Introduction to Kubernetes

JJTECH INC 2023

Topics

- Definition of concepts
- Introduction to Kubernetes
- K8s Architecture
- Kubernetes Installation
- K8s Pods
- K8s Namepsaces
- Running K8s Workloads (Deployments, Replicasets, Daemonsets, Statefulsets)
- K8s Updates (Update strategies)
- K8s Selectors
- K8s Services and Ingresses
- K8s Storage and Persistence

Definition of key concepts

- **K8s Cluster:** A Kubernetes cluster is a group of nodes that work together to run containerized applications and manage the underlying infrastructure. A K8s cluster is a combination of Master/control plane and worker nodes
- **Containers:** Containers are lightweight, isolated environments that encapsulate an application and its dependencies. They provide a consistent and reproducible runtime environment, enabling applications to run reliably across different computing environments.
- **Container Runtime:** A container runtime is a software component responsible for running and managing containers on a host system. It is an essential part of the containerization technology stack and provides the necessary functionality to execute container images and manage their lifecycle. E.g Docker, Containerd
- **Container orchestration:** This refers to the management and coordination of containers within a distributed environment. It involves automating the deployment, scaling, scheduling, and management of containerized applications across a cluster of machines. E.g Kubernetes, Docker swarm, Apache Mesos

Introduction to Kubernetes

- Kubernetes, also known as K8s, is an open-source container orchestration platform th automates the deployment, scaling, and management of containerized applications.
- It was originally developed by Google and is no maintained by the Cloud Native Computing Foundation (CNCF).
- K8s was first announced in June 2014 and was open-sourced and made available to the public in July 2015

What problems does K8s Solve

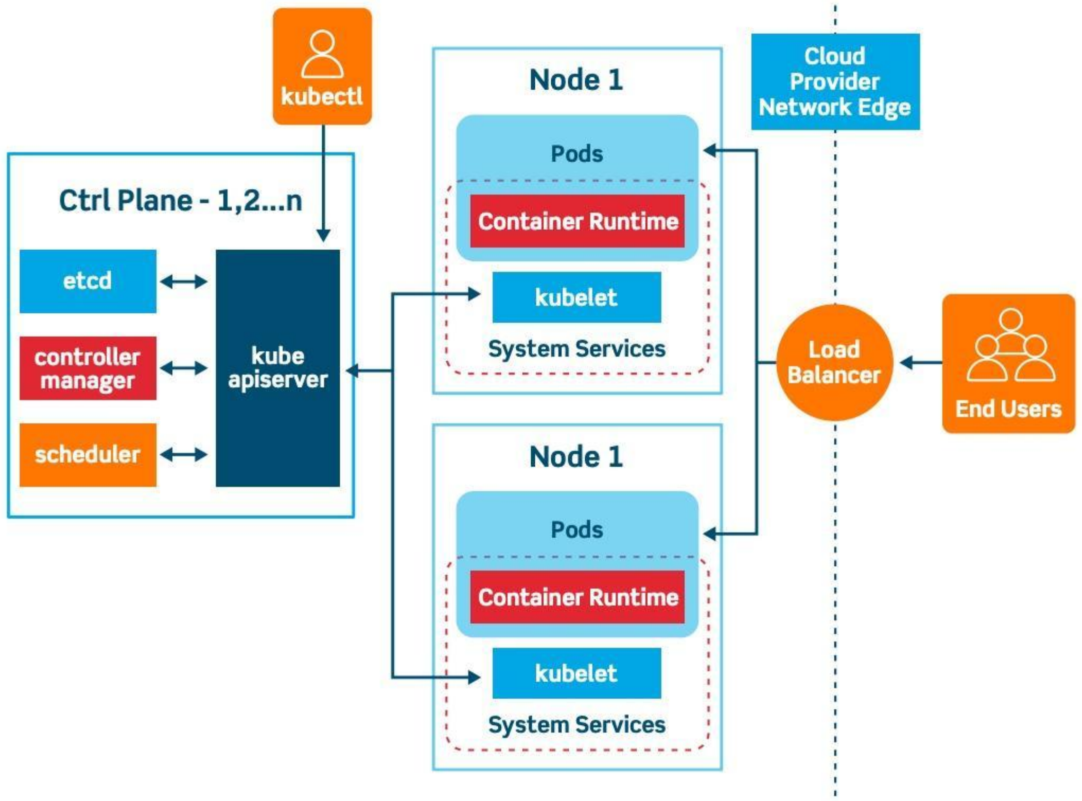
- **Container Orchestration**: Managing a large number of containers manually can be complex and time-consuming. Kubernetes automates the deployment, scaling, and management of containers, making it easier to run applications in a distributed environment.
- **Service Discovery and Load Balancing**: In a dynamic container environment, services need a way to discover each other and communicate. Kubernetes provides service discovery mechanisms and built-in load balancing, ensuring that traffic is properly routed to containers.
- **Fault Tolerance and Self-Healing**: Containers can fail for various reasons, such as hardware issues or software bugs. Kubernetes continuously monitors the health of containers and automatically restarts or replaces failed instances to maintain the desired state of the application.
- **Scalability and Resource Efficiency**: Kubernetes enables horizontal scaling by allowing the dynamic creation and scaling of container replicas based on resource demands. This ensures optimal utilization of resources while handling increased traffic or workload.

- **Container Orchestration**: Managing a large number of containers manually can be complex and time-consuming. Kubernetes automates the deployment, scaling, and management of containers, making it easier to run applications in a distributed environment.
- **Service Discovery and Load Balancing**: In a dynamic container environment, services need a way to discover each other and communicate. Kubernetes provides service discovery mechanisms and built-in load balancing, ensuring that traffic is properly routed to containers.
- **Fault Tolerance and Self-Healing**: Containers can fail for various reasons, such as hardware issues or software bugs. Kubernetes continuously monitors the health of containers and automatically restarts or replaces failed instances to maintain the desired state of the application.
- **Scalability and Resource Efficiency**: Kubernetes enables horizontal scaling by allowing the dynamic creation and scaling of container replicas based on resource demands. This ensures optimal utilization of resources while handling increased traffic or workload.

Kubernetes Architecture

•**Master Node**: The master node is responsible for managing and controlling the cluster. It oversees the overall state of the cluster, schedules workloads, and coordinates communication between various components. The master node typically consists of the following components:

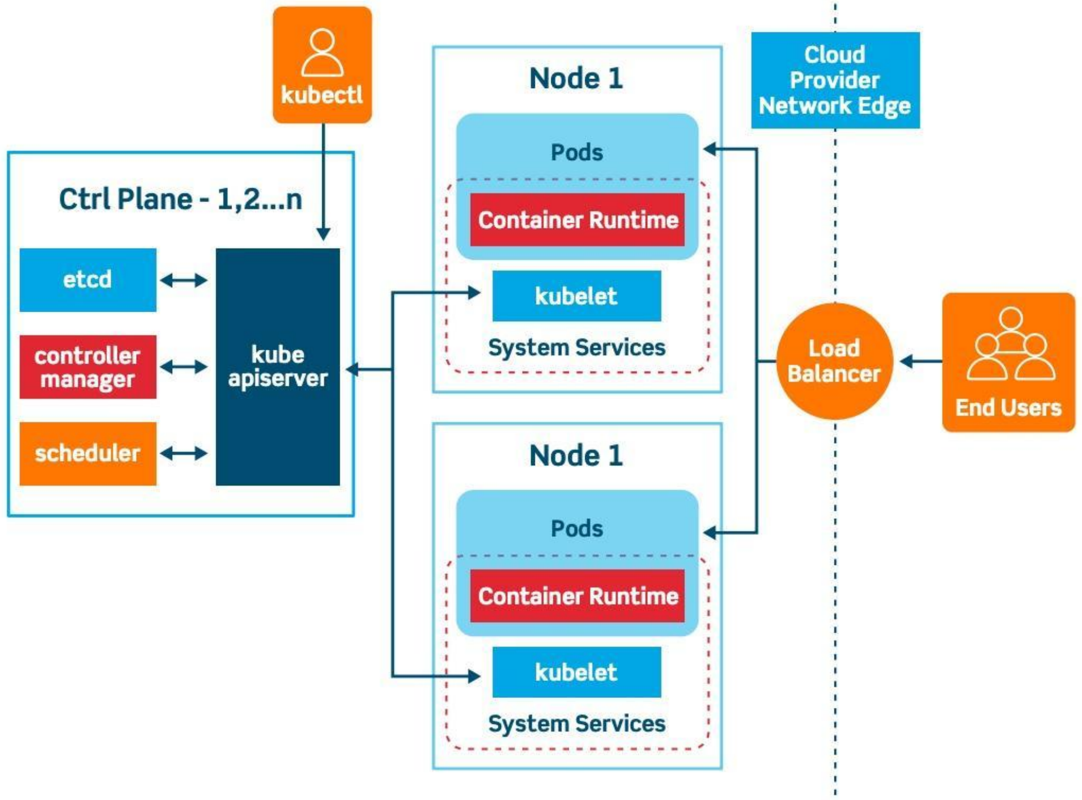
- **API Server**: The API server acts as the primary control plane component, serving as the central management point for all cluster operations. It exposes the Kubernetes API, which allows users and other components to interact with the cluster.
- **Scheduler**: The scheduler assigns pods to nodes based on resource requirements, availability, and other constraints. It determines the optimal placement of pods across the cluster to achieve efficient resource utilization.
- **Controller Manager**: The controller manager runs various controllers responsible for managing different aspects of the cluster. Examples include the replication controller for maintaining desired pod replicas, the node controller for monitoring and managing nodes, and the service controller for managing services.
- **etcd**: etcd is a distributed key-value store that serves as the cluster's primary datastore. It stores the persistent state of the cluster, including configuration data, API objects, and cluster health information.



Kubernetes Architecture

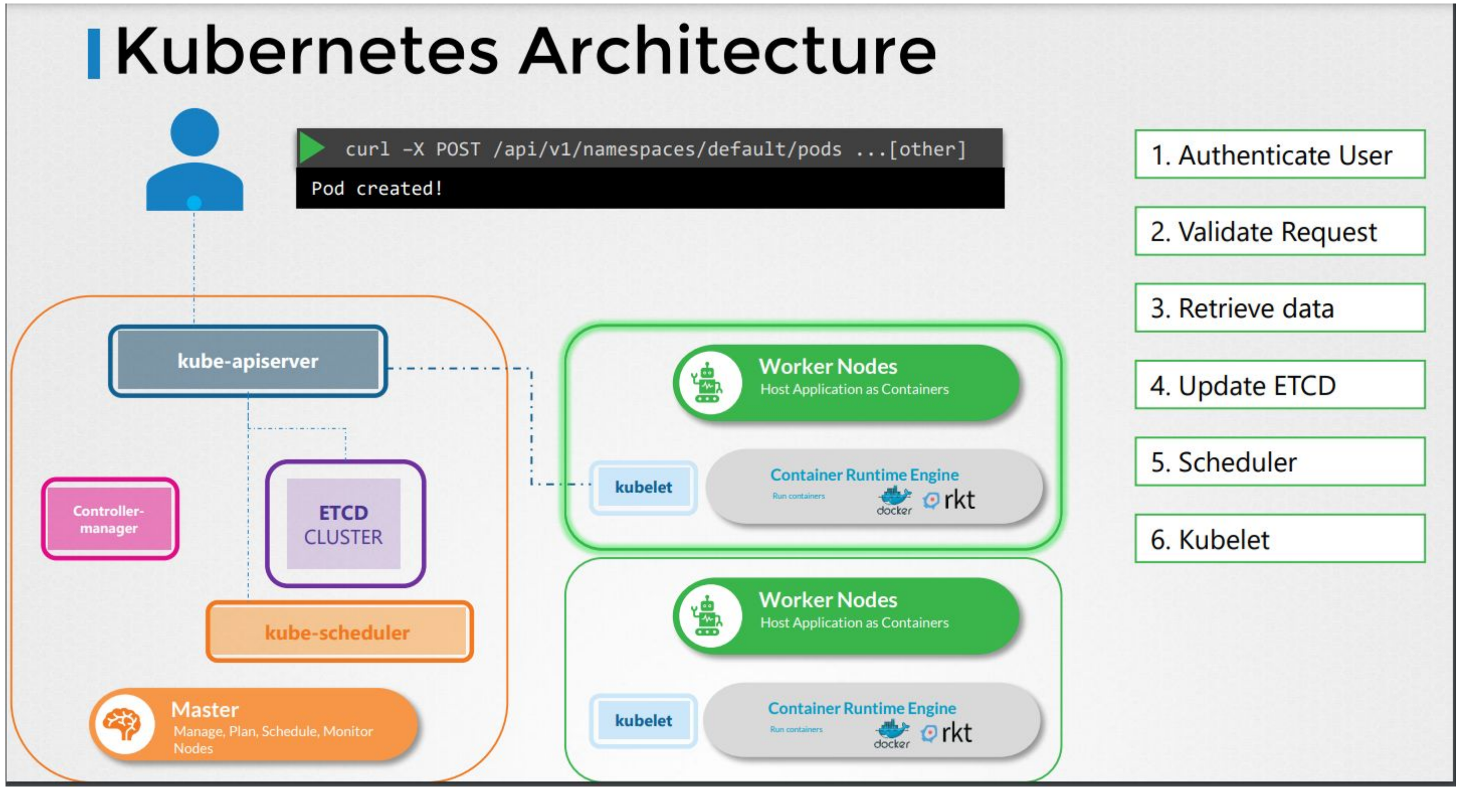
•**Worker Nodes**: Worker nodes are the compute nodes in the cluster where containers are deployed and run. Each worker node typically consists of the following components:

- **Kubelet**: The kubelet is an agent running on each node and responsible for managing containers on that node. It communicates with the API server, receives pod specifications, and ensures that the containers are running and healthy.
- **Container Runtime**: The container runtime, such as Docker or containerd, is responsible for pulling and running container images. It provides the underlying infrastructure for running containers on the node.
- **Kube-proxy**: The kube-proxy enables network communication to and from the pods. It sets up network routing rules, load balances traffic, and provides service discovery capabilities within the cluster.
- **Pods:** A pod is the basic scheduling unit in Kubernetes. It represents one or more co-located containers sharing the same network namespace and storage. Pods are created and scheduled onto worker nodes based on resource requirements and constraints defined in their specifications.



Kube Api Server

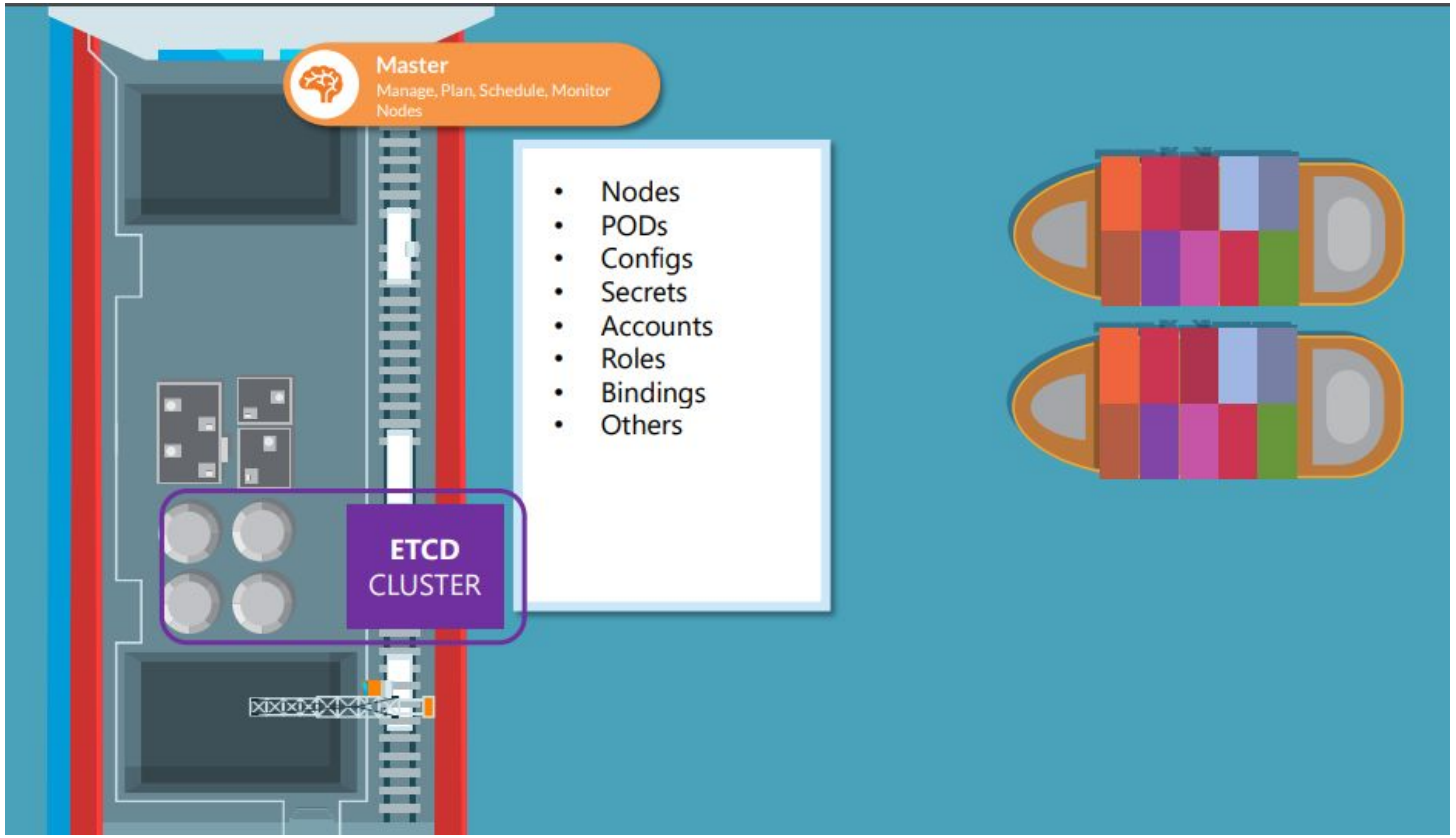
- It exposes the Kubernetes API, allowing users and other components to interact with the cluster.
- It is the entry point for cluster management and resource manipulation.
- It is the only component that interacts directly with the etcd datastore
- It is responsible for handling requests, maintaining cluster state, enforcing security policies, and providing an interface for users and components to interact with the Kubernetes ecosystem.



```
curl -X POST /api/v1/namespaces/default/pods ...[other]
Pod created!
```

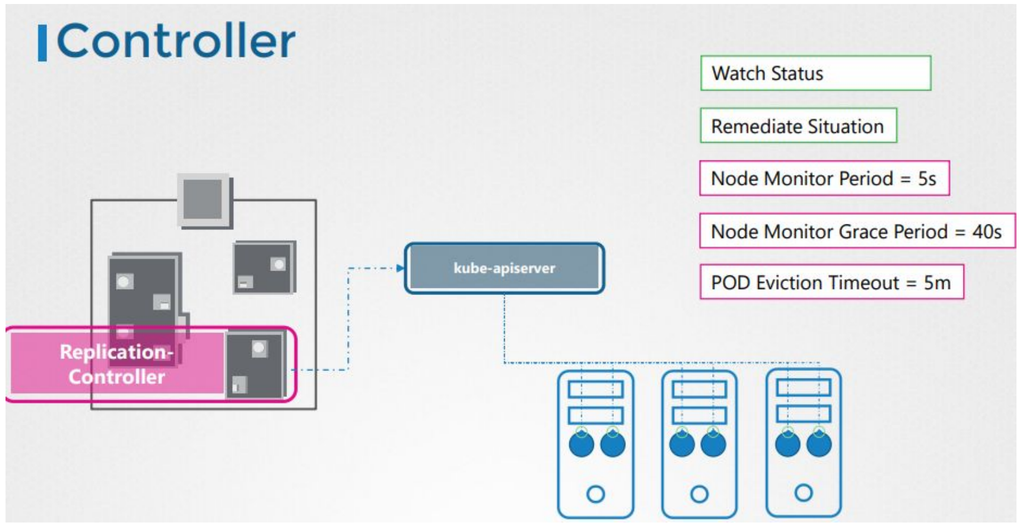
ETCD Cluster

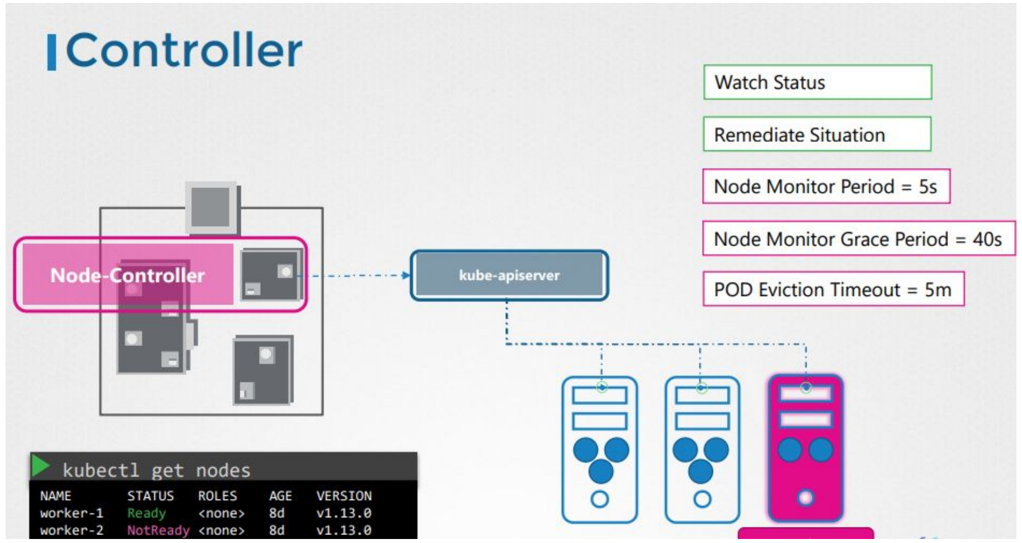
- ETCD stores cluster state information
- Etcd is a highly available and consistent datastore that serves as the primary source of truth for Kubernetes across multiple nodes in the cluster
- Etcd allows for backup and restore, ensuring critical cluster state info is preserved



Controller Manager

- The kube-controller-manager is responsible for maintaining the desired state of the cluster, reacting to changes, and ensuring the proper functioning and health of various Kubernetes resources and components.
- The kube-controller-manager runs multiple controllers that monitor and reconcile the desired state of various Kubernetes objects and resources.
- Node Controller: The node controller monitors the state of nodes in the cluster and performs actions based on changes, such as detecting node failures, marking nodes as ready or not ready, and taking actions to maintain the desired number of nodes.
- Replication Controller/Replica Set Controller: These controllers ensure that the desired number of pod replicas specified in a deployment or replica set are running and maintain the desired state by creating or deleting pods as necessary.
- Service Controller: The service controller ensures that services in the cluster are correctly configured and that they route traffic to the appropriate pods based on service selectors and endpoints.
- Namespace Controller: The namespace controller is responsible for creating and managing namespaces in the cluster, enforcing resource quotas, and maintaining isolation between different components or user groups.

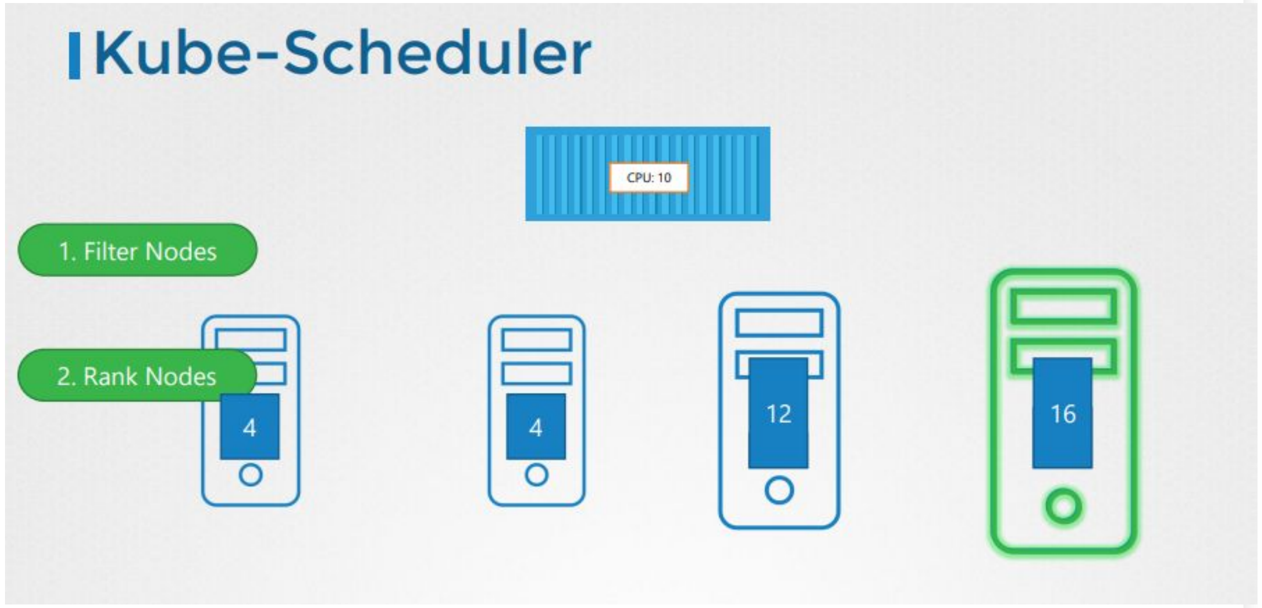




```
▶ kubectl get nodes
NAME        STATUS    ROLES   AGE   VERSION
worker-1    Ready     <none>  8d    v1.13.0
worker-2    NotReady  <none>  8d    v1.13.0
```

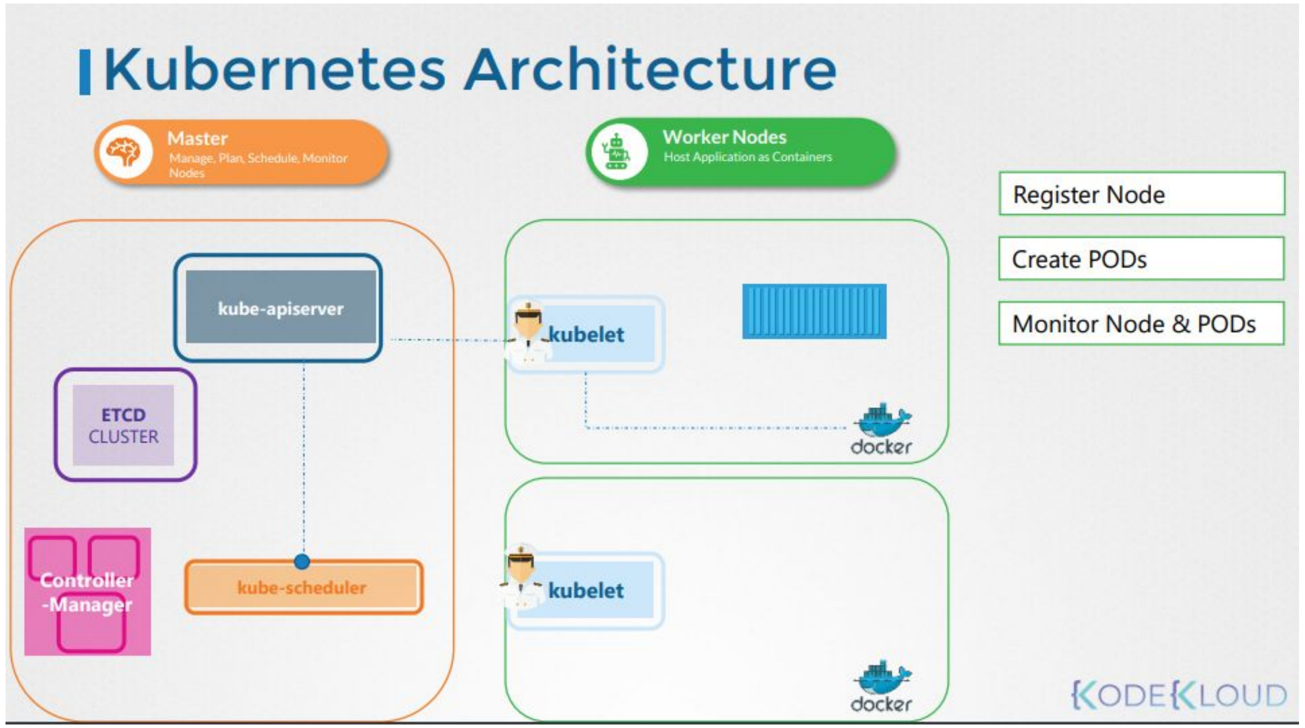
Kube Scheduler

- kube-scheduler, is a component of the Kubernetes control plane responsible for assigning pods to available nodes in the cluster based on resource requirements, placement constraints, and other policies.
- kube-scheduler evaluates various factors such as resource availability, node capacity, affinity/anti-affinity rules, pod interdependencies, and other scheduling constraints to determine the most suitable node for each pod. It aims to balance the workload across the cluster and optimize resource utilization.
- kube-scheduler supports various scheduling policies and strategies. These include spreading pods across different nodes, packing multiple pods onto a single node, or considering affinity and anti-affinity rules to co-locate or separate pods based on node or pod attributes.
- kube-scheduler assesses the fitness of each node in the cluster for hosting a particular pod. It considers factors like available resources (CPU, memory), requested resources by the pod, node conditions, node labels, taints, and tolerations.



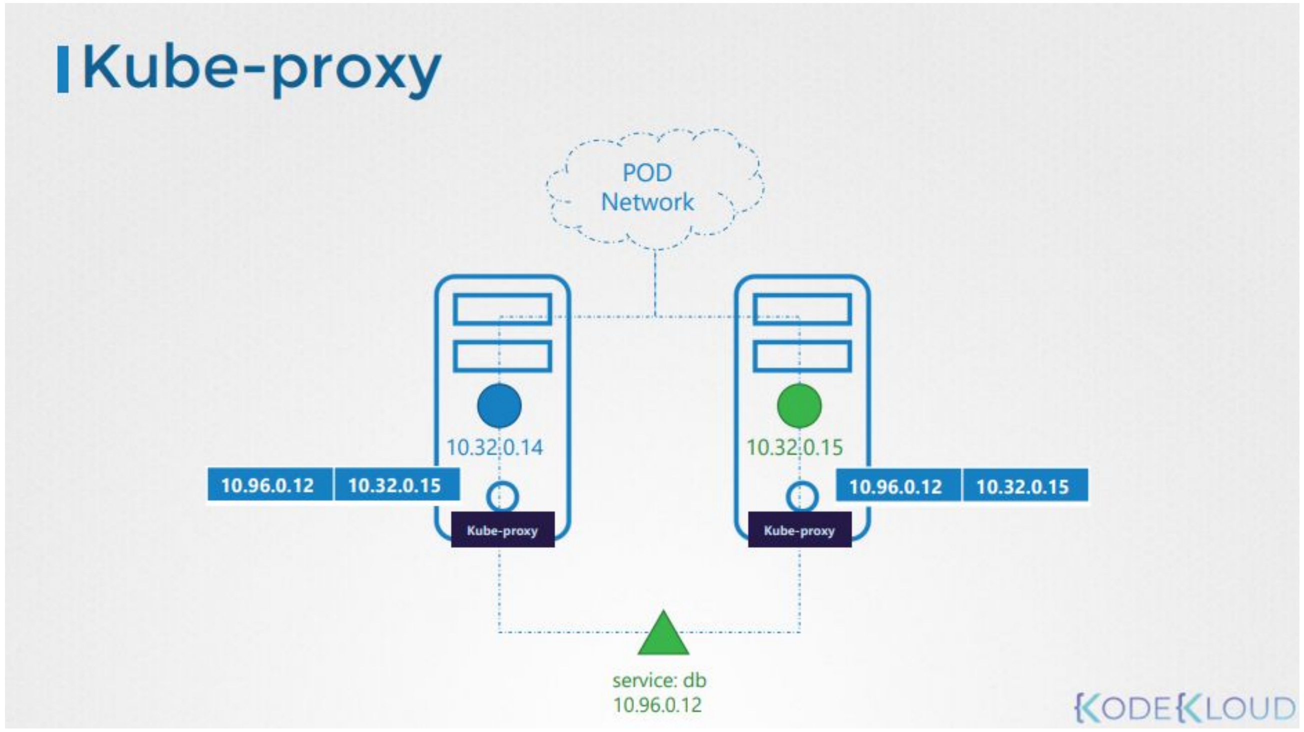
Kubelet

- The kubelet is an essential component of a Kubernetes worker node. It runs on each node in the cluster and is responsible for managing and maintaining the state of individual pods.
- The kubelet is responsible for executing and managing pods on a node. It communicates with the Kubernetes API server to receive instructions and manifests for the pods scheduled on that node.
- The kubelet interacts with the container runtime, such as Docker or containerd, to create and manage containers that make up the pods.
- The kubelet regularly monitors the health of pods and their containers. It determines if the containers are running and ready to accept traffic. If a container or pod fails its health checks, the kubelet takes appropriate action, such as restarting the container or marking the pod as unhealthy.
- The kubelet reports the status of the node to the Kubernetes control plane. It provides information about the node's available resources, capacity, and current conditions. This information is used by the scheduler and other components to make scheduling decisions.
- The kubelet ensures that the required container images are available on the node. It pulls container images from the container registry and caches them locally on the node for efficient use.



Kube-Proxy

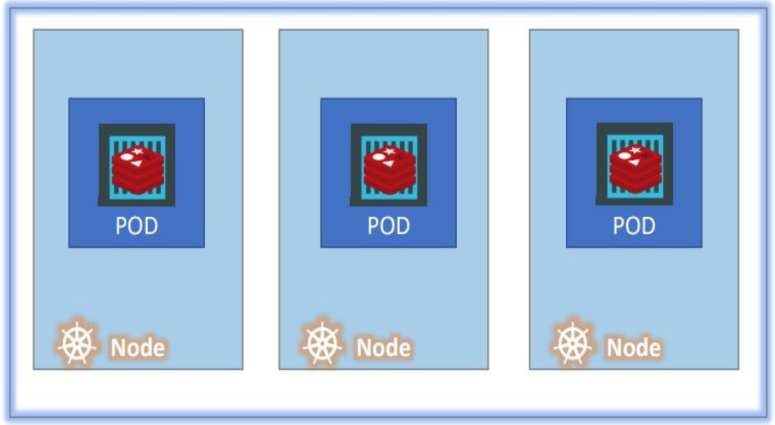
- The kube-proxy is a component of Kubernetes that runs on each node in the cluster. Its primary responsibility is to manage network connectivity to services within the cluster
- The kube-proxy simplifies service access and load balancing within the Kubernetes cluster.
- It ensures that requests are correctly routed to the appropriate pods and provides a seamless and reliable experience for accessing services in the cluster.



Pods

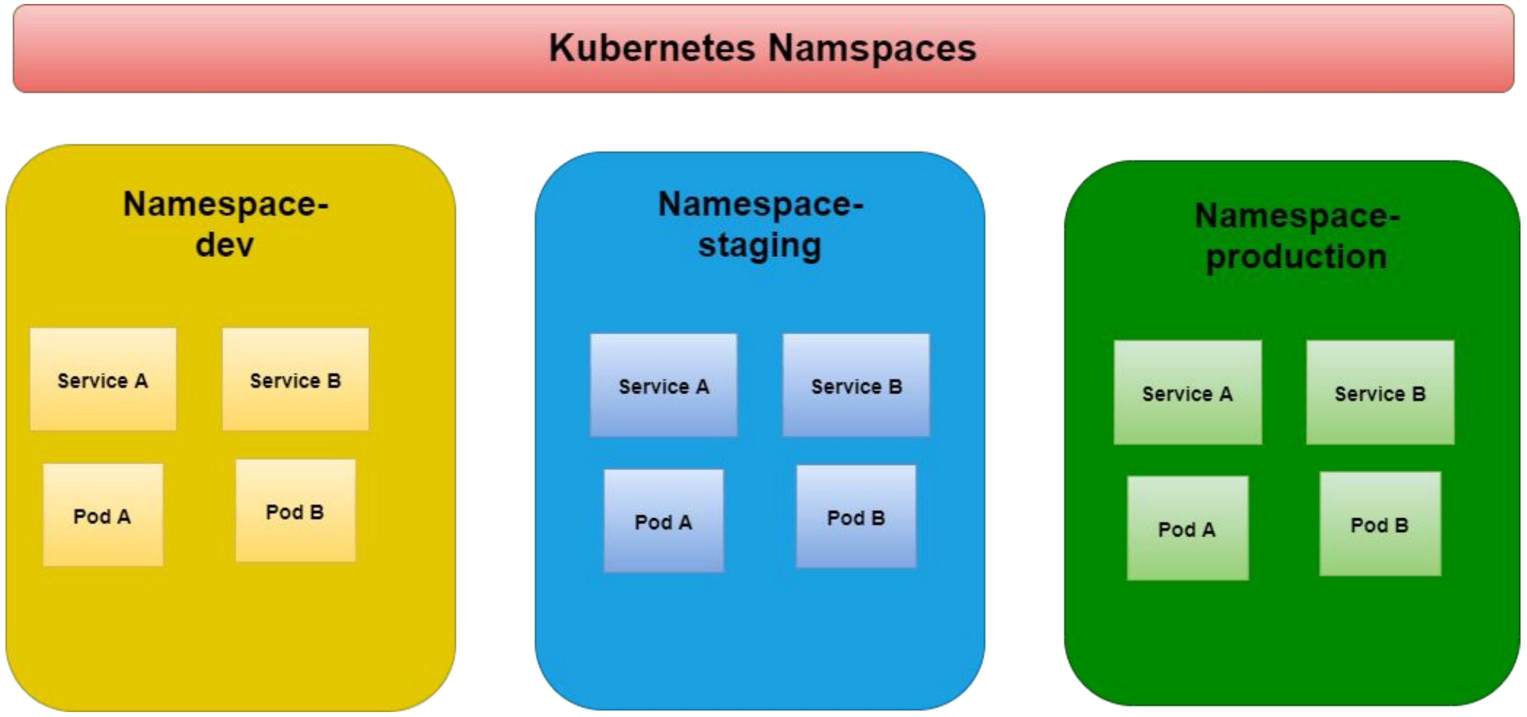
- A pod is the single instance of an application
- A pod is defined as a YAML or JSON manifest that describes the desired state of the pod, including the container images, resource requirements, network settings, environment variables, and other configuration details.
- You can have a single container per pod as well as run multiple containers within a pod
- Pods provide a unit of encapsulation and isolation for application processes and enable efficient resource management, scalability, and resilience within the cluster.
- Pods have their lifecycle managed by the Kubernetes control plane. They can be created, scheduled to nodes, started, stopped, and restarted based on the desired state defined in the pod specification.
- Pods are not self-healing; if a pod fails, it is terminated and a new one is created to replace it

POD



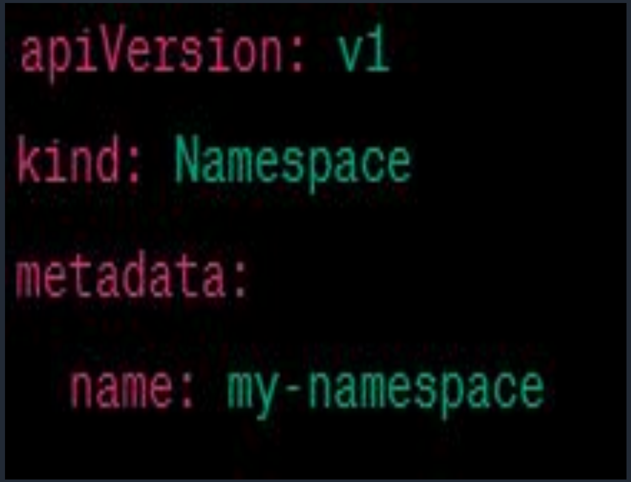
K8S Namepaces

Namespaces in K8s is a logical virtual cluster that provides a way to divide and isolate resources within a cluster. It is used to group and organize objects such as pods, services, deployments, and other Kubernetes resources.



```
kubectl api-resources --namespaced=false # list resources that are not namespace bound
kubectl api-resources --namespaced=true # list resources that are namespace bound
```

K8S Namespace Types



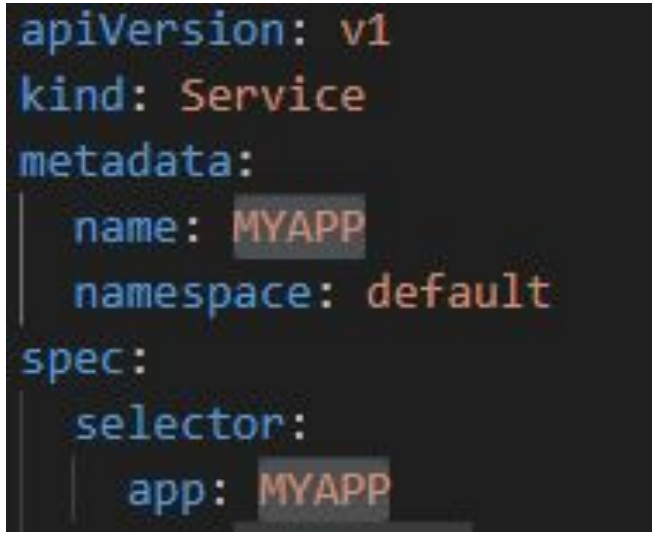
```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

- Default Namespace:
  - The "default" namespace is created automatically in every Kubernetes cluster.
  - If a resource is not explicitly specified with a namespace, it is created in the default namespace by default.
  - This namespace is typically used for resources that are not specific to any particular namespace.
- System Namespaces:
  - Kubernetes provides a set of system namespaces for managing core components and services.
  - Examples of system namespaces include "kube-system" and "kube-public".
  - The "kube-system" namespace hosts essential cluster components like kube-dns, kube-proxy, and the Kubernetes API server. it is advised NOT TO CREATE OR MODIFY ANYTHING namespace.
  - The "kube-public" namespace is readable by all users and can be used for exposing cluster information or resources.
  - **kube-node-lease:** This namespace determines the availability of a node.
- Custom Application Namespaces:
  - These namespaces are created by users or administrators to isolate resources for specific applications, teams, or projects.
  - Custom namespaces are typically used to group related resources and provide logical separation within a cluster.
  - Examples of custom namespaces could be "development", "production", "frontend", "backend", etc.

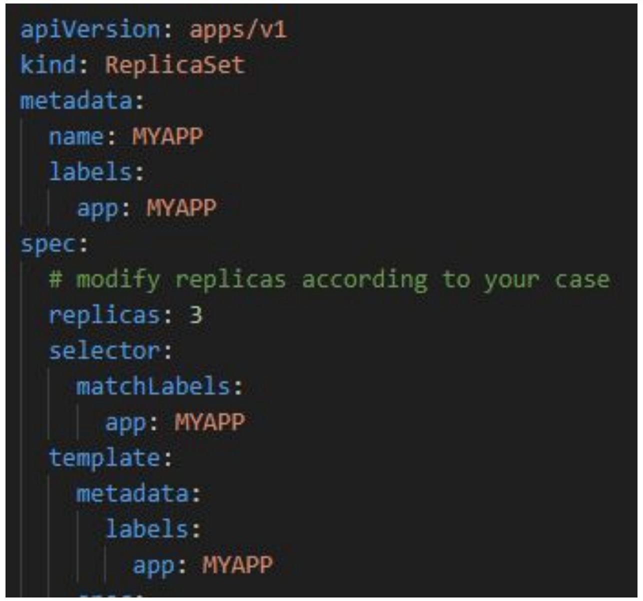
- Readmore: https://www.geeksforgeeks.org/kubernetes-namespaces/

Labels and Selectors

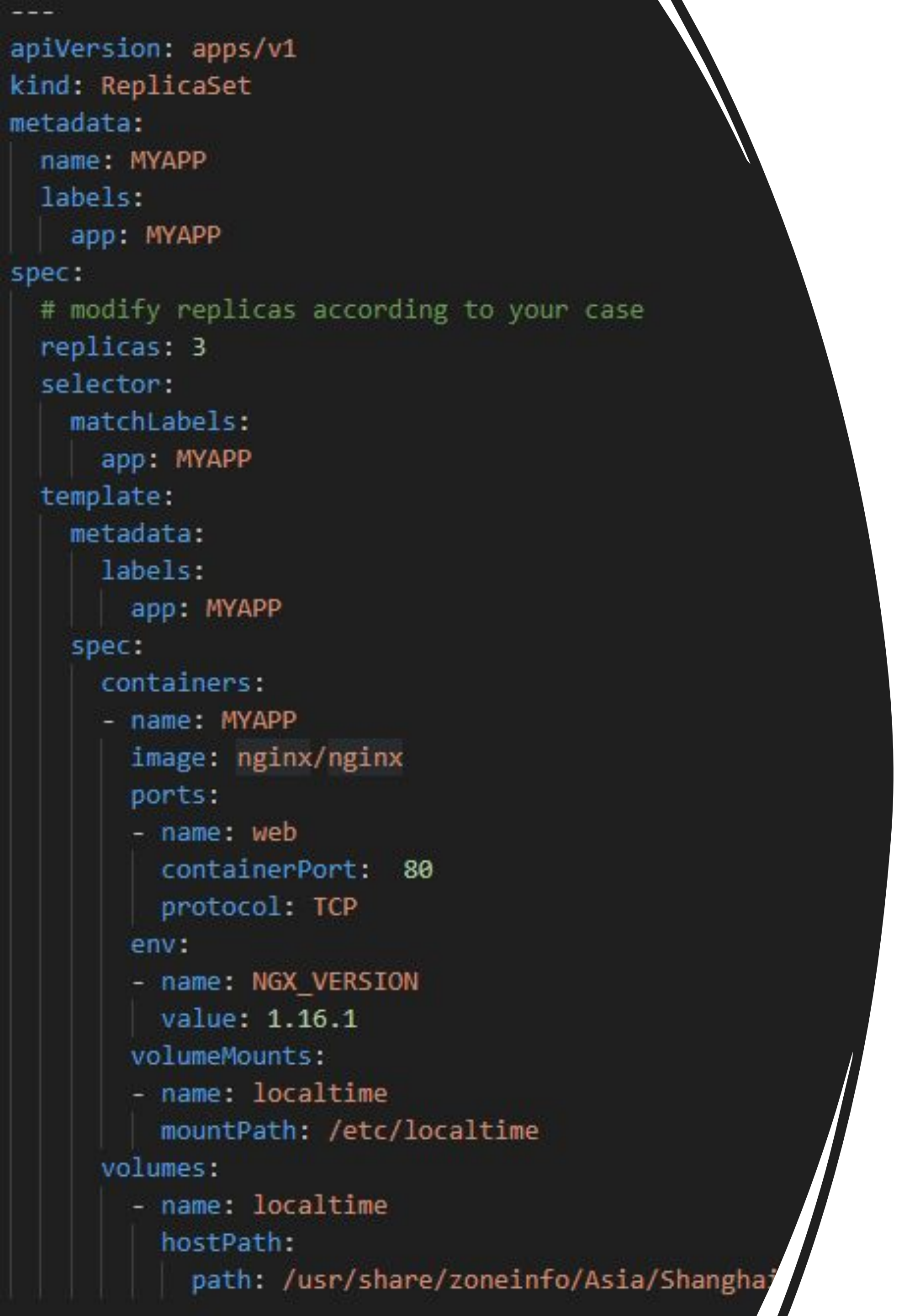
- In Kubernetes, labels and selectors are used to identify and group resources.
- Labels are key-value pairs attached to Kubernetes objects, such as pods, services, and deployments etc … .
- You can attach multiple labels to an object, and labels can be used for various purposes, such as identifying environments, versions, roles, or any other meaningful categorization.
- Selectors are used to query and select resources based on their labels.



```yaml
apiVersion: v1
kind: Service
metadata:
  name: MYAPP
  namespace: default
spec:
  selector:
    app: MYAPP
```



```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: MYAPP
  labels:
    app: MYAPP
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      app: MYAPP
  template:
    metadata:
      labels:
        app: MYAPP
```

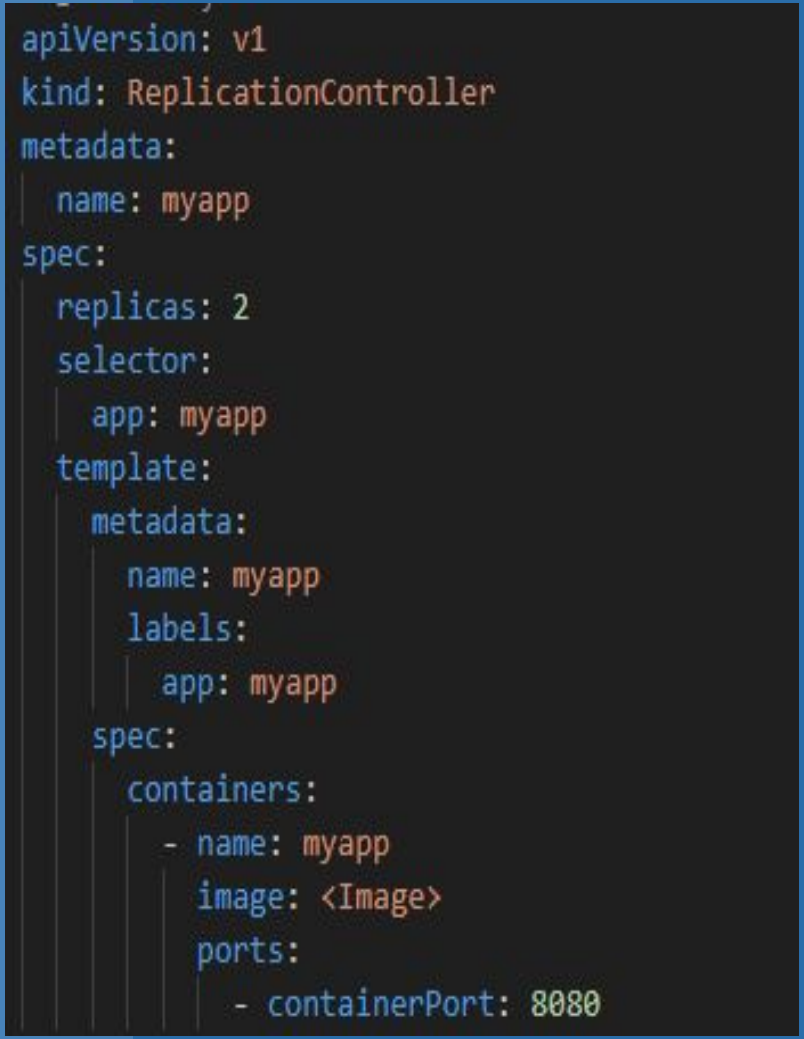


```yaml
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: MYAPP
  labels:
    app: MYAPP
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      app: MYAPP
  template:
    metadata:
      labels:
        app: MYAPP
    spec:
      containers:
      - name: MYAPP
        image: nginx/nginx
        ports:
        - name: web
          containerPort:  80
          protocol: TCP
        env:
        - name: NGX_VERSION
          value: 1.16.1
        volumeMounts:
        - name: localtime
          mountPath: /etc/localtime
      volumes:
      - name: localtime
        hostPath:
          path: /usr/share/zoneinfo/Asia/Shanghai
```

ReplicaSets and Replication Controllers

- ReplicaSets and Replication Controllers are two Kubernetes objects used for managing the replication and scaling of pods. While they serve similar purposes, ReplicaSets are the recommended approach for managing pod replication in newer Kubernetes versions.
- Readmore: https://www.geeksforgeeks.org/kubernetes-replication-controller/

ReplicaSets and Replication Controllers



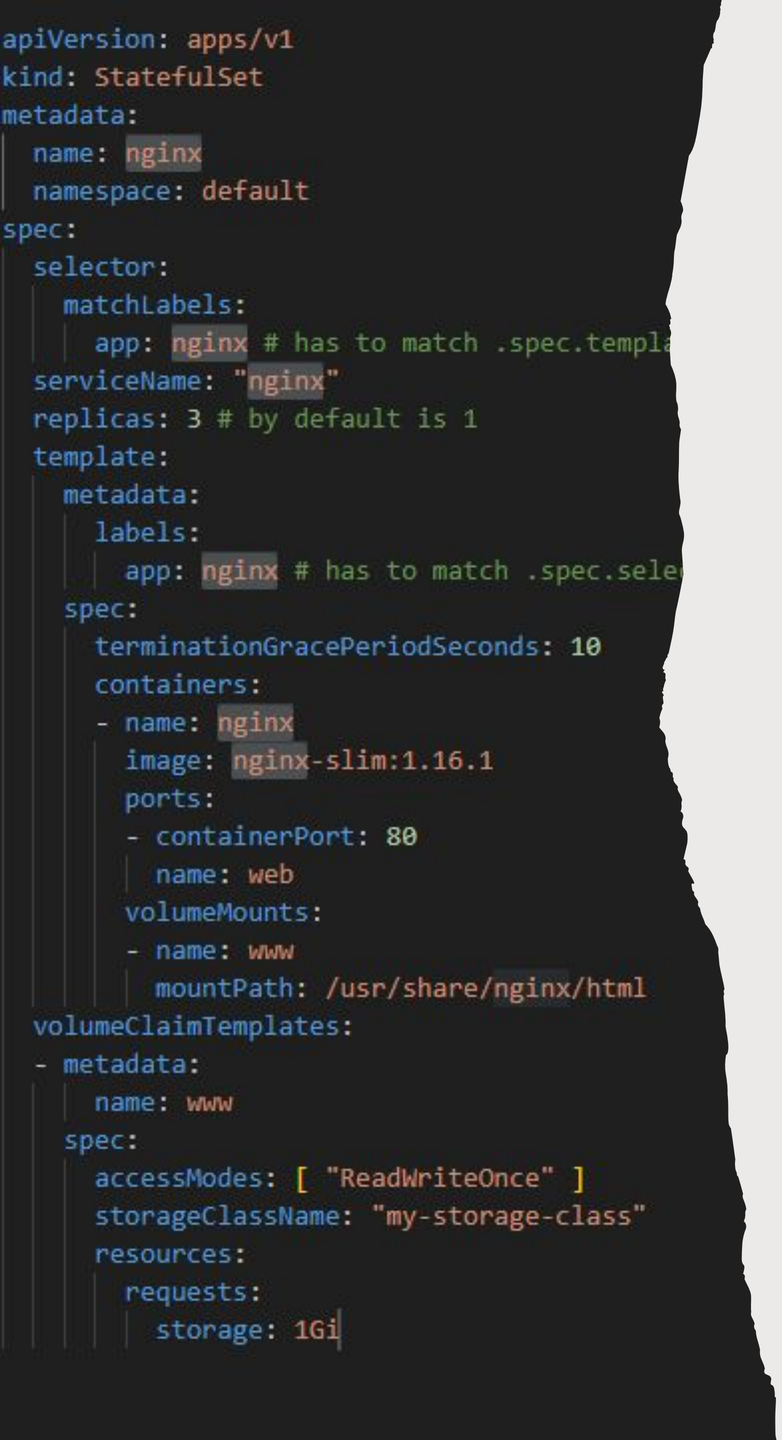
```
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp
spec:
  replicas: 2
  selector:
    app: myapp
  template:
    metadata:
      name: myapp
      labels:
        app: myapp
    spec:
      containers:
        - name: myapp
          image: <Image>
          ports:
            - containerPort: 8080
```

Replication Controllers:

- Replication Controllers ensure that a specified number of pod replicas are running at all times.
- They are responsible for maintaining the desired state by creating or deleting pods as needed to match the desired replica count.
- Replication Controllers use labels and selectors to identify and manage the pods associated with them.
- However, ReplicaSets have largely replaced Replication Controllers in newer Kubernetes versions.

ReplicaSets:

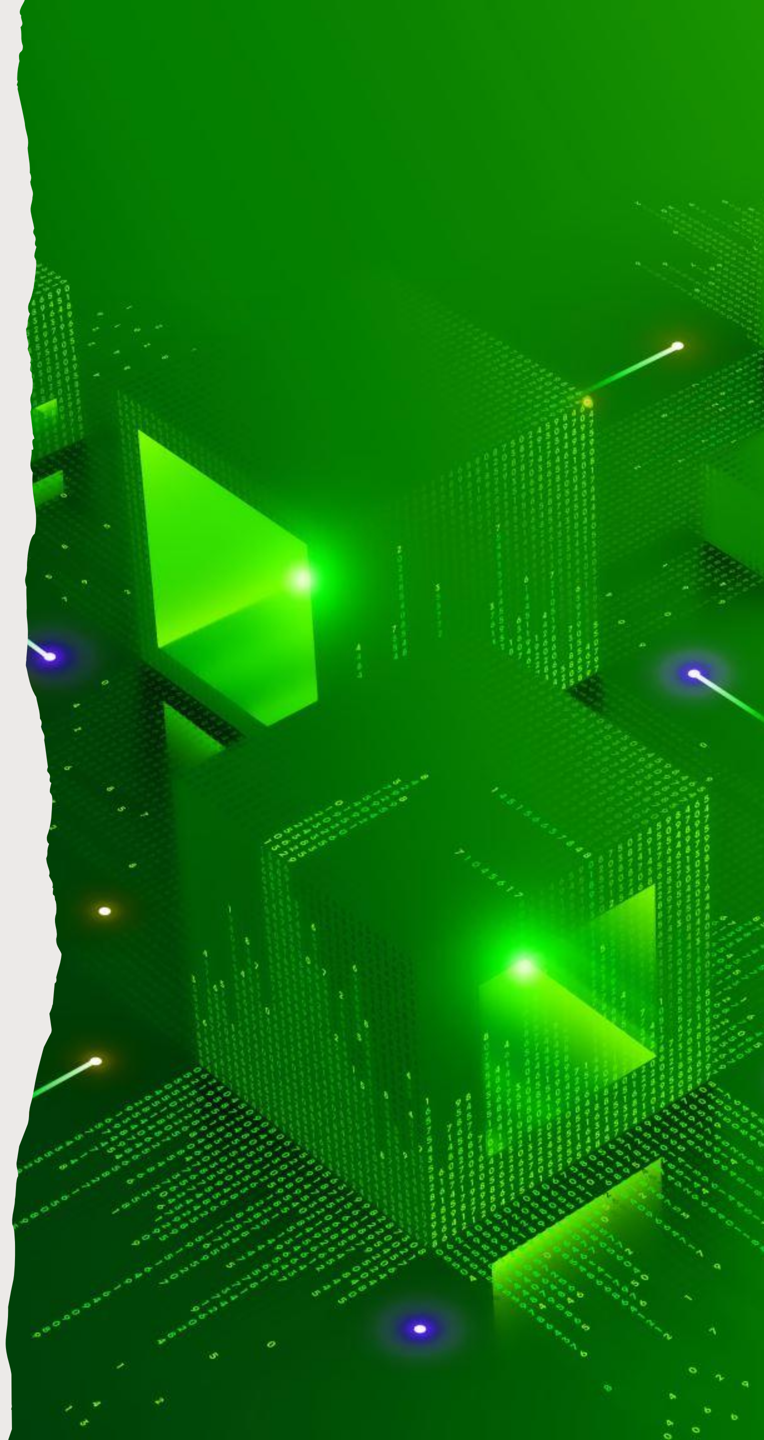
- ReplicaSets are the successor to Replication Controllers and provide more advanced functionality.
- Like Replication Controllers, ReplicaSets ensure a specified number of pod replicas are running.
- ReplicaSets offer enhanced label-based selectors and support more complex pod matching expressions.
- They also support more flexible rolling updates and scaling strategies compared to Replication Controllers.
- ReplicaSets are part of the Kubernetes Deployments API, making them an integral part of application deployment and management.

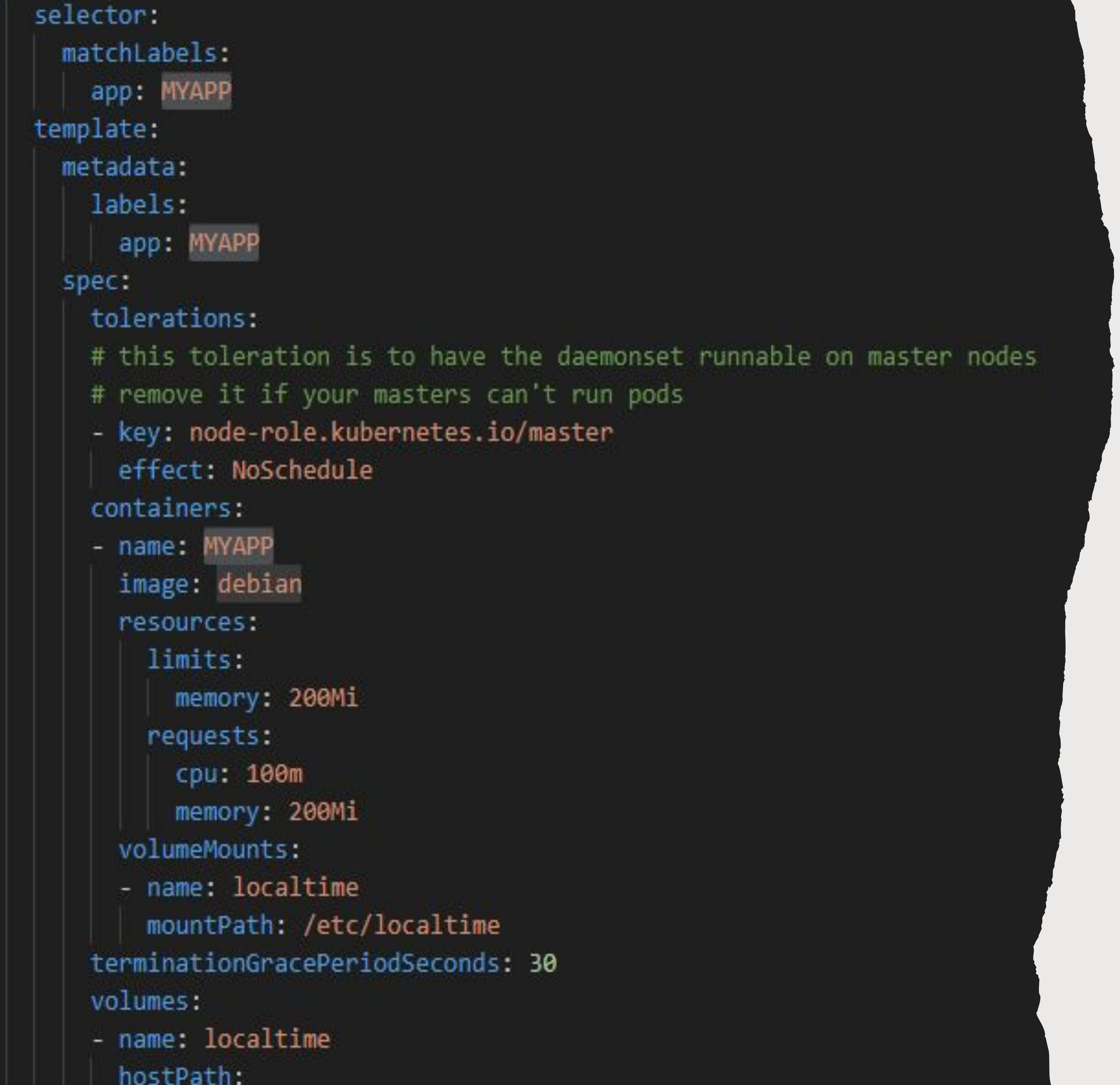


```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
  namespace: default
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.templa
  serviceName: "nginx"
  replicas: 3 # by default is 1
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.sele
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: nginx-slim:1.16.1
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

Statefulsets

- StatefulSets are a Kubernetes object designed for managing stateful applications that require stable network identities and persistent storage.
- StatefulSets are well-suited for applications such as databases, message queues, and distributed systems that require stable network identities and persistent storage.
- StatefulSets provide the necessary guarantees for maintaining state and data consistency in a Kubernetes environment, allowing stateful applications to run effectively and reliably

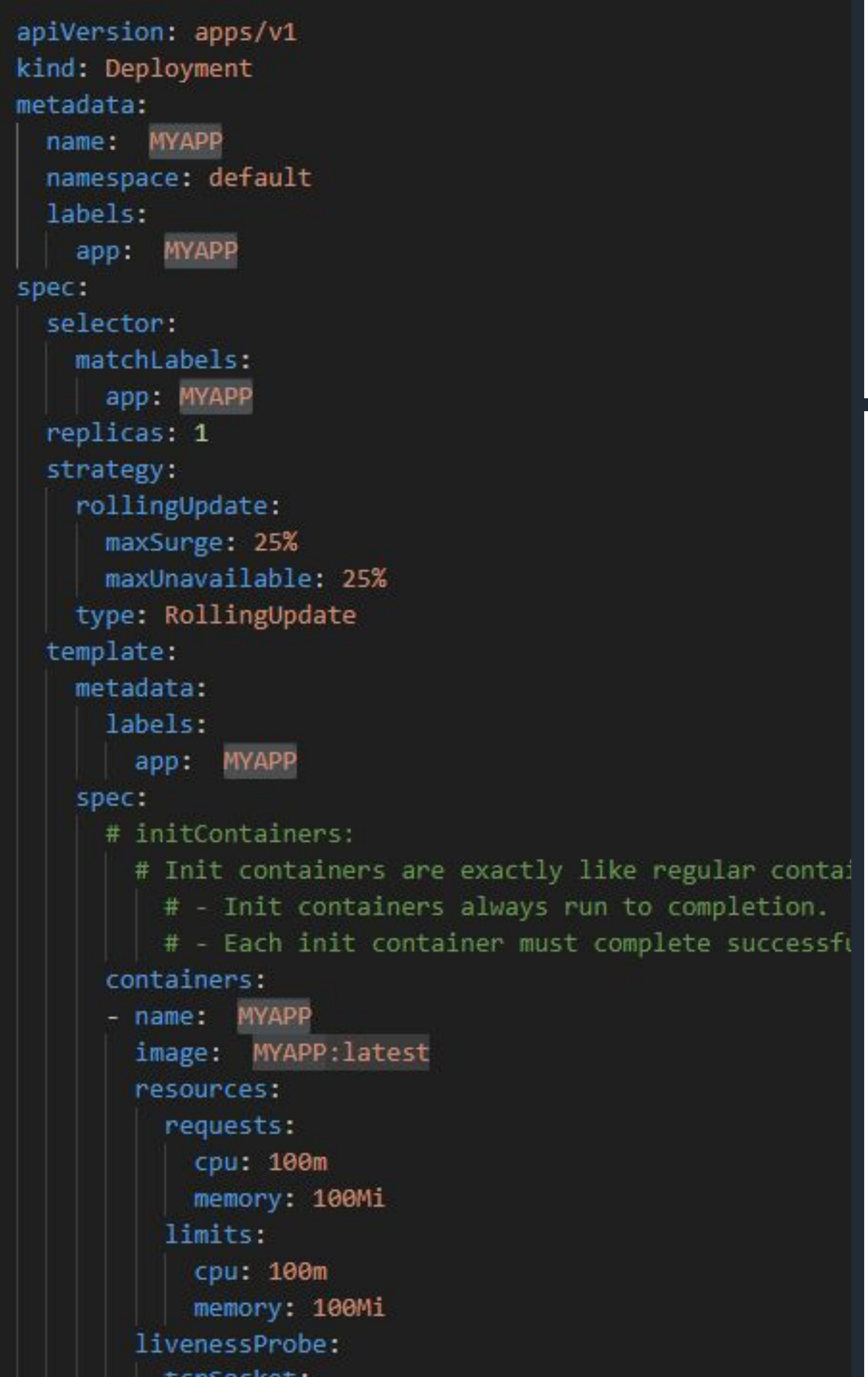




```yaml
selector:
  matchLabels:
    app: MYAPP
template:
  metadata:
    labels:
      app: MYAPP
  spec:
    tolerations:
    # this toleration is to have the daemonset runnable on master nodes
    # remove it if your masters can't run pods
    - key: node-role.kubernetes.io/master
      effect: NoSchedule
    containers:
    - name: MYAPP
      image: debian
      resources:
        limits:
          memory: 200Mi
        requests:
          cpu: 100m
          memory: 200Mi
      volumeMounts:
      - name: localtime
        mountPath: /etc/localtime
    terminationGracePeriodSeconds: 30
    volumes:
    - name: localtime
      hostPath:
```

DaemonSets

- DaemonSets in Kubernetes are a type of workload object used to ensure that a specific pod is running on every node in the cluster.
- Daemonsets are primarily used for deploying system-level daemons or infrastructure services that should be available on all nodes, such as log collectors, monitoring agents, or network proxies.



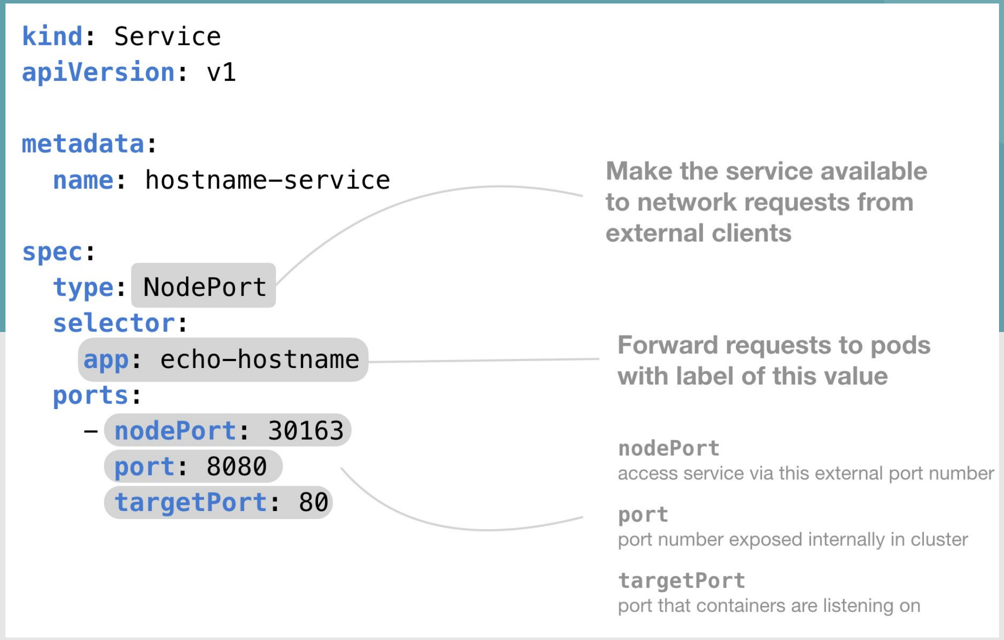
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  MYAPP
  namespace: default
  labels:
    app:  MYAPP
spec:
  selector:
    matchLabels:
      app: MYAPP
  replicas: 1
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app:  MYAPP
    spec:
      # initContainers:
        # Init containers are exactly like regular contai
          # - Init containers always run to completion.
          # - Each init container must complete successfu
      containers:
      - name:  MYAPP
        image:  MYAPP:latest
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
          limits:
            cpu: 100m
            memory: 100Mi
        livenessProbe:
```

Deployments

- In Kubernetes, Deployments are a high-level object used to manage the deployment and scaling of applications.
- Deployments provide a declarative way to define the desired state of the application and ensure that the desired number of replicas (pods) are running and available at all times.
- Deployments provide a robust and flexible way to manage the lifecycle of applications in Kubernetes. They abstract away the complexities of managing ReplicaSets, rolling updates, and scaling, making it easier to deploy and manage applications with built-in self-healing and rollback capabilities.
- Deployments are widely used for managing stateless applications or components of stateful applications that can be easily scaled and updated.

K8S Service

- A Kubernetes service enables communication between various components within a Kubernetes cluster.
- Services provide a stable network identity and an endpoint that other applications can use to interact with pods running in the cluster.
- Services abstract away the underlying details of individual pod IP addresses and provide a single access point for clients.
- Once you create a service, other components within the cluster can use the service name and port to communicate with the pods behind the service, regardless of their dynamic IP addresses



```yaml
kind: Service
apiVersion: v1

metadata:
  name: hostname-service

spec:
  type: NodePort
  selector:
    app: echo-hostname
  ports:
    - nodePort: 30163
      port: 8080
      targetPort: 80
```

Make the service available to network requests from external clients

Forward requests to pods with label of this value

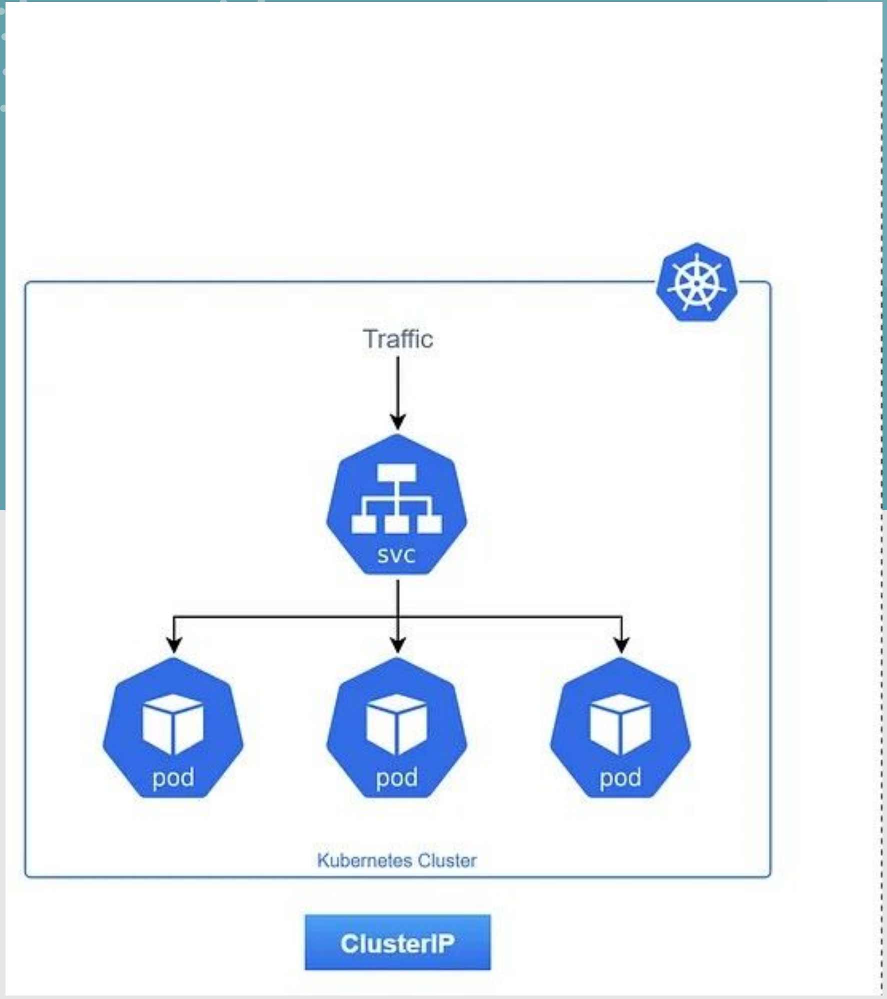
`nodePort`
access service via this external port number

`port`
port number exposed internally in cluster

`targetPort`
port that containers are listening on

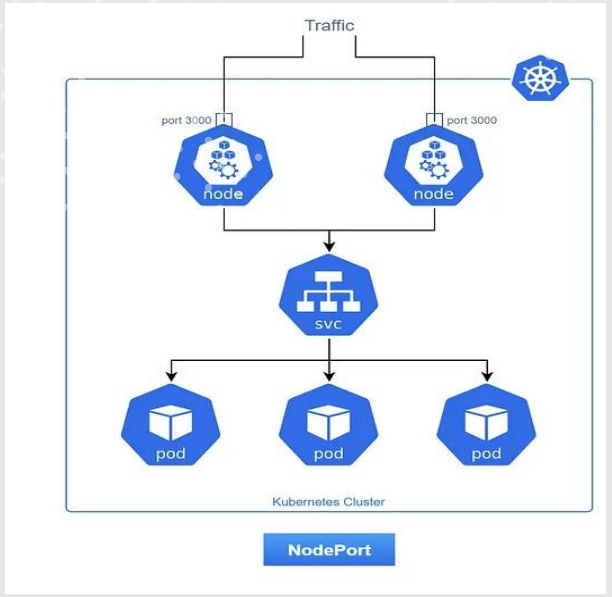
K8S SERVICE TYPES: CLUSTER IP

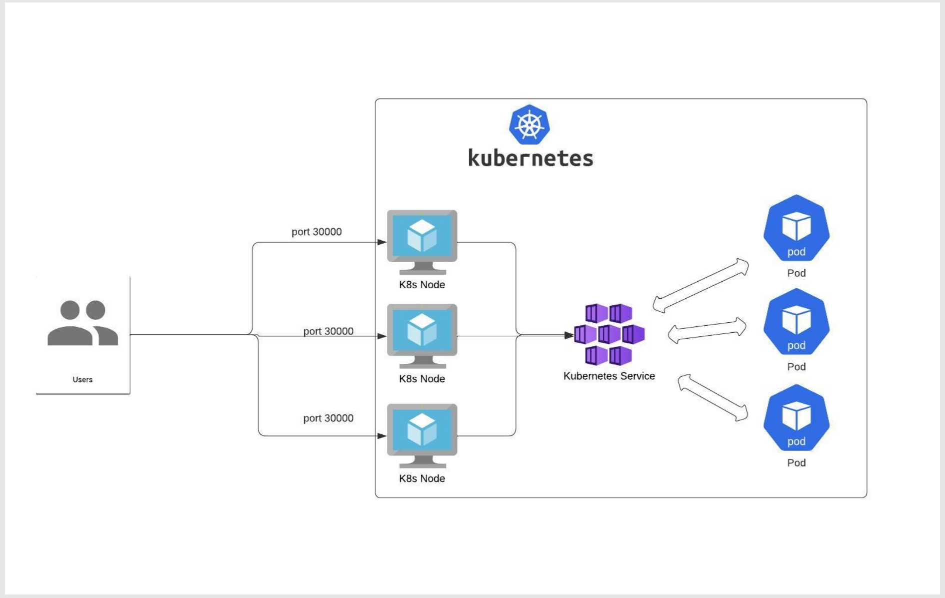
This is the default service type. It assigns a stable IP address to the service within the cluster. The service is only accessible from within the cluster, making it suitable for internal communication between pods.



NODEPORT

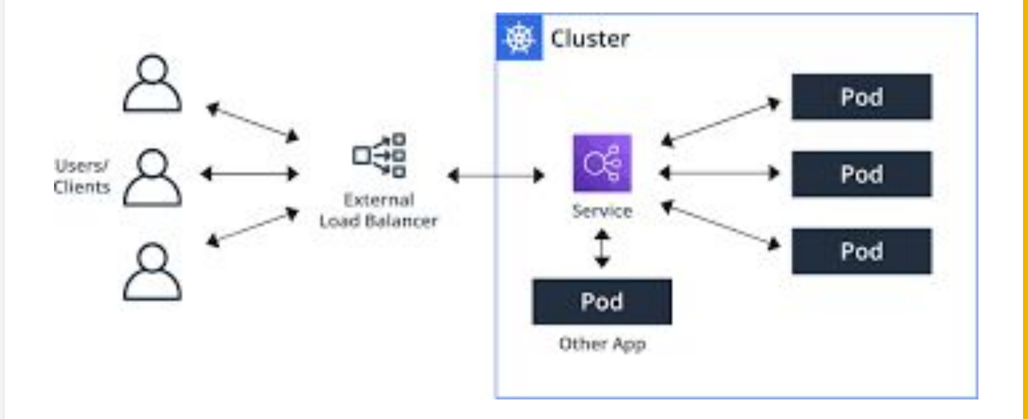
This service type exposes the service on a static port on each node of the cluster. It enables access to the service from outside the cluster by forwarding traffic to the appropriate pods. The allocatable port range is 30000 - 32767

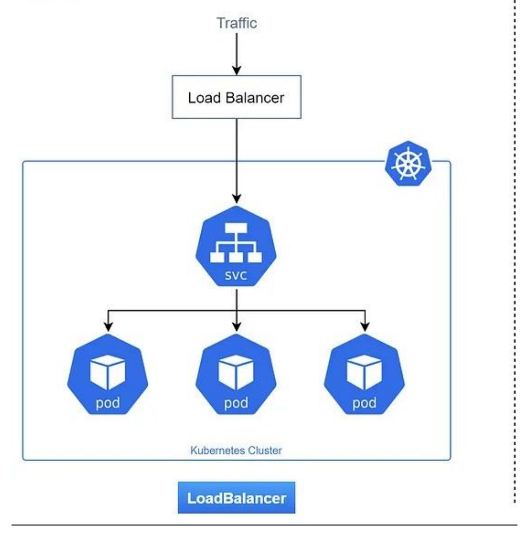




LOADBALANCER

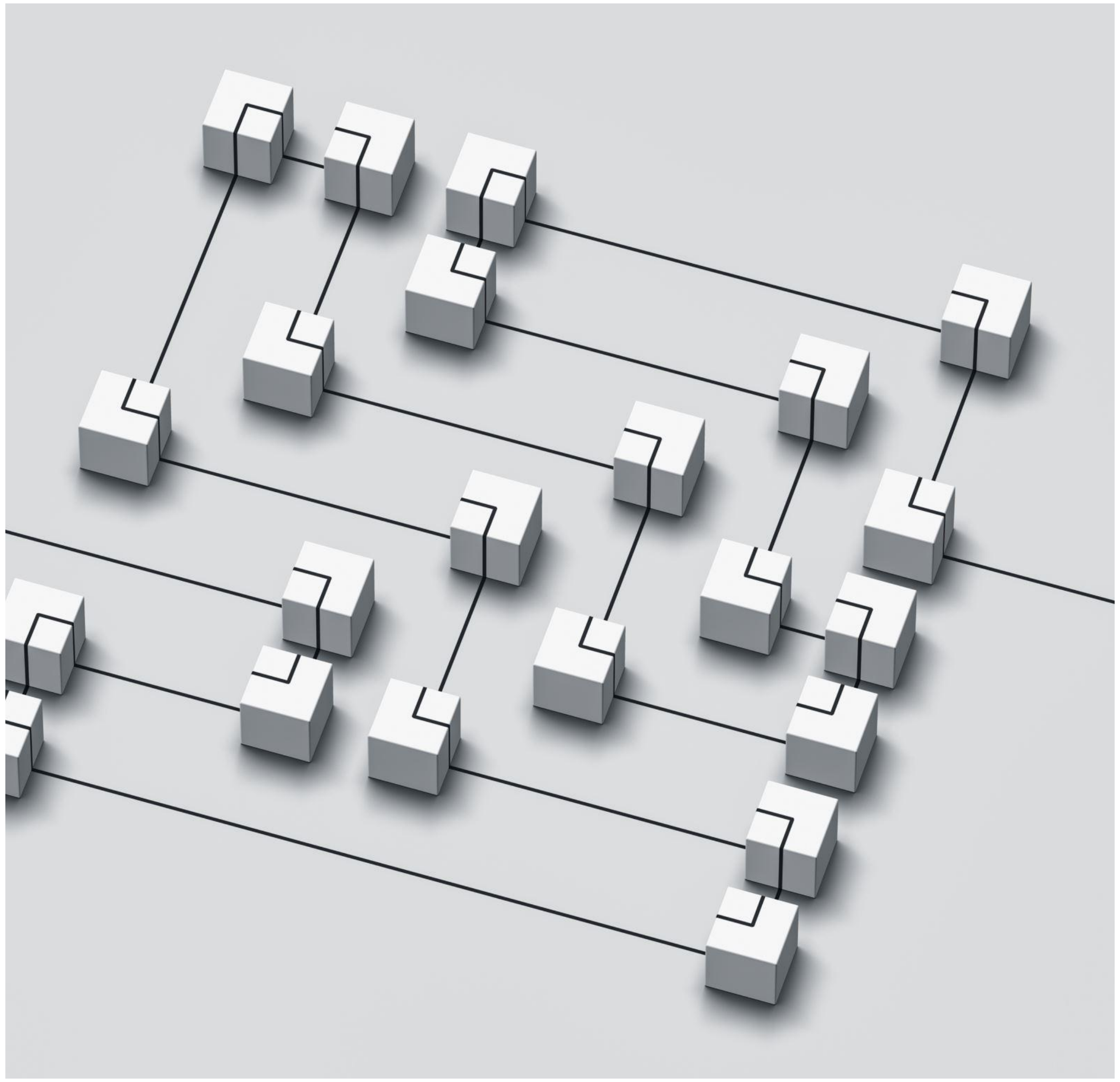
This type creates an external load balancer in the underlying infrastructure (e.g., cloud provider) and forwards traffic to the service. It provides external access to the service, typically by assigning an external IP address.





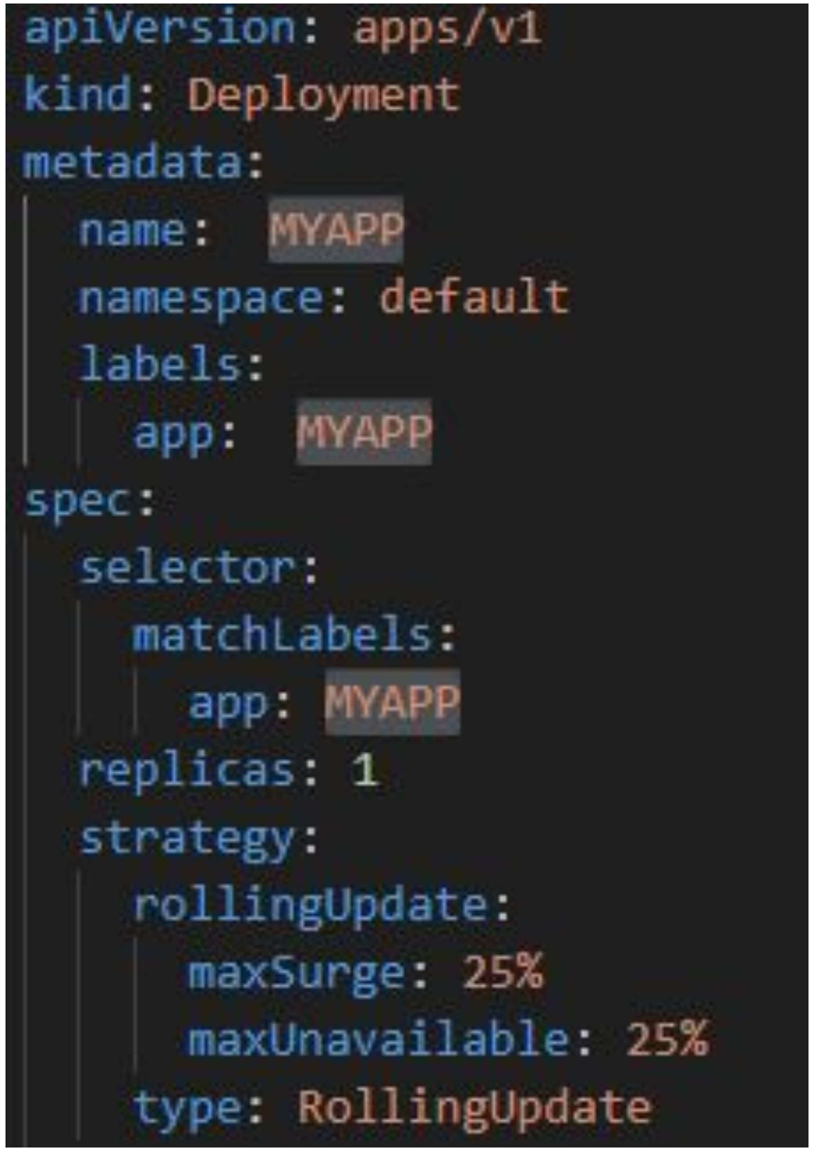
Deployment Strategies

- Kubernetes provides several deployment strategies that define how new versions of applications are rolled out and updated while ensuring high availability and minimal disruption. These deployment strategies offer different trade-offs in terms of deployment speed, downtime, and resource utilization.



Deployment Strategy Types

1. **Rolling Update:** This is the default deployment strategy in Kubernetes. Rolling updates gradually replace old pods with new ones, ensuring that a certain number of pods are available and healthy at all times. It allows for controlled and gradual updates, minimizing downtime and maintaining application availability during the update process.
2. **Blue-Green Deployment:** In a blue-green deployment, two identical environments, referred to as blue and green, are created. The blue environment represents the current production version, while the green environment is the new version. Traffic is initially directed to the blue environment. Once the green environment is fully deployed and validated, traffic is switched to the green environment, effectively swapping the production traffic from blue to green.
3. **Canary Deployment:** Canary deployment involves rolling out a new version of an application to a small subset of users or traffic. The canary release allows for testing the new version in a production-like environment with limited impact. If the canary deployment is successful, the new version can be gradually rolled out to a larger audience or traffic. If issues are detected, the deployment can be easily rolled back.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name:  MYAPP
  namespace: default
  labels:
    app:  MYAPP
spec:
  selector:
    matchLabels:
      app: MYAPP
  replicas: 1
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
```

K8S INGRESS

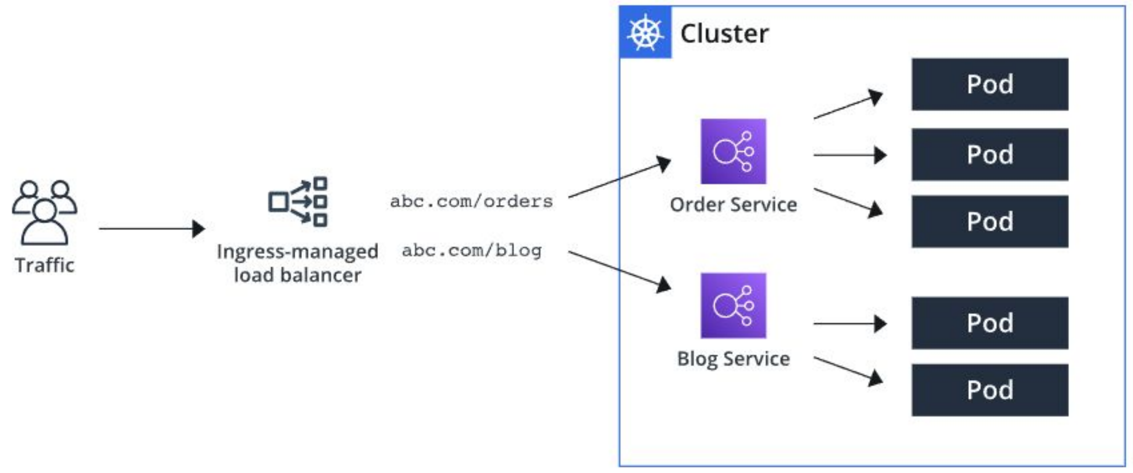
- Kubernetes Ingress manages external access to services within a Kubernetes cluster. It provides a way to configure and manage HTTP and HTTPS routes, load balancing, SSL/TLS termination, and other traffic routing rules for applications running in the cluster.
- Ingress acts as an entry point for incoming traffic and allows you to define rules for routing requests to different services based on the requested hostnames, paths, or other criteria.
- To use Ingress, you need an Ingress controller, which is a separate component responsible for implementing the Ingress rules and managing the underlying load balancer.
- Ingress controllers can be deployed as a separate Kubernetes deployment or as a managed service provided by cloud providers.

Details here:

https://kubernetes-sigs.github.io/aws-load-balancer-controller/v2.2/guide/ingress/annotations/

https://kubernetes-sigs.github.io/aws-load-balancer-controller/v2.2/guide/ingress/annotations/

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

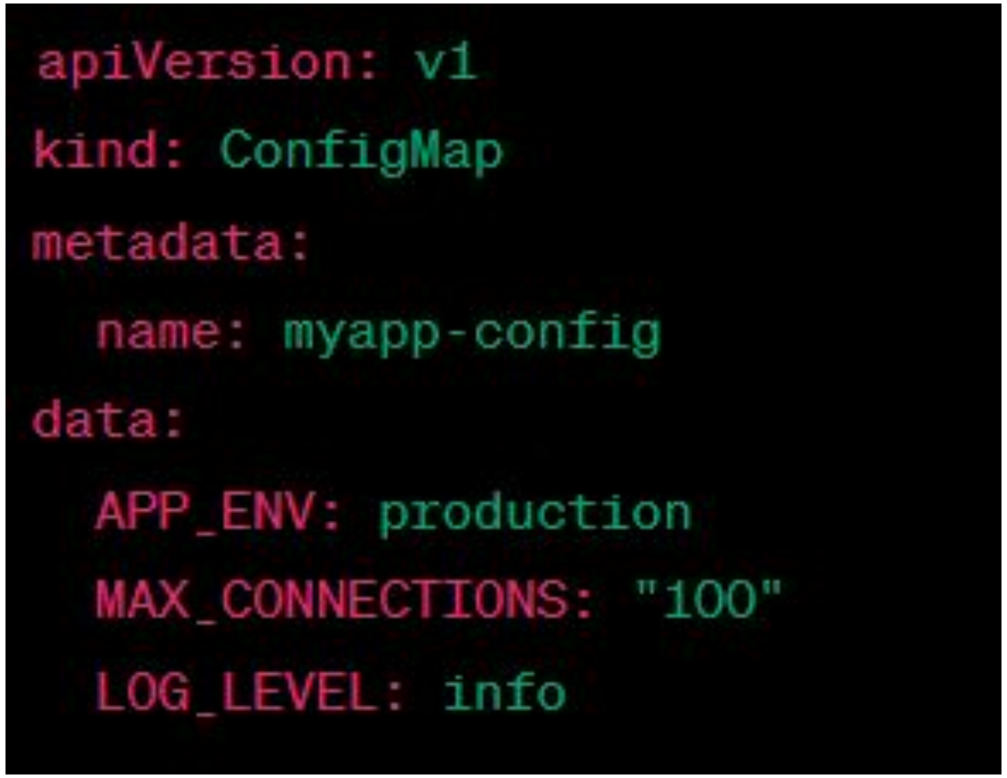


ConfigMaps and Secrets

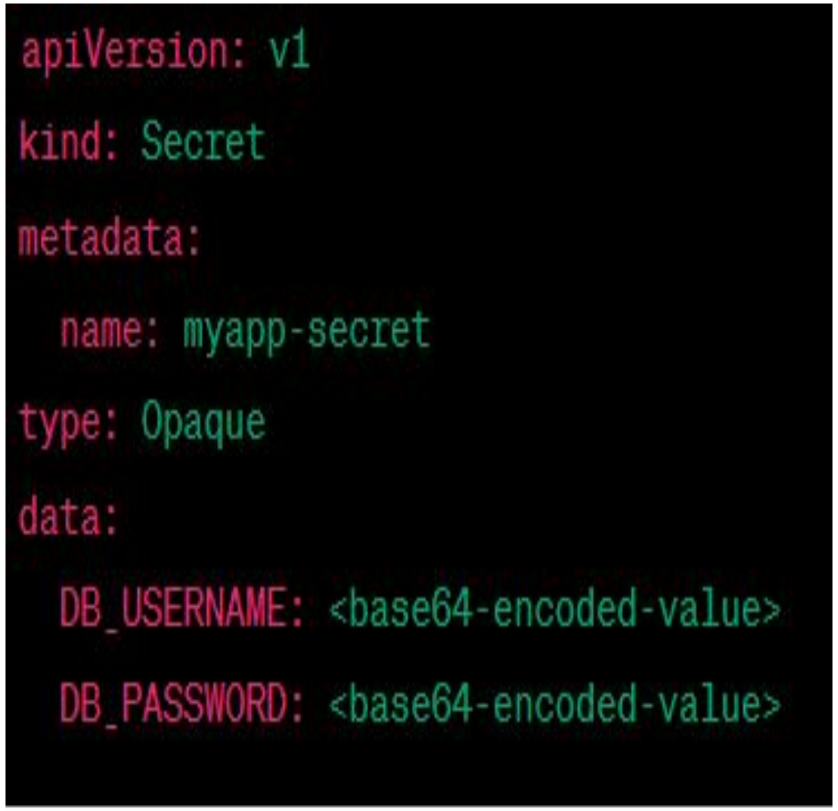
- In Kubernetes, ConfigMaps and Secrets are used to manage configuration data and sensitive information, respectively.
- They provide a way to decouple configuration and sensitive data from your application code, making it easier to manage and maintain.
- You can reference ConfigMaps or Secrets in your pod or deployment YAML files to provide configuration or sensitive data to your application containers.
- Remember to handle Secrets with care, as they store sensitive information. It's important to ensure appropriate access controls and encryption mechanisms are in place to protect the Secrets within your Kubernetes cluster.

ConfigMaps

- ConfigMaps are used to store non-sensitive configuration data in key-value pairs.
- ConfigMaps can be created manually or from configuration files, and they can be used to configure applications at runtime.
- ConfigMaps can store data such as environment variables, command-line arguments, configuration files, or any other configuration data needed by your application.
- ConfigMaps can be mounted as volumes in pods or accessed as environment variables within containers.
- ConfigMaps are stored in plain text and are typically used for non-sensitive data that can be shared across multiple pods or containers.



```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: myapp-config
data:
  APP_ENV: production
  MAX_CONNECTIONS: "100"
  LOG_LEVEL: info
```



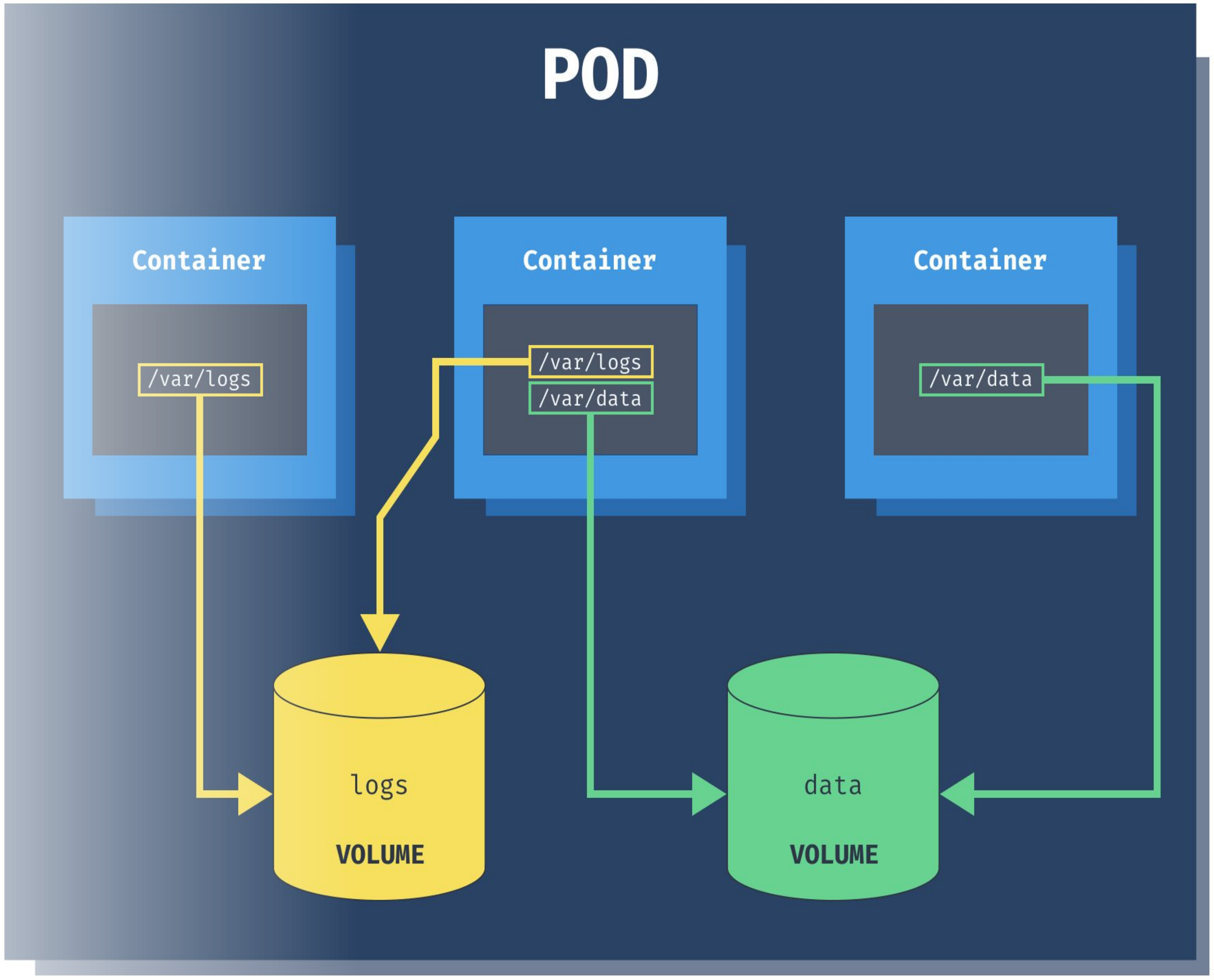
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: myapp-secret
type: Opaque
data:
  DB_USERNAME: <base64-encoded-value>
  DB_PASSWORD: <base64-encoded-value>
```

Secrets

- Secrets are used to store sensitive information, such as passwords, API keys, or TLS certificates.
- Secrets are similar to ConfigMaps but provide an extra layer of security by encoding and storing the data in a base64-encoded format.
- Secrets can be created manually or from files, and they can be used to securely pass sensitive data to your application.
- Secrets can be mounted as volumes in pods or accessed as environment variables within containers, similar to ConfigMaps.
- Secrets are designed to be used for sensitive data that should be encrypted at rest and in transit.

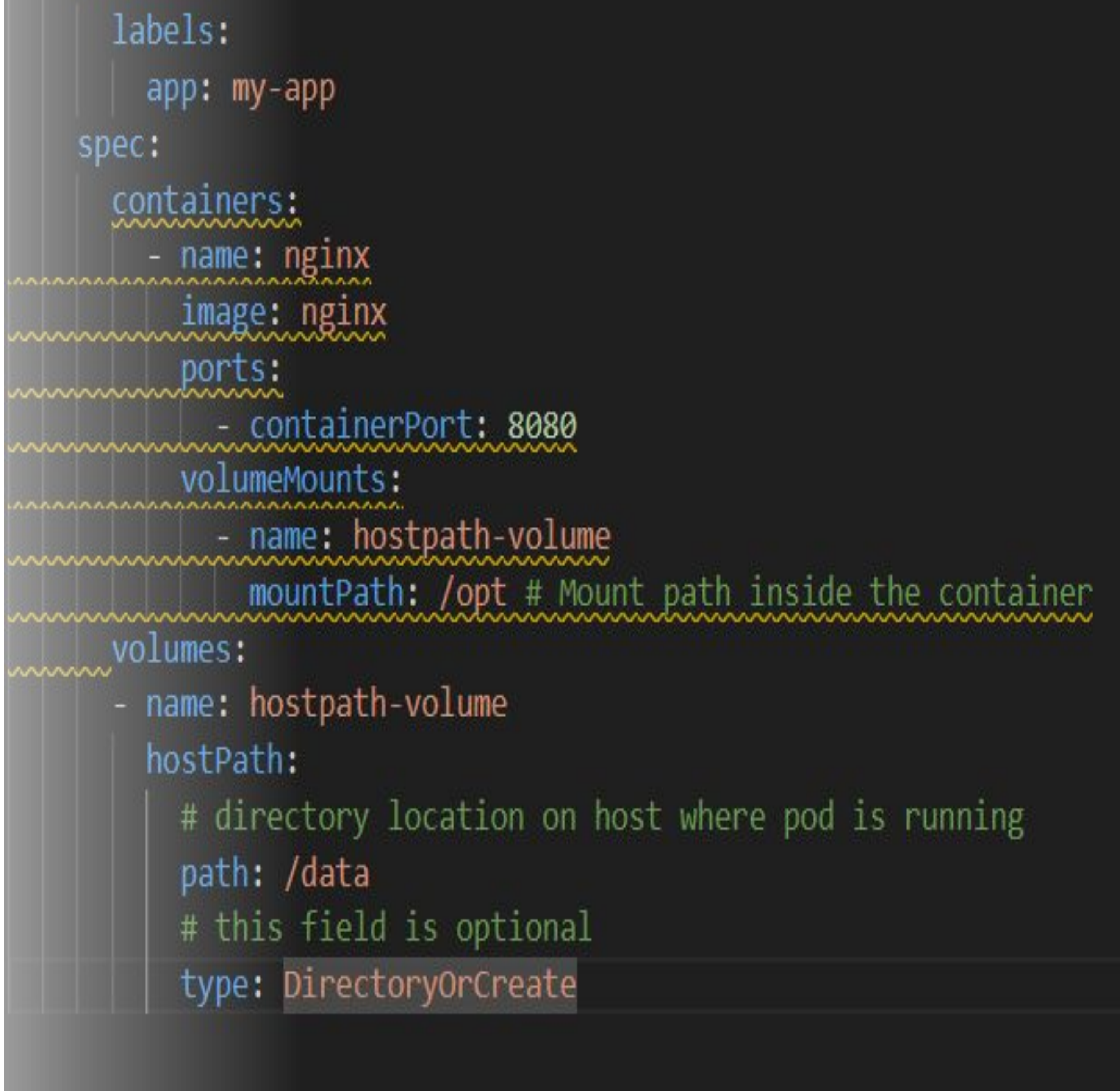
K8s Volumes

In Kubernetes, volumes are used to provide persistent storage for containers running within pods. Volumes decouple the lifecycle of the data from the lifecycle of the pod, allowing data to persist even if the pod is terminated or restarted.



HostPath Volume Type

A HostPath volume mounts a file or directory from the host node's filesystem into the pod. It allows containers to access files or directories on the underlying node. Note that using HostPath volumes can limit pod portability and is generally not recommended in production environments.



```yaml
  labels:
    app: my-app
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 8080
      volumeMounts:
        - name: hostpath-volume
          mountPath: /opt # Mount path inside the container
  volumes:
    - name: hostpath-volume
      hostPath:
        # directory location on host where pod is running
        path: /data
        # this field is optional
        type: DirectoryOrCreate
```

Container Storage Interfaces

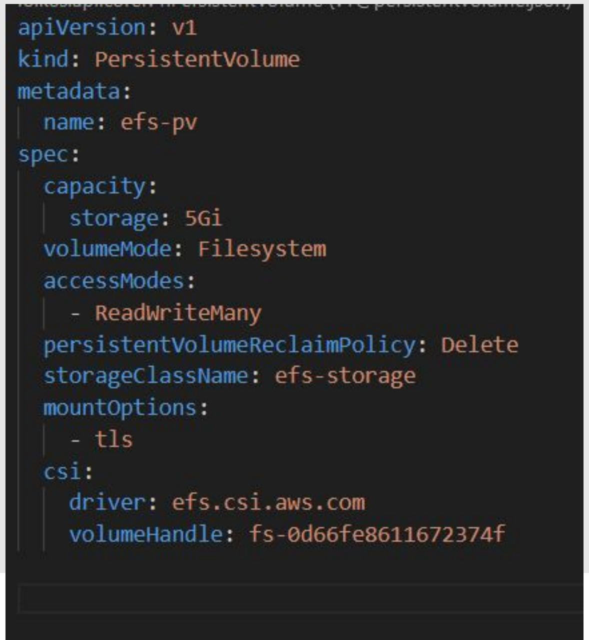
CSI (Container Storage Interface) volumes allow integration with external storage systems and enable the use of storage plugins provided by third-party vendors. CSI volumes provide a standardized way to provision and manage storage resources in Kubernetes. E.g EFS

https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/

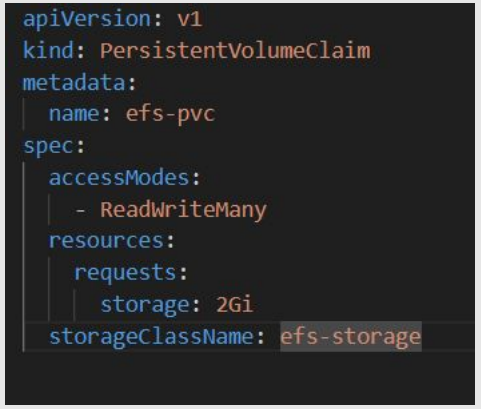
Persistent Volumes & Persistent Volume Claims

PersistentVolume (PV) and PersistentVolumeClaim (PVC): PersistentVolumes are cluster-wide storage resources provisioned by an administrator, while PersistentVolumeClaims are requests made by users for storage resources. PVs and PVCs decouple storage provisioning from the pod specification, allowing for dynamic storage allocation and management. PVs and PVCs can be used with various storage providers like AWS EBS, Azure Disk, NFS, etc.

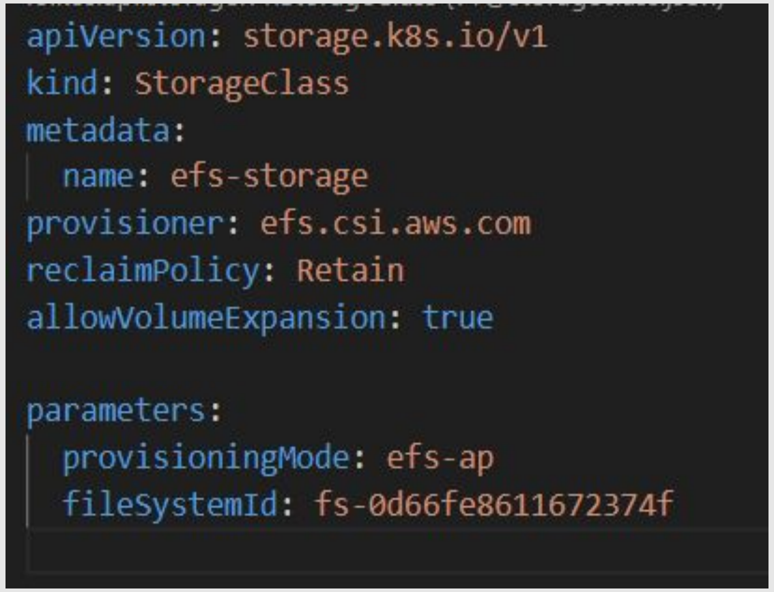
- https://kubernetes.io/docs/concepts/storage/persistent-volumes/



```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-pv
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Delete
  storageClassName: efs-storage
  mountOptions:
    - tls
  csi:
    driver: efs.csi.aws.com
    volumeHandle: fs-0d66fe8611672374f
```



```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: efs-pvc
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 2Gi
  storageClassName: efs-storage
```



```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: efs-storage
provisioner: efs.csi.aws.com
reclaimPolicy: Retain
allowVolumeExpansion: true

parameters:
  provisioningMode: efs-ap
  fileSystemId: fs-0d66fe8611672374f
```

Storage class

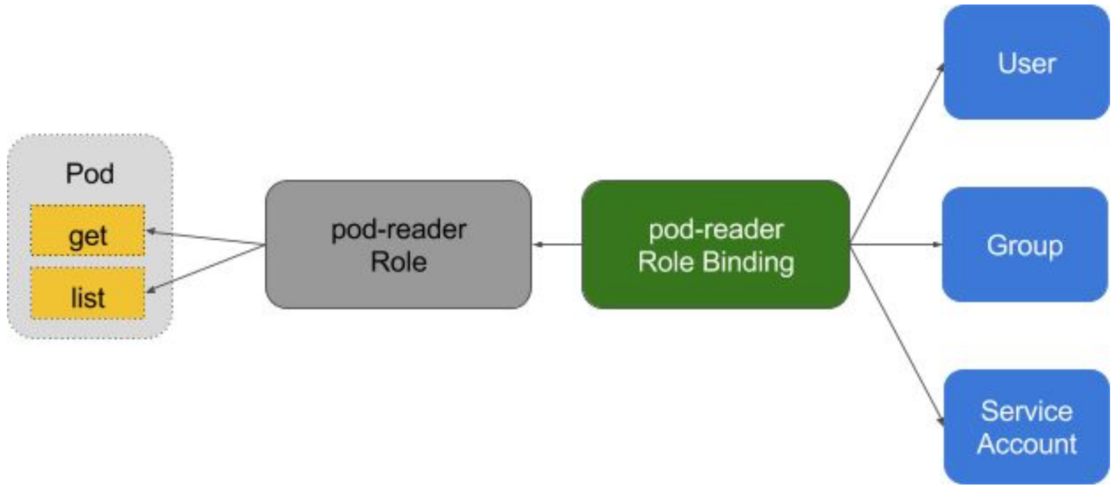
A StorageClass is an object that defines the storage provisioner and parameters for dynamically provisioning PersistentVolumes (PVs). It acts as an abstraction layer between users and the underlying storage infrastructure

- https://kubernetes.io/docs/concepts/storage/storage-classes/

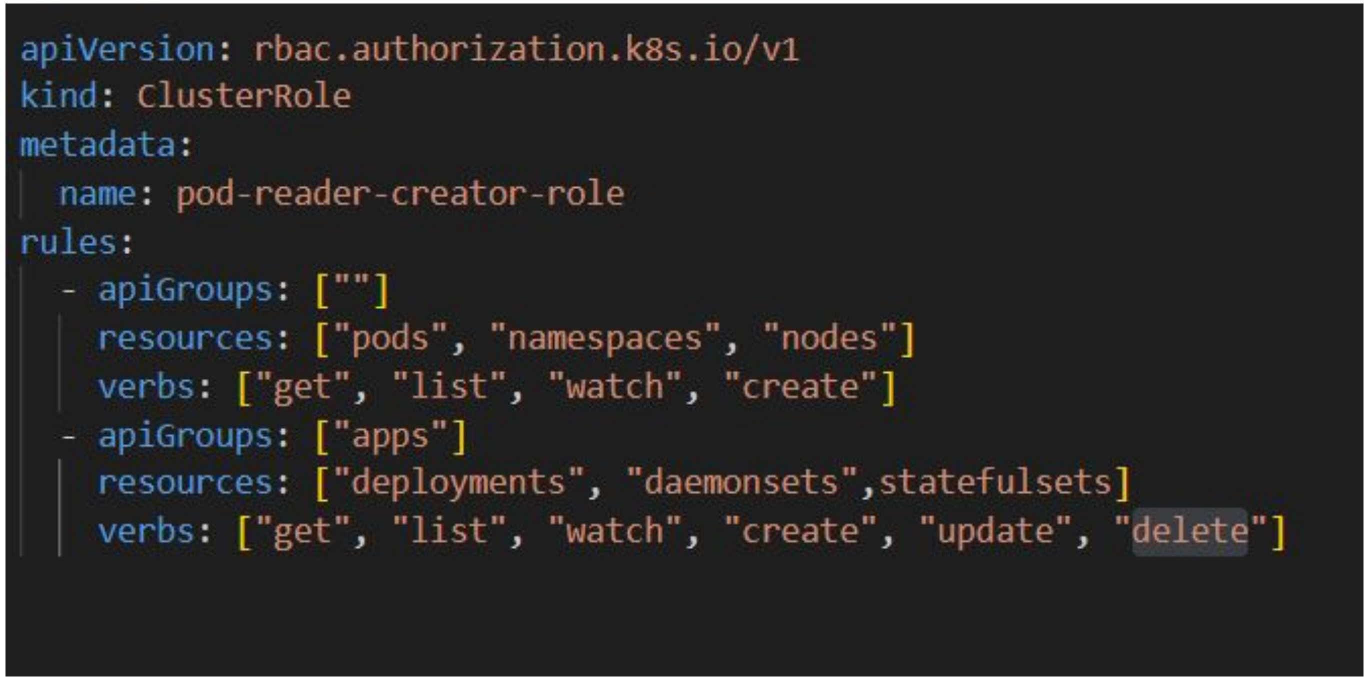
K8s Security (RBAC)

Kubernetes Role-Based Access Control (RBAC) is a security mechanism that provides fine-grained control over user permissions and access to resources within a Kubernetes cluster. RBAC allows you to define roles, role bindings, and cluster roles to manage access at various levels

https://kubernetes.io/docs/reference/access-authn-authz/rbac/



Roles and ClusterRoles

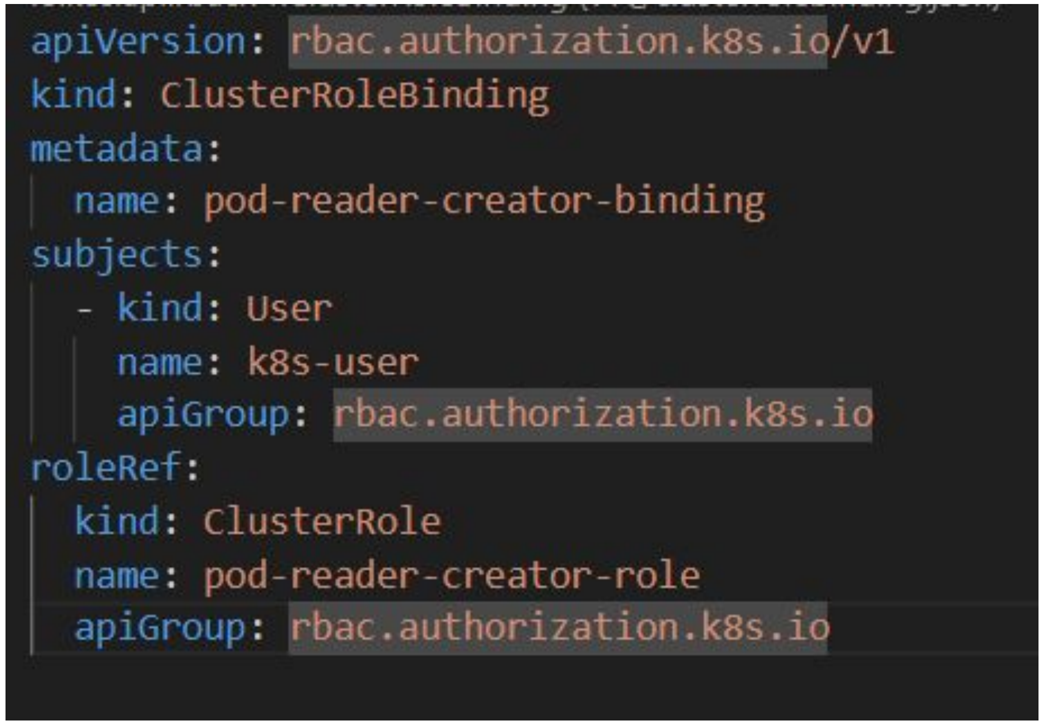


```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-reader-creator-role
rules:
  - apiGroups: [""]
    resources: ["pods", "namespaces", "nodes"]
    verbs: ["get", "list", "watch", "create"]
  - apiGroups: ["apps"]
    resources: ["deployments", "daemonsets",statefulsets]
    verbs: ["get", "list", "watch", "create", "update", "delete"]
```

https://kubernetes.io/docs/reference/access-authn-authz/rbac/

- Roles are a set of rules that define the permissions and access rights for a specific set of resources within a namespace. For example, you can create a Role that allows read-only access to Pods and Services in a particular namespace. **Roles are namespaced resources**.
- Cluster Roles are similar to Roles but are not limited to a specific namespace. They define permissions and access rights across the entire cluster. **Cluster Roles can be used to grant permissions for cluster-wide resources** or to manage cluster-level operations.

RoleBinding and Cluster RoleBinding



```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pod-reader-creator-binding
subjects:
  - kind: User
    name: k8s-user
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: pod-reader-creator-role
  apiGroup: rbac.authorization.k8s.io
```

- Role Binding and Cluster Role Binding are resources used to associate roles (or cluster roles) with subjects, granting them the permissions defined by the roles
- Role Binding associates a role with one or more subjects within a specific namespace, meaning they are created within a specific namespace.
- Cluster Role Binding is similar to Role Binding but works at the cluster level, not limited to a specific namespace. They are used to associate a cluster role with subjects across the entire cluster.

K8s Kubeconfig

- The Kubernetes configuration file, often referred to as the kubeconfig file, is used to configure access to a cluster or clusters. The file specifies the cluster, user credentials, and context information required to interact with a Kubernetes cluster. The kubeconfig file is typically located at **$HOME/.kube/config**
- By default, **kubectl** looks for a file named **config** in the **$HOME/.kube** directory
- The **kubectl** command-line tool uses kubeconfig files to find the information it needs to choose a cluster and communicate with the API server of a cluster

https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/

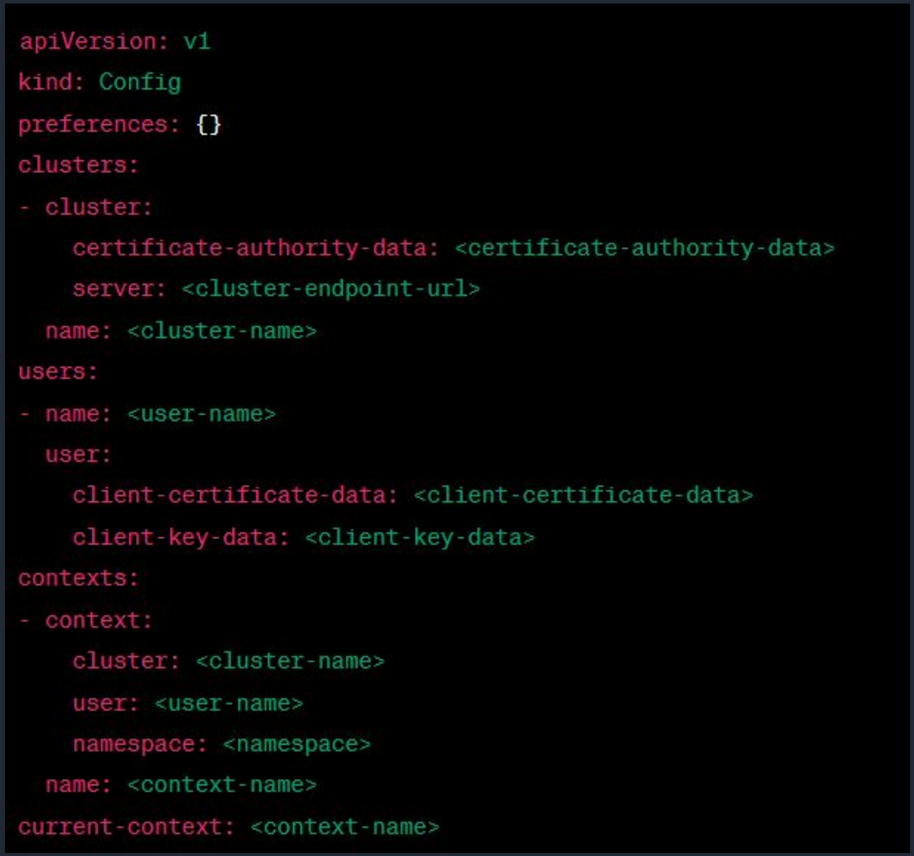
Key Kubeconfig Components

Clusters: This section defines the available clusters and their connection information. It includes the cluster's name, server endpoint URL, and certificate authority data (or paths to the CA files).

Users: This section specifies the user credentials or authentication mechanisms to access the Kubernetes cluster. It can include client certificates, bearer tokens, or other authentication details.

Contexts: This section associates a cluster, user, and namespace to form a context. A context defines the active environment for interacting with the cluster. It specifies which cluster to use, which user to authenticate as, and the default namespace to operate within.

Current Context: The **Current Context** field indicates the current active context in the kubeconfig file. When you run **kubectl** commands, it uses the configuration from the context specified here.



```yaml
apiVersion: v1
kind: Config
preferences: {}
clusters:
- cluster:
    certificate-authority-data: <certificate-authority-data>
    server: <cluster-endpoint-url>
  name: <cluster-name>
users:
- name: <user-name>
  user:
    client-certificate-data: <client-certificate-data>
    client-key-data: <client-key-data>
contexts:
- context:
    cluster: <cluster-name>
    user: <user-name>
    namespace: <namespace>
  name: <context-name>
current-context: <context-name>
```

Thank You