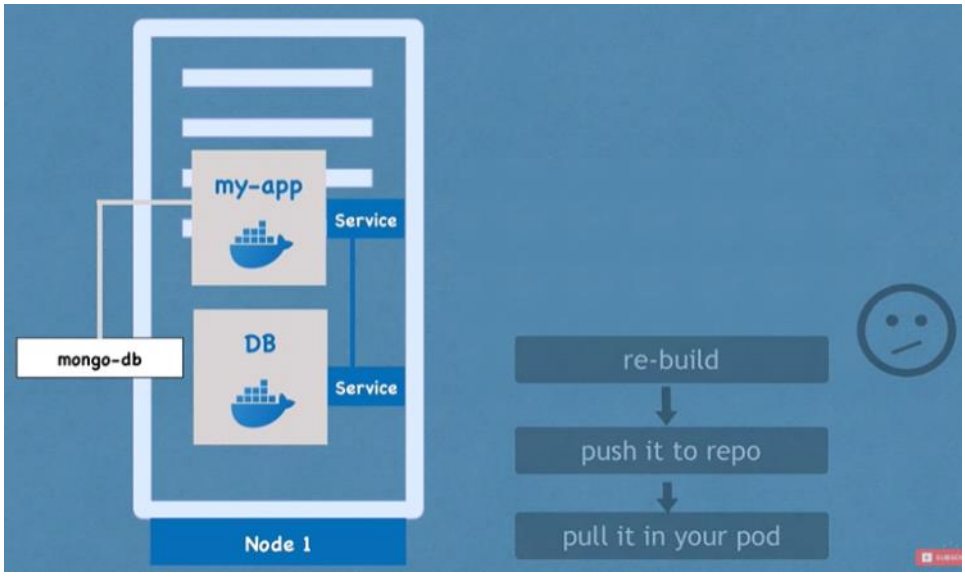


- Here suppose think that Node 1 is the server , where that server runs two pods , one pod is for the application and other pod is for the database , and each pod consists of it's own container. Here to access the application and database we need service. When we want to access the service outside of the node , we use External service and if we want to access the service internally in the application but not from outside the application we use Internal service.
- And as we can see, the pattern of External service is in the form of IP Address : Port number but it will be not good for the user to access in that way, so we use Ingress



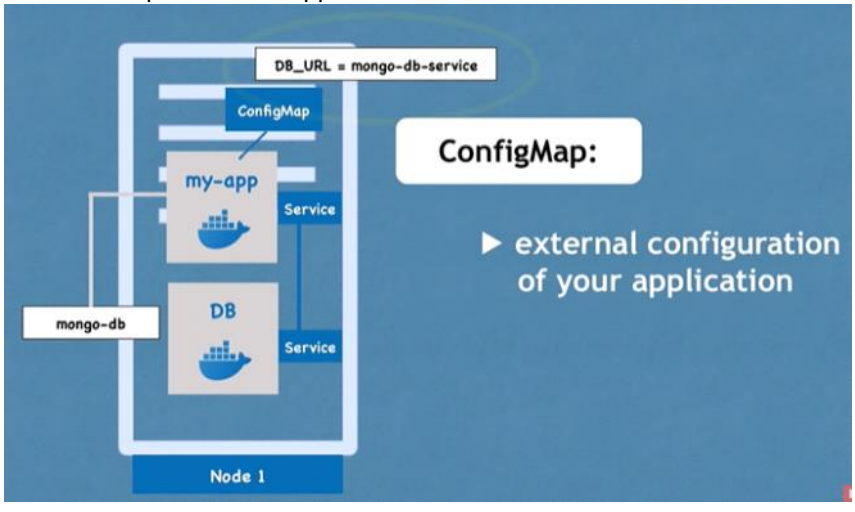
Ingress

- In general if we use the IP address and port number , it goes directly to the service number of the application pod , but when we use URL with domain name , it goes to the ingress and then redirected to the service address of the pod and runs the application . This ensures encapsulation for the application because it doesn't expose the port number and IP address to the End user.



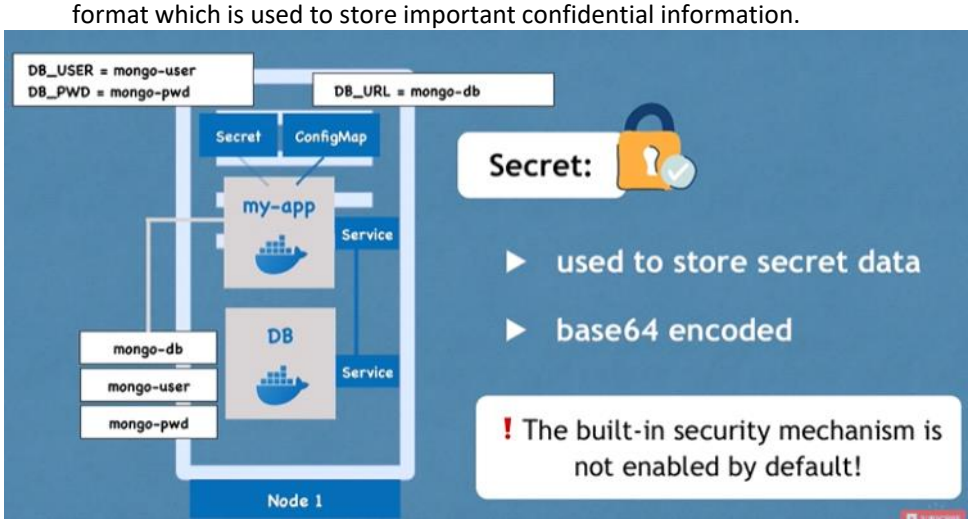
ConfigMap:

- Suppose consider a case where we are in the application and want to access the database , then the URL of the database is present inside the application. And whenever we need to change that link , we need to change the file of the application and rebuild the image , push it into the repository and pull the changed image into the pod. So for a small URL change , now we are doing such more amount of work . So to prevent that , we are creating a file externally to the pod called configmap which contains all the details of the connection and the URL's that are required to the application.



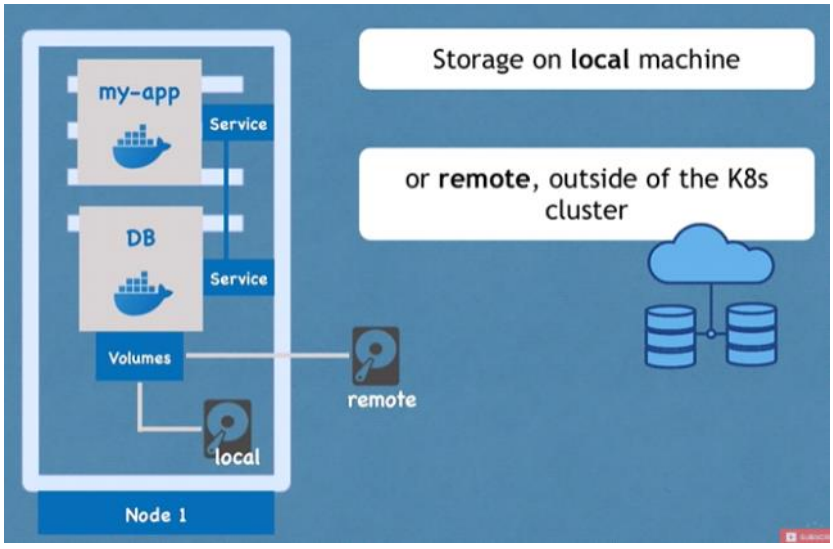
Secret:

- But now when we want to store some confidential information , we couldn't store that in the configmap as we should not store the passwords in the plain purpose.
- We have to store the credential's in the encoded format for that we are using the secret file format which is used to store important confidential information.



- We can store information such as environment variables or properties file in configmap and secret.

Volume:

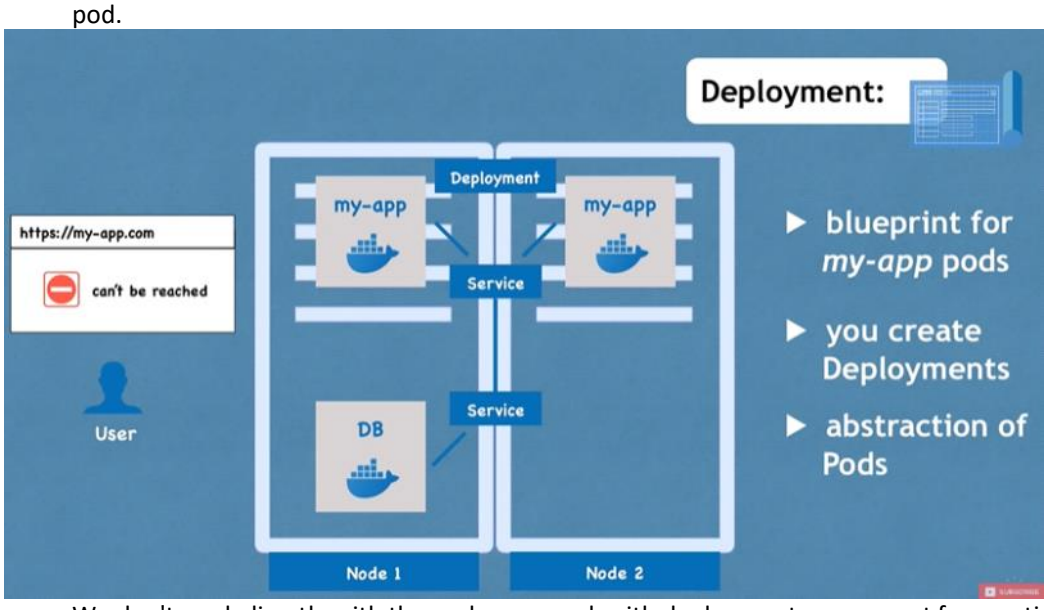


- Suppose , consider a scenario where we are having a Database pod which consists some data and if that pod got restarted then all the data in it will be re initialized in the sense , it would be empty and all the data is lost. So we want something to store the data and persist the data.
- Volumes of Kubernetes is used for persisting the data. And that volume can be connected to the local storage of the host machine or the remote storage inside the cluster or a cloud storage outside the cluster. These external storages helps in persisting the data of the database.

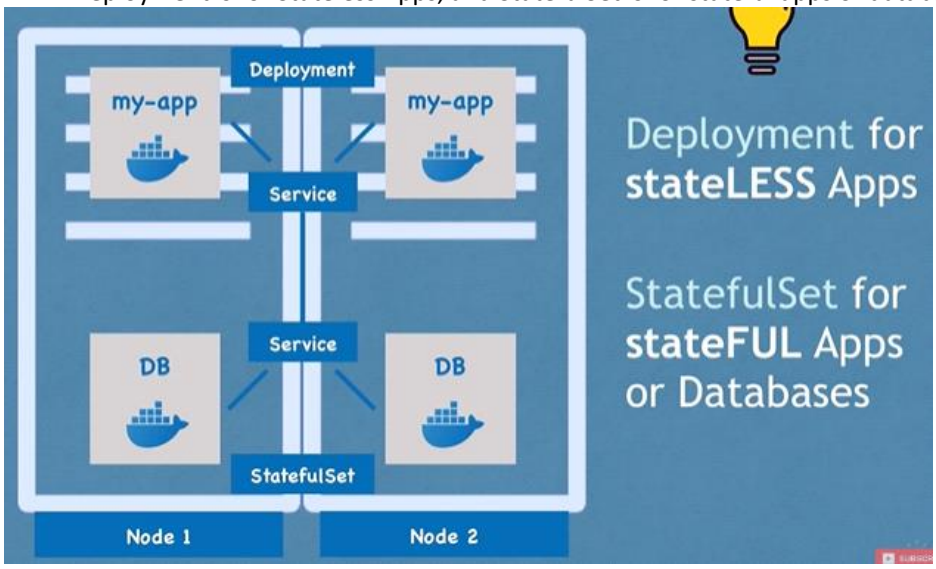
- Kubernetes doesn't manage the data persistence by itself, so to enable the data persistence

we must add the local or remote or external storage by ourself .

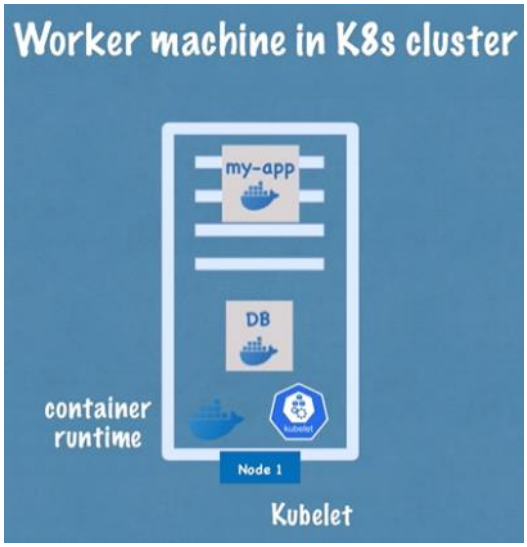
- Using this volume concept we have managed the database persistence if it dies, then what about the application, what will happen if the application pod dies.
- If application pod dies , we may face downtime of the application. Hence we would replicate everything on multiple servers , so we can prevent downtime. Even if one application pod dies, we can use other application pods of other servers to manage the downtime.
- All the replica's will be connected to the same service as they perform the same function , replicas can also work as the load balancer for the application.
- Here as we said , we want to have multiple pods, but we wouldn't create multiple pods. We just create a blueprint for the my-app pods. For this purpose we have a component called deployment which is used to declare number of replicas we need to create for the application pod.



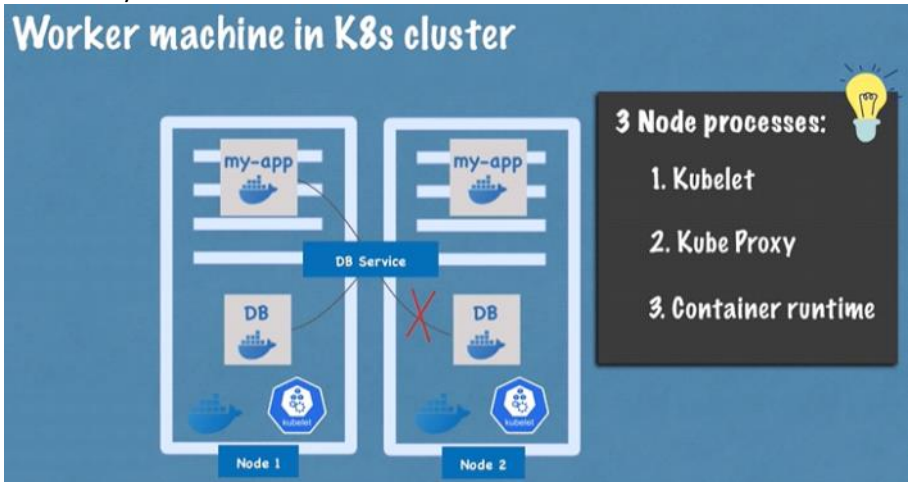
- We don't work directly with the pods , we work with deployment component for creation of pods and creating replicas
- For the my-app application pods we can create replicas, but we could not create multiple instances or replicas for the database through deployment component. Because my-app application is stateless because we no need to have to consider state of the application for the deployment , where as while coming to the database we must consider database state , because multiple database replicas must have it's own state to avoid data inconsistency. We need to track down the state of replicas and make changes for the database or the database will be inconsistent.
- Deployment is for Stateless Apps, and StatefulSet is for stateful apps or databases.



- Now we have two replicas of the my-app application pod and two replicas of DB pod, both are load balanced , even if one pod of a server crashes the other replica will handle the request. So we also decreased the downtime.
- In Kubernetes architecture we have master and slave nodes.



- Here in every node we are having some components , for example container runtime and kubelet. Container runtime is the software which is used to create the images and run the container. Kubelet Schedules the pods and the container underneath. Kubelet is process of Kubernetes, kubelet interacts with both the container and the node. Kubelet starts the pod with a container inside, it also assign the resources to the pod. Every node in the Kubernetes have its own container runtime and kubelet to be installed.
- We use forwarding requests from services to the pod using kube proxy.
- For making the Kubernetes cluster to work properly, we need to have three to be installed on every node .

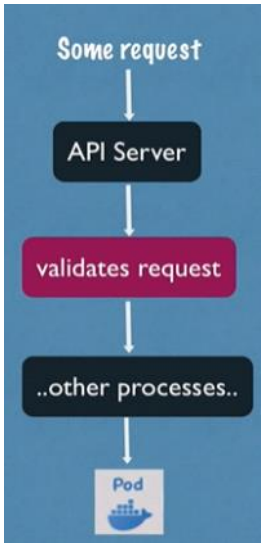


- They are Kubelet , Kube proxy and container runtime.

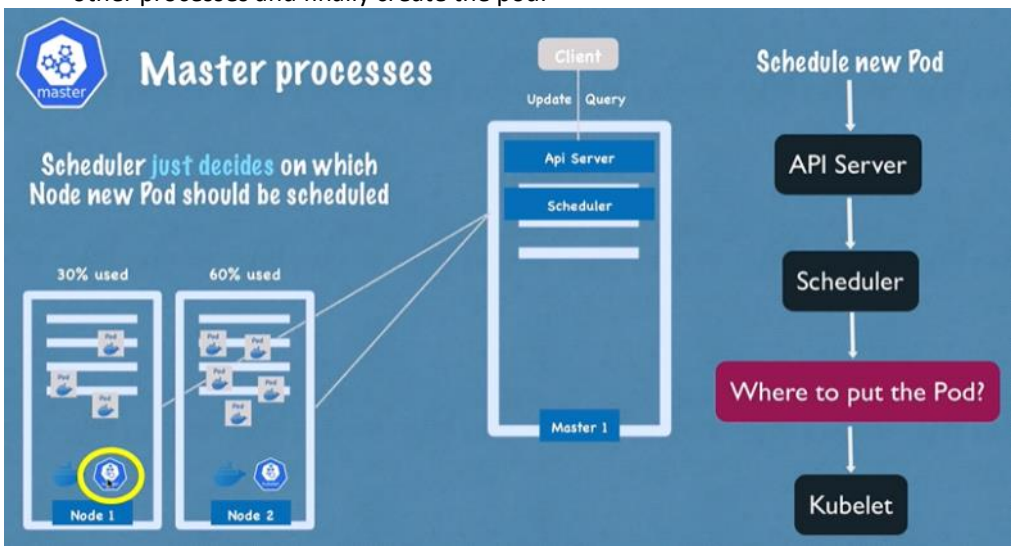
We need to interact with the cluster for schedule the pod or monitor the pod or re schedule or restart the pod or join a new node we use the master node, master node does all these process. All these managing process are done by the master nodes.

Master nodes

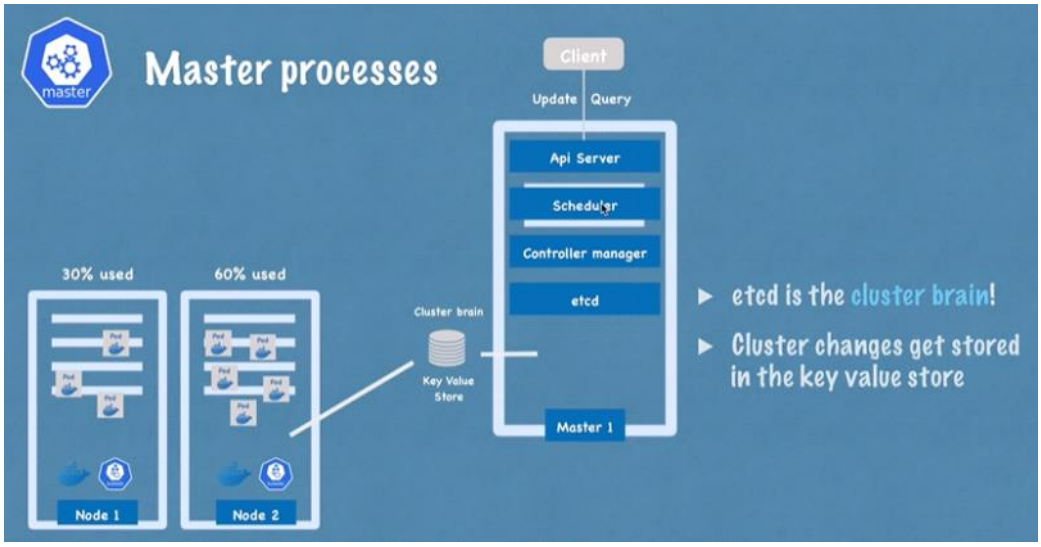
- Master nodes have 4 processes run on every master node .
- Suppose if we want to deploy a new application on the cluster , then client contacts the master node's API server , so it would contact the worker nodes to create what client needs.
- If the client want to update or query in the cluster the API server acts as the gateway and acts as the gatekeeper for authentication. So only authorized users can manipulate that.



- If we want to process any request to create a pod, we send that request to the API server and then the request will be validated by API server and if everything is fine it send the request to other processes and finally create the pod.



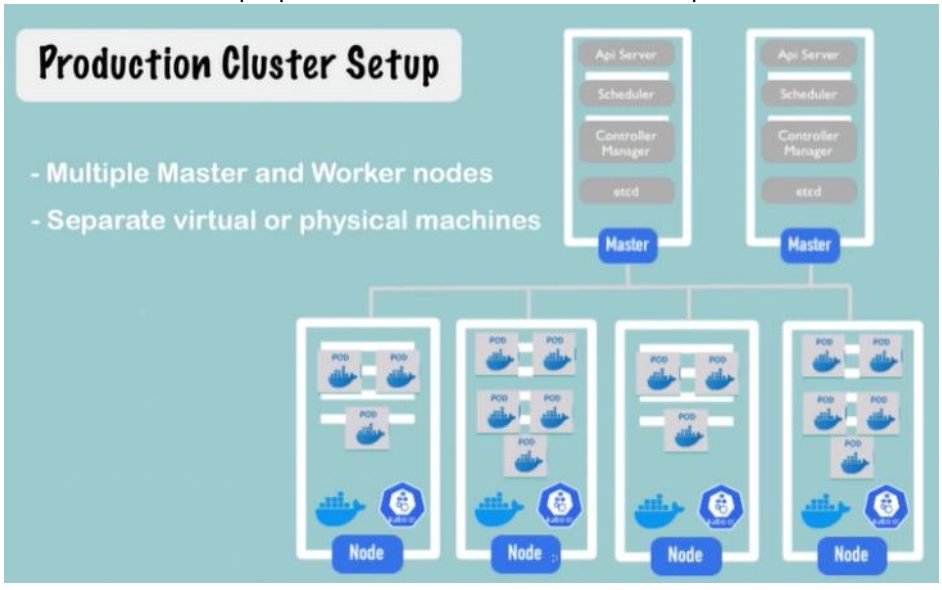
- Another important process of master node is the Scheduler , suppose think , we got a request to schedule a new pod from the client then the request is received by the API server and if all the validations done it goes through the scheduler . This scheduler have a mechanism to find the best way to do it. In our case to put a pod , it checks , where we have resources available and schedules in that way towards the node. And the actual service that runs the request is Kubectl which takes the request and executes it and create a new pod in that node.
- We have another important component or process of the master node that is controller manager.
- Controller manager detects the state changes in the nodes, suppose if any pod has been stopped or dissolved then controller manager detects the state changes and initiate the process for restarting the pod. It sends a request for a scheduler to schedule the pod and similar to the scheduler process, it checks where the pod need to be assigned for the best use and then corresponding to the node the following kubelet will process the request and create the pod on that server or node.



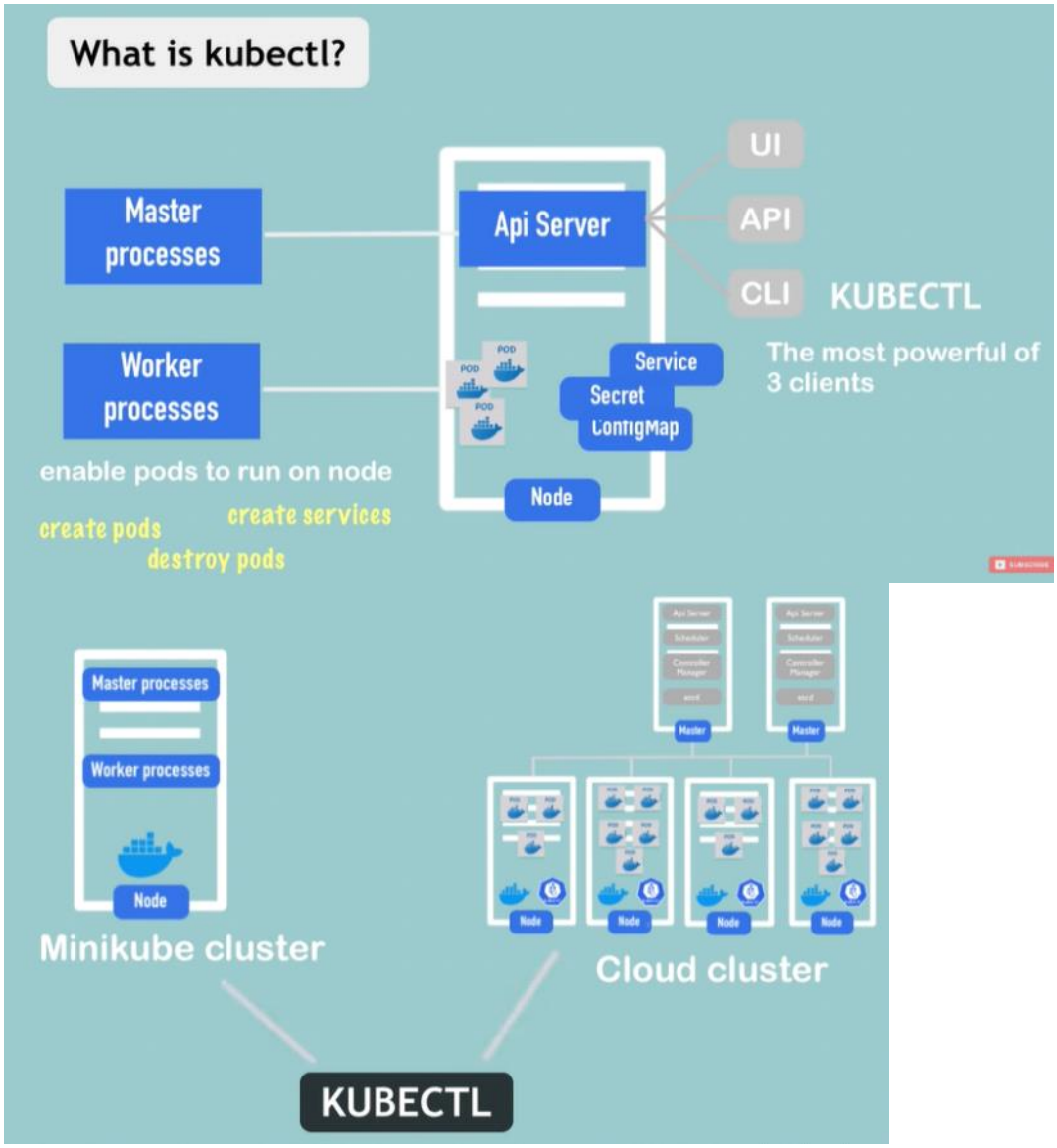
- etcd is the cluster brain where it stores the cluster changes in key value storage format.
- Due to this etcd we are able to know what resources are available and did the cluster state change , health of the cluster.
- Etcd doesn't store the application data.
- Kubernetes have multiple master nodes, and etcd is the distributed data across all the master nodes
- In a typical cluster we would have two master nodes and three slave or worker nodes. Master nodes require less resources and worker nodes does the main processing and hence it requires more resources.

Minikube:

When we set up a production cluster we will have multiple master nodes and worker nodes.



- Suppose we have such cluster setup and if we need to test that on our local machine then it is difficult to run these many individual virtual machines on the system. So , to solve this issue, we have a concept called Minikube . Minikube is a single node in which all the master processes and worker processes run on one machine.
- And Minikube have docker pre installed and it runs on the local system using the virtual box. All the nodes of the cluster runs in this single Minikube node in the virtual box.
- Minikube is a single node Kubernetes cluster.
- But to interact with the cluster, and create new components or configure it. To provide those instructions we use Kubectl.
- Kubectl is the command line tool for Kubernetes.
- Here the interaction is enabled by the API server of master processes of Minikube.
- We can interact with the cluster only through API server using UI , API , CLI. And Kubernetes is a CLI tool which helps in creating destroying pods .
- And the actual action is done by worker processes inside the Minikube. Master processes get the task validate it and send the work to the worker processes and worker nodes perform the task.



Minikube have Kubectl as dependencies, so we no need to install Kubectl again.
If those are installed type these following commands to check those are installed or not

- minikube
- Kubectl

To create and start cluster we need to declare the virtual machine on the command line

- minikube start --vm-driver=hyperkit

It tells, start the minikube using the following virtual machine.

- Docker is pre installed with the minikube , we no need to install docker separately.
- Kubectl used to interact with Kubernetes cluster.

To get status of nodes:

- kubectl get nodes

To know the status of minikube:

- minikube status

```
[~]$ kubectl get nodes
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   21h   v1.17.0
[~]$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
[~]$ kubectl version
Client Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.1", GitCommit:"d224
476cd0730baca2b6e357d144171ed74192d6", GitTreeState:"clean", BuildDate:"2020-01-15T15:50:
25Z", GoVersion:"go1.13.6", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.0", GitCommit:"7013
2b0f130acc0bed193d9ba59dd186f0e634cf", GitTreeState:"clean", BuildDate:"2019-12-07T21:12:
17Z", GoVersion:"go1.13.4", Compiler:"gc", Platform:"linux/amd64"}
```

Basic Kubectl commands

(https://gitlab.com/nanuchi/youtube-tutorial-series/-/blob/master/basic-kubectl-commands/cli-commands.md#debugging)

- Replica set is managing the replicas of a pod.

Layers of Abstraction:

- Deployment manages a replica set and pods.
- Replica set manages all the replicas of that pod.

- Pod is a abstraction of container.

Everything below the deployment must be handled by the Kubernetes.

Command:

kubectl edit deployment nginx-depl

If we execute the following command we get a auto generated configuration file with default values. So we can edit it if it is needed to do.

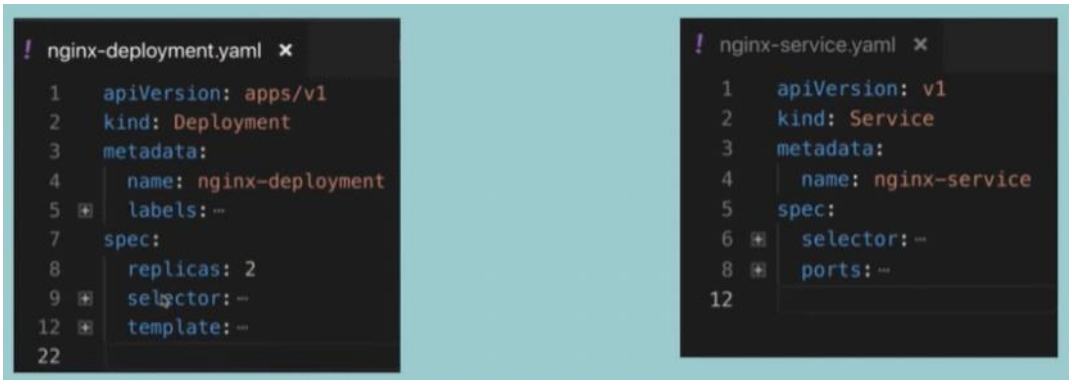

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will b
e
# reopened with the relevant failures.
#
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: "2020-01-23T11:20:43Z"
  generation: 3
  labels:
    app: nginx-depl
  name: nginx-depl
  namespace: default
  resourceVersion: "55942"
  selfLink: /apis/apps/v1/namespaces/default/deployments/nginx-depl
  uid: e6bf6b5b-d56a-4a99-b85d-9c5a56c46113
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
"/var/folders/y3/bvgmrxg950x0f1z4zt3pby3c0000gn/T/kubectl-edit-808e9.yaml" 67L, 1866C
```

Here we could not add or execute commands from the command line every time , so , we are creating another configuration file and apply that to the kubectl using "kubectl apply -f [configuration filename]" command

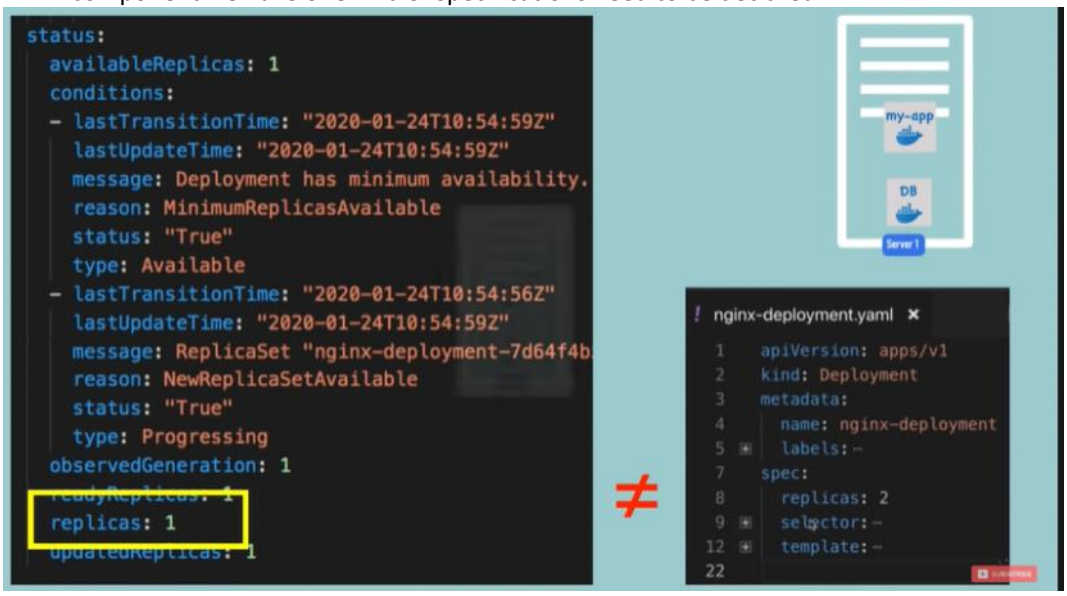
- This is the following configuration file

```
Terminal  Shell  Edit  View  Window  Help
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.16
        ports:
        - containerPort: 80
~
```

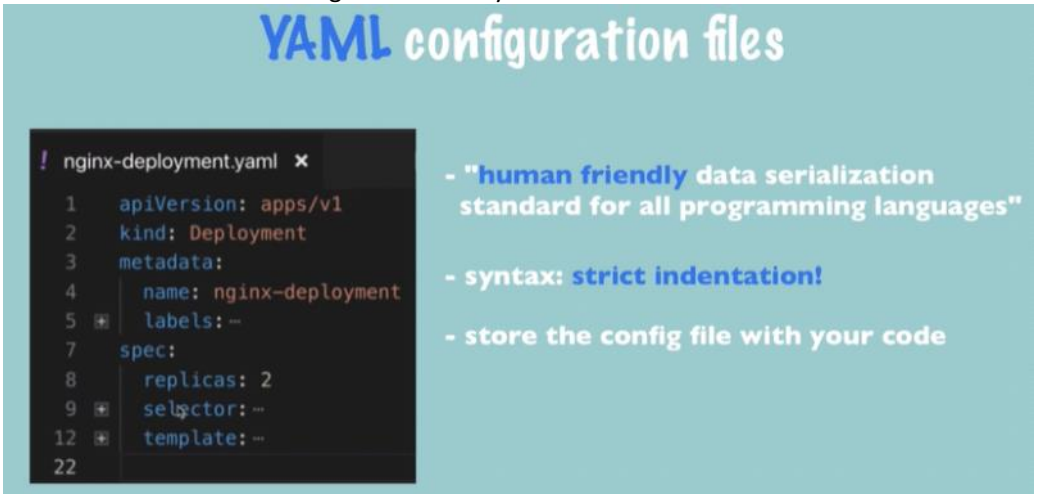
- We are creating a deployment component, how many replicas of pods we need to create , blueprint for pods , specification for the deployment and specification for pods , port numbers are being declared.
- In configuration file we have three parts.
    - Metadata of the component that we are creating
    - Specification of every kind of configuration that we want to apply to the component. Declaring Kind of the component and the apiVersion
    - Status (will be automatically generated)



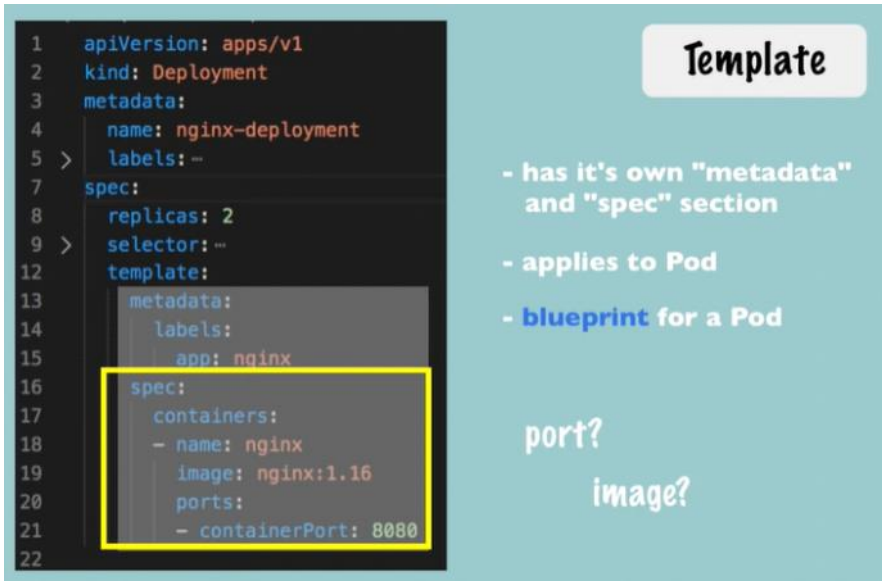
- The specification section varies from one kind to the other kind of the component, for deployment we have one kind of specifications need to be declared and for service component we have one kind of specifications need to be declared.



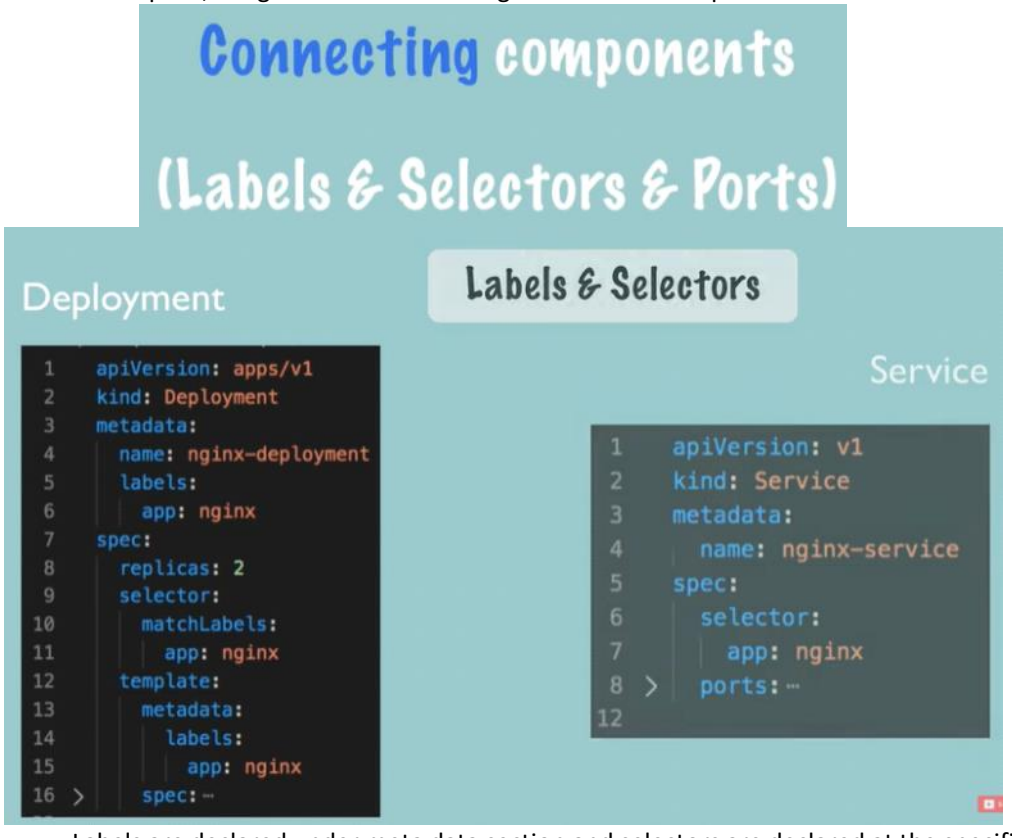
- Here , we declare our own specifications for the components and then when Kubernetes runs it check does the specification of the component and the status are matching or not. And if not matched , the Kubernetes creates a new node by itself and match the specification with the status. This status executes continuously and checks is there any changes to be resolved. We get this status data from etcd. ETCD holds the current status of Kubernetes component.
- The format of the configuration file is yaml.



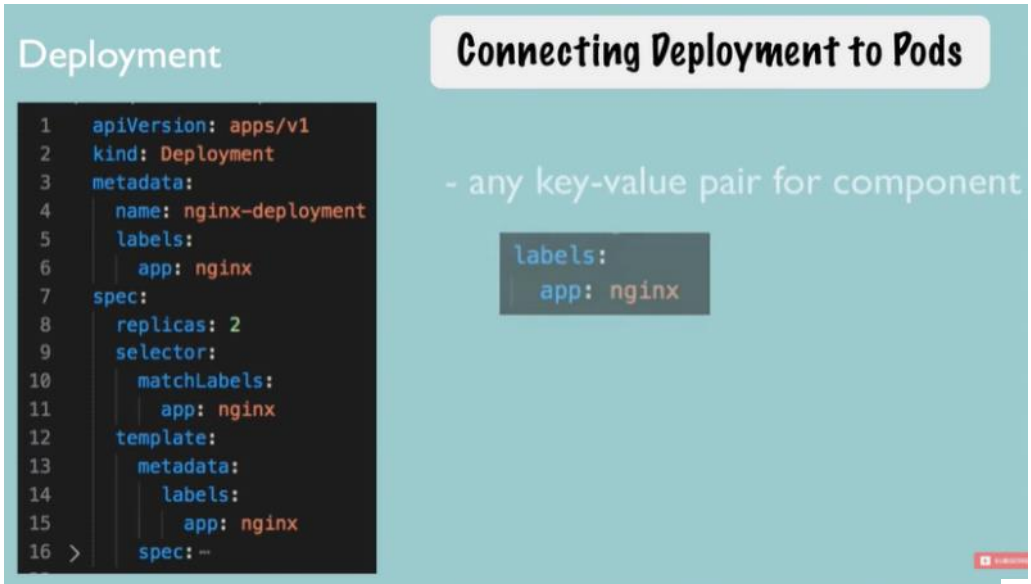
- Or we can store the config files in own git repository , but the general practice is to keep the config files with our code.

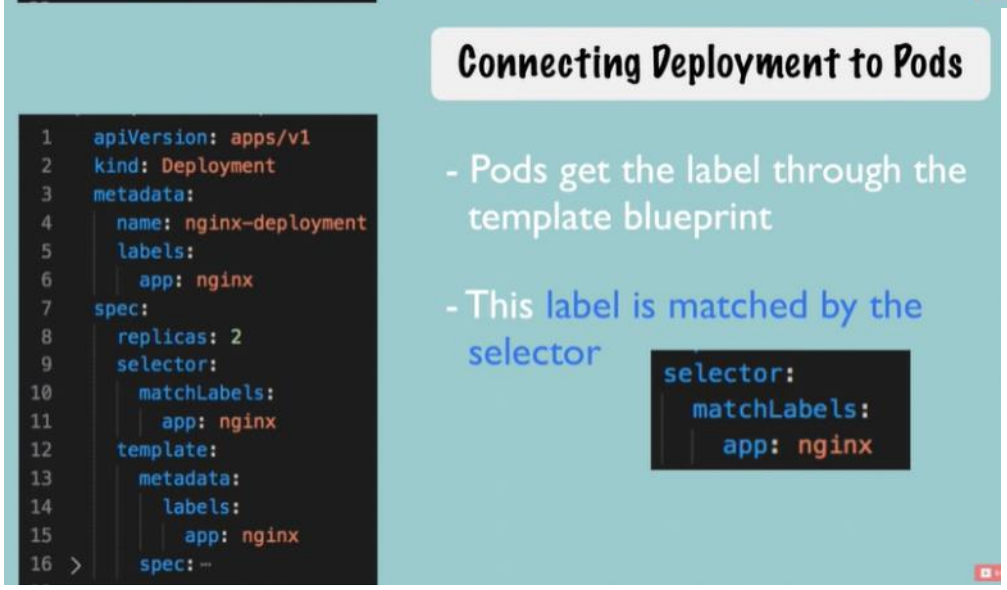


- As we know deployment manages the replica set and the pods. We declare the specifications and template of the pod inside the deployment component's specification as shared.
- Kind represents the type of the component
- Spec stores the specification of the deployment
- Template determines the template of the pod , to create the replica. And each pod has it's own meta data so we declare it under template's meta data.
- As we need to declare specification of the pod , we declare it under spec section under the template of the deployment component's specification.
- The shaded part in the image applies to the pod, the template is the blueprint for a pod. We define port , image name and following details in the template section.

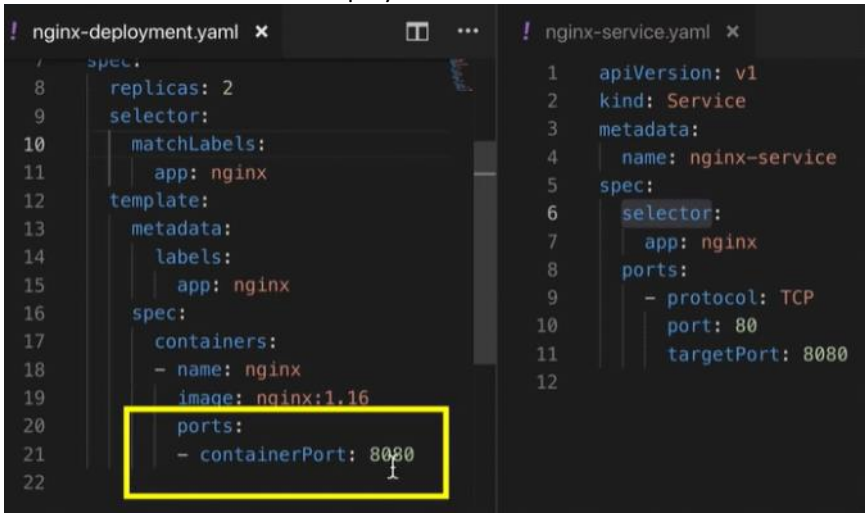


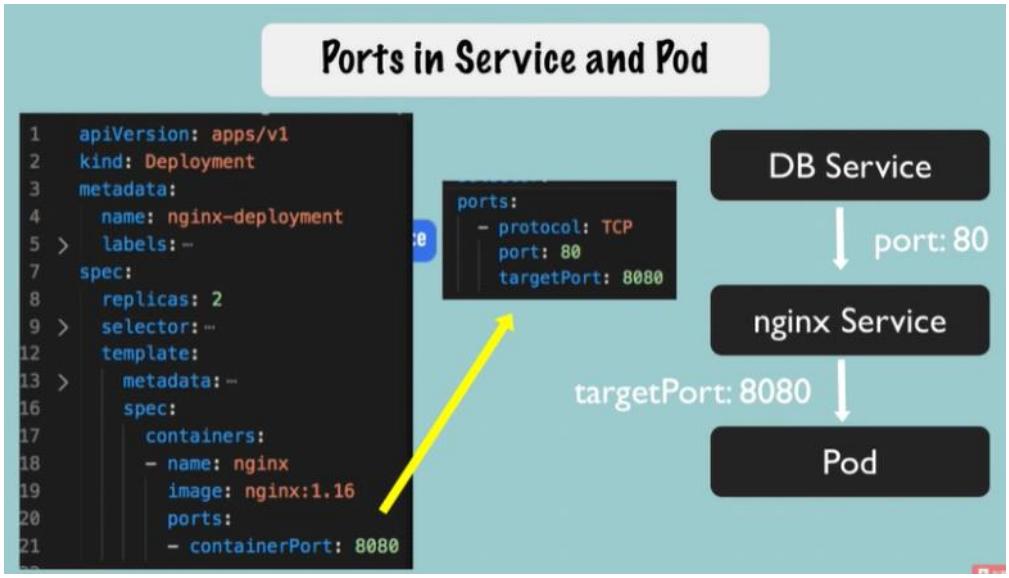
- Labels are declared under meta data section and selectors are declared at the specification section of the component.



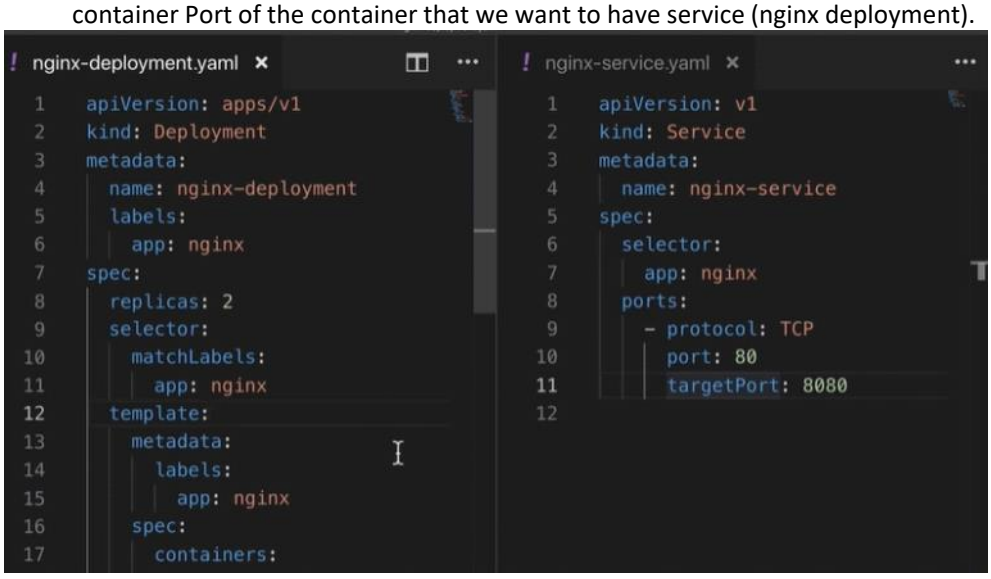


- Here in first image we are declaring label as nginx as the deployment component name, and under it we are creating a template for the pods where we declared labels as nginx, this establishes a link between the deployment component and the replica templates. And we are also declaring the selector for all the pods as the nginx in match labels.
- And if we want to connect the service with the deployment we use selector name as the key. Selector of the service maps to the label names of the deployment and establish a link between services and deployment.

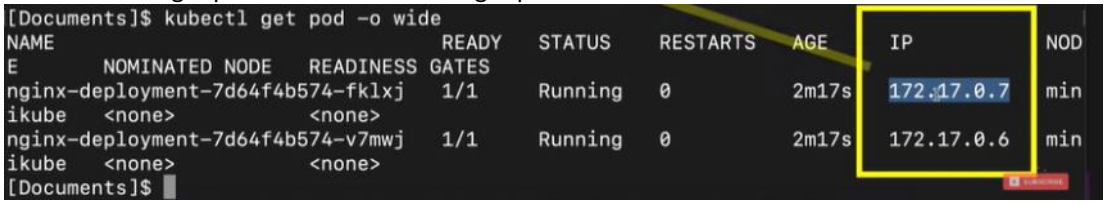




- Here we are having the port id's in both the service pod and deployment pod , for example DB service will call the nginx service that we declared and nginx runs at port:80 then that service must serve some containers and pods , and to locate them we are giving a target Port for the containers where they are located, and we must define with the same port address in the container Port of the container that we want to have service (nginx deployment).



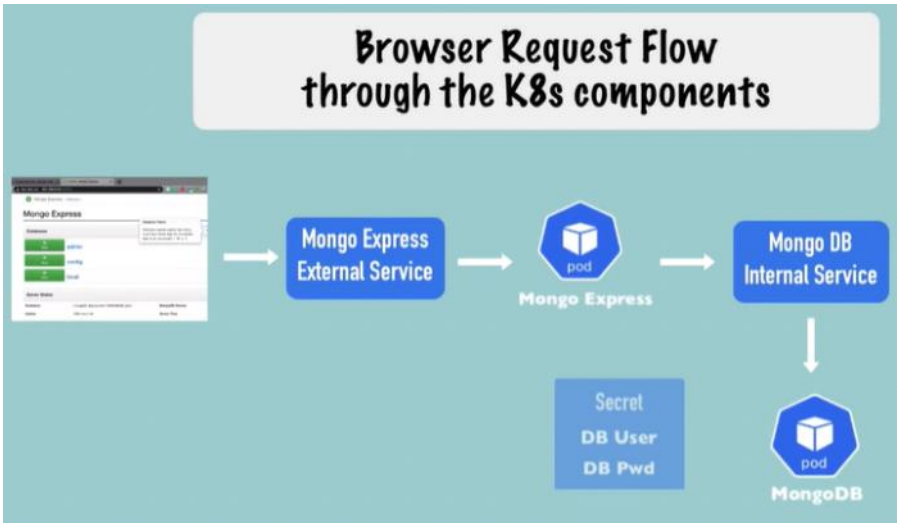
- "kubectl get pod -o wide" is used to get pod details more clearer



How to get status of the deployment and save it as yaml file.

- It get status from etcd.

- Deleting deployment configuration file using command line kubectl.



- To convert a value into base64 we use the following

```
[~]$ echo -n 'username' | base64
dXNlcm5hbWU=
```

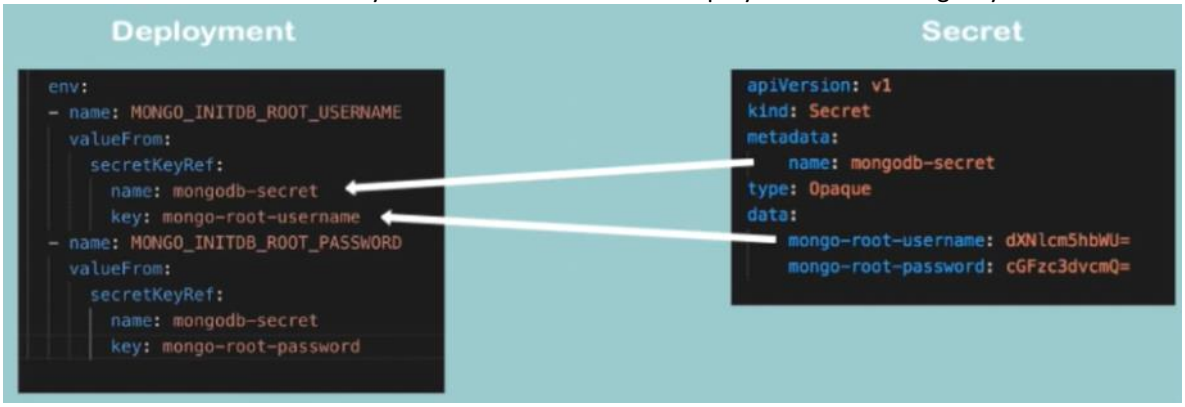
- While we want to store sensitive information in the secret yaml file we need to store sensitive information in base 64 , so first we need to convert the data into base 64 and store that base64 value inside the secret yaml file.
- Apply or create the component

```
[k8s-configuration]$ kubectl apply -f mongo-secret.yaml
secret/mongodb-secret created
```

- To checkout the component

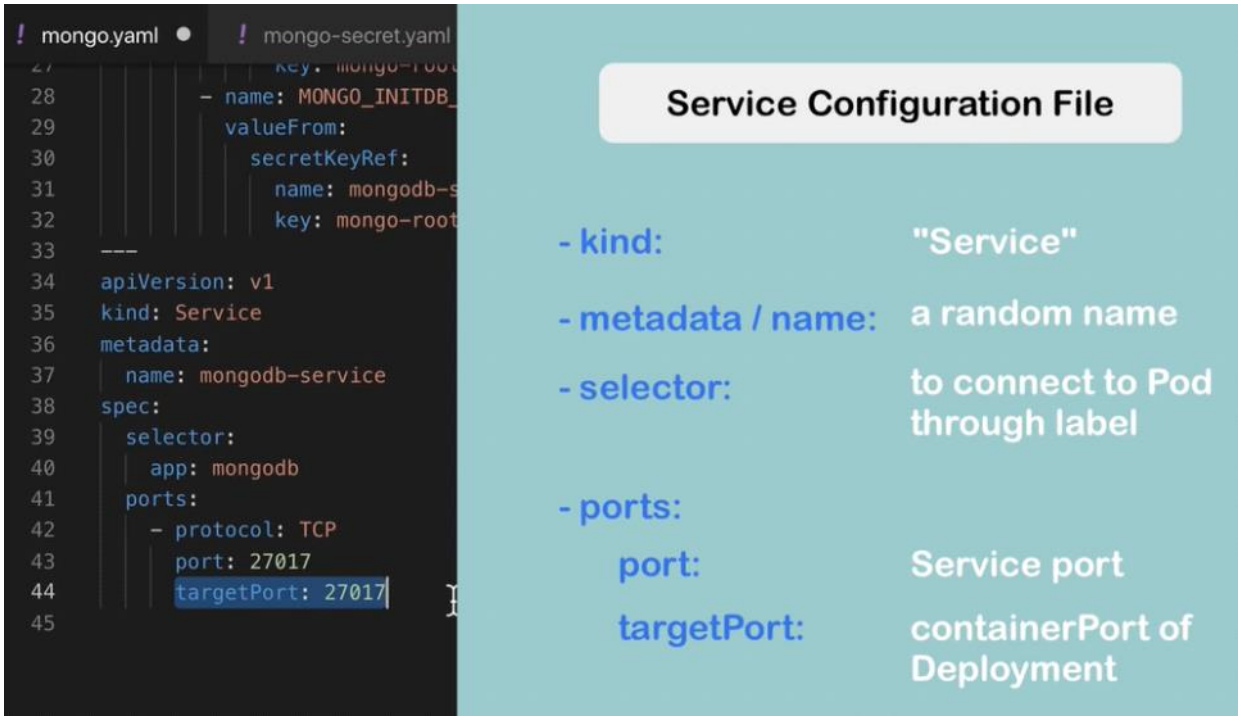
```
[k8s-configuration]$ kubectl get secret
NAME                  TYPE                                  DATA   AGE
default-token-4clzs   kubernetes.io/service-account-token   3      22m
mongodb-secret        Opaque                                2      68s
```

- We need to use the secret yaml file from secret to the deployment in following way

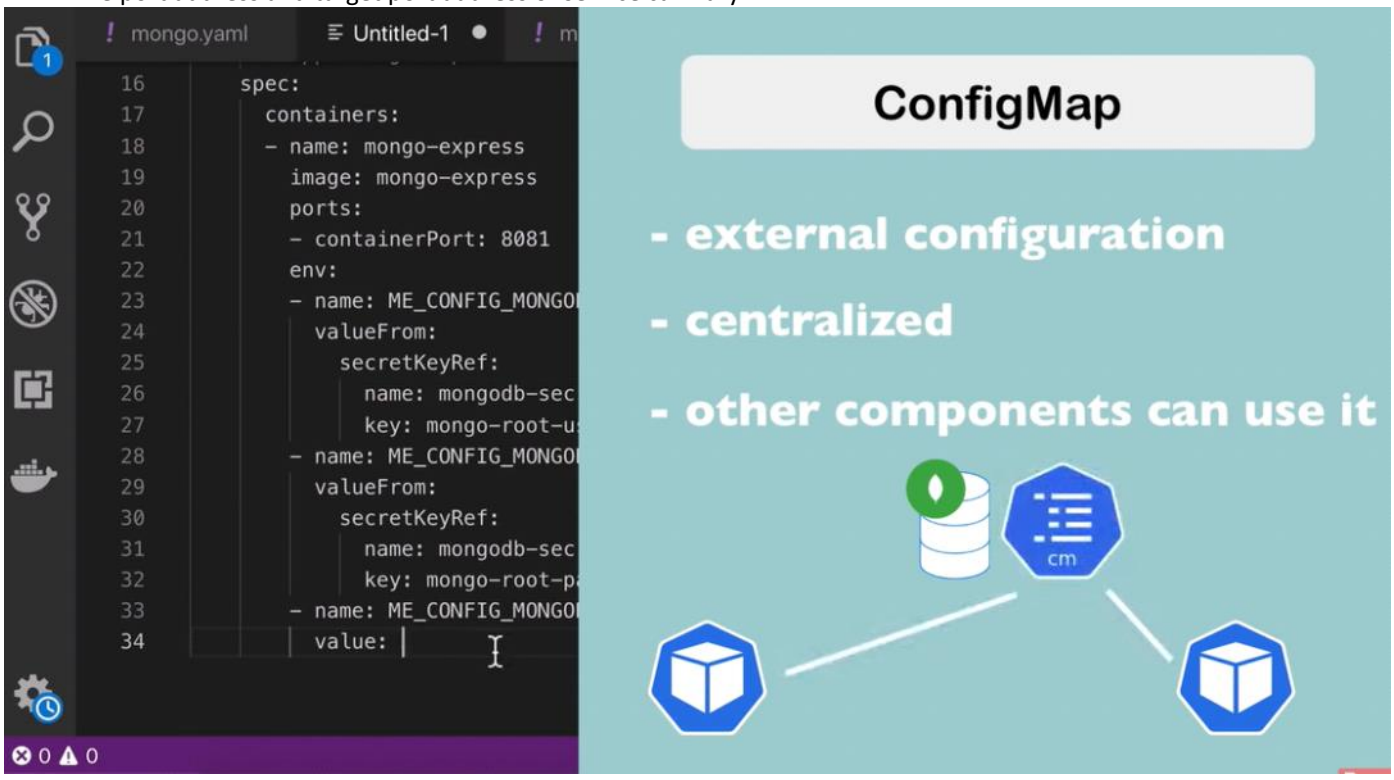


In the same yaml file we can declare multiple components , but they must be separated by three lines(dashes) .It denotes that , we are going to create a new component from there.

- As deployment and service are related they are kept in a single file.
- Service connect to the pod using the selector in name in the service to the label name in the deployment.

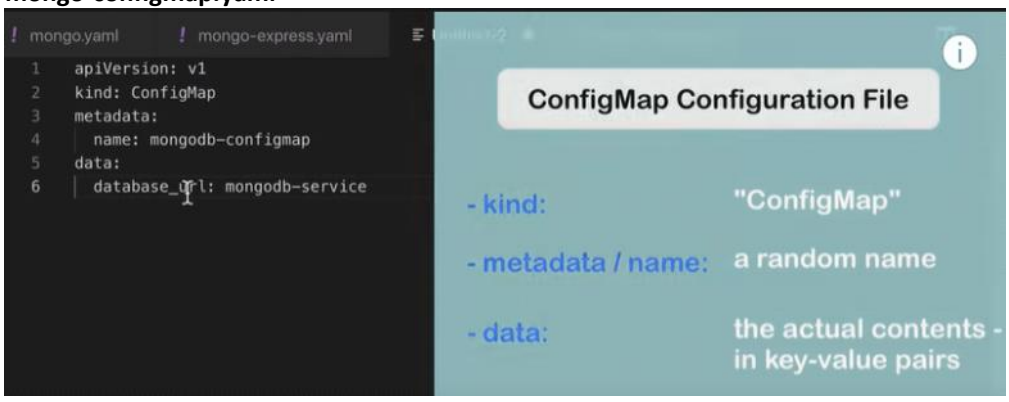


- The target port of the service must match with the container port of the deployment. As they are connected and service is actually servicing the pod.
- The port address and target port address of service can vary.

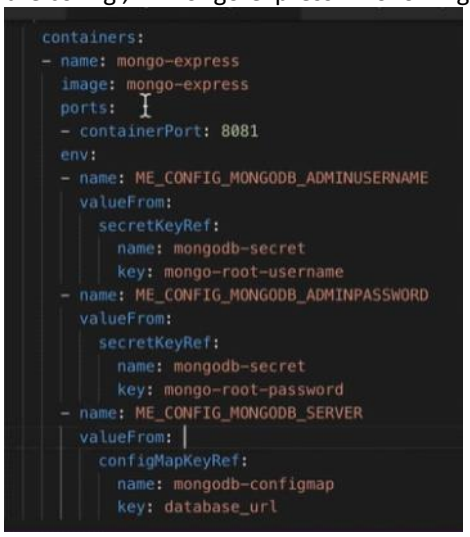


Now we want to define database service URL in the config map , we do it as following.

mongo-configmap.yaml

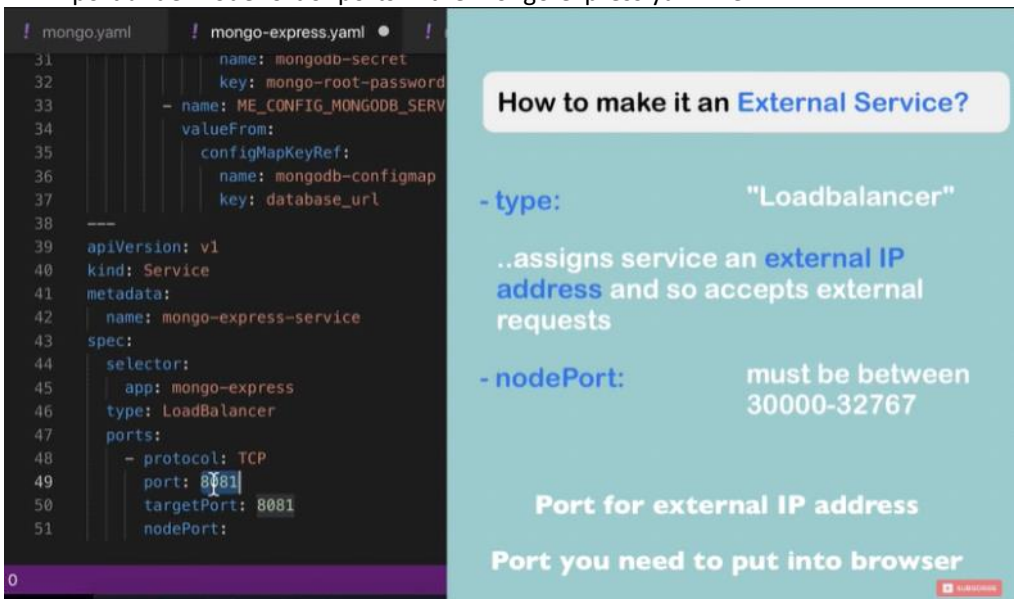


Here we want to refer the configmap to the main mongo-express.yaml , so we need to reference in this way. The above image is mongo-configmap and the below image is mongo-express.yaml. Refer the config , in mongo-express in following way.



```yaml
containers:
- name: mongo-express
  image: mongo-express
  ports:
  - containerPort: 8081
  env:
  - 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
```

- In the above image we have referenced three keys from other files, two keys from secret file and one key is from configmap. All these three are environmental variables that mongo needs to connect.
- Suppose if we want to access the application on the external port then we must declare that port under nodePort of ports in the mongo-express.yaml file.

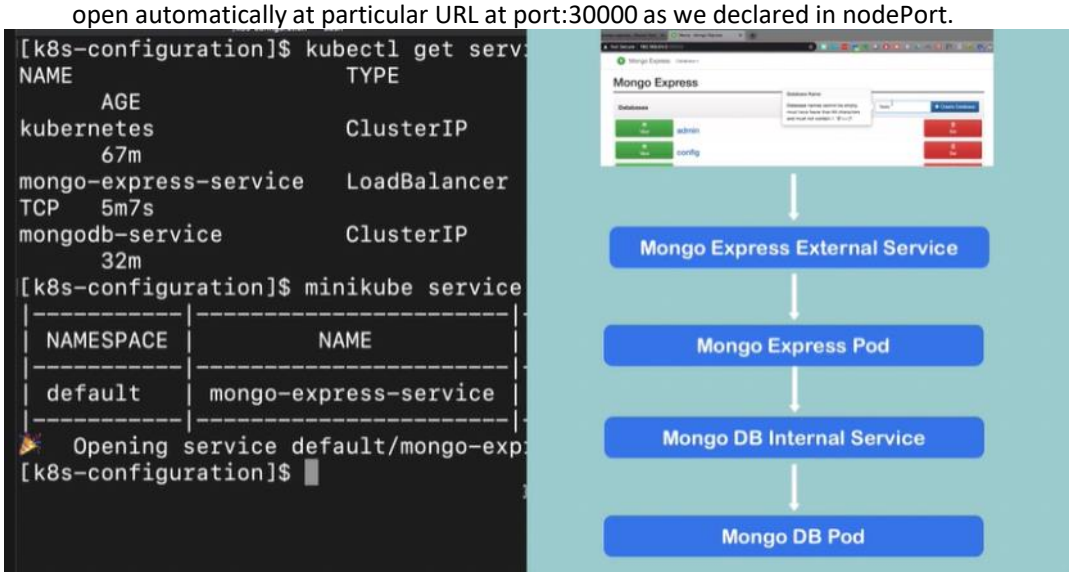


- Nodeport we declared as 30000
- If we declare the node port we can access that port through the browser. And creates the external service.
- Internal service or Cluster IP is default for the service
- If we want to declare external service we must define the type as the load balancer.

```
[k8s-configuration]$ kubectl get service
NAME                  TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)
      AGE
kubernetes            ClusterIP      10.96.0.1       <none>        443/TCP
      62m
mongo-express-service   LoadBalancer   10.96.178.16    <pending>     8081:30000/
TCP   6s
mongodb-service       ClusterIP      10.96.86.105    <none>        27017/TCP
      26m
```

```
[k8s-configuration]$ minikube service mongo-express-service
|-----------|-----------------------|-------------|---------------------------|
| NAMESPACE |         NAME          | TARGET PORT |            URL            |
|-----------|-----------------------|-------------|---------------------------|
| default   | mongo-express-service |             | http://192.168.64.5:30000 |
|-----------|-----------------------|-------------|---------------------------|
   Opening service default/mongo-express-service in default browser...
[k8s-configuration]$
```

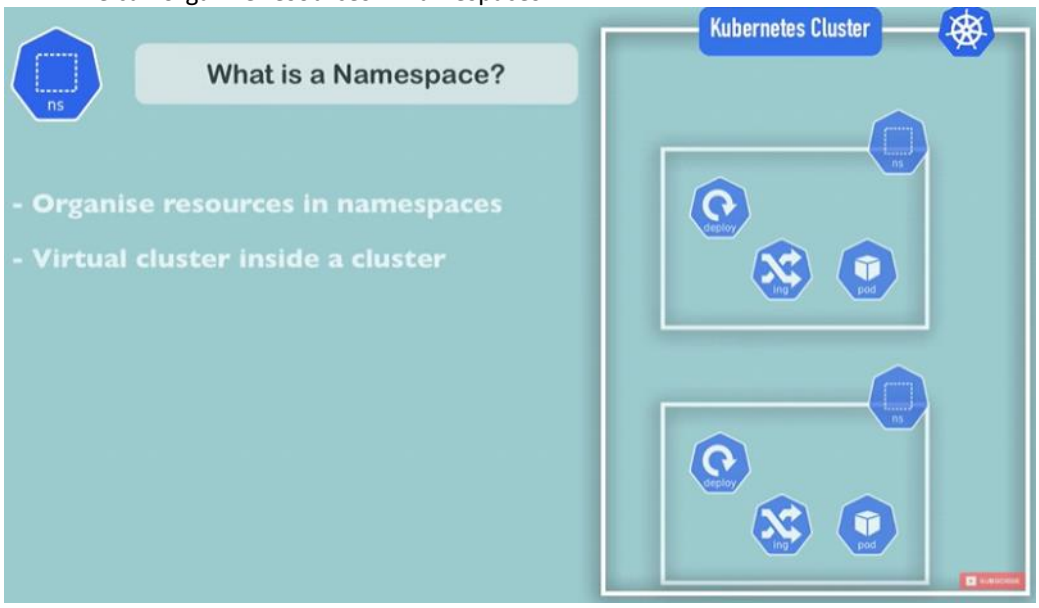
- Now after we execute this command the external service will be triggered and browser will open automatically at particular URL at port:30000 as we declared in nodePort.



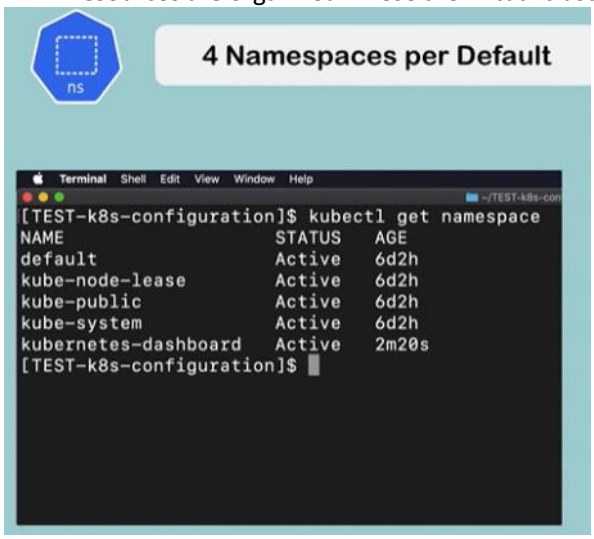
Using this process , we have created mongo Express in the minikube and executing it in locally on the server.

Kubernetes Namespaces

- We can organize resources in namespaces

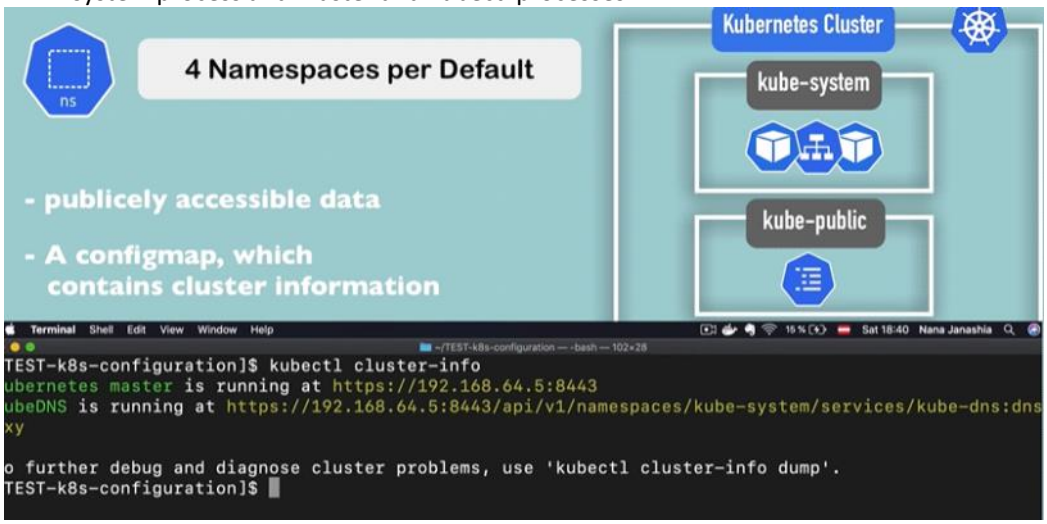


- Kubernetes cluster is cluster and under that we can have multiple virtual clusters where the resources are organized. These are virtual cluster inside a Kubernetes cluster.

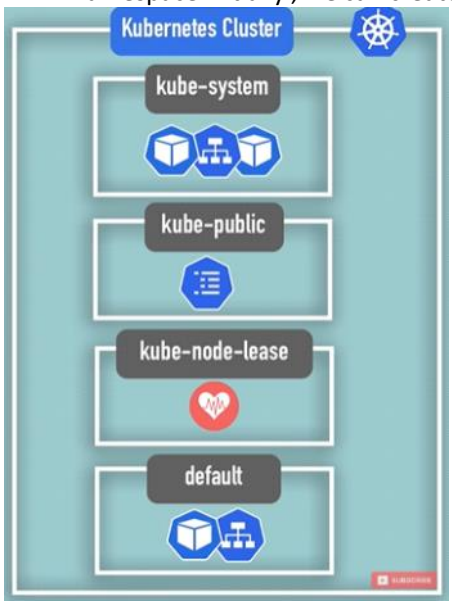


- Kubernetes automatically creates 4 namespaces as default when creating cluster , when we use following command we can find names of the namespaces.

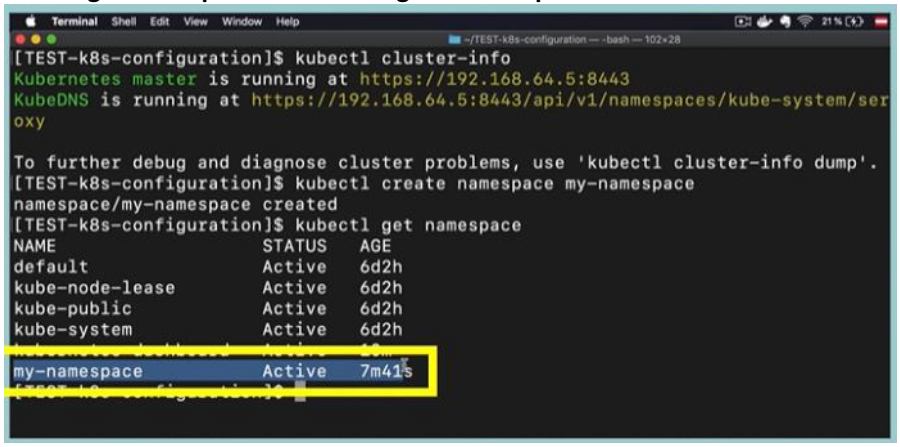
- Kubernetes-dashboard is being installed while we install minikube
- In kube-system namespace, we should not create or modify anything . Because they are system process and master and kubectl processes.



- In kube public namespace we contain publicly accessible data , for example a configmap , which contains cluster information.
- In kube-node-lease namespace we contain heartbeats of nodes and each node has associated lease object in namespace , it determines the availability of a node.
- In default namespace, the resources that we create are located here, if we don't create namespace initially , we can create new namespace through this default namespaces.



Creating a namespace and checking the namespace



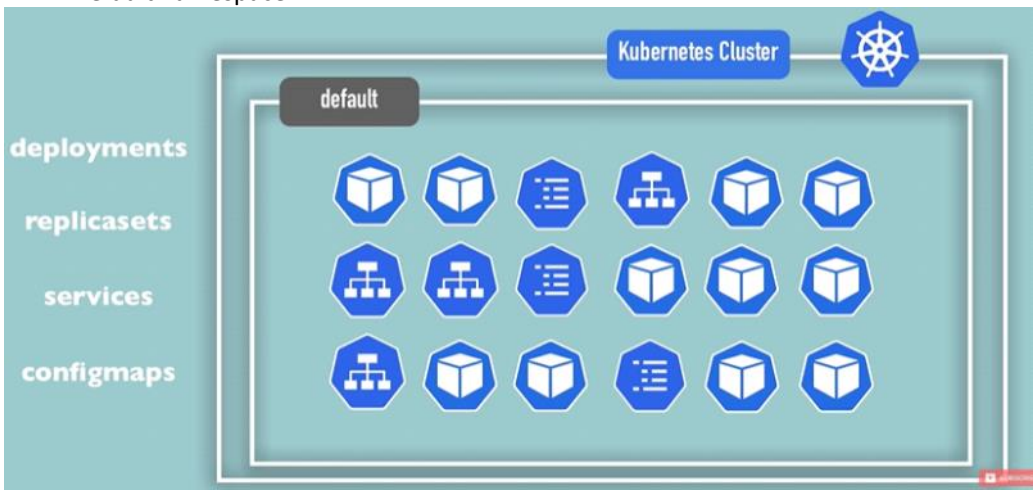
- We can create namespace through command line or else we can create it with configuration file.

What is the need of namespaces?

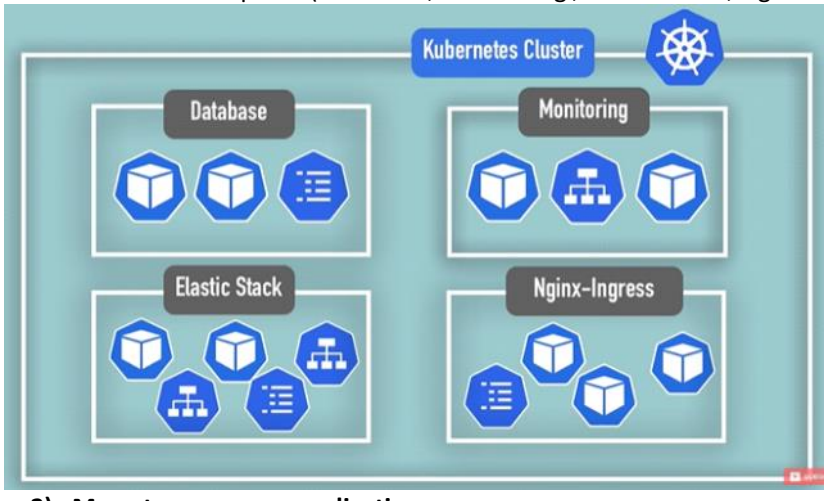
Use cases:

1)Resources group in Namespaces

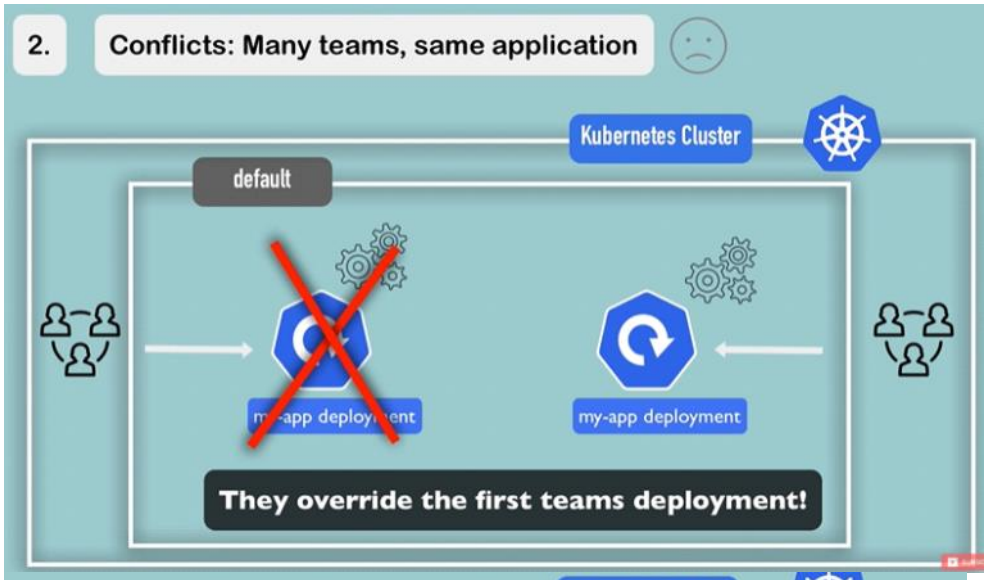
- Suppose when we are working on a complex application and we are having many pods , many config files , many secret files, more replicas and more services. And when we want to summarize the namespace to validate the functionality and resource utilization , it won't be good in default namespace.
- So we group the resources in the namespaces in such a way the type of the files , like database , monitoring , nginx-ingress.
- We should not use namespaces , if it is not a small project or if it is up to 10 users.
- But even if it is small project, we require namespaces sometimes, then we could use default namespaces.
- Default namespace

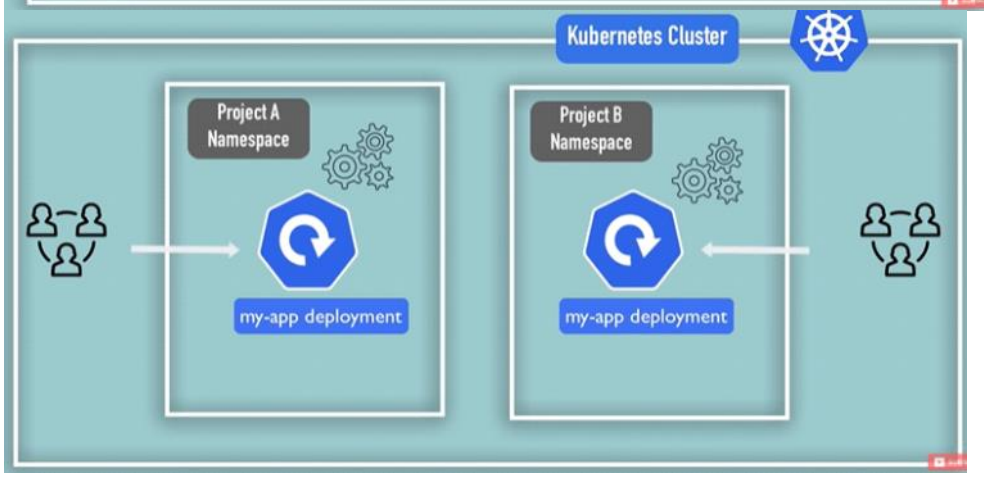


- As the default namespace overhead , we are creating multiple namespaces under Kubernetes cluster.
- Created namespaces (Database , Monitoring , Elastic Stack , Nginx-Ingress)



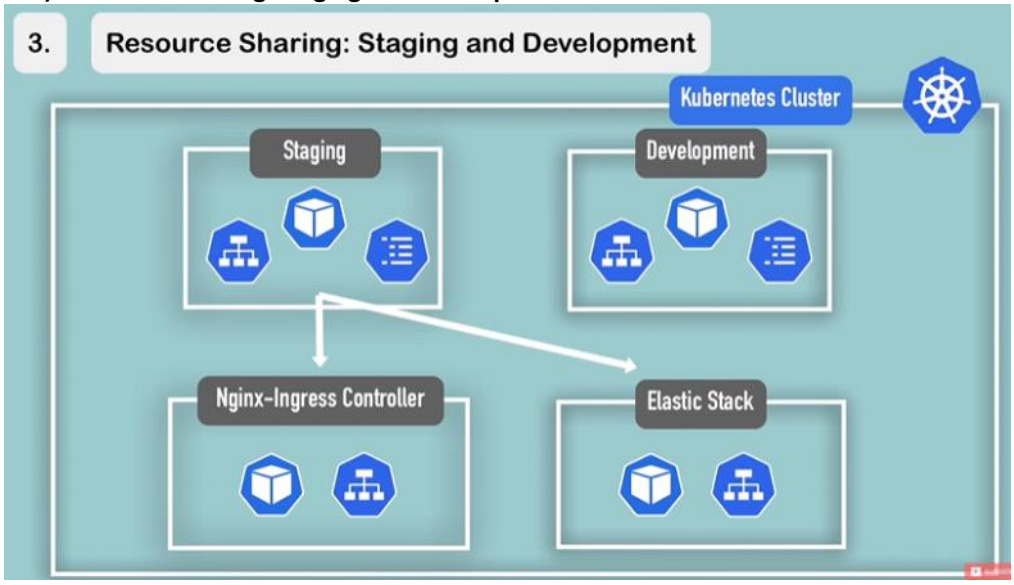
2) Many teams, same application



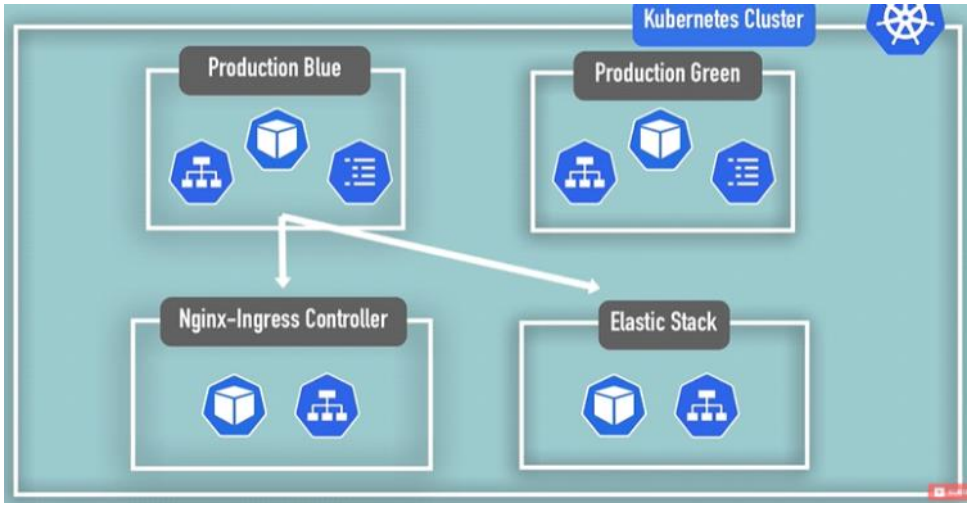


- Suppose if we are having multiple deployments with same name under in default namespace, which may be created by different teams , then the second team that is going to upload the deployment or if we use any automated process of deployment using Jenkins will overwrite the first teams deployment as they are under same name and same namespace.
- So we are creating multiple namespaces for each project, now there will be no issue even if the deployment name matches, one team project doesn't affect other team's project.

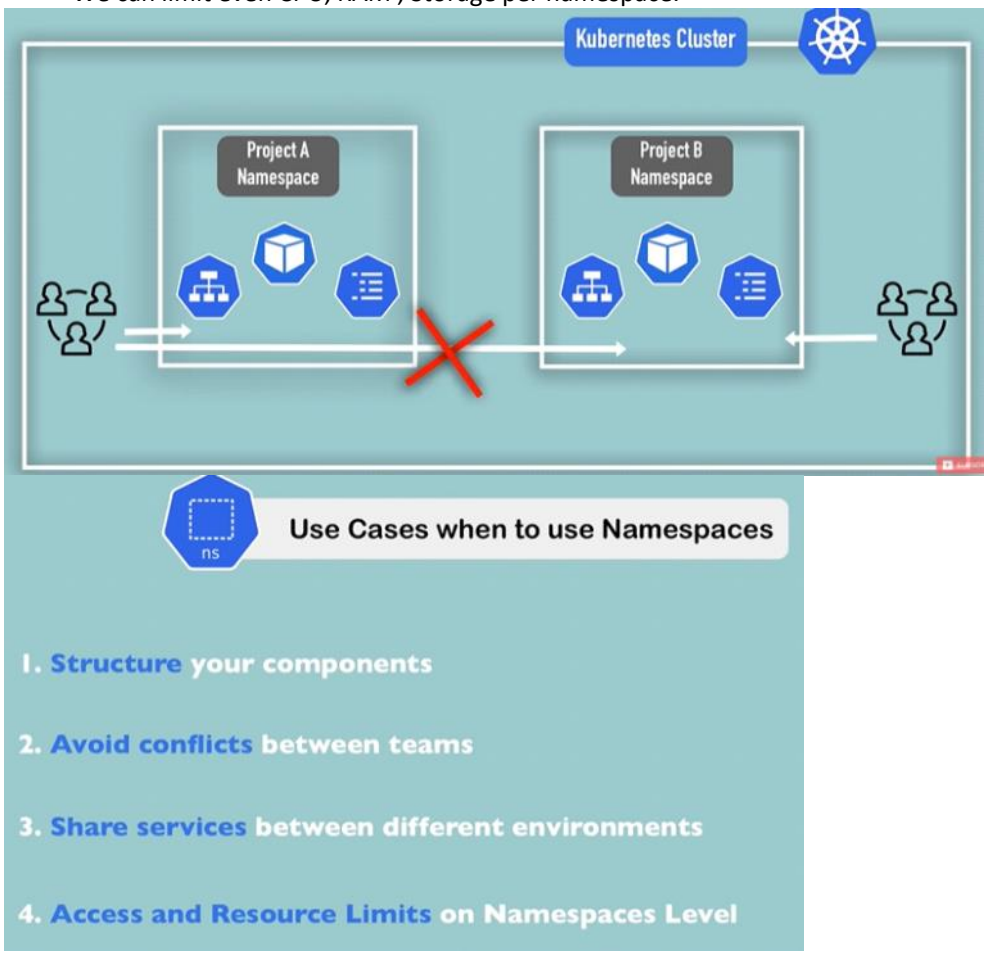
3) Resource sharing: staging and development



- Here we have multiple namespaces like staging and development under the same Kubernetes cluster , so that we can use those services by all other namespaces under the same Kubernetes cluster.

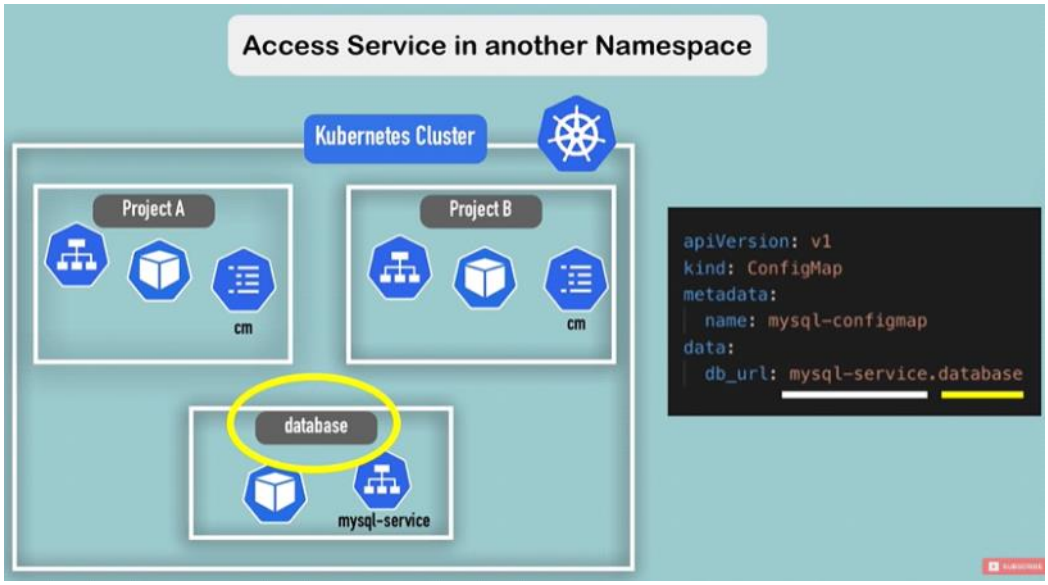


- Here production blue and green are one of latest version and one is the old version , keeping these two under Kubernetes cluster makes other namespaces to access them.
- By creating namespaces, we can limit access and resource limits on namespaces.
- We can limit even CPU, RAM , Storage per namespace.

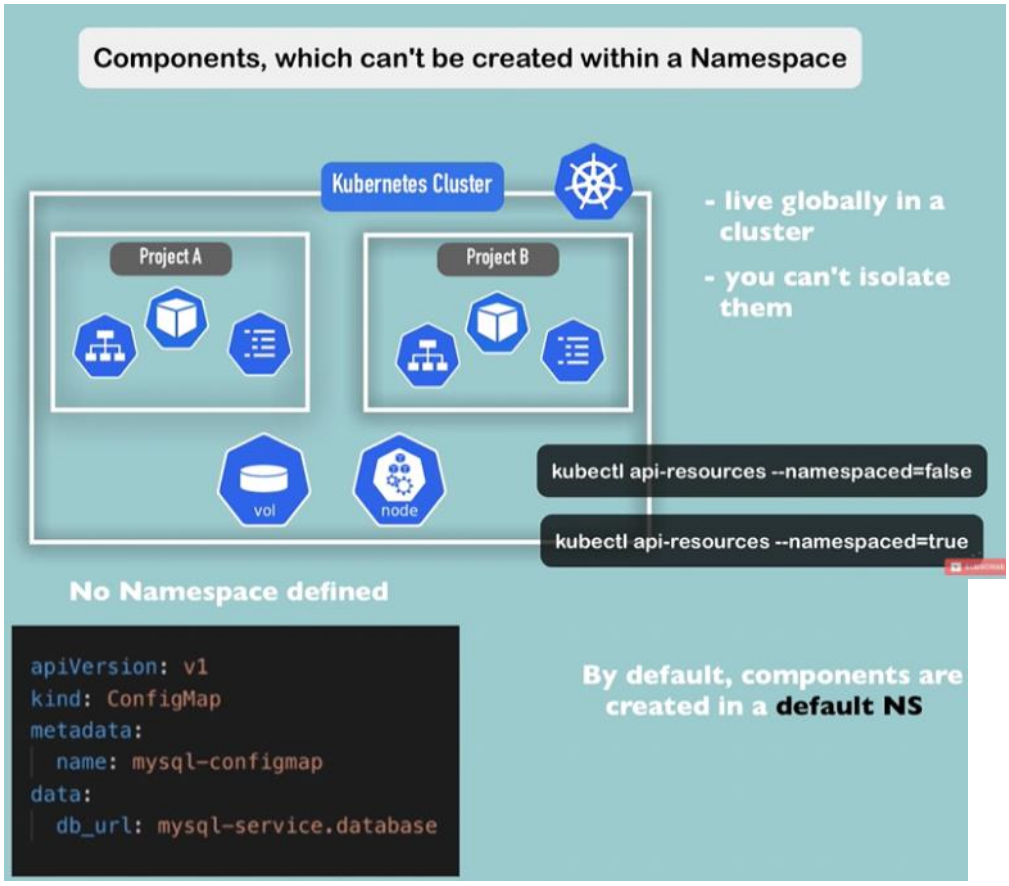


Restrictions:

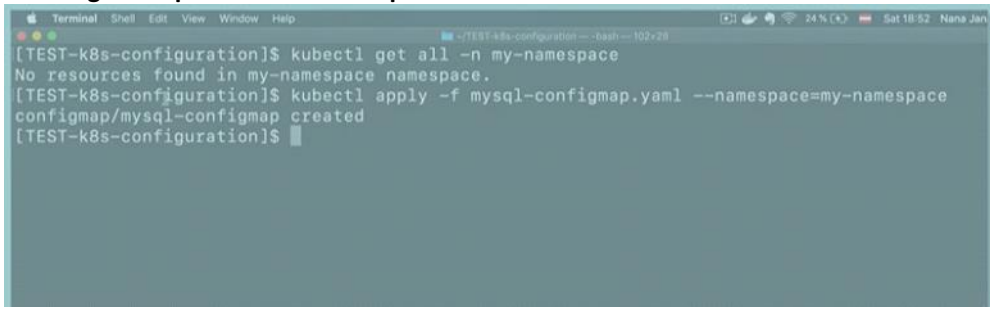
- Each Namespace must define it's own configmap, secret, it cannot be shared between groups.
- We can share service from namespace to another namespace.



- Some components which can be created within a namespace, because some components can't live locally under a namespace, they must define globally in the cluster. Examples are volume



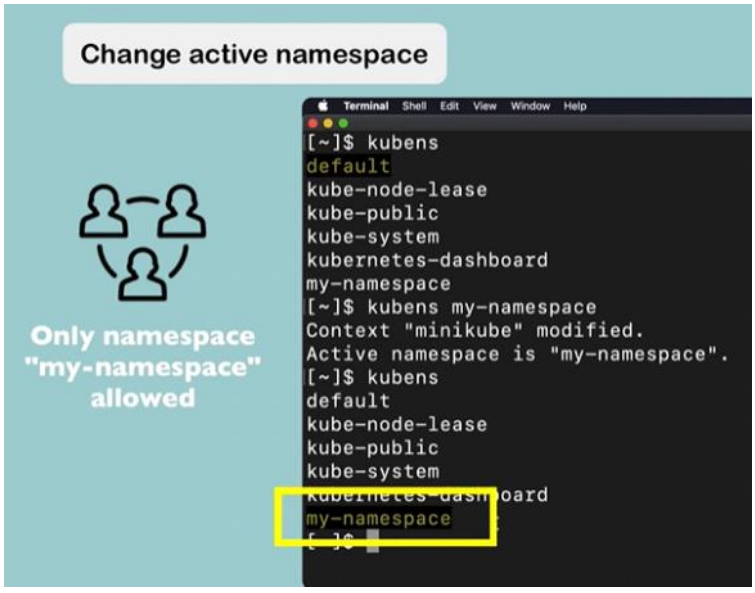
Creating a component in a Namespace



- We must externally define the namespace , if we want to create it under a namespace, or we can create it using yaml file.
- This is the configuration file with namespace declared in it, in above image we didn't declare any namespace.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-configmap
  namespace: my-namespace
data:
  db_url: mysql-service.database
```

- This is the preferable method as it is well documented and it supports automation as everything is declared inside the yaml files.
- We need to use --namespace=my-namespace repeatedly while we create any component under a namespace, so doing it repeatedly is not nice. For declaring a namespace before all the components we use kubens which is external to kubectl
- Kubectx is used to install kubens.



- Kubens fix the active namespace to what ever we want, so that we no need to define namespace name externally every time when we create any component.
- In the above image , we have default namespace as default then we are changing it to my-namespace , so now what every component we create will be under my-namespace namespace without defining externally through command line.

Kube Python Practical

Saturday, May 14, 2022 12:23 PM

To install Kubernetes in python , first we need to install the Kubernetes into python using pip package manager using "pip install kubernetes"

Dynamic Client

- The Kubernetes Python client framework supports the ability to watch a cluster for API object events including ADDED , MODIFIED , DELETED generated when an object is created, updated, and removed respectively
- Kubernetes stores your configuration under ~/. kube/config (default location). In it you will find context definition for every cluster you may have access to. A field called current-context defines your current context.

kubernetes.config.kube_config.load_kube_config(*config_file=None, context=None, client_configuration=None, persist_config=True*)

Loads authentication and cluster information from kube-config file and stores them in kubernetes.client.configuration.

Parameters:
- **config_file** – Name of the kube-config file.
- **context** – set the active context. If is set to None, current_context from config file will be used.
- **client_configuration** – The kubernetes.client.Configuration to set configs to.
- **persist_config** – If True, config file will be updated when changed (e.g GCP token refresh).

- Kubernetes provide an open model through the kube-apiserver, without splitting an internal and external interface, we can interact with the cluster and any other system to integrate both from the same application (Controller) and even use custom resources to describe our unique operations.
- So to interact with the Kubernetes , we need to communicate to the API server of Kubernetes , we could use an HTTP client for it, but this is a difficult task to accomplish. Kubernetes itself provides own clients to interact with the API server. One of that is typed client , this interface provides exclusive methods for each resource on Kubernetes (think of Pods, Deployments, Services, everything!) and operation (Create, Get, List, Watch, Update, Patch and Delete). It is preferable to use this client whenever required.
- But dynamic client provides some more features than typed client.
- To write applications using the Kubernetes REST API, you do not need to implement the API calls and request/response types yourself. You can use a client library for the programming language you are using.
- Client libraries often handle common tasks such as authentication for you. Most client libraries can discover and use the Kubernetes Service Account to authenticate if the API client is running inside the Kubernetes cluster, or can understand the kubeconfig file format to read the credentials and the API Server address.

Example: We are fetching a node through python through Kubernetes client module call api_client , with particular custom headers and printing the node's apiVersion and kind that we fetched through api_client and custom headers.

```
from kubernetes import config, dynamic
from kubernetes.client import api_client

def main():
    # Creating a dynamic client
    client = dynamic.DynamicClient(
        api_client.ApiClient(configuration=config.load_kube_config())
    )

    # fetching the node api
    api = client.resources.get(api_version="v1", kind="Node")

    # Creating a custom header
    params = {'header_params': {'Accept': 'application/json;as=PartialObjectMetadataList;v=v1;g=meta.k8s.io'}}

    resp = api.get(**params)

    # Printing the kind and apiVersion after passing new header params.
    print("%s\t\t\t%s" %("VERSION", "KIND"))
    print("%s\t\t%s" %(resp.apiVersion, resp.kind))


if __name__ == "__main__":
    main()
```

Example: Kubernetes Configmap Creation using dynamic client and list , patch , delete the configmap using python

```
from kubernetes import config, dynamic
from kubernetes.client import api_client


def main():
    # Creating a dynamic client
    client = dynamic.DynamicClient(
        api_client.ApiClient(configuration=config.load_kube_config())
    )

    # fetching the configmap api
    api = client.resources.get(api_version="v1", kind="ConfigMap")

    configmap_name = "test-configmap"

    configmap_manifest = {
        "kind": "ConfigMap",
        "apiVersion": "v1",
        "metadata": {
            "name": configmap_name,
            "labels": {
                "foo": "bar",
            },
        },
        "data": {
            "config.json": '{"command":"/usr/bin/mysqld_safe"}',
            "frontend.cnf": "[mysqld]\nbind-address = 10.0.0.3\n",
        },
    }

    # Creating configmap `test-configmap` in the `default` namespace

    configmap = api.create(body=configmap_manifest, namespace="default")
    print("\n[INFO] configmap `test-configmap` created\n")

    # Listing the configmaps in the `default` namespace

    configmap_list = api.get(
        name=configmap_name, namespace="default", label_selector="foo=bar"
    )

    print("NAME:\n%s\n" % (configmap_list.metadata.name))
    print("DATA:\n%s\n" % (configmap_list.data))

    # Updating the configmap's data, `config.json`

    configmap_manifest["data"]["config.json"] = "{}"

    configmap_patched = api.patch(
        name=configmap_name, namespace="default", body=configmap_manifest
    )

    print("\n[INFO] configmap `test-configmap` patched\n")
    print("NAME:\n%s\n" % (configmap_patched.metadata.name))
    print("DATA:\n%s\n" % (configmap_patched.data))

    # Deleting configmap `test-configmap` from the `default` namespace

    configmap_deleted = api.delete(name=configmap_name, body={}, namespace="default")
    print("\n[INFO] configmap `test-configmap` deleted\n")


if __name__ == "__main__":
    main()
```

Example: Creation of Kubernetes deployment using dynamic client and rolling restart of the deployment and listing and deletion of the deployment.

```
from kubernetes import config, dynamic
from kubernetes.client import api_client
import datetime
import pytz

def main():
    # Creating a dynamic client
    client = dynamic.DynamicClient(
        api_client.ApiClient(configuration=config.load_kube_config())
    )

    # fetching the deployment api
    api = client.resources.get(api_version="apps/v1", kind="Deployment")

    name = "nginx-deployment"
```

```
    deployment_manifest = {
        "apiVersion": "apps/v1",
        "kind": "Deployment",
        "metadata": {"labels": {"app": "nginx"}, "name": name},
        "spec": {
            "replicas": 3,
            "selector": {"matchLabels": {"app": "nginx"}},
            "template": {
                "metadata": {"labels": {"app": "nginx"}},
                "spec": {
                    "containers": [
                        {
                            "name": "nginx",
                            "image": "nginx:1.14.2",
                            "ports": [{"containerPort": 80}],
                        }
                    ]
                },
            },
        },
    }

    # Creating deployment `nginx-deployment` in the `default` namespace

    deployment = api.create(body=deployment_manifest, namespace="default")

    print("\n[INFO] deployment `nginx-deployment` created\n")

    # Listing deployment `nginx-deployment` in the `default` namespace

    deployment_created = api.get(name=name, namespace="default")
    print("%s\t%s\t\t\t%s\t%s" % ("NAMESPACE", "NAME", "REVISION", "RESTARTED-AT"))
    print(
        "%s\t\t%s\t%s\t\t%s\n"
        % (
            deployment_created.metadata.namespace,
            deployment_created.metadata.name,
            deployment_created.metadata.annotations,
            deployment_created.spec.template.metadata.annotations,
        )
    )

    # Patching the `spec.template.metadata` section to add `kubectl.kubernetes.io/restartedAt` annotation
    # In order to perform a rolling restart on the deployment `nginx-deployment`

    deployment_manifest["spec"]["template"]["metadata"] = {
        "annotations": {
            "kubectl.kubernetes.io/restartedAt": datetime.datetime.utcnow()
            .replace(tzinfo=pytz.UTC)
            .isoformat()
        }
    }

    deployment_patched = api.patch(
        body=deployment_manifest, name=name, namespace="default"
    )

    print("\n[INFO] deployment `nginx-deployment` restarted\n")
    print(
        "%s\t%s\t\t\t%s\t\t\t\t\t\t%s"
        % ("NAMESPACE", "NAME", "REVISION", "RESTARTED-AT")
    )
    print(
        "%s\t\t%s\t%s\t\t%s\n"
        % (
            deployment_patched.metadata.namespace,
            deployment_patched.metadata.name,
            deployment_patched.metadata.annotations,
            deployment_patched.spec.template.metadata.annotations,
        )
    )

    # Deleting deployment `nginx-deployment` from the `default` namespace

    deployment_deleted = api.delete(name=name, body={}, namespace="default")

    print("\n[INFO] deployment `nginx-deployment` deleted\n")


if __name__ == "__main__":
    main()
```

Example: creating a custom resource called namespace through python code and list , path and delete the namespace through python client.

```
from kubernetes import config, dynamic
from kubernetes import client as k8s_client
from kubernetes.dynamic.exceptions import ResourceNotFoundError
from kubernetes.client import api_client
import time

def list_ingressroute_for_all_namespaces(group, version, plural):
    custom_object_api = k8s_client.CustomObjectsApi()

    list_of_ingress_routes = custom_object_api.list_cluster_custom_object(
        group, version, plural
    )
    print(
        "%s\t\t\t%s\t\t\t%s\t\t%s\t\t\t\t%s"
        % ("NAME", "NAMESPACE", "FQDN", "TLS", "STRATEGY")
    )
    for item in list_of_ingress_routes["items"]:
        print(
            "%s\t%s\t\t%s\t%s\t%s"
            % (
                item["metadata"]["name"],
                item["metadata"]["namespace"],
                item["spec"]["virtualhost"]["fqdn"],
                item["spec"]["virtualhost"]["tls"],
                item["spec"]["strategy"]
            )
        )
def create_namespace(namespace_api, name):
    namespace_manifest = {
        "apiVersion": "v1",
        "kind": "Namespace",
        "metadata": {"name": name, "resourceversion": "v1"},
    }
    namespace_api.create(body=namespace_manifest)


def delete_namespace(namespace_api, name):
    namespace_api.delete(name=name)
def main():
    # Creating a dynamic client
    client = dynamic.DynamicClient(
        api_client.ApiClient(configuration=config.load_kube_config())
    )

    # fetching the custom resource definition (CRD) api
    crd_api = client.resources.get(
        api_version="apiextensions.k8s.io/v1", kind="CustomResourceDefinition"
    )

    namespace_api = client.resources.get(api_version="v1", kind="Namespace")

    # Creating a Namespaced CRD named "ingressroutes.apps.example.com"
    name = "ingressroutes.apps.example.com"
    crd_manifest = {
        "apiVersion": "apiextensions.k8s.io/v1",
        "kind": "CustomResourceDefinition",
        "metadata": {"name": name, "namespace": "default"},
        "spec": {
            "group": "apps.example.com",
            "versions": [
                {
                    "name": "v1",
                    "schema": {
                        "openAPIV3Schema": {
                            "properties": {
                                "spec": {
                                    "properties": {
                                        "strategy": {"type": "string"},
                                        "virtualhost": {
                                            "properties": {
                                                "fqdn": {"type": "string"},
                                                "tls": {
                                                    "properties": {
                                                        "secretName": {"type": "string"}
                                                    },
                                                    "type": "object",
                                                },
                                            },
                                            "type": "object",
                                        },
                                    },
                                    "type": "object",
                                }
                            },
                            "type": "object",
                        }
                    },
                    "served": True,
                    "storage": True,
                }
            ],
```

```
        "scope": "Namespaced",
        "names": {
            "plural": "ingressroutes",
            "listKind": "IngressRouteList",
            "singular": "ingressroute",
            "kind": "IngressRoute",
            "shortNames": ["ir"],
        },
    },
}

crd_creation_response = crd_api.create(crd_manifest)
print(
    "\n[INFO] custom resource definition `ingressroutes.apps.example.com` created\n"
)
print("%s\t\t%s" % ("SCOPE", "NAME"))
print(
    "%s\t%s\n"
    % (crd_creation_response.spec.scope, crd_creation_response.metadata.name)
)

# Fetching the "ingressroutes" CRD api

try:
    ingressroute_api = client.resources.get(
        api_version="apps.example.com/v1", kind="IngressRoute"
    )
except ResourceNotFoundError:
    # Need to wait a sec for the discovery layer to get updated
    time.sleep(2)

ingressroute_api = client.resources.get(
    api_version="apps.example.com/v1", kind="IngressRoute"
)

# Creating a custom resource (CR) `ingress-route-*`, using the above CRD `ingressroutes.apps.example.com`

namespace_first = "test-namespace-first"
namespace_second = "test-namespace-second"
 create_namespace(namespace_api, namespace_first)
 create_namespace(namespace_api, namespace_second)

 ingressroute_manifest_first = {
     "apiVersion": "apps.example.com/v1",
     "kind": "IngressRoute",
     "metadata": {
         "name": "ingress-route-first",
         "namespace": namespace_first,
     },
     "spec": {
         "virtualhost": {
             "fqdn": "www.google.com",
             "tls": {"secretName": "google-tls"},
         },
         "strategy": "RoundRobin",
     },
 }

 ingressroute_manifest_second = {
     "apiVersion": "apps.example.com/v1",
     "kind": "IngressRoute",
     "metadata": {
         "name": "ingress-route-second",
         "namespace": namespace_second,
     },
     "spec": {
         "virtualhost": {
             "fqdn": "www.yahoo.com",
             "tls": {"secretName": "yahoo-tls"},
         },
         "strategy": "RoundRobin",
     },
 }

 ingressroute_api.create(body=ingressroute_manifest_first, namespace=namespace_first)
 ingressroute_api.create(body=ingressroute_manifest_second, namespace=namespace_second)
 print("\n[INFO] custom resources `ingress-route-*` created\n")
# Listing the `ingress-route-*` custom resources

list_ingressroute_for_all_namespaces(
    group="apps.example.com", version="v1", plural="ingressroutes"
)

# Patching the ingressroutes custom resources

ingressroute_manifest_first["spec"]["strategy"] = "Random"
ingressroute_manifest_second["spec"]["strategy"] = "WeightedLeastRequest"

patch_ingressroute_first = ingressroute_api.patch(
    body=ingressroute_manifest_first, content_type="application/merge-patch+json"
)
patch_ingressroute_second = ingressroute_api.patch(
    body=ingressroute_manifest_second, content_type="application/merge-patch+json"
)

print(
    "\n[INFO] custom resources `ingress-route-*` patched to update the strategy\n"
)
list_ingressroute_for_all_namespaces(
    group="apps.example.com", version="v1", plural="ingressroutes"
)

# Deleting the ingressroutes custom resources

delete_ingressroute_first = ingressroute_api.delete(
    name="ingress-route-first", namespace=namespace_first
)
delete_ingressroute_second = ingressroute_api.delete(
    name="ingress-route-second", namespace=namespace_second
)

print("\n[INFO] custom resources `ingress-route-*` deleted")
```

```
    # Deleting the namespaces

    delete_namespace(namespace_api, namespace_first)
    time.sleep(4)
    delete_namespace(namespace_api, namespace_second)
    time.sleep(4)

    print("\n[INFO] test namespaces deleted")

    # Deleting the ingressroutes.apps.example.com custom resource definition

    crd_api.delete(name=name)
    print(
        "\n[INFO] custom resource definition `ingressroutes.apps.example.com` deleted"
    )


if __name__ == "__main__":
    main()
```

Example: How to list cluster nodes using dynamic client

```
from kubernetes import config, dynamic
from kubernetes.client import api_client


def main():
    # Creating a dynamic client
    client = dynamic.DynamicClient(
        api_client.ApiClient(configuration=config.load_kube_config())
    )

    # fetching the node api
    api = client.resources.get(api_version="v1", kind="Node")

    # Listing cluster nodes

    print("%s\t\t%s\t\t%s" % ("NAME", "STATUS", "VERSION"))
    for item in api.get().items:
        node = api.get(name=item.metadata.name)
        print(
            "%s\t%s\t\t%s\n"
            % (
                node.metadata.name,
                node.status.conditions[3]["type"],
                node.status.nodeInfo.kubeProxyVersion,
            )
        )
if __name__ == "__main__":
    main()
```

Example: Demonstrates the creation , listing & deletion of a namespaced replication controller using dynamic client.

```
from kubernetes import config, dynamic
from kubernetes.client import api_client


def main():
    # Creating a dynamic client
    client = dynamic.DynamicClient(
        api_client.ApiClient(configuration=config.load_kube_config())
    )

    # fetching the replication controller api
    api = client.resources.get(api_version="v1", kind="ReplicationController")

    name = "frontend-replication-controller"
```

```
    replication_controller_manifest = {
        "apiVersion": "v1",
        "kind": "ReplicationController",
        "metadata": {"labels": {"name": name}, "name": name},
        "spec": {
            "replicas": 2,
            "selector": {"name": name},
            "template": {
                "metadata": {"labels": {"name": name}},
                "spec": {
                    "containers": [
                        {
                            "image": "nginx",
                            "name": "nginx",
                            "ports": [{"containerPort": 80, "protocol": "TCP"}],
                        }
                    ]
                },
            },
        },
    }
    # Creating replication-controller `frontend-replication-controller` in the `default` namespace
    replication_controller = api.create(
        body=replication_controller_manifest, namespace="default"
    )

    print("\n[INFO] replication-controller `frontend-replication-controller` created\n")

    # Listing replication-controllers in the `default` namespace
    replication_controller_created = api.get(name=name, namespace="default")

    print("%s\t%s\t\t\t\t\t%s" % ("NAMESPACE", "NAME", "REPLICAS"))
    print(
        "%s\t\t%s\t\t%s\n"
        % (
            replication_controller_created.metadata.namespace,
            replication_controller_created.metadata.name,
            replication_controller_created.spec.replicas,
        )
    )
    # Deleting replication-controller `frontend-service` from the `default` namespace

    replication_controller_deleted = api.delete(name=name, body={}, namespace="default")

    print("[INFO] replication-controller `frontend-replication-controller` deleted\n")


if __name__ == "__main__":
    main()
```

Example: Creation of Kubernetes service using dynamic client and list , patch , delete that service

```
from kubernetes import config, dynamic
from kubernetes.client import api_client


def main():
    # Creating a dynamic client
    client = dynamic.DynamicClient(
        api_client.ApiClient(configuration=config.load_kube_config())
    )

    # fetching the service api
    api = client.resources.get(api_version="v1", kind="Service")

    name = "frontend-service"

    service_manifest = {
        "apiVersion": "v1",
        "kind": "Service",
        "metadata": {"labels": {"name": name}, "name": name, "resourceversion": "v1"},
        "spec": {
            "ports": [
                {"name": "port", "port": 80, "protocol": "TCP", "targetPort": 80}
            ],
            "selector": {"name": name},
        },
    }
    # Creating service `frontend-service` in the `default` namespace

    service = api.create(body=service_manifest, namespace="default")

    print("\n[INFO] service `frontend-service` created\n")

    # Listing service `frontend-service` in the `default` namespace
    service_created = api.get(name=name, namespace="default")

    print("%s\t%s" % ("NAMESPACE", "NAME"))
    print(
        "%s\t\t%s\n"
        % (service_created.metadata.namespace, service_created.metadata.name)
    )

    # Patching the `spec` section of the `frontend-service`

    service_manifest["spec"]["ports"] = [
        {"name": "new", "port": 8080, "protocol": "TCP", "targetPort": 8080}
    ]

    service_patched = api.patch(body=service_manifest, name=name, namespace="default")
    print("\n[INFO] service `frontend-service` patched\n")
    print("%s\t%s\t\t\t%s" % ("NAMESPACE", "NAME", "PORTS"))
    print(
        "%s\t\t%s\t%s\n"
        % (
            service_patched.metadata.namespace,
            service_patched.metadata.name,
            service_patched.spec.ports,
        )
    )

    # Deleting service `frontend-service` from the `default` namespace
    service_deleted = api.delete(name=name, body={}, namespace="default")

    print("\n[INFO] service `frontend-service` deleted\n")


if __name__ == "__main__":
    main()
```

Example: How to create a config map and use its data in pods

- ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. In this notebook we would learn how to create a ConfigMap and also how to use its data in Pods as seen in

How to create a ConfigMap and use its data in Pods

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. In this notebook we would learn how to create a ConfigMap and also how to use its data in Pods as seen in https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/

```
from kubernetes import client, config
from kubernetes.client.rest import ApiException
```

Load config from default location

```
config.load_kube_config()
```

Create API endpoint instance and API resource instances

```
api_instance = client.CoreV1Api()
cmap = client.V1ConfigMap()
```

Create key value pair data for the ConfigMap

```
cmap.metadata = client.V1ObjectMeta(name="special-config")
cmap.data = {}
cmap.data["special.how"] = "very"
cmap.data["special.type"] = "charm"
```

Create ConfigMap

```
api_instance.create_namespaced_config_map(namespace="default", body=cmap)
```

Create API endpoint instance and API resource instances for test Pod

```
pod = client.V1Pod()
spec = client.V1PodSpec()
pod.metadata = client.V1ObjectMeta(name="dapi-test-pod")
```

Initialize test Pod container

```
container = client.V1Container()
container.name = "test-container"
container.image = "gcr.io/google_containers/busybox"
container.command = ["/bin/sh", "-c", "env"]
```

Define Pod environment variables with data from ConfigMaps

```
container.env = [client.V1EnvVar(name="SPECIAL_LEVEL_KEY"), client.V1EnvVar(name="SPECIAL_TYPE_KEY")]
container.env[0].value_from = client.V1EnvVarSource()
container.env[0].value_from.config_map_key_ref = client.V1ConfigMapKeySelector(name="special-config", key="special.how")

container.env[1].value_from = client.V1EnvVarSource()
container.env[1].value_from.config_map_key_ref = client.V1ConfigMapKeySelector(name="special-config", key="special.type")

spec.restart_policy = "Never"
spec.containers = [container]
pod.spec = spec
```

Create Pod

```
api_instance.create_namespaced_pod(namespace="default",body=pod)
```

View ConfigMap data from Pod log

```
log = ""
try:
    log = api_instance.read_namespaced_pod_log(name="dapi-test-pod", namespace="default")
except ApiException as e:
    if str(e).find("ContainerCreating") != -1:
        print("Creating Pod container.\nRe-run current cell.")
    else:
        print("Exception when calling CoreV1Api->read_namespaced_pod_log: %s\n" % e)

for line in log.split("\n"):
    if line.startswith("SPECIAL"):
        print(line)
```

Delete ConfigMap

```
api_instance.delete_namespaced_config_map(name="special-config", namespace="default", body=cmap)
```

Delete Pod

```
api_instance.delete_namespaced_pod(name="dapi-test-pod", namespace="default", body=client.V1DeleteOptions())
```

Example: How to create a deployment with 3 replica sets and add , delete, patch operations on the deployment.

In this notebook, we show you how to create a Deployment with 3 ReplicaSets. These ReplicaSets are owned by the Deployment and are managed by the Deployment controller. We would also learn how to carry out RollingUpdate and RollBack to new and older versions of the deployment.

```
from kubernetes import client, config
```

Load config from default location

```
config.load_kube_config()
apps_api = client.AppsV1Api()
```

Create Deployment object

```
deployment = client.V1Deployment()
```

Fill required Deployment fields (apiVersion, kind, and metadata)

```
deployment.api_version = "apps/v1"
deployment.kind = "Deployment"
deployment.metadata = client.V1ObjectMeta(name="nginx-deployment")
```

A Deployment also needs a .spec section

```
spec = client.V1DeploymentSpec()
spec.replicas = 3
```

Add Pod template in .spec.template section

```
spec.template = client.V1PodTemplateSpec()
spec.template.metadata = client.V1ObjectMeta(labels={"app": "nginx"})
spec.template.spec = client.V1PodSpec()
```

Pod template container description

```
container = client.V1Container()
container.name="nginx"
container.image="nginx:1.7.9"
container. ports = [client.V1ContainerPort(container_port=80)]
```

```
spec.template.spec.containers = [container]
deployment.spec = spec
```

Create Deployment

```
apps_api.create_namespaced_deployment(namespace="default", body=deployment)
```

Update container image

```
deployment.spec.template.spec.containers[0].image = "nginx:1.9.1"
```

Apply update (RollingUpdate)

```
apps_api.replace_namespaced_deployment(name="nginx-deployment", namespace="default", body=deployment)
```

Delete Deployment

```
apps_api.delete_namespaced_deployment(name="nginx-deployment", namespace="default", body=client.V1DeleteOptions(propagation_policy="Foreground", gra
```

Example: How to start a pod

In this notebook, we show you how to create a single container Pod.

Start by importing the Kubernetes module

```
from kubernetes import client, config
```

If you are using a proxy, you can use the *client Configuration* to setup the host that the client should use. Otherwise read the kubeconfig file.

```
config.load_incluster_config()
```

Pods are a stable resource in the V1 API group. Instantiate a client for that API group endpoint.

```
v1=client.CoreV1Api()
```

```
pod=client.V1Pod()
spec=client.V1PodSpec()
pod.metadata=client.V1ObjectMeta(name="busybox")
```

In this example, we only start one container in the Pod. The container is an instnace of the *V1Container* class.

```
container=client.V1Container()
container.image="busybox"
container.args=["sleep", "3600"]
container.name="busybox"
```

The specification of the Pod is made of a single container in its list.

```
spec.containers = [container]
pod.spec = spec
```

Get existing list of Pods, before the creation of the new Pod.

```
ret = v1.list_namespaced_pod(namespace="default")
for i in ret.items:
    print("%s  %s  %s" % (i.status.pod_ip, i.metadata.namespace, i.metadata.name))
```

You are now ready to create the Pod.

```
v1.create_namespaced_pod(namespace="default",body=pod)
```

```
ret = v1.list_namespaced_pod(namespace="default")
for i in ret.items:
    print("%s  %s  %s" % (i.status.pod_ip, i.metadata.namespace, i.metadata.name))
```

Delete the Pod

You refer to the Pod by name, you need to add its namespace and pass some *delete* options.

```
v1.delete_namespaced_pod(name="busybox", namespace="default", body=client.V1DeleteOptions())
```

Example: How to create and use a secret

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. In this notebook, we would learn how to create a Secret and how to use Secrets as files from a Pod as seen in https://kubernetes.io/docs/concepts/configuration/secret/#using-secrets

```
from kubernetes import client, config
```

Load config from default location

```
config.load_kube_config()
client.configuration.assert_hostname = False
```

Create API endpoint instance and API resource instances

```
api_instance = client.CoreV1Api()
sec  = client.V1Secret()
```

Fill required Secret fields

```
sec.metadata = client.V1ObjectMeta(name="mysecret")
sec.type = "Opaque"
sec.data = {"username": "bXl1c2VybmFtZQ==", "password": "bXlwYXNzd29yZA=="}
```

Create Secret

```
api_instance.create_namespaced_secret(namespace="default", body=sec)
```

Create test Pod API resource instances

```
pod = client.V1Pod()
spec = client.V1PodSpec()
pod.metadata = client.V1ObjectMeta(name="mypod")
container = client.V1Container()
container.name = "mypod"
container.image = "redis"
```

Add volumeMount which would be used to hold secret

```
volume_mounts = [client.V1VolumeMount()]
volume_mounts[0].mount_path = "/data/redis"
volume_mounts[0].name = "foo"
container.volume_mounts = volume_mounts
```

Create volume required by secret

```
spec.volumes = [client.V1Volume(name="foo")]
spec.volumes[0].secret = client.V1SecretVolumeSource(secret_name="mysecret")
```

```
spec.containers = [container]
pod.spec = spec
```

Create the Pod

```
api_instance.create_namespaced_pod(namespace="default",body=pod)
```

View secret being used within the pod

Wait for atleast 10 seconds to ensure pod is running before executing this section.

```
user = api_instance.connect_get_namespaced_pod_exec(name="mypod", namespace="default", command=[ "/bin/sh", "-c", "cat /data/redis/username" ], stderr
print(user)
passwd = api_instance.connect_get_namespaced_pod_exec(name="mypod", namespace="default", command=[ "/bin/sh", "-c", "cat /data/redis/password" ], stde
print(passwd)
```

Delete Pod

```
api_instance.delete_namespaced_pod(name="mypod", namespace="default", body=client.V1DeleteOptions())
```

Delete Secret

```
api_instance.delete_namespaced_secret(name="mysecret", namespace="default", body=sec)
```

Example: How to create a service

In this notebook, we show you how to create a Service. A service is a key Kubernetes API resource. It defines a networking abstraction to route traffic to a particular set of Pods using a label selection.

```
from kubernetes import client, config
```

Load config from default location

```
config.load_kube_config()
```

Create API endpoint instance

```
api_instance = client.CoreV1Api()
```

Create API resource instances

```
service = client.V1Service()
```

Fill required Service fields (apiVersion, kind, and metadata)

```
service.api_version = "v1"
service.kind = "Service"
service.metadata = client.V1ObjectMeta(name="my-service")
```

Provide Service .spec description

Set Service object named **my-service** to target TCP port **9376** on any Pod with the **'app'='MyApp'** label. The label selection allows Kubernetes to determine which Pod should receive traffic when the service is used.

```
spec = client.V1ServiceSpec()
spec.selector = {"app": "MyApp"}
spec.ports = [client.V1ServicePort(protocol="TCP", port=80, target_port=9376)]
service.spec = spec
```

Create Service

```
api_instance.create_namespaced_service(namespace="default", body=service)
```

Delete Service

```
api_instance.delete_namespaced_service(name="my-service", namespace="default")
```

Managing kubernetes objects using common resource operations with the python client

Some of these operations include;

- `create_xxxx` : create a resource object. Ex `create_namespaced_pod` and `create_namespaced_deployment` , for creation of pods and deployments respectively. This performs operations similar to `kubectl create` .
- `read_xxxx` : read the specified resource object. Ex `read_namespaced_pod` and `read_namespaced_deployment` , to read pods and deployments respectively. This performs operations similar to `kubectl describe` .
- `list_xxxx` : retrieve all resource objects of a specific type. Ex `list_namespaced_pod` and `list_namespaced_deployment` , to list pods and deployments respectively. This performs operations similar to `kubectl get` .
- `patch_xxxx` : apply a change to a specific field. Ex `patch_namespaced_pod` and `patch_namespaced_deployment` , to update pods and deployments respectively. This performs operations similar to `kubectl patch` , `kubectl label` , `kubectl annotate` etc.
- `replace_xxxx` : replacing a resource object will update the resource by replacing the existing spec with the provided one. Ex `replace_namespaced_pod` and `replace_namespaced_deployment` , to update pods and deployments respectively, by creating new replacements of the entire object. This performs operations similar to `kubectl rolling-update` , `kubectl apply` and `kubectl replace` .
- `delete_xxxx` : delete a resource. This performs operations similar to `kubectl delete` .

```
from kubernetes import client, config
```

Load config from default location.

```
config.load_kube_config()
```

Create API endpoint instance as well as API resource instances (body and specification).

```
api_instance = client.AppsV1Api()
dep = client.V1Deployment()
spec = client.V1DeploymentSpec()
```

Fill required object fields (apiVersion, kind, metadata and spec).

```
name = "my-busybox"
dep.metadata = client.V1ObjectMeta(name=name)

spec.template = client.V1PodTemplateSpec()
spec.template.metadata =  client.V1ObjectMeta(name="busybox")
spec.template.metadata.labels = {"app":"busybox"}
spec.template.spec = client.V1PodSpec()
dep.spec = spec

container = client.V1Container()
container.image = "busybox:1.26.1"
container.args = ["sleep", "3600"]
container.name = name
spec.template.spec.containers = [container]
```

Create Deployment using create_xxxx command for Deployments.

```
api_instance.create_namespaced_deployment(namespace="default",body=dep)
```

Use list_xxxx command for Deployment, to list Deployments.

```
deps = api_instance.list_namespaced_deployment(namespace="default")
for item in deps.items:
    print("%s  %s" % (item.metadata.namespace, item.metadata.name))
```

Use read_xxxx command for Deployment, to display the detailed state of the created Deployment resource.

```
api_instance.read_namespaced_deployment(namespace="default",name=name)
```

Use patch_xxxx command for Deployment, to make specific update to the Deployment.

```
dep.metadata.labels = {"key": "value"}
api_instance.patch_namespaced_deployment(name=name, namespace="default", body=dep)
```

Use replace_xxxx command for Deployment, to update Deployment with a completely new version of the object.

```
dep.spec.template.spec.containers[0].image = "busybox:1.26.2"
api_instance.replace_namespaced_deployment(name=name, namespace="default", body=dep)
```

Use delete_xxxx command for Deployment, to delete created Deployment.

```
api_instance.delete_namespaced_deployment(name=name, namespace="default", body=client.V1DeleteOptions(propagation_policy="Foreground", grace_period_se
```

Logging Module in python

Thursday, May 19, 2022 8:16 PM