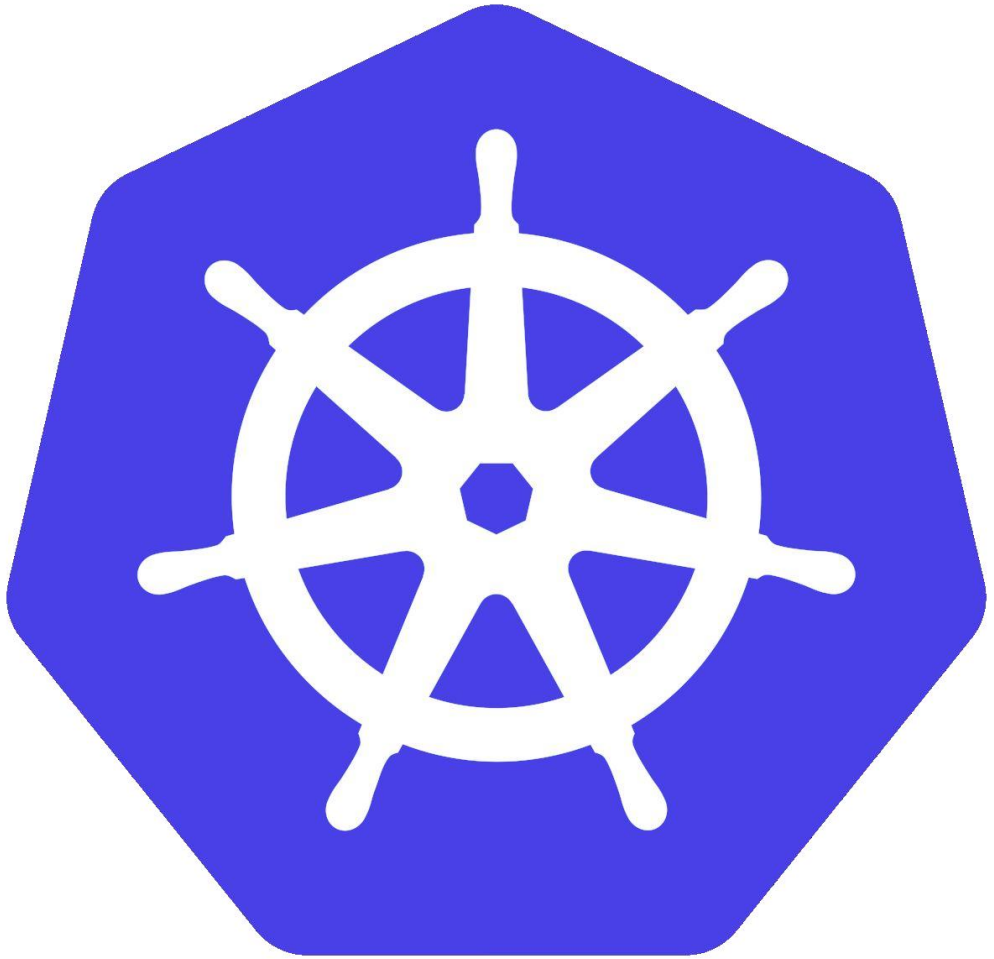


# Kubernetes

**Repo that contains all the work,**
https://github.com/james-jasvin/K8s-Repo

Refer for better understanding:
Techworld with Nana

# K8s YAML Configuration file

- There’s the **apiVersion** and **kind** attributes in the config, kind specifies **what K8s object** you want to create here and apiVersion is used to indicate which **version of the Kubernetes API** you're using to create this object.
- Each K8s configuration file has 3 parts,

1. Metadata (metadata)
2. Specification (spec)
3. Status

**1. Metadata (metadata)**

- Includes name of the object.(Includes other stuff as well but the most important metadata element is the name.)

**2. Specification (spec)**

- **The spec defines the desired state of the K8s object.**
- Includes many specifications related to the pods to be created such as the number of replicas to be maintained, the image to be used by the containers created and so on.
- The attributes in the spec are specific to the kind specified, so depending on whether you are talking about a deployment or a service, etc. the attributes inside spec will change.
- K8s compares the desired state (spec) against the current status of the cluster to determine whether everything is up-to-the-mark or not.This is the basis for the self-healing feature of K8s

**3. Status**

- Automatically generated and maintained by Kubernetes.

# Deployment Configuration

Done via YAML, every configuration
in K8s is specified via YAML.

**kubectl apply**
kubectl apply -f <file-name>.yaml
So you can keep updating YAML and applying it via kubectl to see the new configuration being used to replace the existing Deployment.

```yaml
apiVersion: apps/v1
# The apiVersion field is used to specify the version of the Kubernetes API that should be used
kind: Deployment
# Type of component you are deploying
metadata:
  name: nginx-deployment
  # Name of the Deployment created
  labels:
  # Selector Labels: Used to match the Deployment with its ReplicaSet
  # These labels are key-value pairs that can include any key and any value
    app: nginx
spec:
# Specification for the Deployment
  replicas: 2
  # Number of Replicas to create
  selector:
    # To link the Deployment to its Pods, you specify which labels a particular Pod
    # must have to be linked with this Deployment
    matchLabels:
      app: nginx
  template:
  # Template for the Pod
    metadata:
      labels:
      # Must be the same as the Deployment's labels
      # This is how Pods are linked with the Deployment
        app: nginx
    spec:
    # Specification for the Pod
      containers:
      - name: nginx
        # Name of the container, custom-name
        image: nginx:1.16
        # Which image this container uses
        ports:
        - containerPort: 80
        # On which port this container runs
```

## template

- For a Deployment object, it is a sub-attribute of the spec attribute and it describes the pods that will be created.
- A template has its own metadata and spec section. So a template is like the configuration file for a Pod which is nested within the configuration file for a Deployment

Deployment

**Connecting Deployment to Pods**

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

- **selector**: As metadata contains labels and spec contains selectors.
- In a label, you assign any number of key-value pairs to that K8s component, like in this case, weare assigning the key-value pair “app: nginx” to this Deployment component.
- Now this label is stuck to that component in which it is specified.
- We put the same label in the template metadata as well so that the created pods also have the same key-value pair and then specify the same key-value pair in the selector attribute of the spec under the matchLabels sub-attribute. This is done so that K8s is able to tell the Deployment which Pods belong.

# ports

- Another thing that must be specified in a spec for both Service & Pod are the ports (so it's the template's spec, not the Deployment's spec).

For Service,

- **Port** where the Service is accessible at, so if any other Service wants to access this Service, it should send the request at this port. For the Service to function properly it should know where its Pods are and on what port these Pods are running, so that it forward the requests accordingly.
- **targetPort**: On which port are the Pods that the Service is attached to running at

For Deployment,

- **containerPort**: On which port are the Pods running the application at? Clearly, this must be the same as the targetPort.

# K8s Objects

Deployment

Service 

ConfigMap

Secret

Ingress

Volume

StatefulSet

# Services

- Recall: Services are used by Pods to communicate with each other and the outside world by mapping a Deployment (and its Pods) to an IP Address that's not tied to the lifecycle of the Pods themselves.
- Note: A Service is an abstraction over an IP address, not a process.
- Depending on the type of communication there are 4 types of Services,
    1. **ClusterIP (default)**
    2. **Headless**
    3. **LoadBalancer**
    4. **NodePort**

# ClusterIP Service

- ClusterIP services are typically used for **internal communication** within a Kubernetes cluster, rather than for exposing services to the outside world.
- For example, if you have a set of pods running a database, you could create a ClusterIP service to provide a stable IP address for those pods. Then, other pods within the cluster that need to access the database can do so by using the IP address of the ClusterIP service.
- Additionally, ClusterIP services can be load-balanced across multiple pods, allowing for efficient and scalable communication within the cluster.

```yaml
apiVersion: v1
kind: Service
# Type of K8s Object created = Service
metadata:
  name: nginx-service
  # Name of the Service created
spec:
  # No type specified in spec, so service will be ClusterIP
  selector:
    app: nginx
    # Selector labels for the Service, must match with the Deployment that it'll connect to
  ports:
  - protocol: TCP # Default value TCP
    port: 80
    # Port on which this Service will be accessible
    targetPort: 80
    # Port where this Service should redirect to
    # Should match with the Port where the Pods are running
```

# Headless Service

What if…

- Client wants to talk with a specific Pod directly?
- Pods want to talk to a specific Pod?

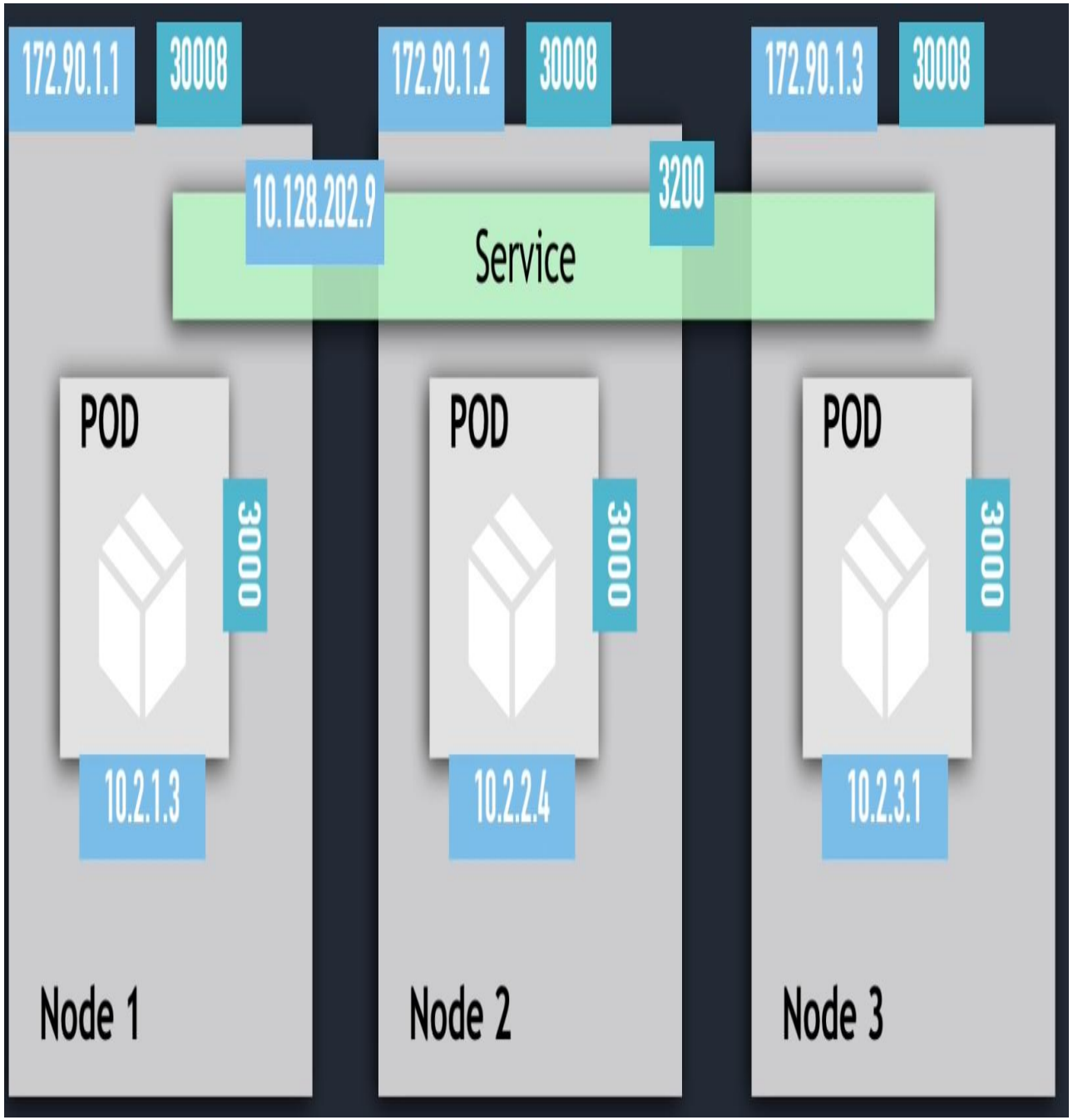
*ClusterIP Service doesn't make sense here because it'll randomly select a Pod on the basis of load balancing.*

- The only thing that you need to do is to specify the clusterIP attribute as None in a normal ClusterIP Service config under the spec attribute.
- By doing this, K8s will not assign a cluster IP address to that service and instead the Service will return the IP address of all the Pods to which it attaches and now the client can do a simple DNS lookup to communicate with the required Pod directly.

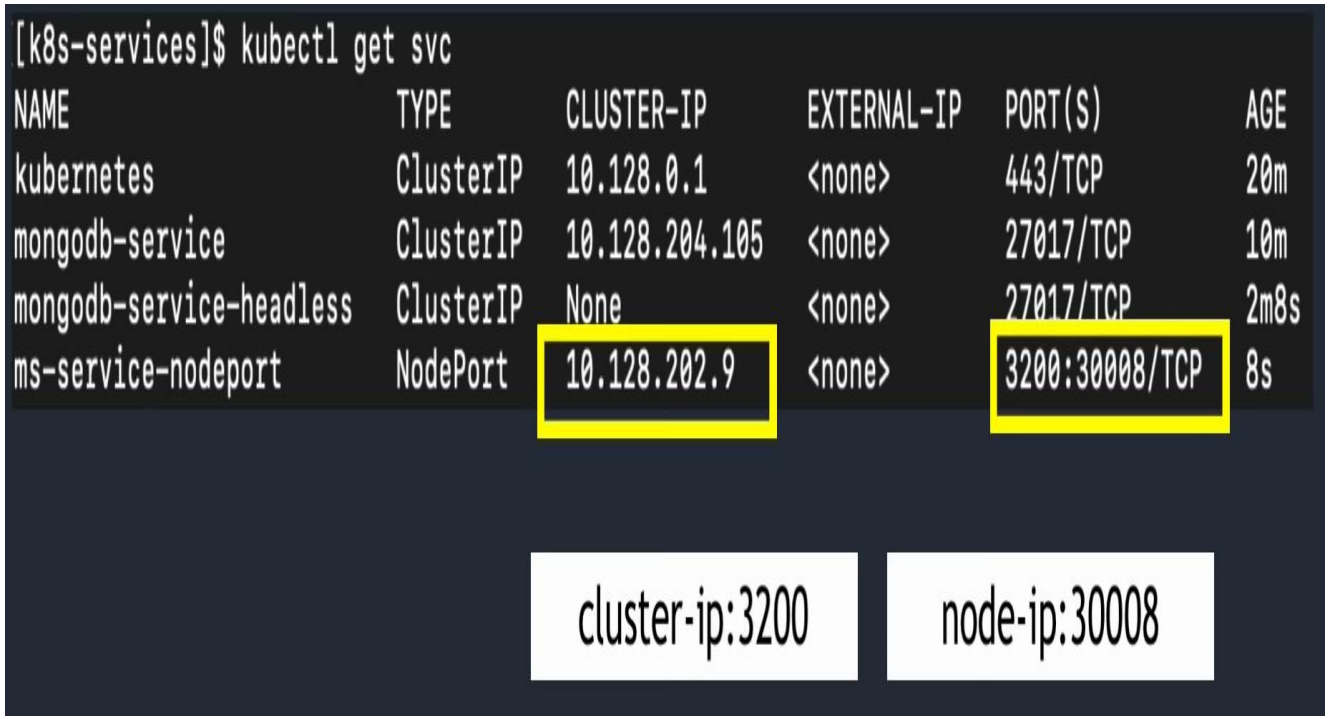
# NodePort Service

- Creates a Service that is **accessible** on a **specified static port** on each Worker Node in the Cluster.
- The ClusterIP service is only accessible within a Cluster whereas with the help of the NodePort service, **external traffic** has access to a **fixed port** on **each Worker Node**.
- This type of Service exposure is **not very efficient** and definitely **isn't secure** because you are allowing **external communication directly with an internal Node's port**, so clients have direct access to the Worker nodes basically.
- NodePort shouldn't be used in Production environments but it's good for debugging while developing your environment without having to configure a LoadBalancer or Ingress, so you can quickly test whether your services are working correctly with minimal configuration.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ms-service-nodeport
spec:
  type: NodePort
  selector:
    app: microservice-one
  ports:
  - port: 3200
    targetPort: 3000
    nodePort: 30008
    # The port address where NodePort service will be accessible at
    # Can only take values in the range of 30000-32767
    # Now the Browser can request microservice-one by specifying the IP Address
    # of the Worker Node followed by the NodePort
```



- But note that the NodePort service has a port parameter as well, this is the port of the ClusterIP address that's automatically created internally by K8s, a NodePort request is actually forwarded to this ClusterIP service for proper functioning.



# LoadBalancer Service

- This is a better alternative to NodePort.
- In this, a Service becomes accessible externally via a Cloud Provider’s Load Balancer.
- A ClusterIP and a NodePort service are created automatically to which the external Load Balancer of the Cloud Provider will route the traffic to.
- The LoadBalancer Service is an extension of the NodePort service which is an extension of the ClusterIP Service.

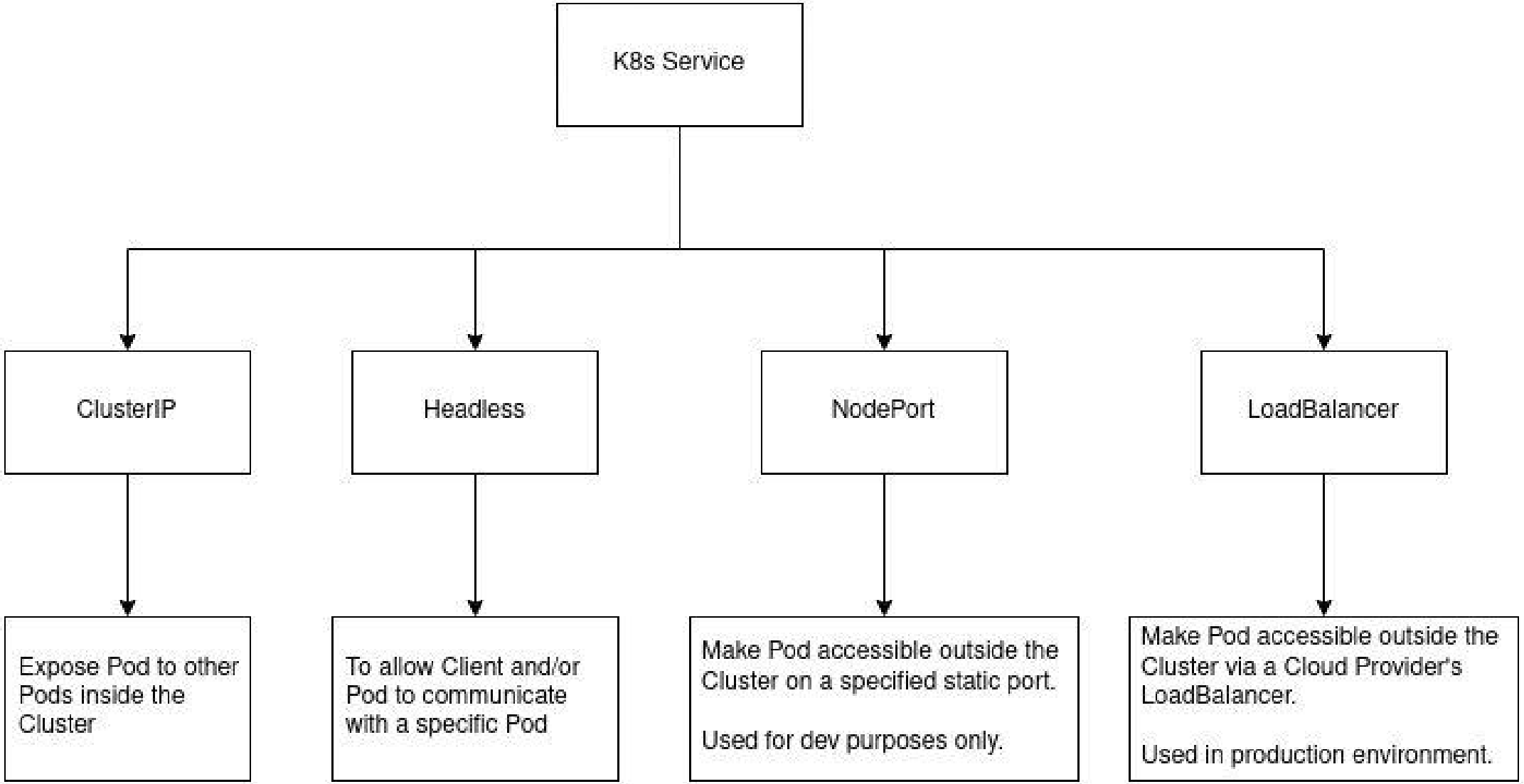
- Once you apply this YAML file, an external IP Address will be assigned to the Service which is how the Service can be accessed by the browser in the external world.

- But because we don’t have any Cloud Provider’s Load Balancer setup, on applying this YAML file the Service’s external IP field will be set to <pending> and remain as it is.

- To resolve this temporarily for our pseudo-cluster with minikube, open a new terminal and run the following command,
  **minikube tunnel**

- Keep this terminal running and you should see that an external IP has been assigned which you can use to access the Service and by extension the Deployment.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ms-service-loadbalancer
spec:
  type: LoadBalancer
  selector:
    app: microservice-one
  ports:
  - port: 3200
    targetPort: 3000
    nodePort: 30010
```


K8s Service
ClusterIP
Headless
NodePort
LoadBalancer
Expose Pod to other Pods inside the Cluster
To allow Client and/or Pod to communicate with a specific Pod
Make Pod accessible outside the Cluster on a specified static port.
Used for dev purposes only.
Make Pod accessible outside the Cluster via a Cloud Provider's LoadBalancer.
Used in production environment.

# K8s Objects

Deployment

Service

ConfigMap 

Secret

Ingress

Volume

StatefulSet

# ConfigMap

- Contains external configuration of your application like deployed URL endpoint, etc.
- The Pod is directly connected to the ConfigMap on the Node and uses it to read the configuration and set itself up.
- So if anything is updated in the ConfigMap, the Pod will automatically implement these changes on a restart and no redeployment is required.
- **Note: Don’t put credentials in ConfigMap!**
- Why not just these variables as environment variables in a Deployment?
- What if the variables are required to be shared across Deployments?
- That’s why ConfigMap is useful as it is a separate component independent of Pods.
- It also makes your design robust to configuration changes

# ConfigMap Configuration

```yaml
apiVersion: v1
kind: ConfigMap
# Type of K8s Object to create = ConfigMap
metadata:
  name: mongo-configmap
  # Name of the ConfigMap object, to be used in the Deployment
data:
# The data field consists of key-value pairs that will represent your
# environment variables
# In this case, a variable with the name "database-url" is created
# with the value "mongodb-service" and note that this is actually the name
# of the MongoDB Service K8s Object, so internally K8s will map this name
# to the ClusterIP Address of the MongoDB Service
  database-url: mongodb-service
```

# Using ConfigMap in a Deployment

```yaml
spec:
      containers:
      - name: mongo-express
        image: mongo-express
        ports:
        - containerPort: 8081
        env:
        - name: ME_CONFIG_MONGODB_SERVER
        # Name that you want to assign the environment variable in the container
          valueFrom:
          # Where are you getting the value of the variable from
            configMapKeyRef:
            # Referencing a ConfigMap for this variable's value
              name: mongo-configmap
              # Name of the ConfigMap
              key: database-url
              # Name of the key in the specified ConfigMap
```

# K8s Objects

Deployment

Service

ConfigMap

Secret 

Ingress

Volume

StatefulSet

# Secret

- This component is pretty much the same as ConfigMap but it is used for **storing secrets** and so it needs to be **encrypted**.
- The configuration file for a Secret looks the same as a ConfigMap but as mentioned it needs to be encrypted, which isn't done by K8s, so we'll have to do the encryption ourselves by specifying the base64 encoded version of the secret text.
- For example,If we have a secret variable called "username" whose value is "jasvin", then we'll put the Base64 version of it in the config file.
- Get the base64 string with the command,

```
jasvin@jasvin-Bravo-15:~$ echo 'jasvin' | base64
amFzdmluCg==
```

# Secret Configuration

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mongo-secret
type: Opaque
data:
  mongo-root-username: dXNlcm5hbWU=
  mongo-root-password: cGFzc3dvcmQ=
```

Note: You'll use this in the Deployment in the same way as with ConfigMap, just the attribute will be secretKeyRef

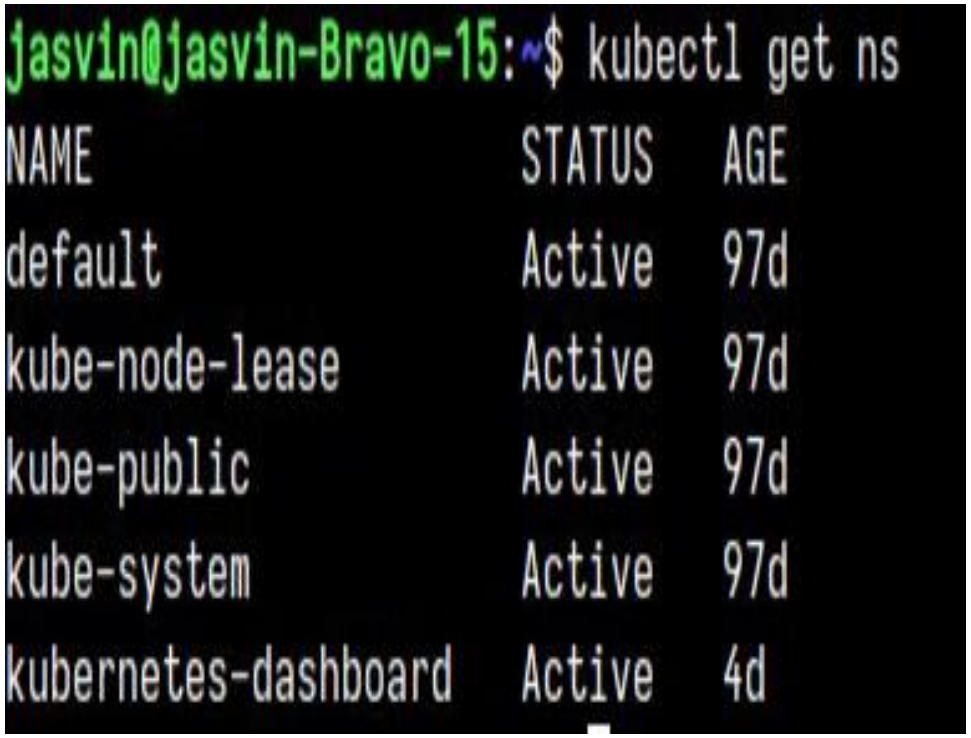
```yaml
- name: ME_CONFIG_MONGODB_ADMINUSERNAME
          valueFrom:
            secretKeyRef:
              name: mongo-secret
              key: mongo-root-username
```

# K8s Namespaces

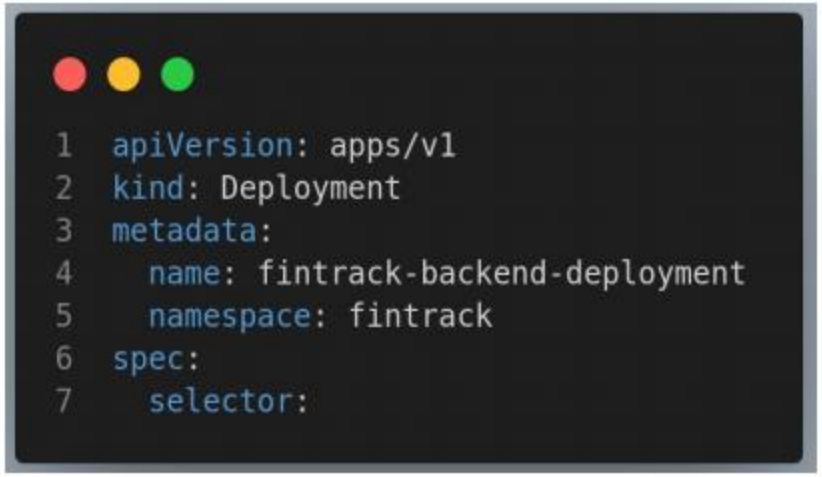
- Used for organizing resources and it is like a virtual cluster within your K8s cluster.
- **kubectl get ns**

  **or**

  **kubectl get namespaces**
- There are 5 namespaces by default
- **kubernetes-dashboard** is only shipped with minikube.[Must run minikube dashboard command first for this to be visible]
- **kube-system** is meant for the system and not for your use, so DO NOT create or modifyanything in it.
- **kube-public** contains publicly accessible data, a ConfigMap which contains cluster info without any authentication.
- **kube-node-lease** contains information about heartbeats of nodes.
- **default** is the namespace where you can do your actual work



- Also you can create and use your own namespaces as well, which is a generally good practice, like creating your own virtual environment with Python and Node projects.
- You can also create namespaces using configuration files which is again a generally better practice.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fintrack-backend-deployment
  namespace: fintrack
spec:
  selector:
```

- And note that all kubectl commands will need to include the relevant namespace name with the -n option now because the current active namespace is default.

  **kubectl apply -f filename.yaml -n <namespace-name>**

- But instead of doing this every time, you can set the current active namespace with the command,

  **kubectl config set-context --current--namespace=<namespace-name>**

- To check the current active namespace,

  **kubectl config view --minify | grep namespace:**

# K8s Objects

Deployment

Service

ConfigMap

Secret

Ingress 

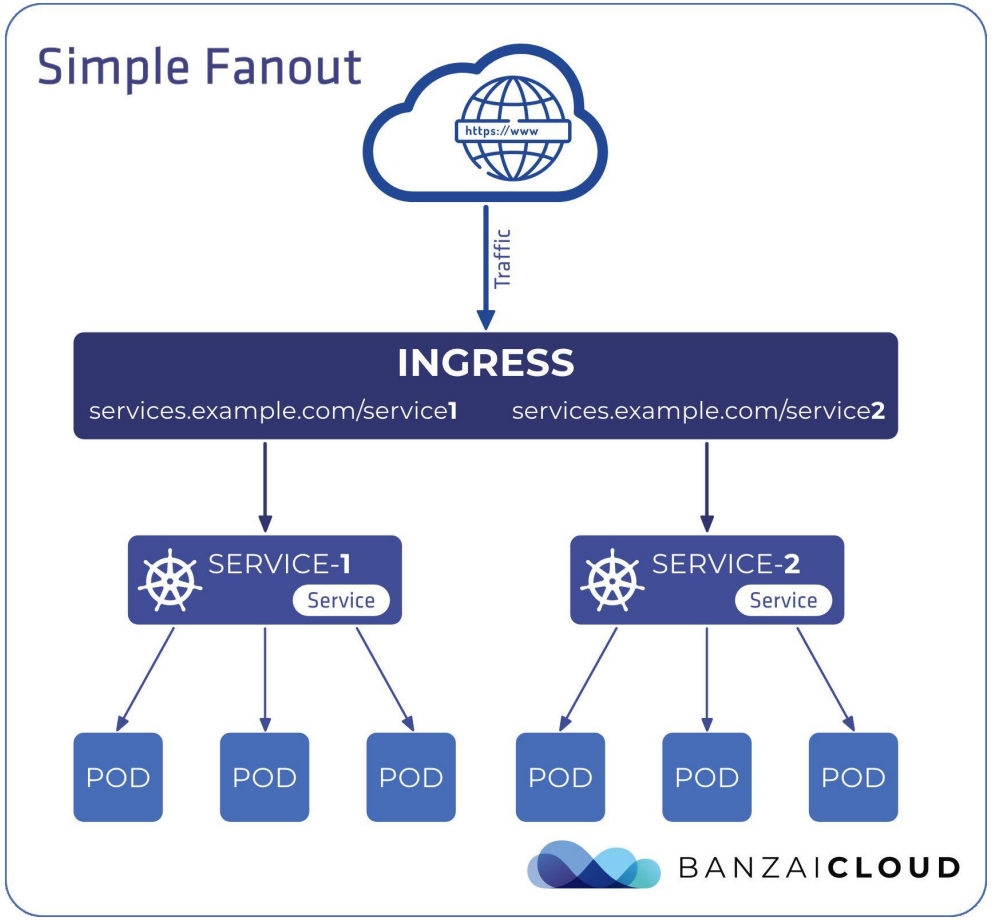
Volume

StatefulSet

# Ingress

- For your app Pod to be accessible outside the Node, you'll type in the IP address of the Node followed by the port address, but this is not very convenient so you'll need domain name addressing and for handling all of this, the Ingress service is used.

- So all client requests would hit the Ingress component which then forwards it to the Internal service and after that normal communication between Service and Deployment takes place.

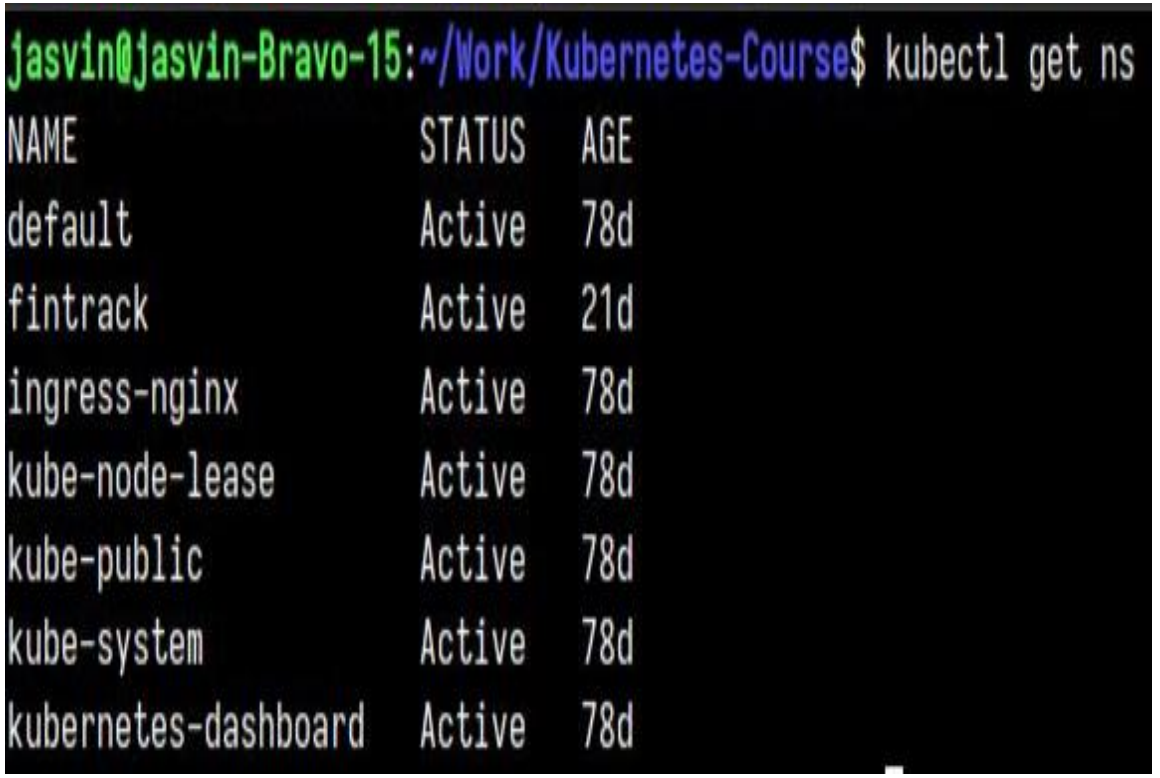


# Ingress Controller

- It is a pod or set of pods that run on your Node which continuously evaluates and processes Ingress rules and performs the necessary redirections.
- It is the real entrypoint to the cluster.
- There are many third-party implementations of Controllers, so it is up to you to decide which one to install. We will use the Nginx Ingress Controller (provided with Minikube).
- You can have an entrypoint Node outside the K8s cluster that redirects to K8s Nodes inside the cluster. It's a very good security practice as no internal components are exposed outside and most often this is a mandatory thing

# Installing Nginx Ingress Controller

- minikube addons enable ingress
- Automatically starts the K8s Nginx implementation of Ingress.
- Can also use this in production environments.
- A new namespace for the nginx-ingress controller is created.



# Setting up Ingress

- Apply the nginx ClusterIP service and the Ingress configuration.
- Wait till an IP address is assigned to the Ingress,

```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl get ingress
NAME            CLASS   HOSTS       ADDRESS        PORTS   AGE
nginx-ingress   nginx   myapp.com   192.168.49.2   80      16m
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$
```

- But this isn't enough to open myapp.com on the browser because you don't actually own the domain or are using a Cloud Provider, so we have to first map myapp.com to this IP address locally.
- For this open the /etc/hosts file and edit as shown.

```
  GNU nano 4.8
127.0.0.1       localhost
127.0.1.1       jasvin-Bravo-15
192.168.49.2    fintrack.com
192.168.49.2    myapp.com

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

# K8s Objects

Deployment

Service

ConfigMap

Secret

Ingress

Volume 

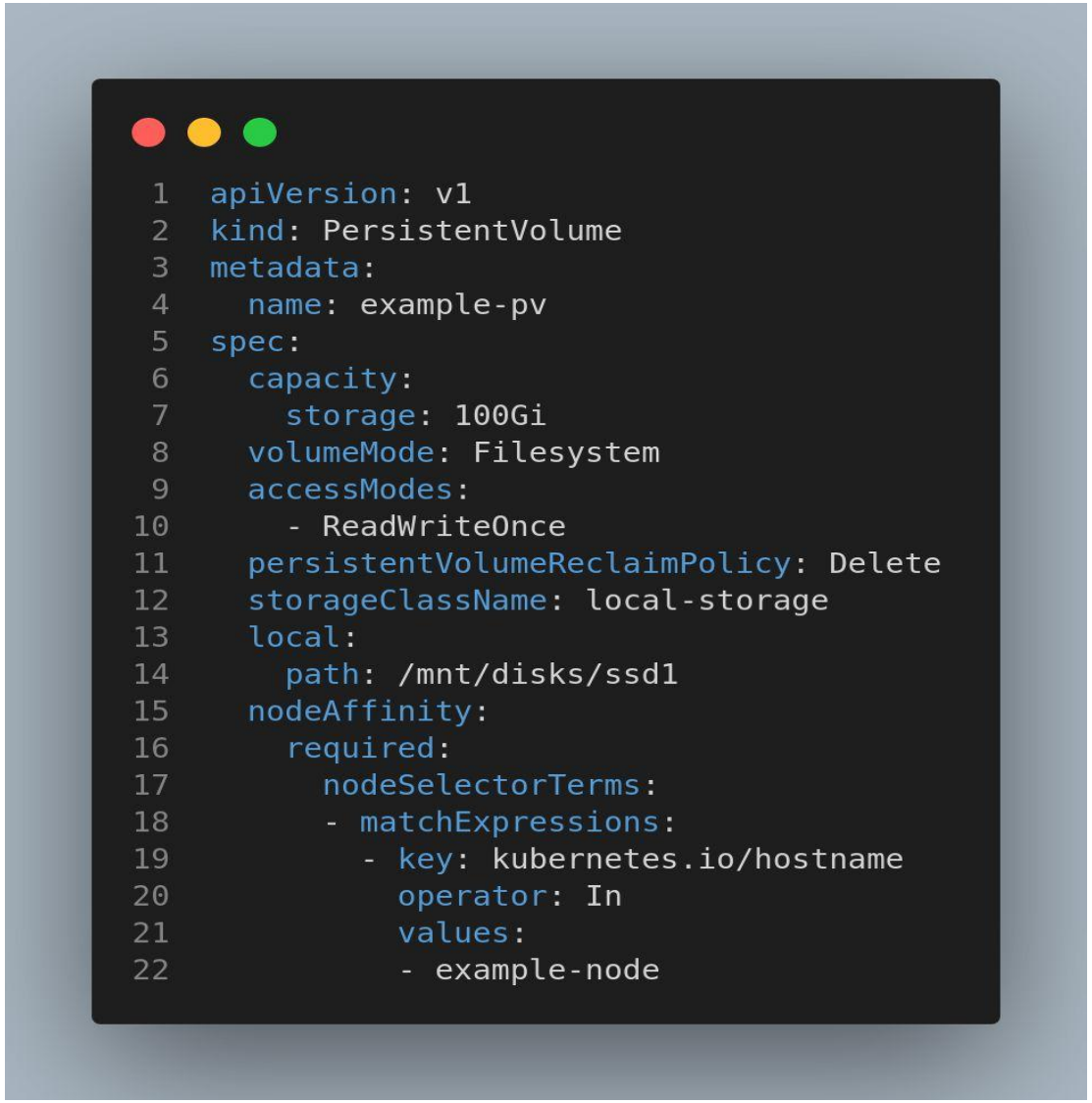
StatefulSet

# Volume

- For persistent data storage. Because if a Pod dies, its associated data will also die, the same problem as with Docker containers.
- A permanent storage location on a hard disk is associated with a Volume.
- A K8s Volume can be on the same Node or even outside the K8s cluster on any kind of remote storage.
- Think of K8s Volumes as external hard drives plugged into the K8s cluster.
- K8s doesn’t manage data persistence, so it is your responsibility to manage the persistence, availability, replication, etc. of the data.

# Persistent Volume (PV)

- K8s Component that allows you to **read/write from a file directory**, useful for session data, logs, etc.
- Think of it as a cluster resource like CPU/RAM.
- It is created via a YAML file with the kind PersistentVolume and needs the spec specifying how much storage has to be provisioned.
- This is an abstract component and you need actual physical storage to house your volume and this can be anywhere, on the cluster, external NFS server, cloud provider, etc.
- K8s doesn’t care where the storage is because it doesn’t provide any services for it, it just provides the Persistent Volume as an interface that can be used to access the underlying data. Management of the data like backups and consistency have to be done by yourself.

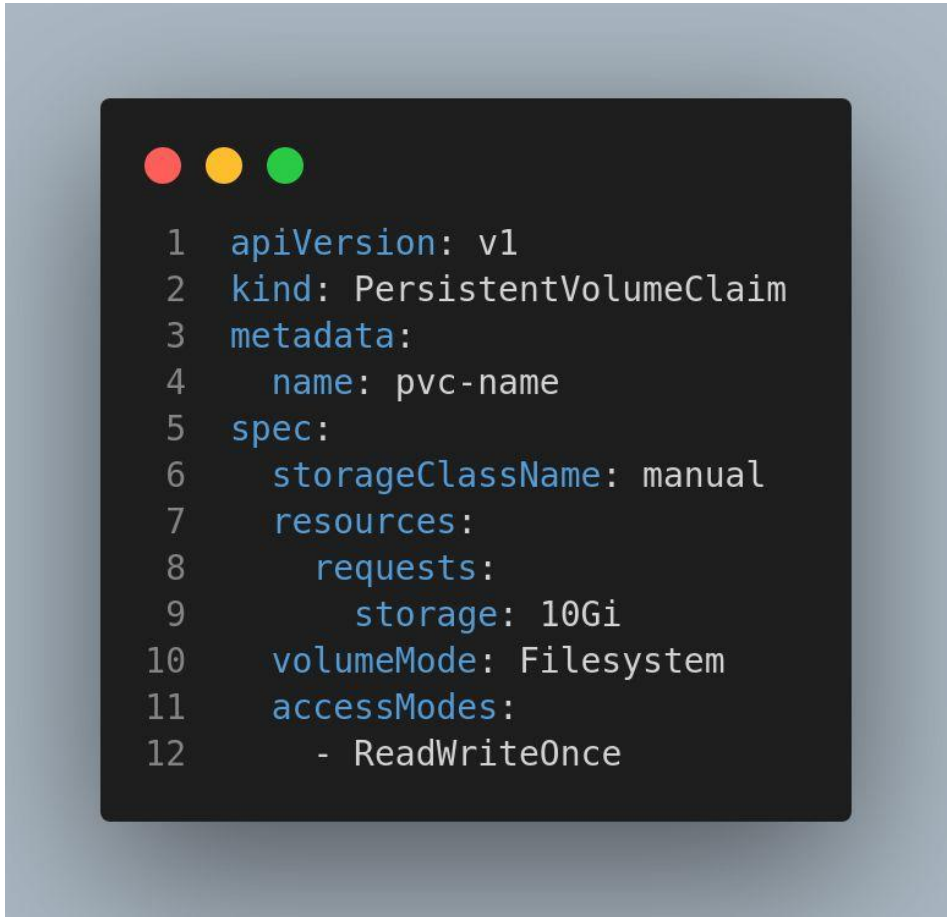


```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
```

- Persistent Volumes need to exist before the Pod that depends on it is created.
- Persistent Volumes are not namespaced, so they exist outside of the cluster and are available to all namespaces within the cluster.

# Persistent Volume Claim (PVC)

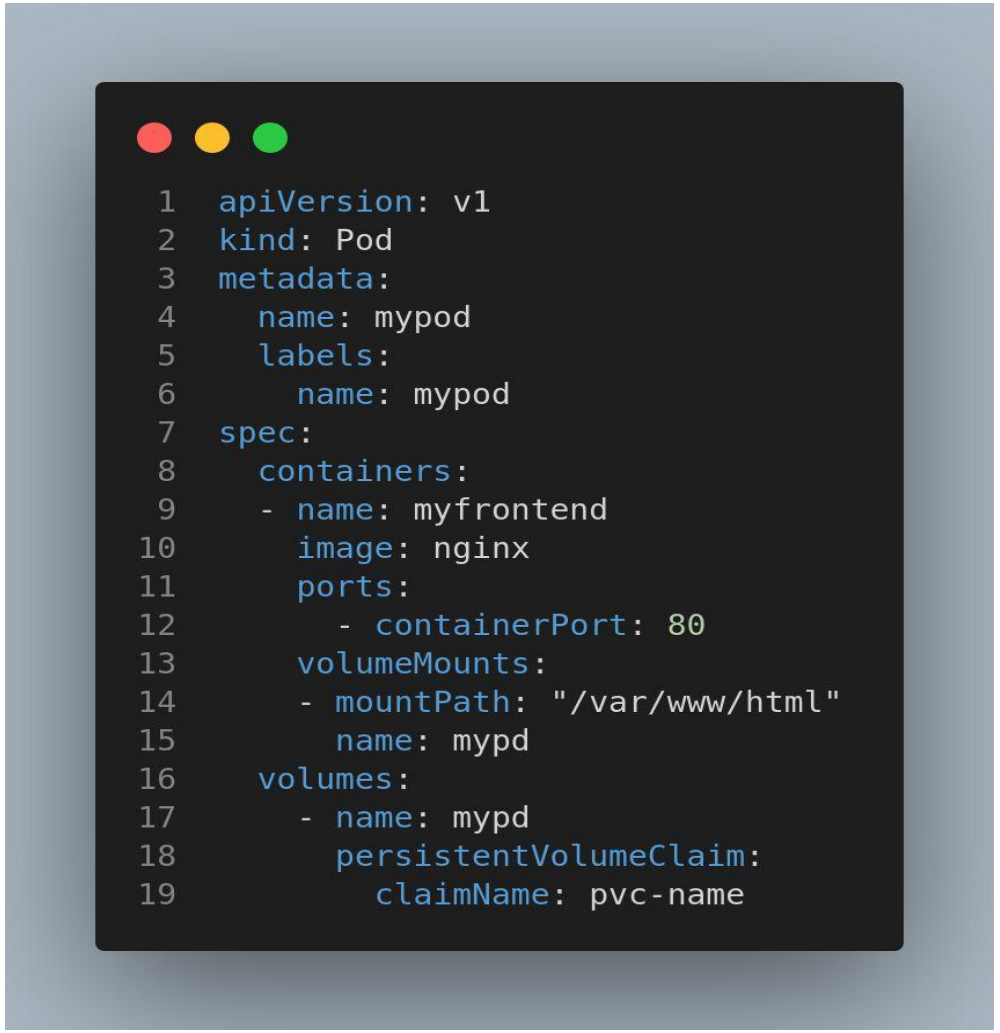
- In order to get access to a Persistent Volume, an Application has to first claim the required amount of Persistent Volume space and for this you have to write a Persistent Volume Claim.
- This component claims the specified amount of storage (10GB) from a volume with specified attributes like accessModes and so on. Whichever existing PersistentVolume satisfies this criteria will be used by the PVC and in turn your application.



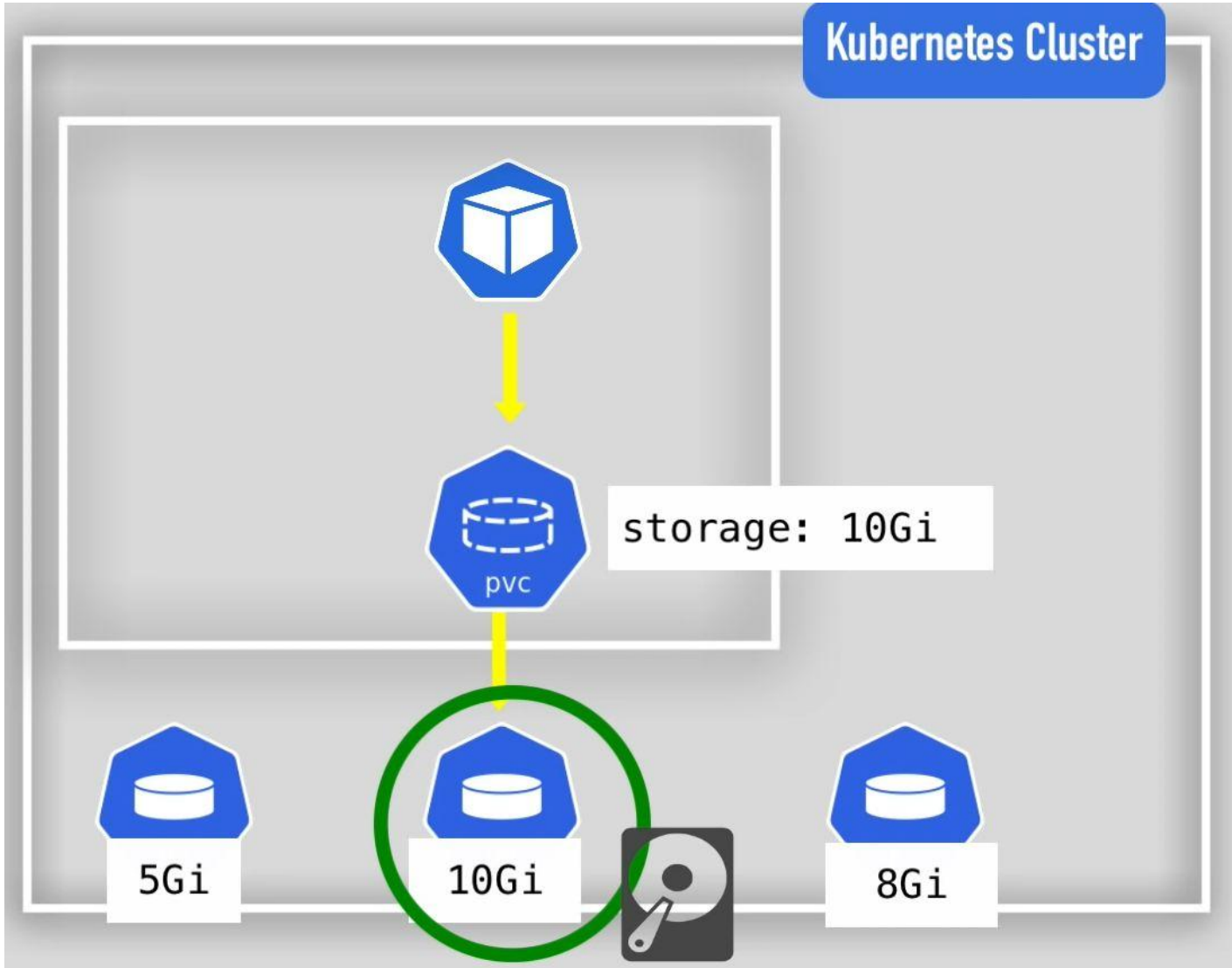
```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-name
spec:
  storageClassName: manual
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
```

- You have to also reference the PVC in your Pod configuration as in the image

- Notice that the Pod's configuration file specifies a PVC, but it does not specify a PV. From the Pod's point of view, the claim is a volume.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  labels:
    name: mypod
spec:
  containers:
  - name: myfrontend
    image: nginx
    ports:
      - containerPort: 80
    volumeMounts:
    - mountPath: "/var/www/html"
      name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc-name
```




Pod requests the volume through the PV claim

Claim tries to find a volume in cluster

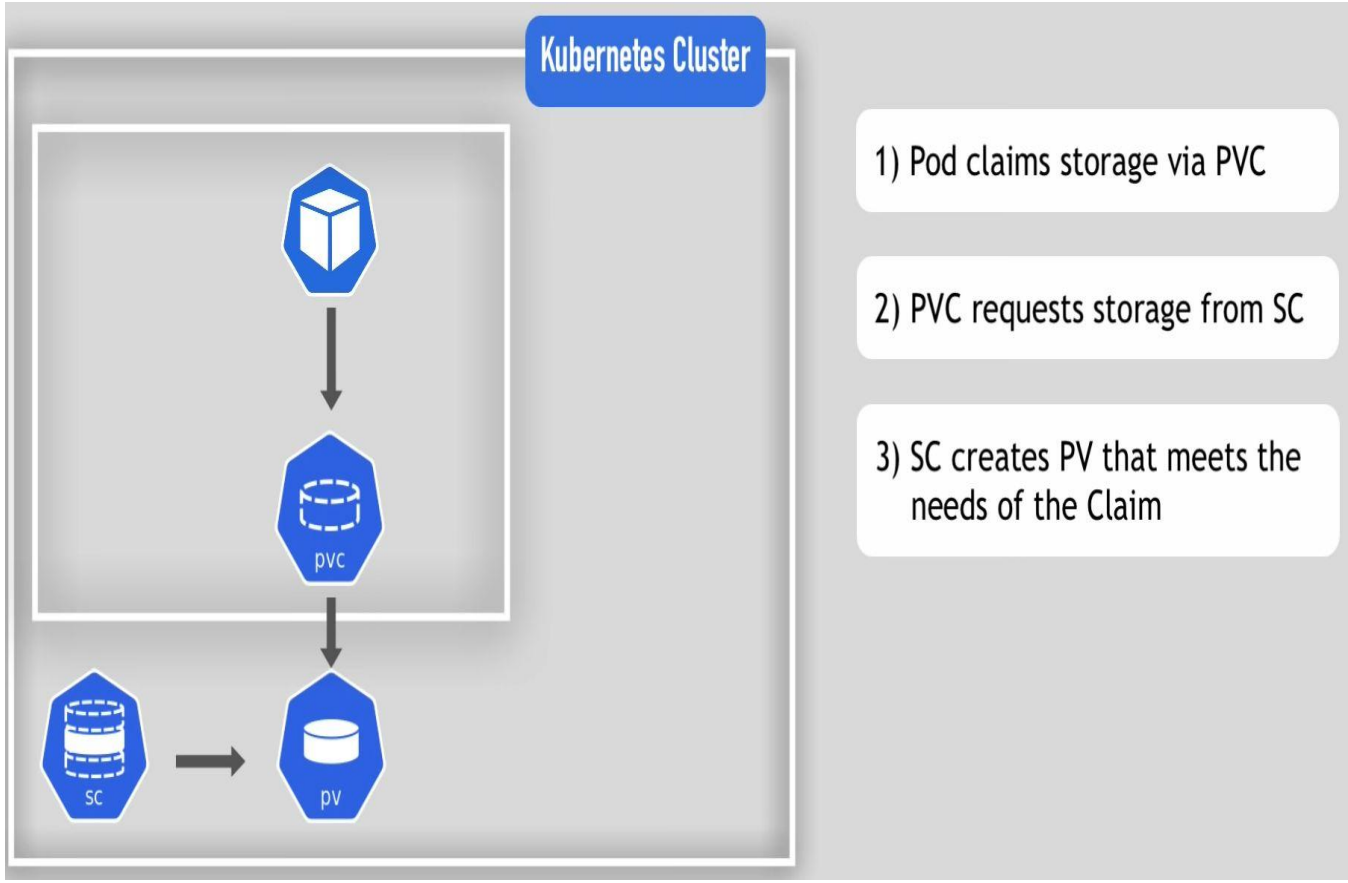
Volume has the actual storage backend

# Storage Class

- The K8s application can require 100s of Pods each of which require several Volumes for which PVs and PVCs have to be provisioned and configured by the Admin (and potentially, the Cloud Provider's team) which is a very laborious manual process that takes time. To make this process efficient, you have the K8s Storage Class component.
- **Storage Class provisions Persistent Volumes dynamically whenever a PVC claims it.**
- **provisioner is the most important attribute because it is the provisioner that is going to be doing all the work of dynamically creating and maintaining PVs.**
- Each storage backend has its own provisioner, prefixed with kubernetes.io.

# Final Overall Flow!

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-name
spec:
  storageClassName: storage-class-name
  resources:
    requests:
      storage: 100Gi
  accessModes:
    - ReadWriteOnce
```



# K8s Objects

Deployment

Service

ConfigMap

Secret

Ingress

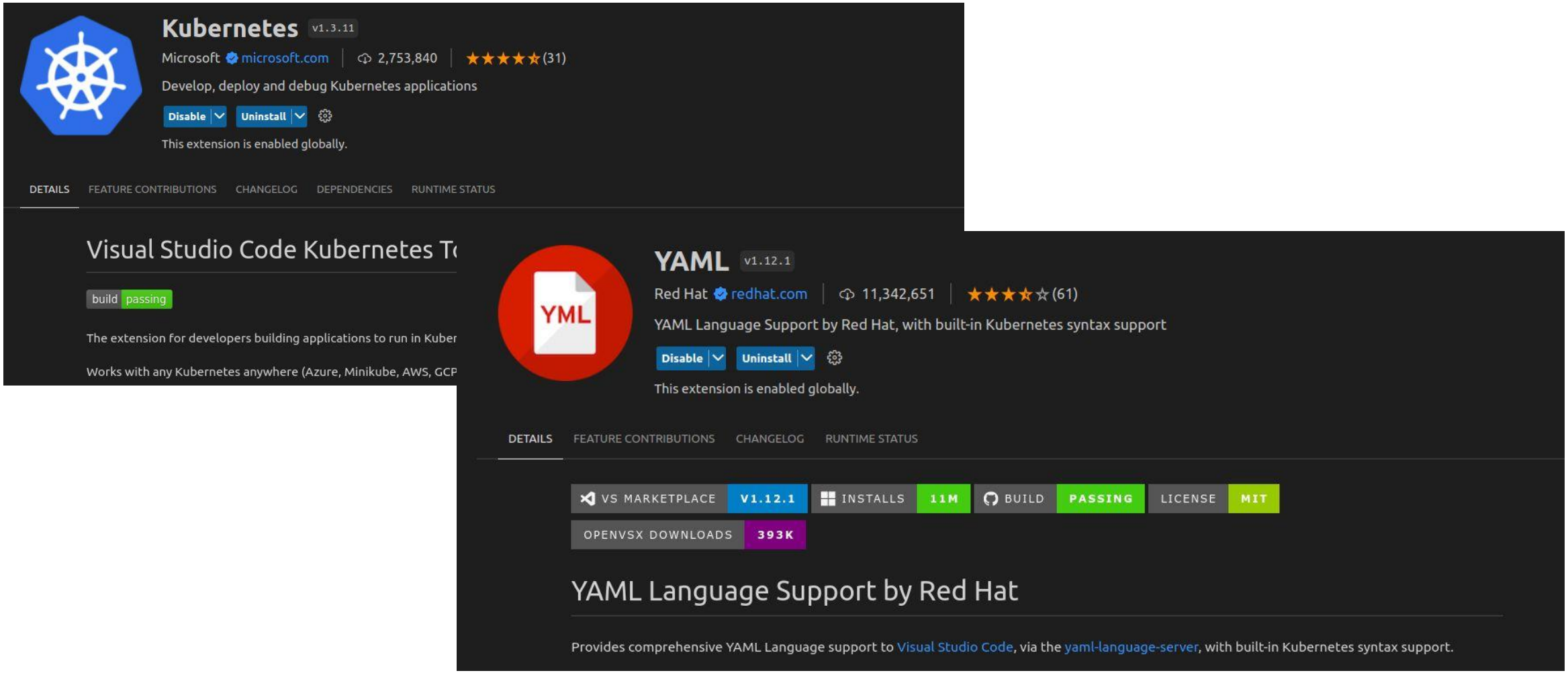
Volume

StatefulSet

# StatefulSet

- It is the alternative to Deployment for stateful applications like database applications.
- Deployment cannot be used here because it doesn’t maintain any state so the Pods are ephemeral (dies quickly).
- However the core issue with using StatefulSet is that,Stateful apps are not perfect for containerized environments but are instead perfect for stateless apps.
- Consider you have a StatefulSet for MySQL, the Pods cannot be deleted or created simultaneously, as in you cannot create 2 StatefulSet Pods at the same time and the same thing for deletion. They cannot be randomly addressed either like Deployment Pods. This is because the replica Pods are not identical, so they have a Pod Identity and assigning this identity is the main task of StatefulSet

# Exercise 9: Install the Kubernetes & YAML VSCode Extensions



# Exercise 10: Write the YAML config shown and create it via kubectl apply

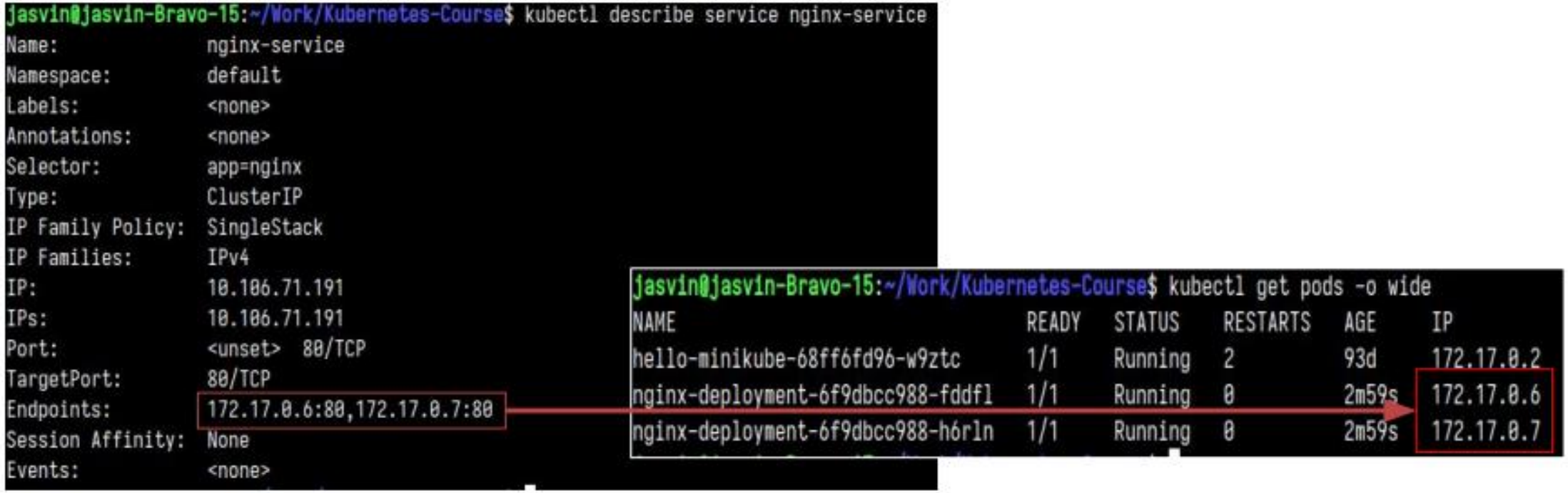
```yaml
apiVersion: apps/v1
# The apiVersion field is used to specify the version of the Kubernetes API that should be used
kind: Deployment
# Type of component you are deploying
metadata:
  name: nginx-deployment
  # Name of the Deployment created
  labels:
  # Selector Labels: Used to match the Deployment with its ReplicaSet
  # These labels are key-value pairs that can include any key and any value
    app: nginx
spec:
# Specification for the Deployment
  replicas: 2
  # Number of Replicas to create
  selector:
    # To link the Deployment to its Pods, you specify which labels a particular Pod
    # must have to be linked with this Deployment
    matchLabels:
      app: nginx
  template:
  # Template for the Pod
    metadata:
      labels:
      # Must be the same as the Deployment's labels
      # This is how Pods are linked with the Deployment
        app: nginx
    spec:
    # Specification for the Pod
      containers:
      - name: nginx
        # Name of the container, custom-name
        image: nginx:1.16
        # Which image this container uses
        ports:
        - containerPort: 80
        # On which port this container runs
```

```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl get all
NAME                                    READY   STATUS              RESTARTS   AGE
pod/nginx-deployment-6f9dbcc988-7c8d6   0/1     ContainerCreating   0          3s
pod/nginx-deployment-6f9dbcc988-hg7g8   0/1     ContainerCreating   0          3s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   77d

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment   0/2     2            0           3s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-deployment-6f9dbcc988   2         2         0       3s
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$
```

```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl describe deployment/nginx-deployment
Name:                   nginx-deployment
Namespace:              default
CreationTimestamp:      Fri, 17 Mar 2023 15:00:29 +0530
Labels:                 app=nginx
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=nginx
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=nginx
  Containers:
   nginx:
    Image:        nginx:1.16
    Port:         80/TCP
    Host Port:    0/TCP
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-6f9dbcc988 (3/3 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  15m   deployment-controller  Scaled up replica set nginx-deployment-6f9dbcc988 to 2
  Normal  ScalingReplicaSet  10s   deployment-controller  Scaled up replica set nginx-deployment-6f9dbcc988 to 3 from 2
```

# Exercise 13: Create a Cluster IP Service

- Create a ClusterIP Service for the nginx Deployment and verify that the two are now connected.



# Exercise 14: Create a NodePort Service

- Create a NodePort Service for the nginx Deployment and open the Pod in the browser.

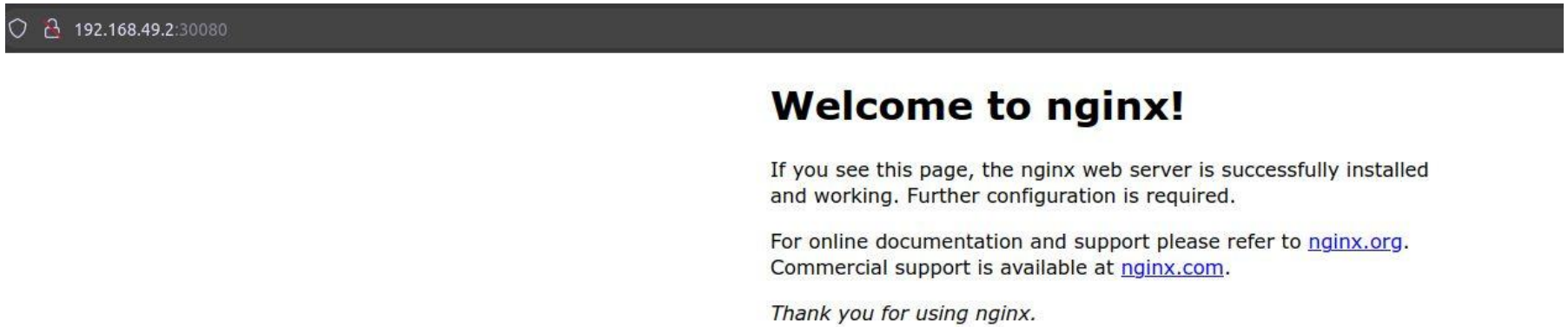
```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl apply -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl apply -f nginx-nodeport-service.yaml
service/nginx-service created
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
pod/nginx-deployment-6f9dbcc988-gjxs2   1/1     Running   0          11s
pod/nginx-deployment-6f9dbcc988-th212   1/1     Running   0          11s
pod/nginx-deployment-6f9dbcc988-xh6mh   1/1     Running   0          11s

NAME                    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
service/kubernetes      ClusterIP   10.96.0.1        <none>        443/TCP        77d
service/nginx-service   NodePort    10.102.144.172   <none>        80:30001/TCP   3s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment   3/3     3            3           11s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-deployment-6f9dbcc988   3         3         3       11s
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$
```

You can open the Deployment (nginx server index.html page) in the browser by typing in <IP-Address-of-PC>:<Node-Port>

But this won't work, why?
Because we are on a minikube cluster, not an actual K8s Cloud Deployment, so we have to first make the service available by executing the minikube service command

```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ minikube service nginx-service
|-----------|---------------|-------------|---------------------------|
| NAMESPACE |     NAME      | TARGET PORT |            URL            |
|-----------|---------------|-------------|---------------------------|
| default   | nginx-service | http/80     | http://192.168.49.2:30080 |
|-----------|---------------|-------------|---------------------------|
🎉  Opening service default/nginx-service in default browser...
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$
```

What the webpage looks like



# Exercise 14: Create a LoadBalancer Service

Create a LoadBalancer Service for the nginx Deployment and open the Pod in the browser.

```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl apply -f nginx-loadbalancer-service.yaml
service/nginx-service created
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl get all
NAME                                    READY   STATUS    RESTARTS      AGE
pod/nginx-deployment-6f9dbcc988-bd7tn   1/1     Running   2 (8h ago)    17h
pod/nginx-deployment-6f9dbcc988-jsbbf   1/1     Running   2 (8h ago)    17h
pod/nginx-deployment-6f9dbcc988-kwqkg   1/1     Running   2 (8h ago)    17h

NAME                    TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/kubernetes      ClusterIP      10.96.0.1       <none>        443/TCP        78d
service/nginx-service   LoadBalancer   10.110.219.95   <pending>     80:30002/TCP   3s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nginx-deployment   3/3     3            3           17h

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/nginx-deployment-6f9dbcc988   3         3         3       17h
replicaset.apps/nginx-deployment-74c7646cbc   0         0         0       17h
```

```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ minikube tunnel
[sudo] password for jasvin:
Status:
        machine: minikube
        pid: 15440
        route: 10.96.0.0/12 -> 192.168.49.2
        minikube: Running
        services: [nginx-service, fintrack-frontend-service]
    errors:
                minikube: no errors
                router: no errors
                loadbalancer emulator: no errors
```

```
jasvin@jasvin-Bravo-15:~/Work/Kubernetes-Course$ kubectl get services
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
kubernetes      ClusterIP      10.96.0.1       <none>          443/TCP        78d
nginx-service   LoadBalancer   10.110.219.95   10.110.219.95   80:30002/TCP   2m16s
```

Open 10.110.219.95:80 in the browser