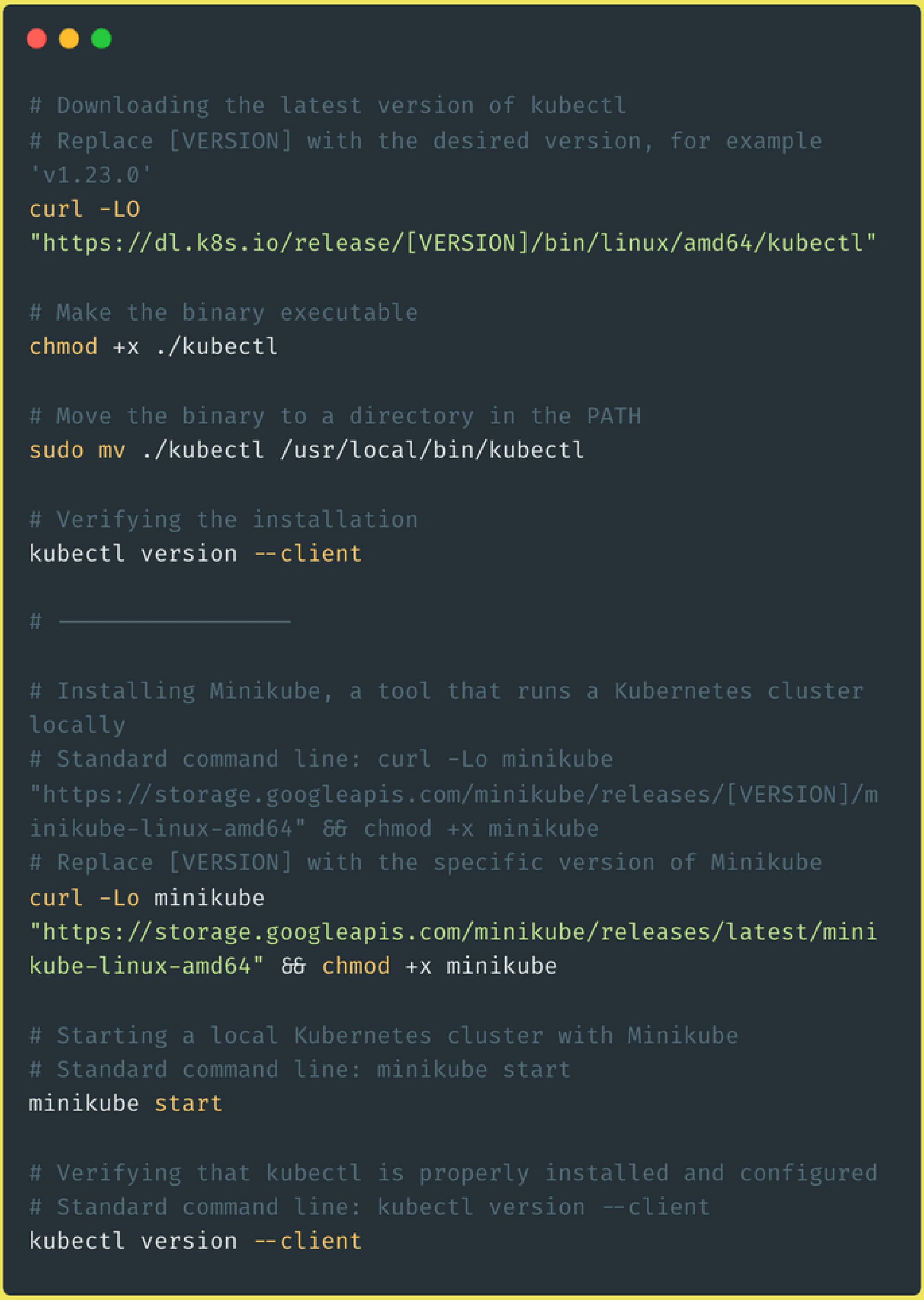
Antony Kervazo-Canut

Kubernetes for Teenagers

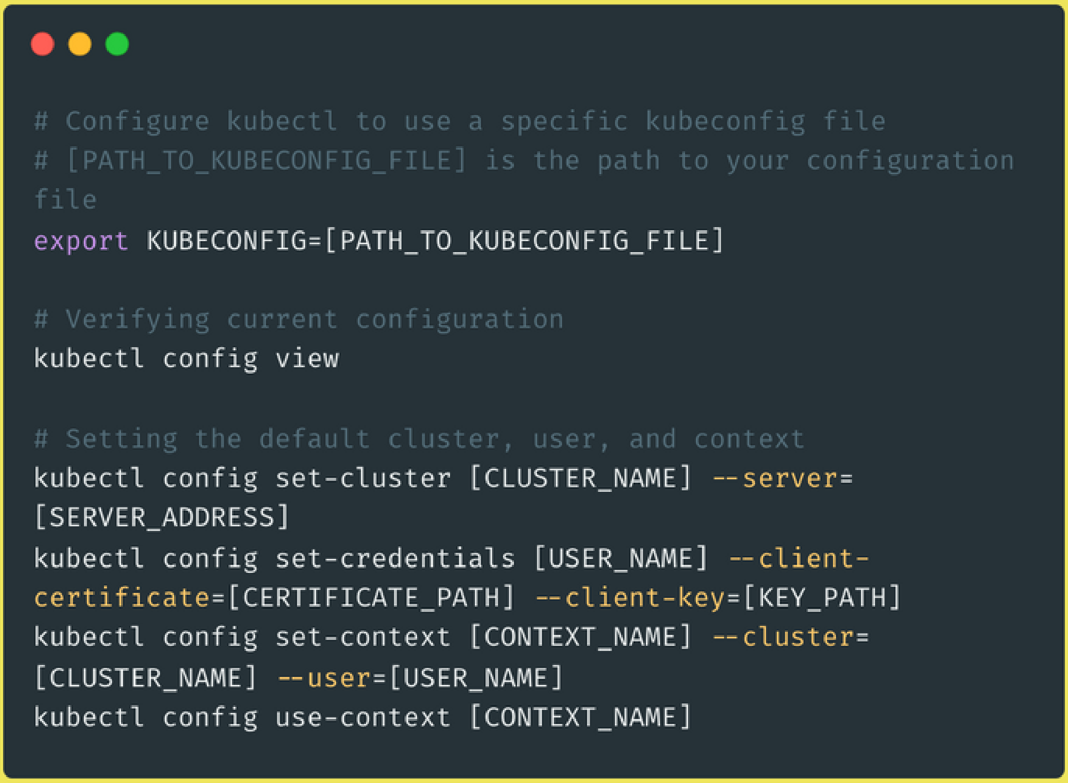
SUITABLE FOR ADULTS

Installation kubectl - minikube



```
# Downloading the latest version of kubectl
# Replace [VERSION] with the desired version, for example 'v1.23.0'
curl -LO "https://dl.k8s.io/release/[VERSION]/bin/linux/amd64/kubectl"

# Make the binary executable
chmod +x ./kubectl

# Move the binary to a directory in the PATH
sudo mv ./kubectl /usr/local/bin/kubectl

# Verifying the installation
kubectl version --client

# ─────────────────

# Installing Minikube, a tool that runs a Kubernetes cluster locally
# Standard command line: curl -Lo minikube "https://storage.googleapis.com/minikube/releases/[VERSION]/minikube-linux-amd64" && chmod +x minikube
# Replace [VERSION] with the specific version of Minikube
curl -Lo minikube "https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64" && chmod +x minikube

# Starting a local Kubernetes cluster with Minikube
# Standard command line: minikube start
minikube start

# Verifying that kubectl is properly installed and configured
# Standard command line: kubectl version --client
kubectl version --client
```

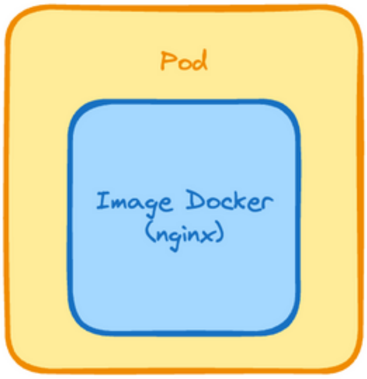
Configuration kubectl



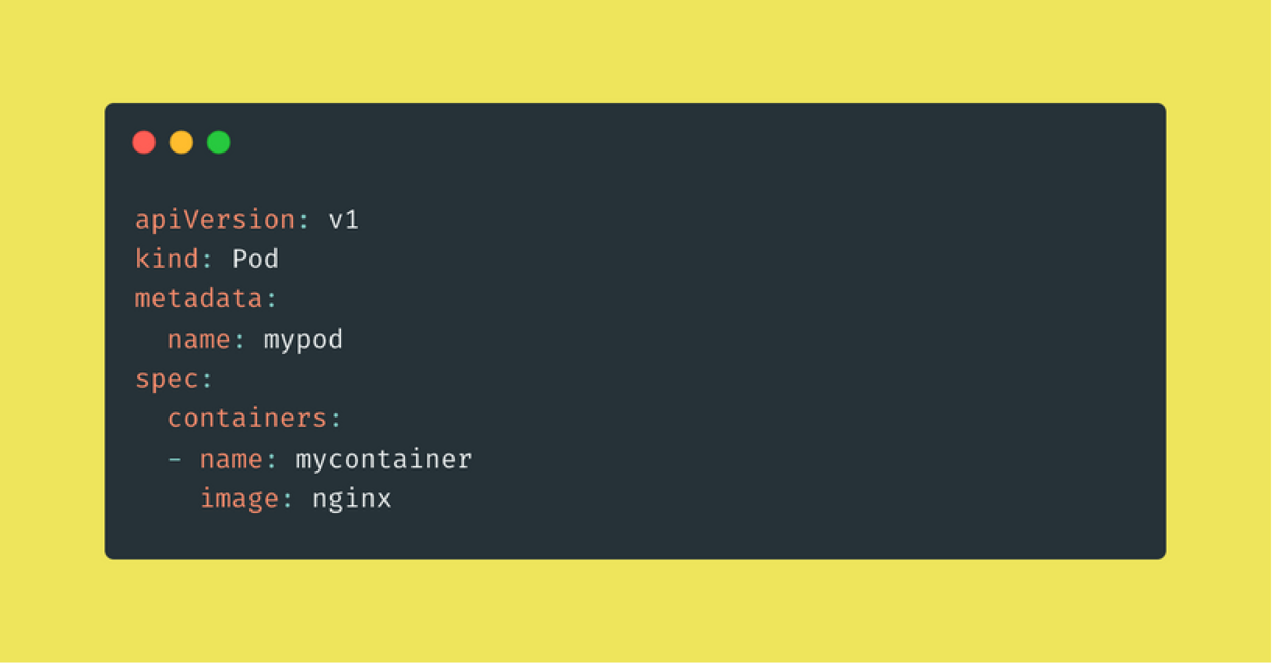
```
# Configure kubectl to use a specific kubeconfig file
# [PATH_TO_KUBECONFIG_FILE] is the path to your configuration file
export KUBECONFIG=[PATH_TO_KUBECONFIG_FILE]

# Verifying current configuration
kubectl config view

# Setting the default cluster, user, and context
kubectl config set-cluster [CLUSTER_NAME] --server=[SERVER_ADDRESS]
kubectl config set-credentials [USER_NAME] --client-certificate=[CERTIFICATE_PATH] --client-key=[KEY_PATH]
kubectl config set-context [CONTEXT_NAME] --cluster=[CLUSTER_NAME] --user=[USER_NAME]
kubectl config use-context [CONTEXT_NAME]
```

Pod

Pods are the smallest deployable units created and managed by Kubernetes. A Pod is a group of one or more containers.



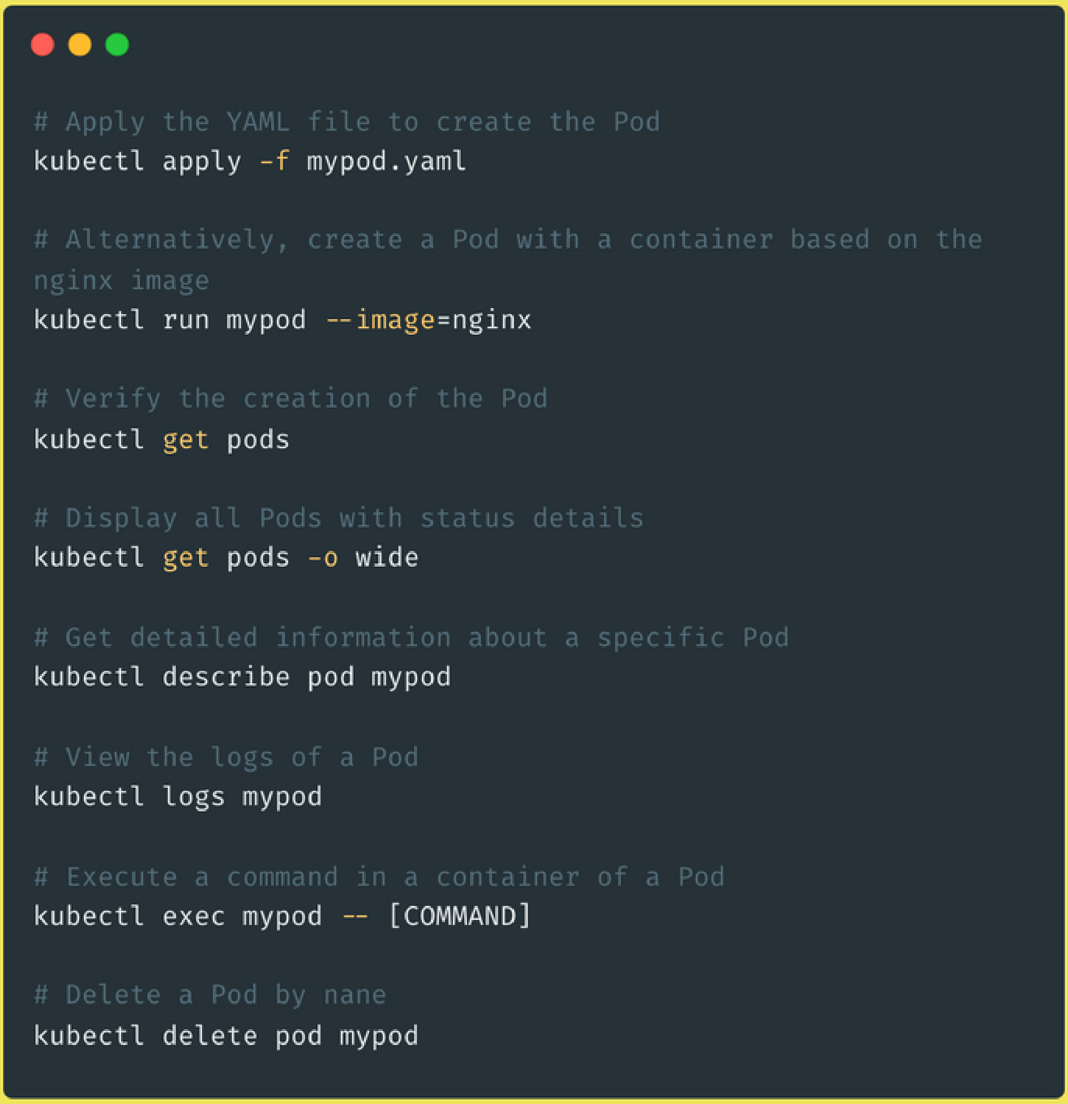
Pods can be created using YAML configuration files, providing more control and flexibility.



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: nginx
```

mypod.yaml

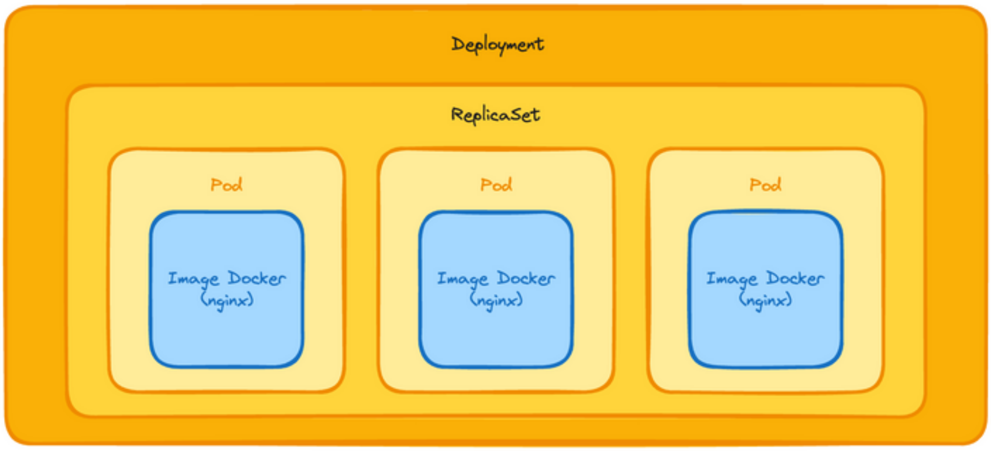
Managing Pods



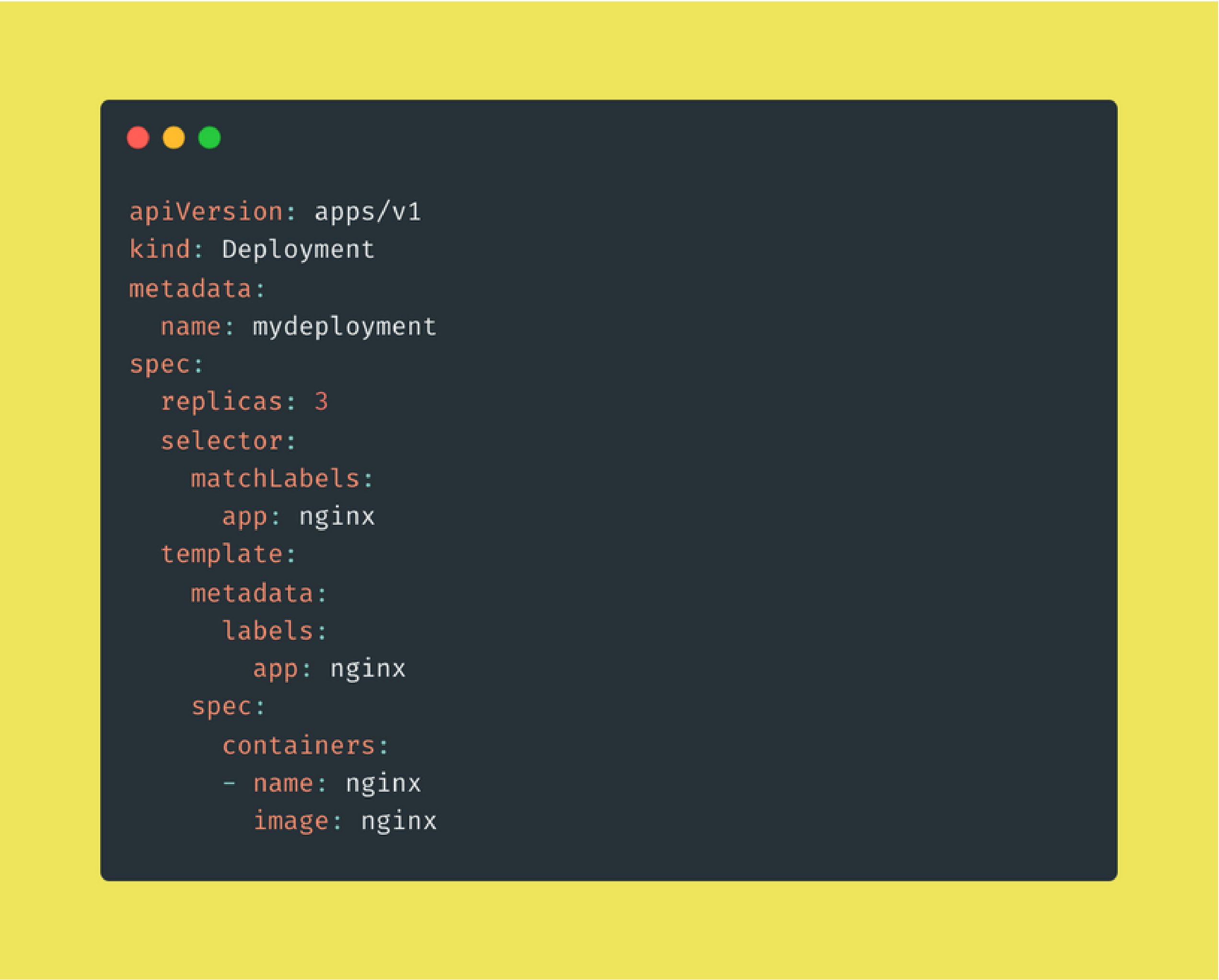
```
# Apply the YAML file to create the Pod
kubectl apply -f mypod.yaml

# Alternatively, create a Pod with a container based on the
nginx image
kubectl run mypod --image=nginx

# Verify the creation of the Pod
kubectl get pods

# Display all Pods with status details
kubectl get pods -o wide

# Get detailed information about a specific Pod
kubectl describe pod mypod

# View the logs of a Pod
kubectl logs mypod

# Execute a command in a container of a Pod
kubectl exec mypod -- [COMMAND]

# Delete a Pod by nane
kubectl delete pod mypod
```

Deployment

A Deployment manages a set of replicas of your application, ensuring its deployment and scaling.

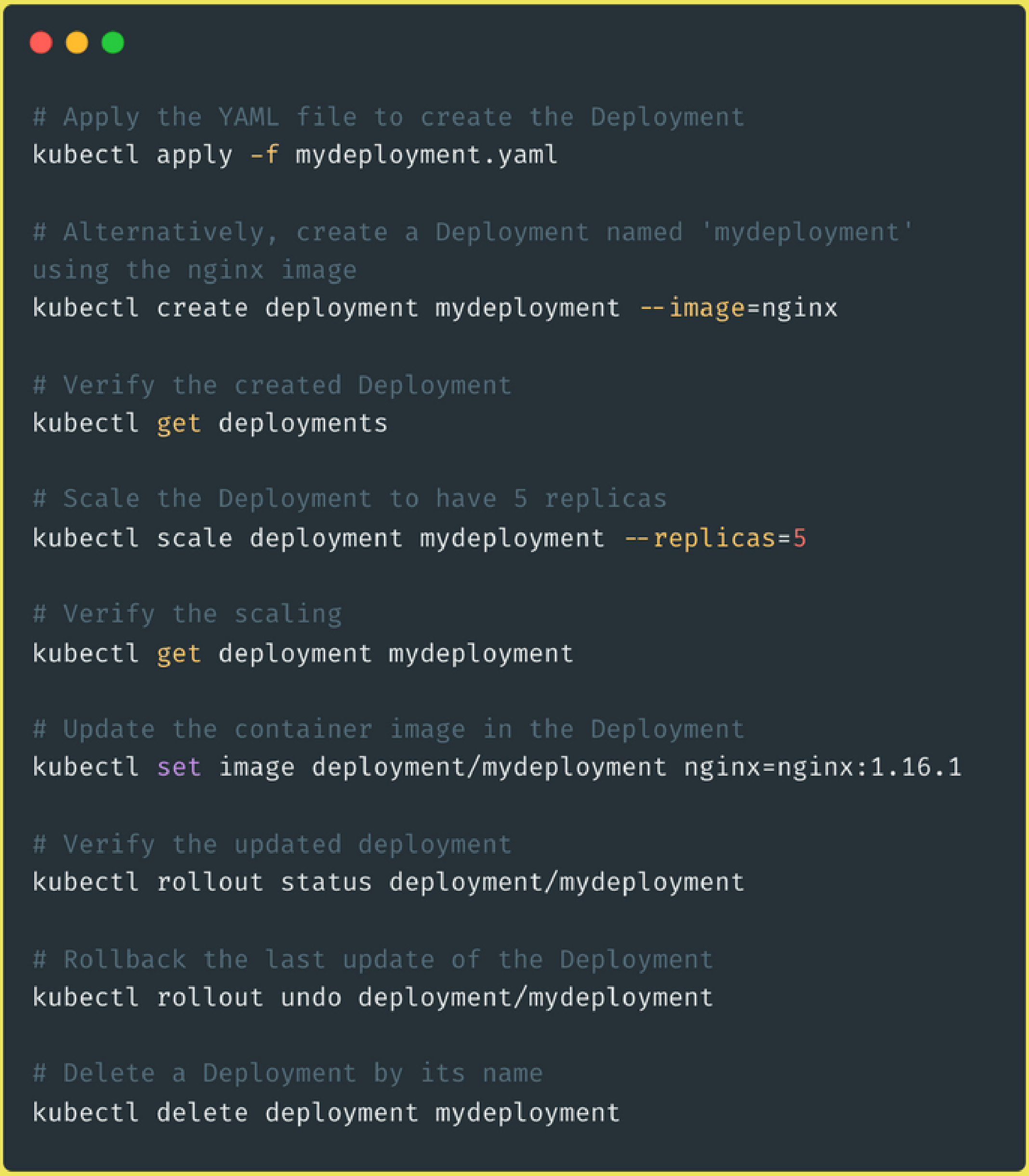


Deployments are often defined and configured via YAML files.



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
```

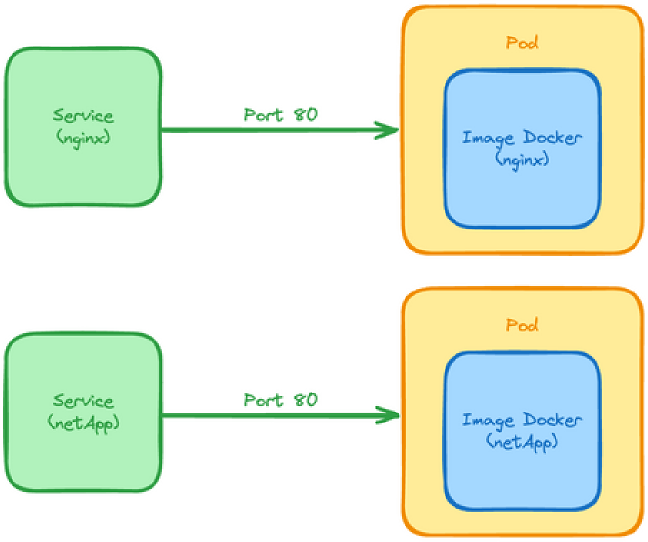
Managing Deployments



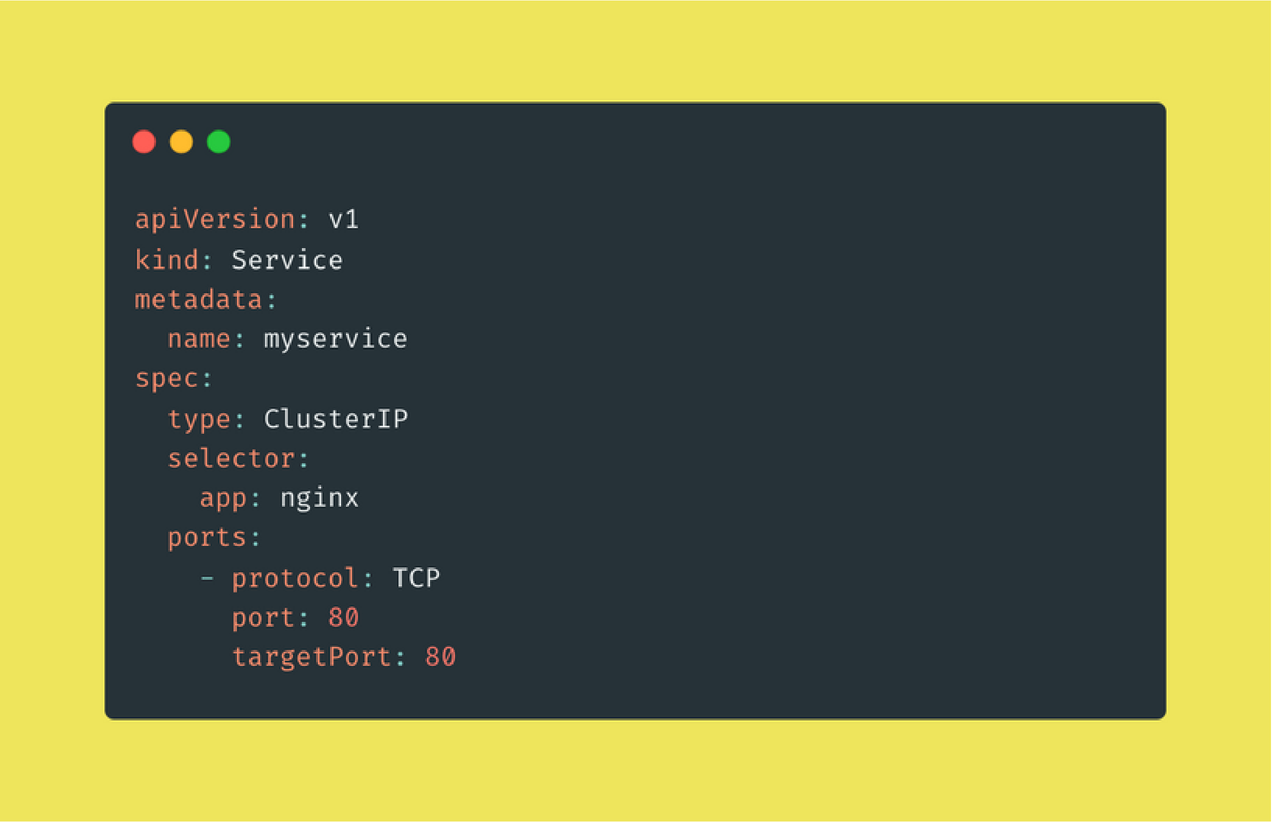
```
# Apply the YAML file to create the Deployment
kubectl apply -f mydeployment.yaml

# Alternatively, create a Deployment named 'mydeployment'
using the nginx image
kubectl create deployment mydeployment --image=nginx

# Verify the created Deployment
kubectl get deployments

# Scale the Deployment to have 5 replicas
kubectl scale deployment mydeployment --replicas=5

# Verify the scaling
kubectl get deployment mydeployment

# Update the container image in the Deployment
kubectl set image deployment/mydeployment nginx=nginx:1.16.1

# Verify the updated deployment
kubectl rollout status deployment/mydeployment

# Rollback the last update of the Deployment
kubectl rollout undo deployment/mydeployment

# Delete a Deployment by its name
kubectl delete deployment mydeployment
```

Service

A Service in Kubernetes is an abstraction that defines a logical set of Pods and a policy by which to access them.

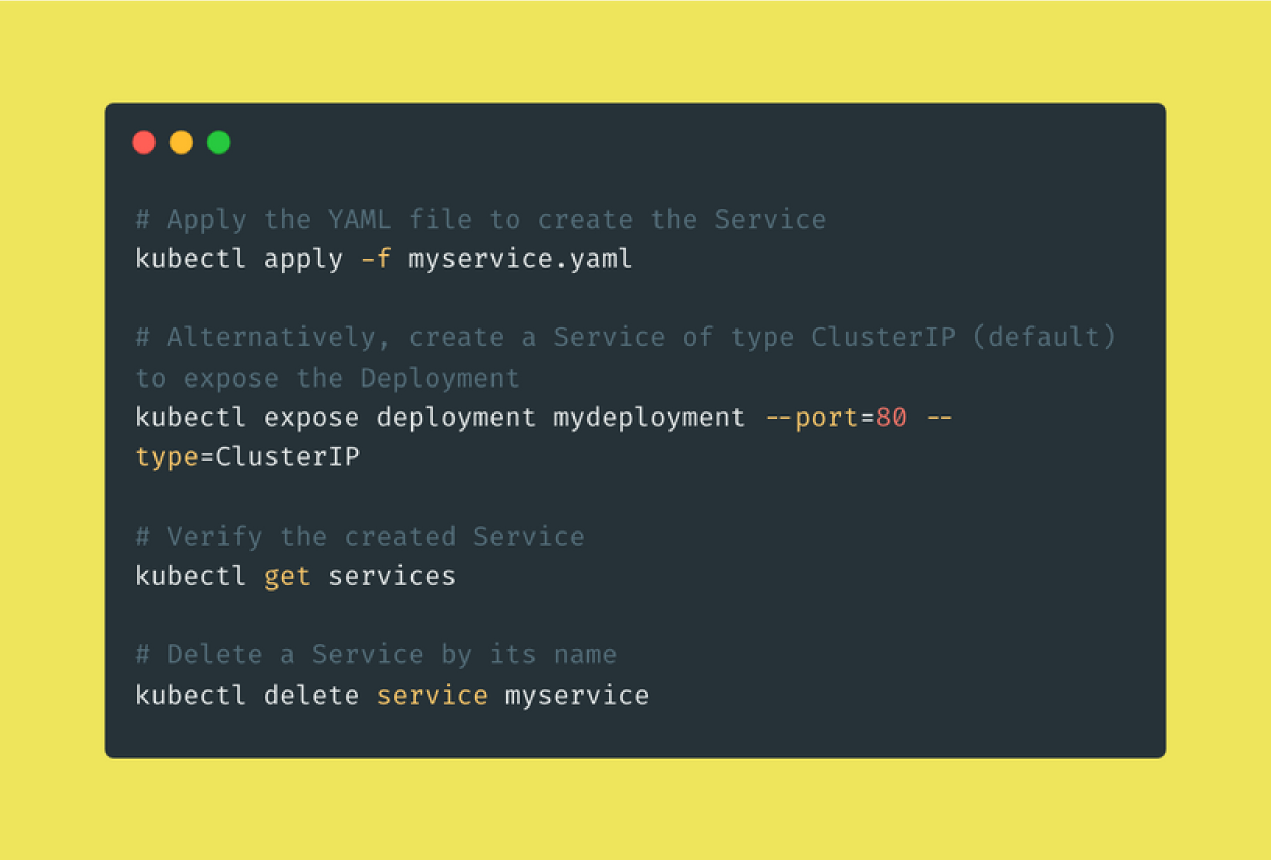


Services can be configured in more detail via YAML files, especially to define different types of Services such as ClusterIP, NodePort, or LoadBalancer.



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

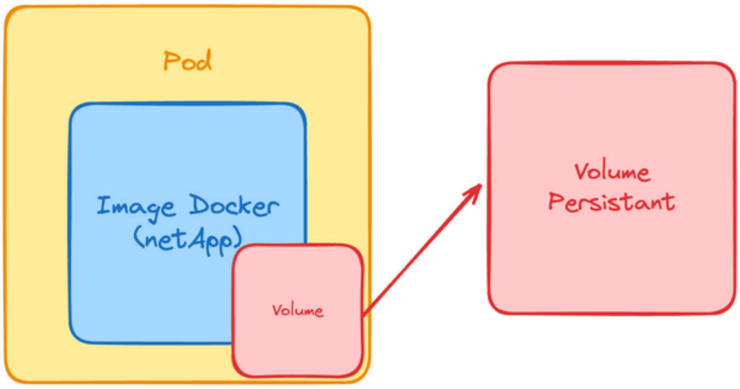
Managing Services

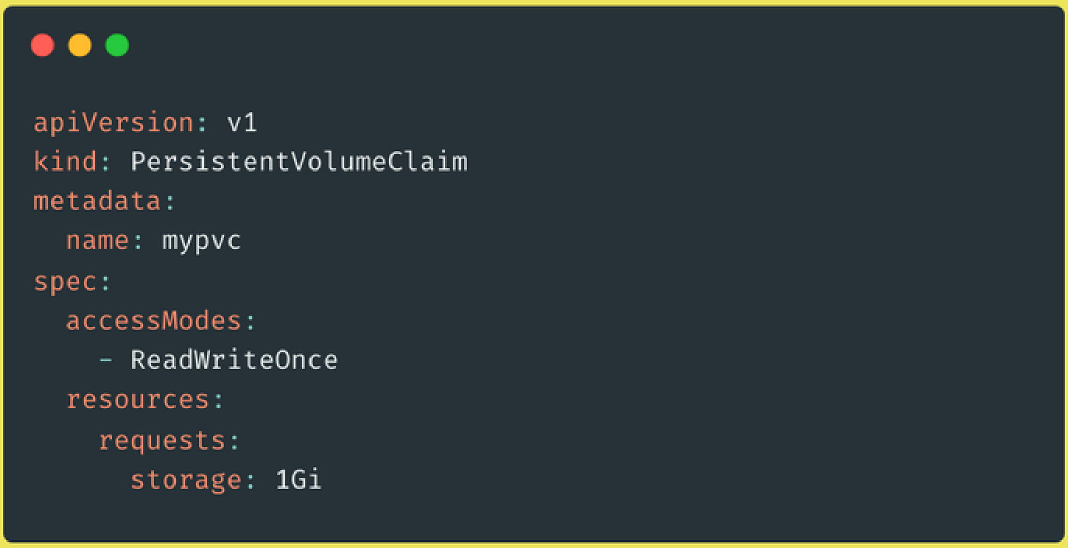


```
# Apply the YAML file to create the Service
kubectl apply -f myservice.yaml

# Alternatively, create a Service of type ClusterIP (default) to expose the Deployment
kubectl expose deployment mydeployment --port=80 --type=ClusterIP

# Verify the created Service
kubectl get services

# Delete a Service by its name
kubectl delete service myservice
```

Volume

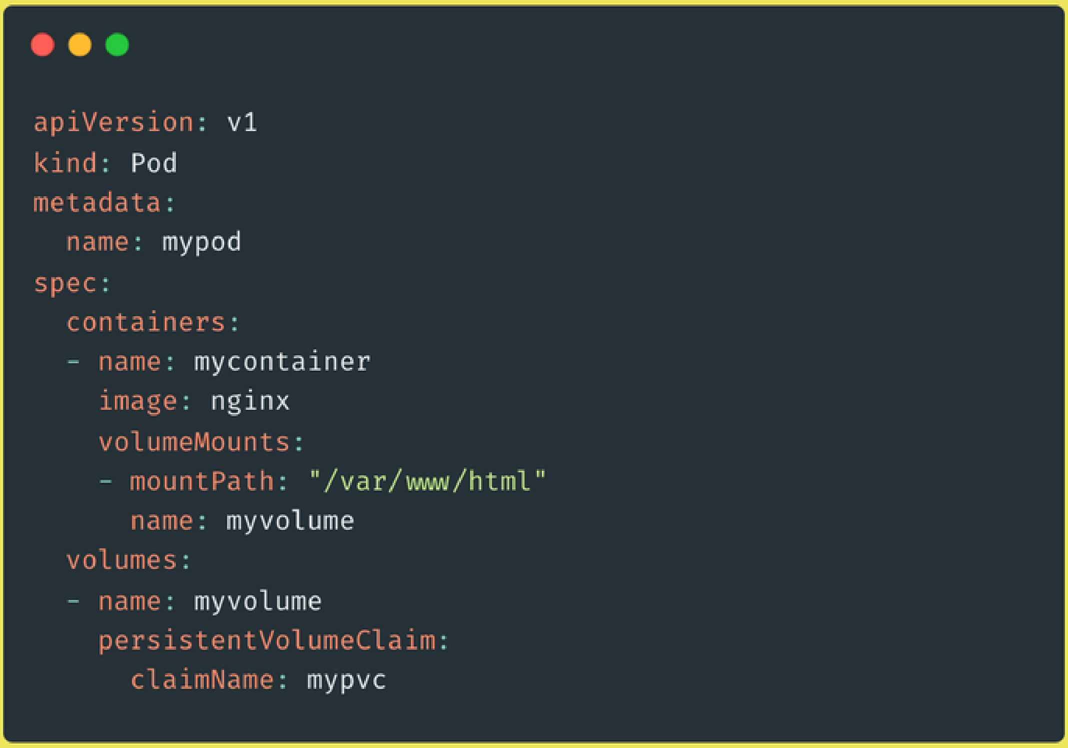
In Kubernetes, a volume is a unit of storage attached to a Pod, existing as long as the Pod exists. A Persistent Volume (PersistentVolume, PV), on the other hand, is a storage resource in the cluster that remains independent of the lifespan of individual Pods. PersistentVolumeClaims (PVCs) are storage requests by users that can be bound to PVs to provide persistent storage.



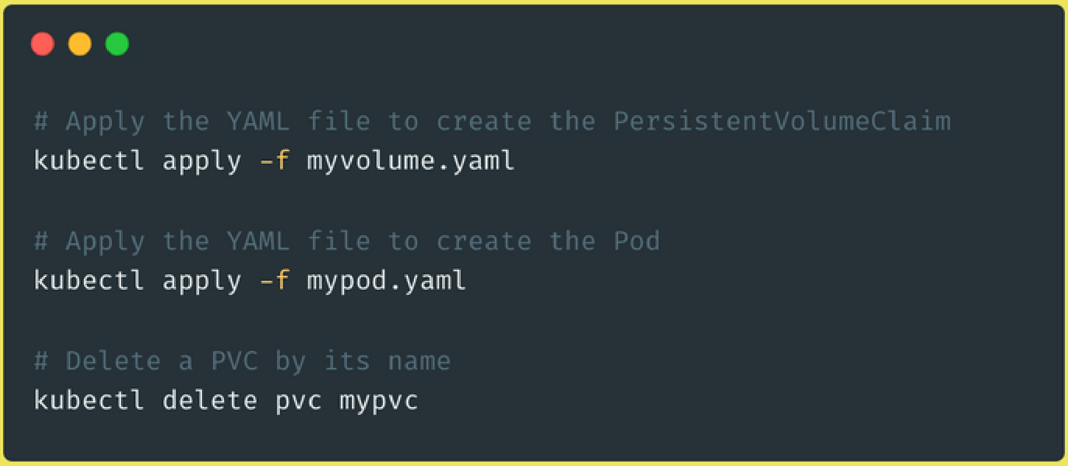


```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Managing Volumes



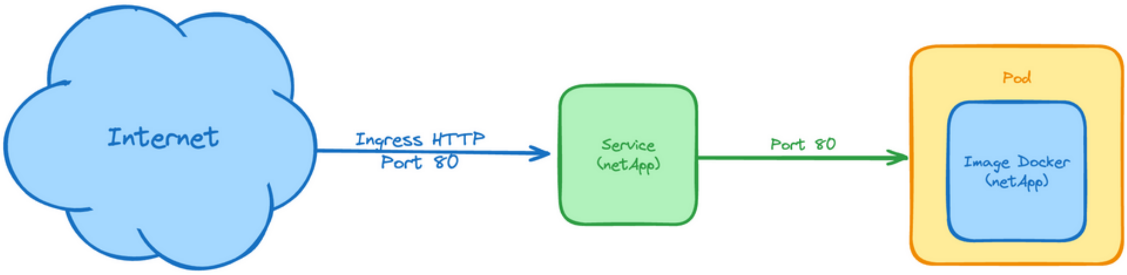
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: nginx
    volumeMounts:
    - mountPath: "/var/www/html"
      name: myvolume
  volumes:
  - name: myvolume
    persistentVolumeClaim:
      claimName: mypvc
```



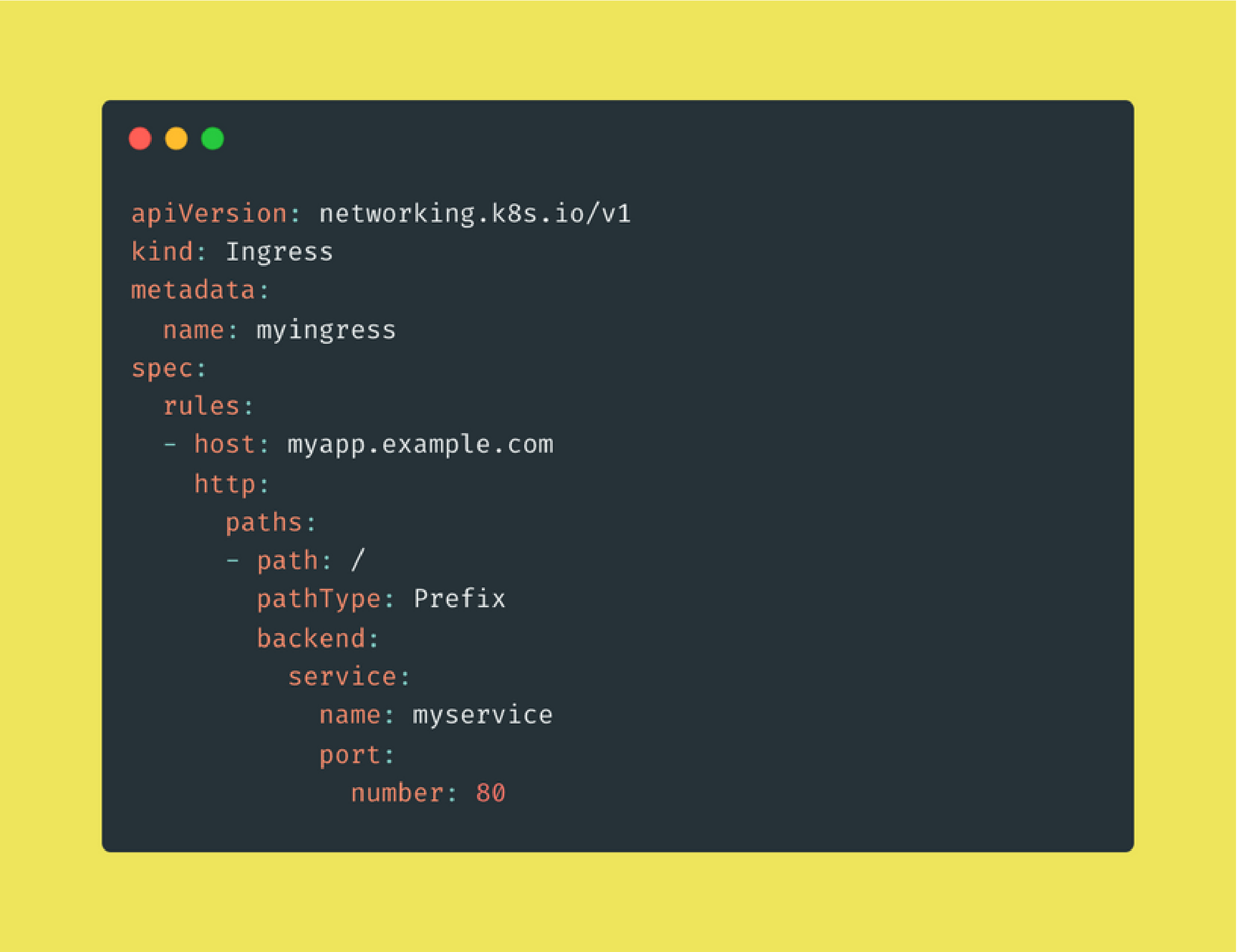
```
# Apply the YAML file to create the PersistentVolumeClaim
kubectl apply -f myvolume.yaml

# Apply the YAML file to create the Pod
kubectl apply -f mypod.yaml

# Delete a PVC by its name
kubectl delete pvc mypvc
```

Ingress

In Kubernetes, networks enable communication between different components, such as Pods, Services, and outside of the cluster.

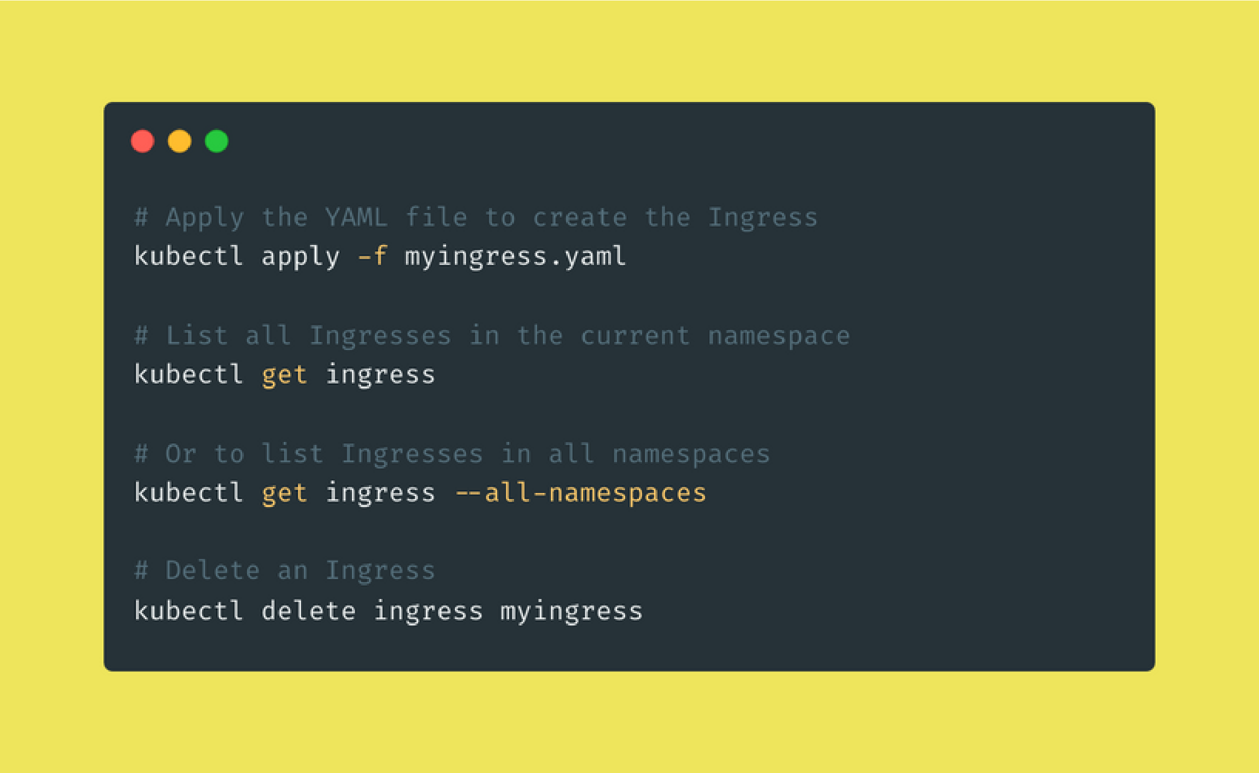


Ingress is a Kubernetes object that manages external access to services in a cluster, typically HTTP.



```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myingress
spec:
  rules:
  - host: myapp.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: myservice
            port:
              number: 80
```

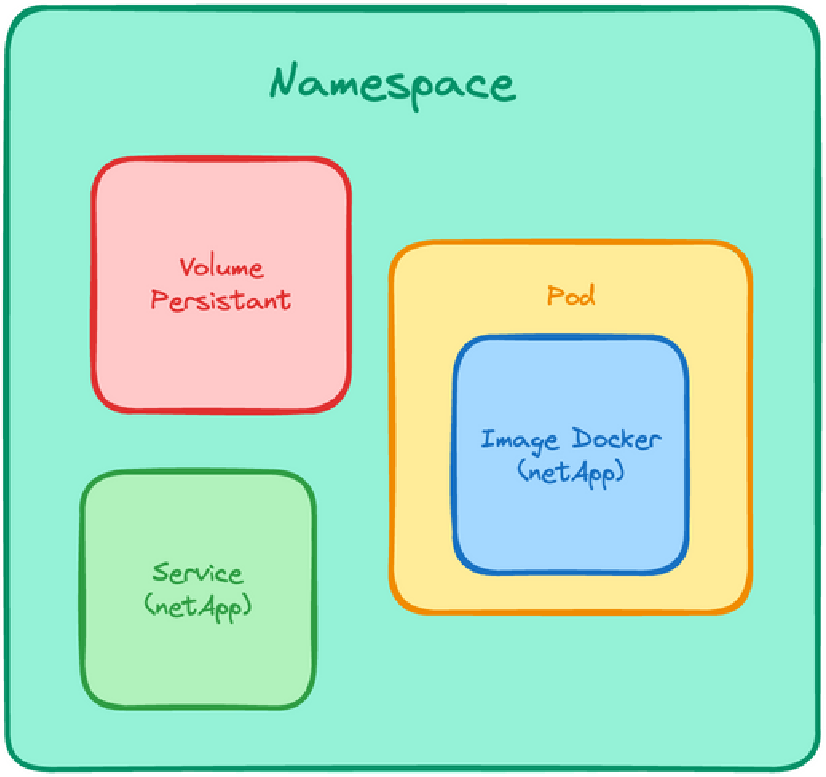
Managing Ingress



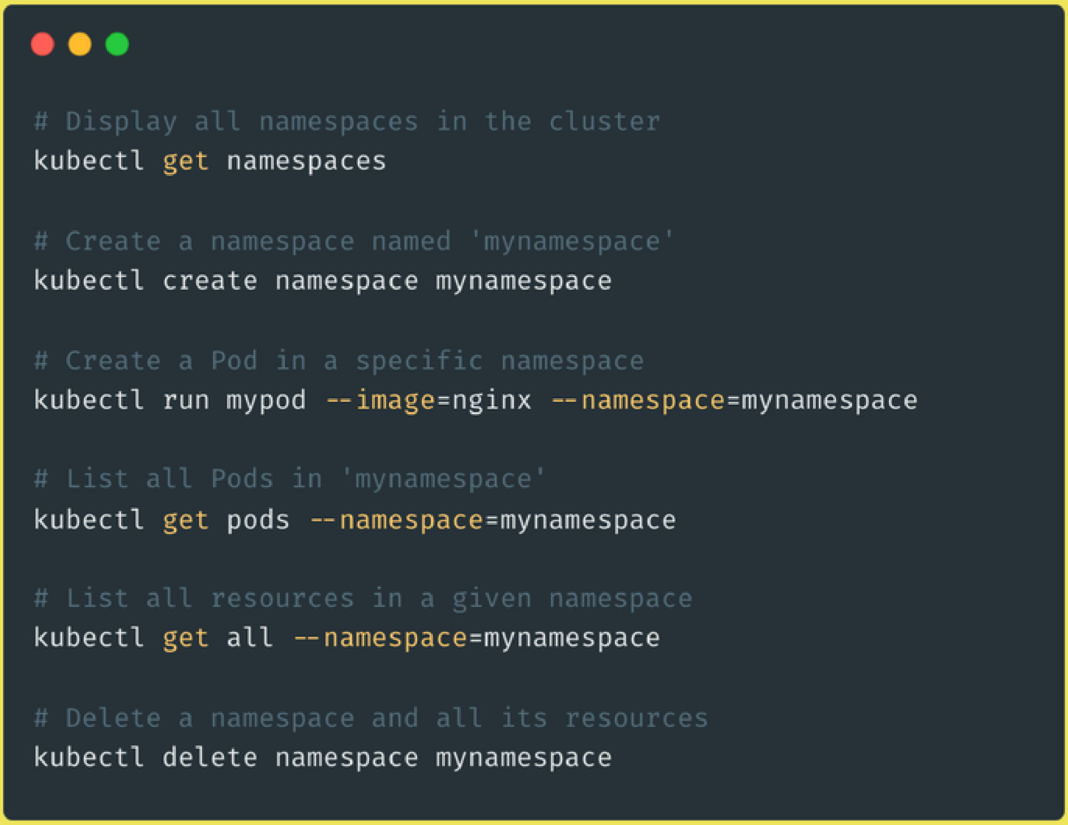
```
# Apply the YAML file to create the Ingress
kubectl apply -f myingress.yaml

# List all Ingresses in the current namespace
kubectl get ingress

# Or to list Ingresses in all namespaces
kubectl get ingress --all-namespaces

# Delete an Ingress
kubectl delete ingress myingress
```

Namespace

Kubernetes namespaces offer a way to divide cluster resources among multiple users and projects. They are useful for creating isolated environments within the same cluster.



Managing Namespaces

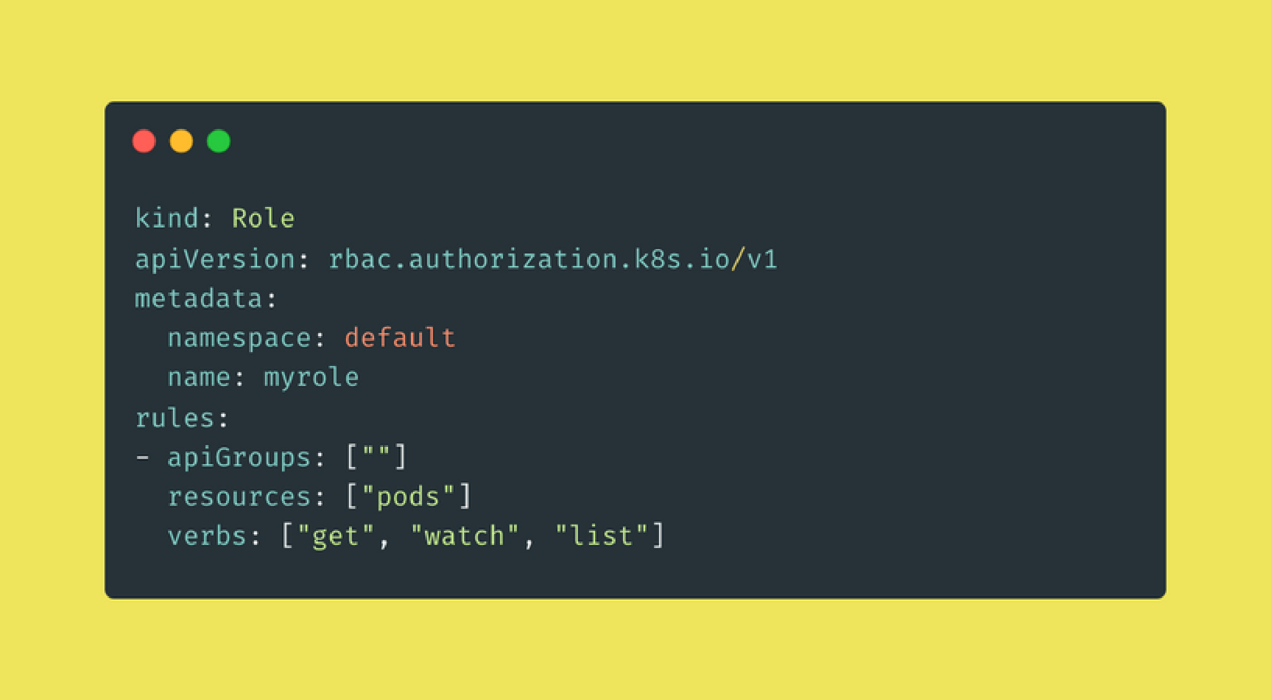


```
# Display all namespaces in the cluster
kubectl get namespaces

# Create a namespace named 'mynamespace'
kubectl create namespace mynamespace

# Create a Pod in a specific namespace
kubectl run mypod --image=nginx --namespace=mynamespace

# List all Pods in 'mynamespace'
kubectl get pods --namespace=mynamespace

# List all resources in a given namespace
kubectl get all --namespace=mynamespace

# Delete a namespace and all its resources
kubectl delete namespace mynamespace
```

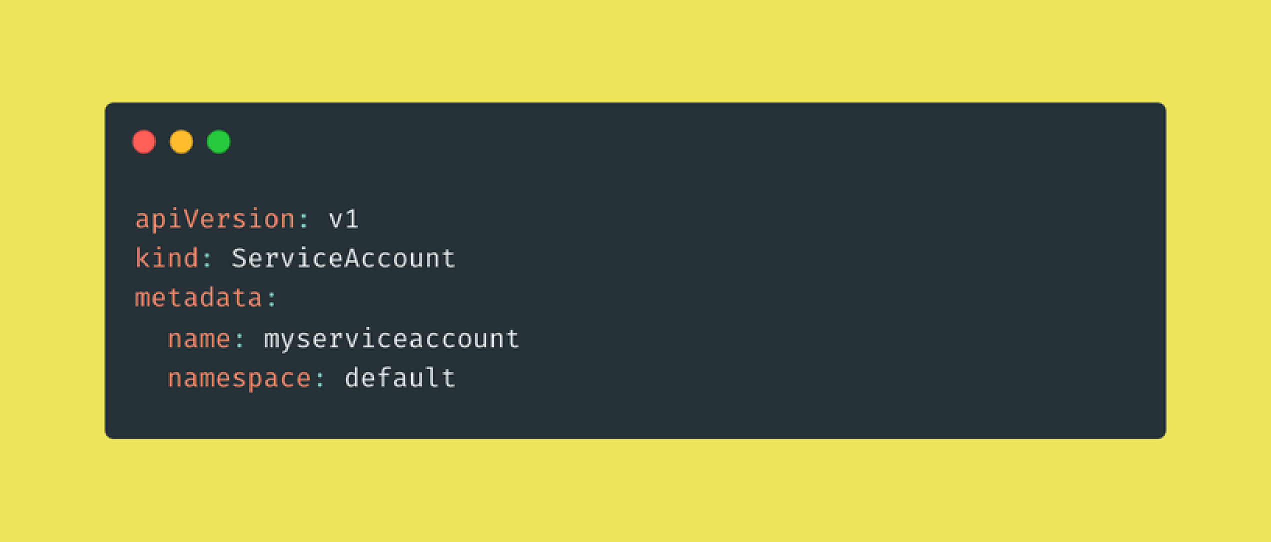
Authentication

Security in Kubernetes heavily relies on the use of tokens for the authentication of users and processes. Tokens can be API tokens, service account tokens, or other forms of identifiers.

Kubernetes uses RBAC (Role-Based Access Control) to manage the permissions of users and service accounts.

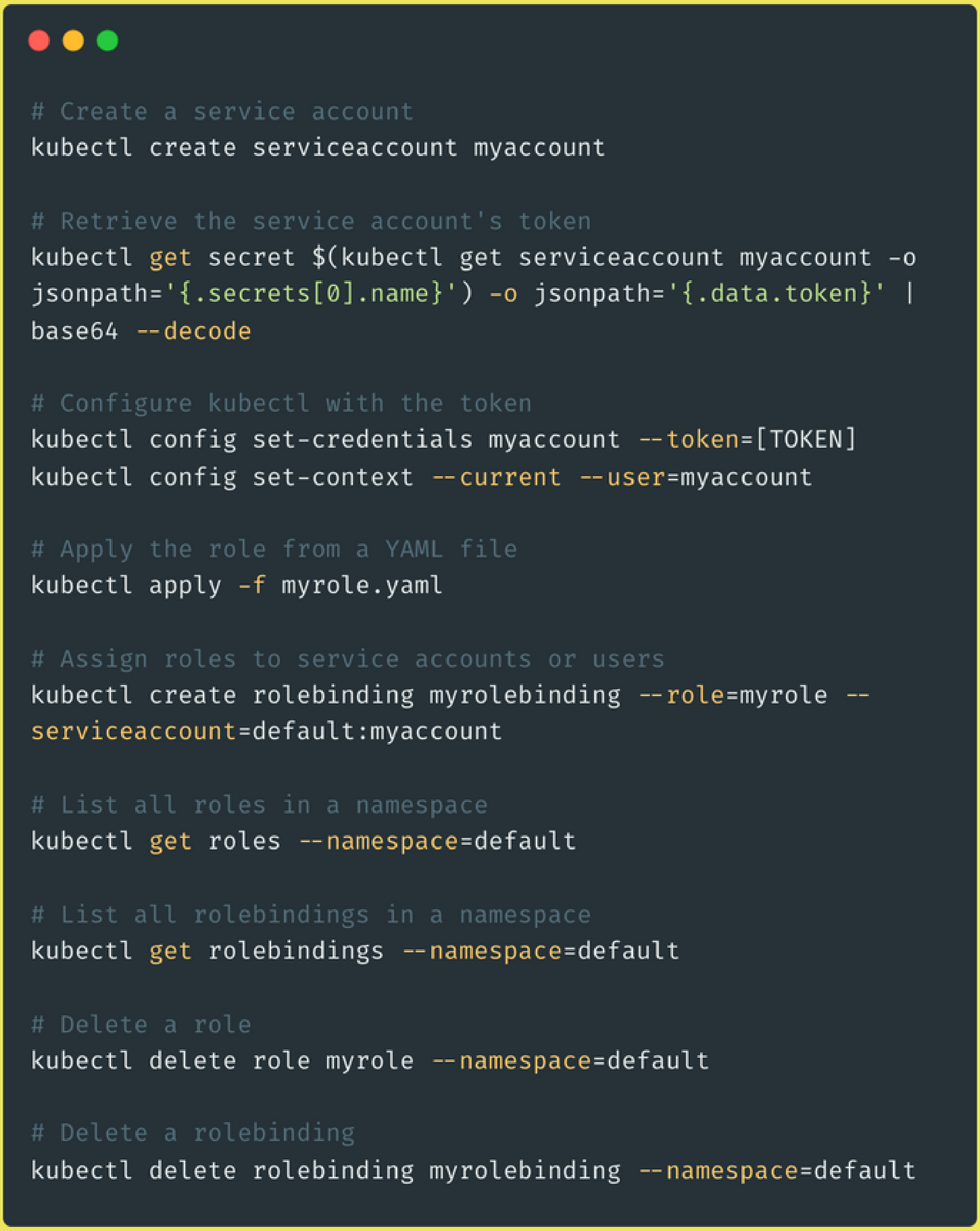


```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: myrole
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```



```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: myserviceaccount
  namespace: default
```

Managing Accounts

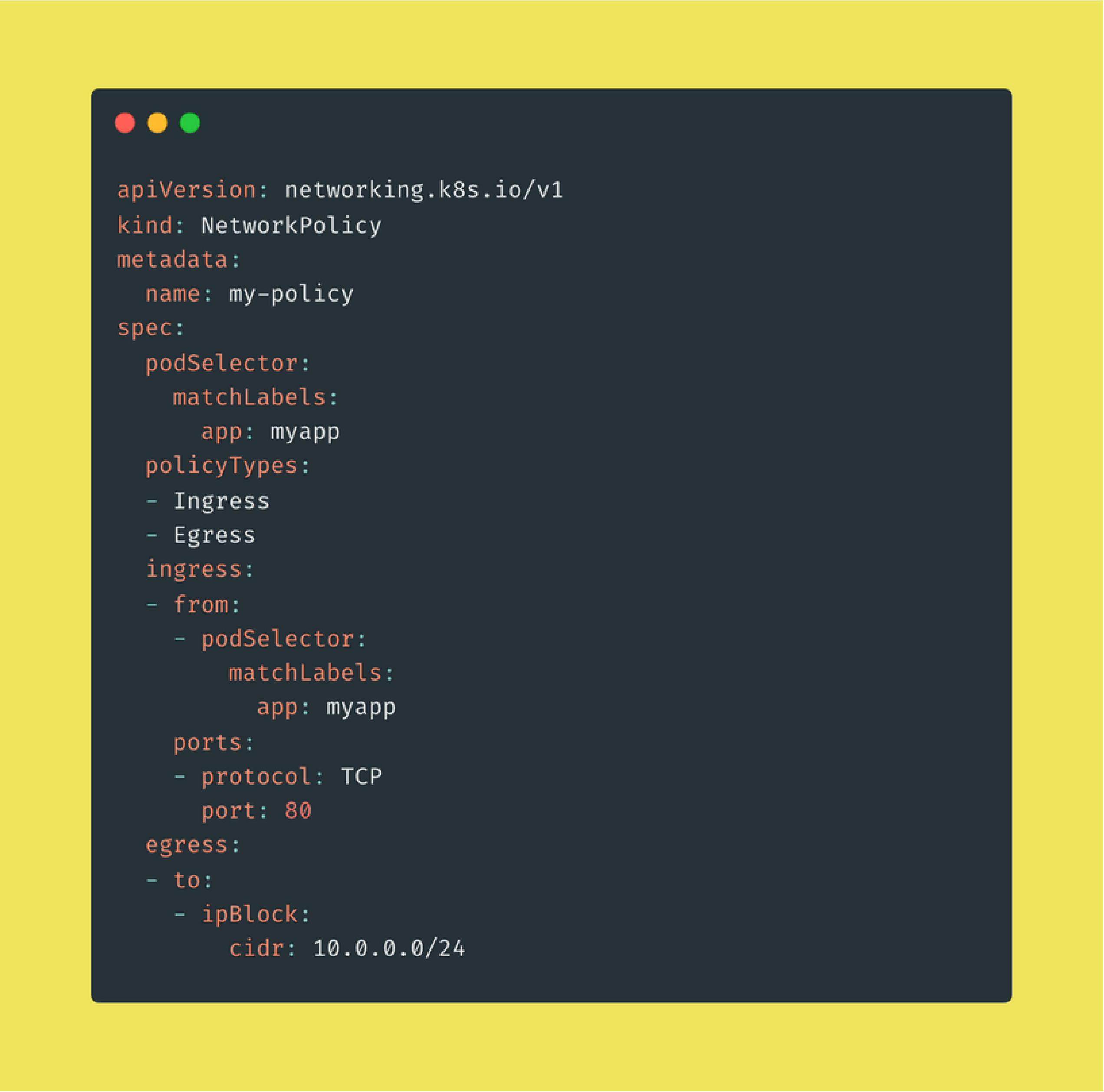


```
# Create a service account
kubectl create serviceaccount myaccount

# Retrieve the service account's token
kubectl get secret $(kubectl get serviceaccount myaccount -o jsonpath='{.secrets[0].name}') -o jsonpath='{.data.token}' | base64 --decode

# Configure kubectl with the token
kubectl config set-credentials myaccount --token=[TOKEN]
kubectl config set-context --current --user=myaccount

# Apply the role from a YAML file
kubectl apply -f myrole.yaml

# Assign roles to service accounts or users
kubectl create rolebinding myrolebinding --role=myrole --serviceaccount=default:myaccount

# List all roles in a namespace
kubectl get roles --namespace=default

# List all rolebindings in a namespace
kubectl get rolebindings --namespace=default

# Delete a role
kubectl delete role myrole --namespace=default

# Delete a rolebinding
kubectl delete rolebinding myrolebinding --namespace=default
```

Network Policies

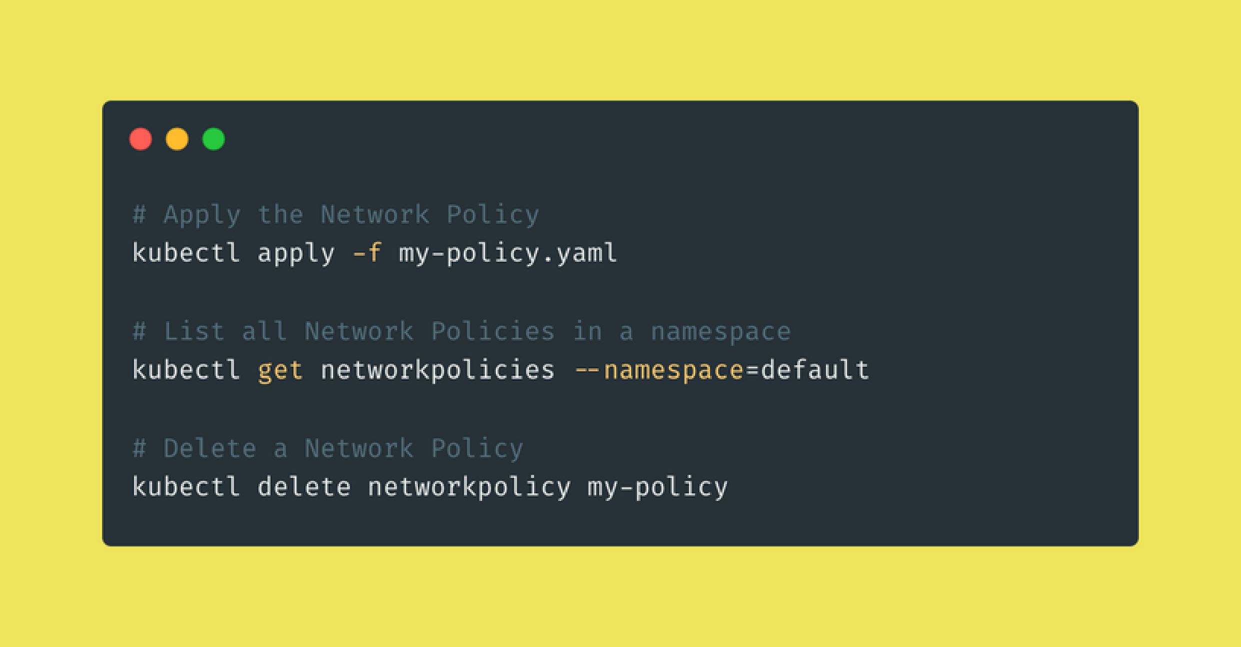
Network Policies in Kubernetes allow controlling how Pods can communicate with each other and with other network endpoints.

Network policies are defined using YAML files that specify the rules for incoming (ingress) and outgoing (egress) traffic.



```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-policy
spec:
  podSelector:
    matchLabels:
      app: myapp
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: myapp
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
```

Managing Network Policies

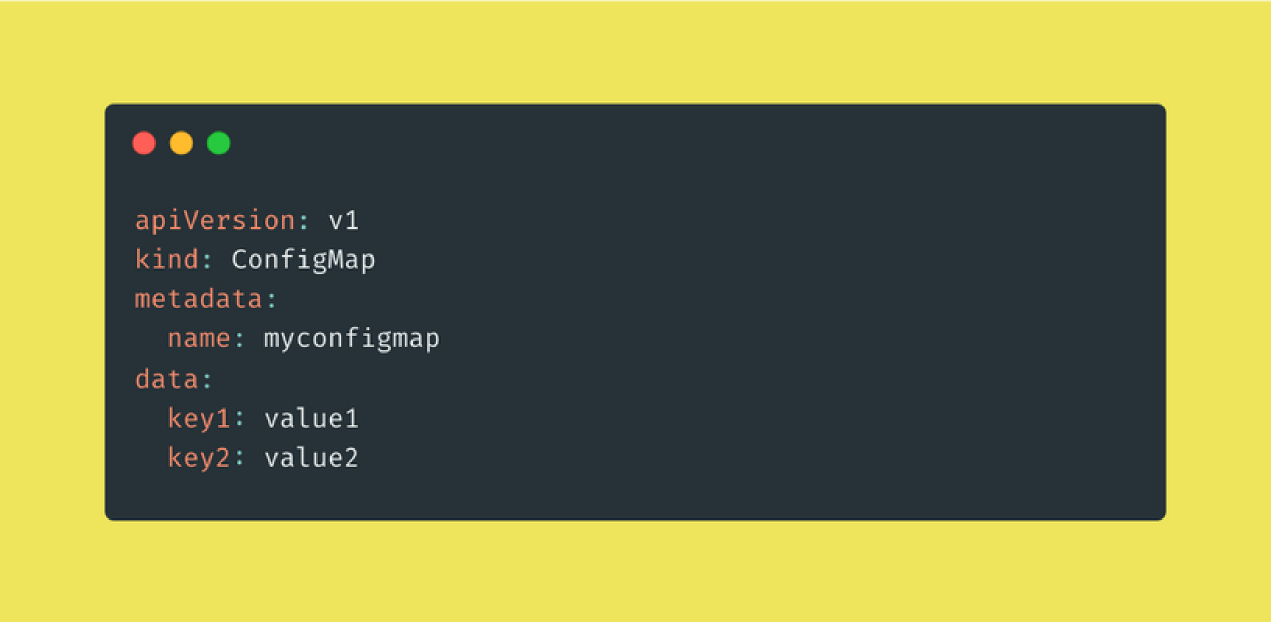


```
# Apply the Network Policy
kubectl apply -f my-policy.yaml

# List all Network Policies in a namespace
kubectl get networkpolicies --namespace=default

# Delete a Network Policy
kubectl delete networkpolicy my-policy
```

ConfigMap

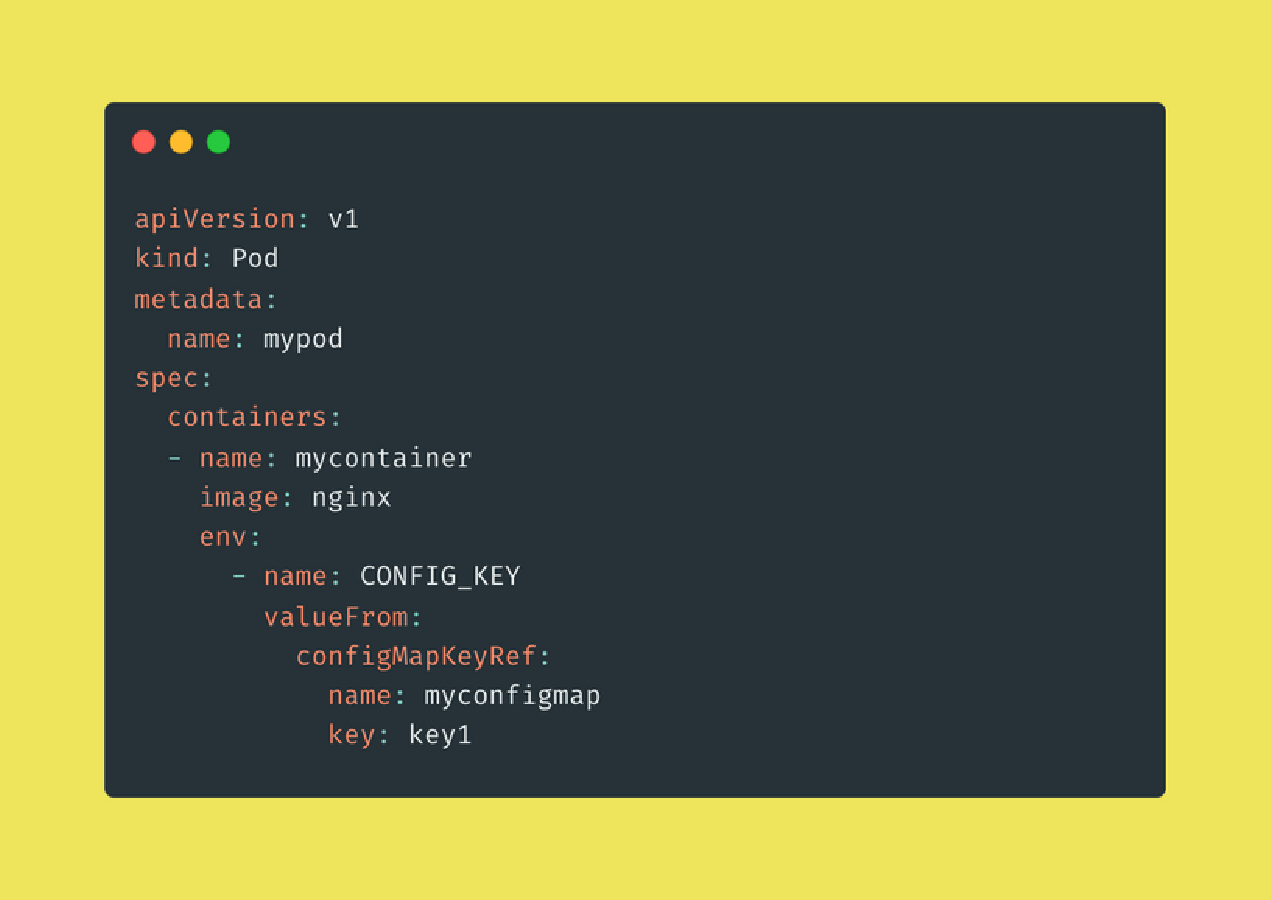
ConfigMaps allow storing configuration data external to Pods, aiding in the management and deployment of applications.

ConfigMaps can be used in Pods as environment variables, command-line arguments, or as configuration files in a volume.



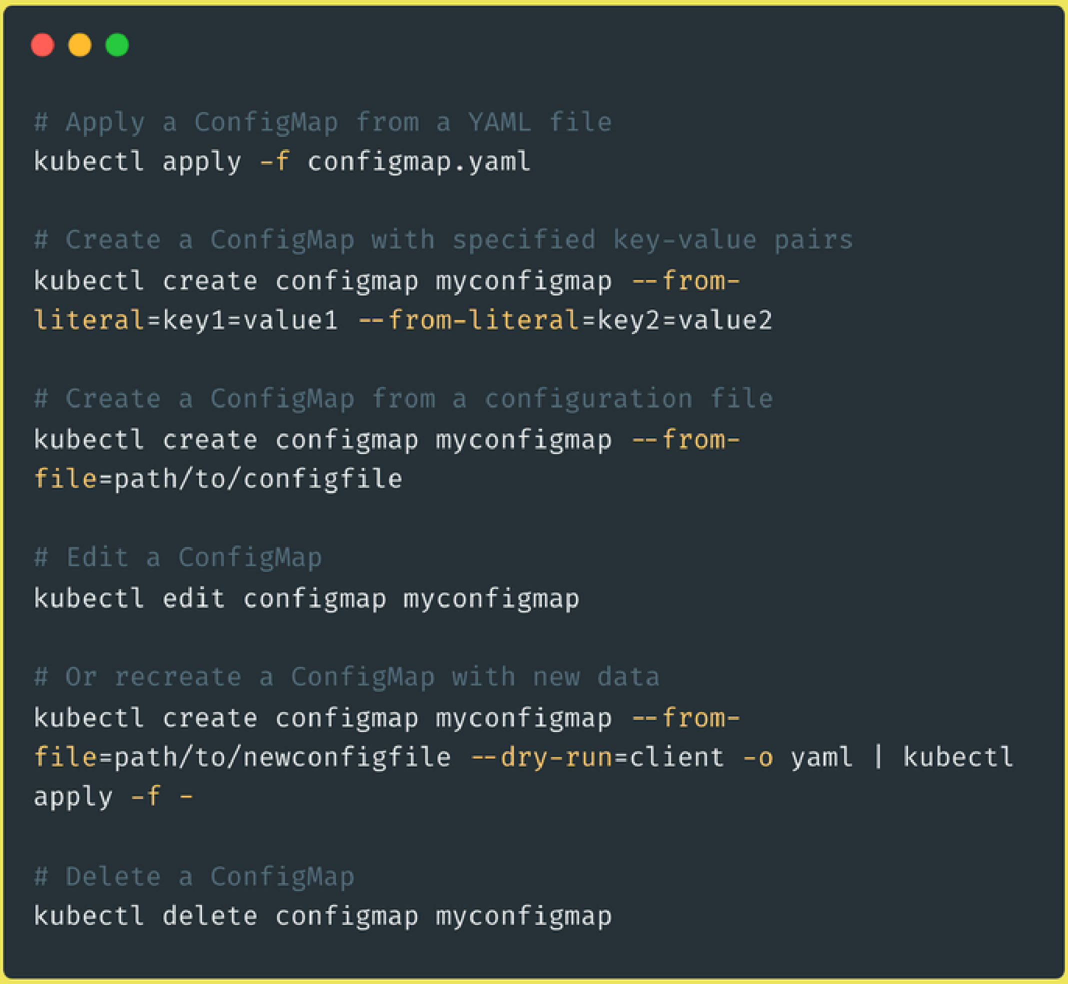
```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  key1: value1
  key2: value2
```

ConfigMap



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: nginx
    env:
      - name: CONFIG_KEY
        valueFrom:
          configMapKeyRef:
            name: myconfigmap
            key: key1
```

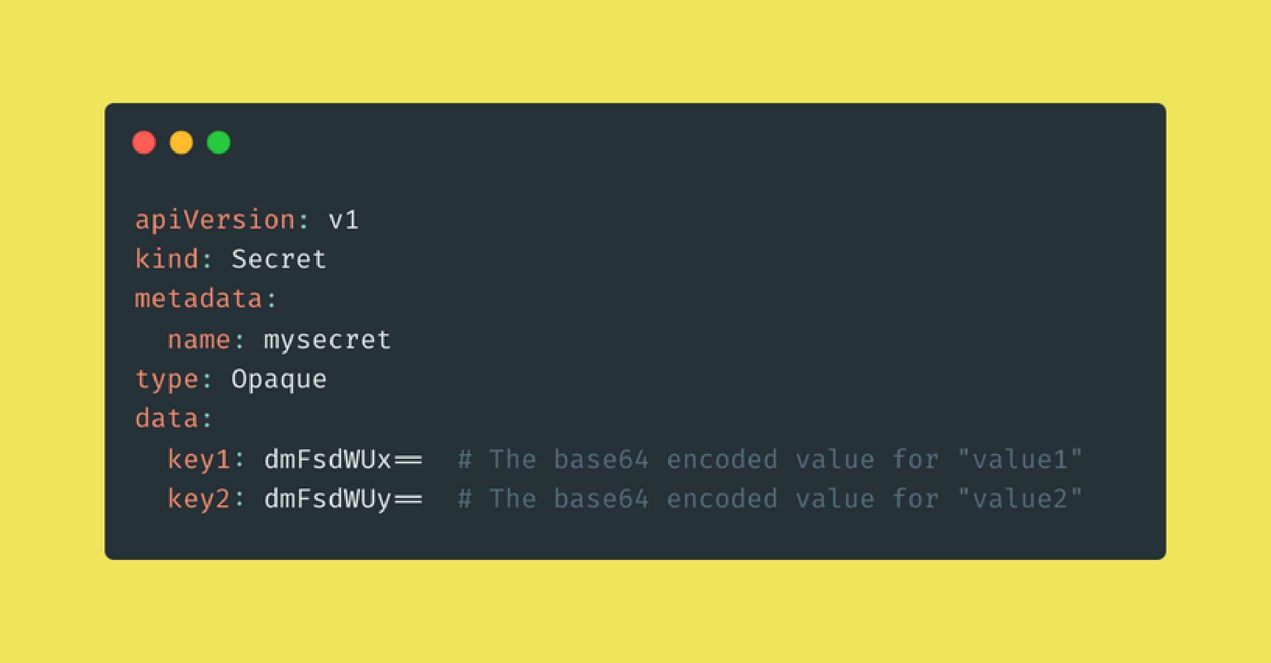
Managing ConfigMaps



```
# Apply a ConfigMap from a YAML file
kubectl apply -f configmap.yaml

# Create a ConfigMap with specified key-value pairs
kubectl create configmap myconfigmap --from-
literal=key1=value1 --from-literal=key2=value2

# Create a ConfigMap from a configuration file
kubectl create configmap myconfigmap --from-
file=path/to/configfile

# Edit a ConfigMap
kubectl edit configmap myconfigmap

# Or recreate a ConfigMap with new data
kubectl create configmap myconfigmap --from-
file=path/to/newconfigfile --dry-run=client -o yaml | kubectl
apply -f -

# Delete a ConfigMap
kubectl delete configmap myconfigmap
```

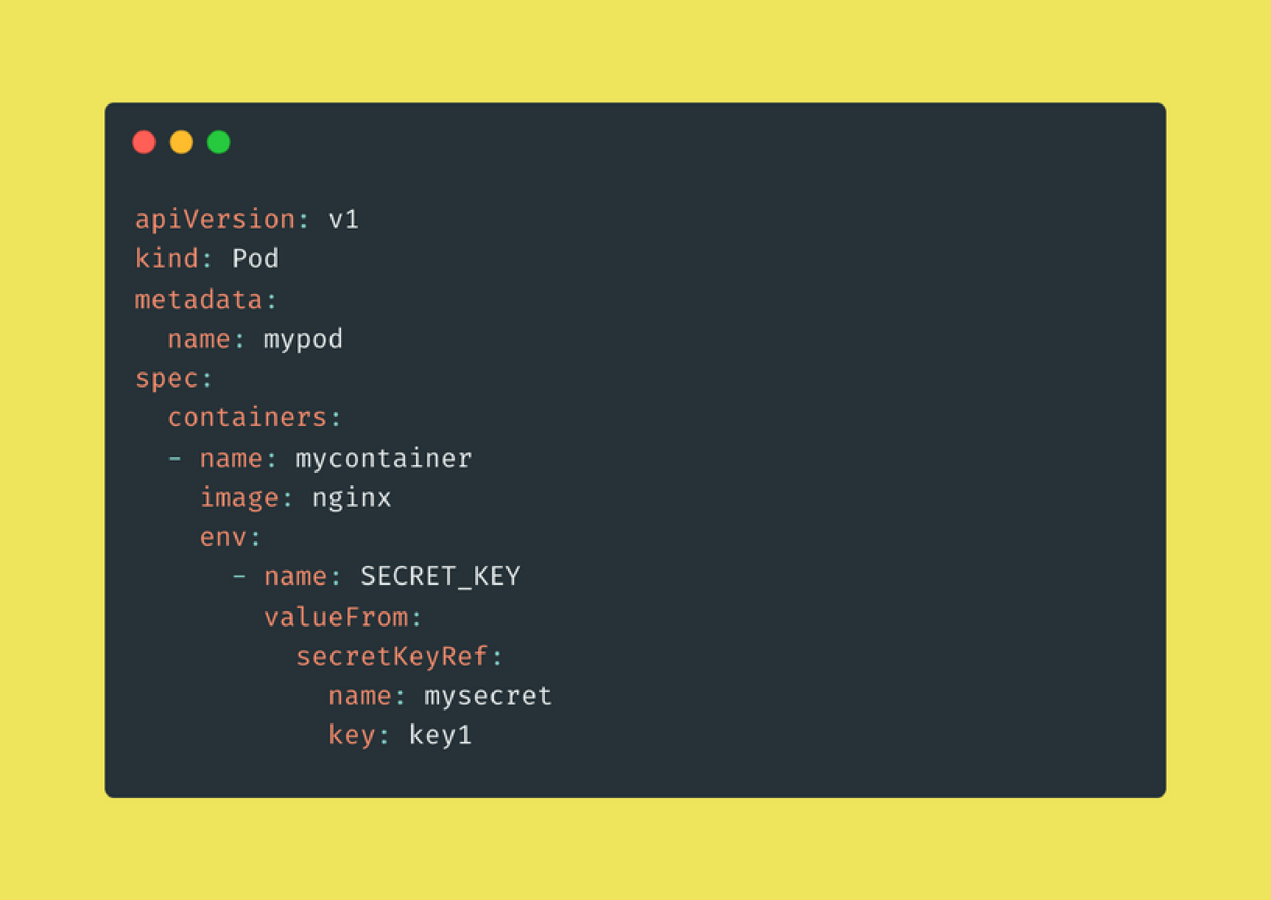
Secret

Kubernetes secrets are used to store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. They allow for the separation of sensitive details from configuration files or container images.



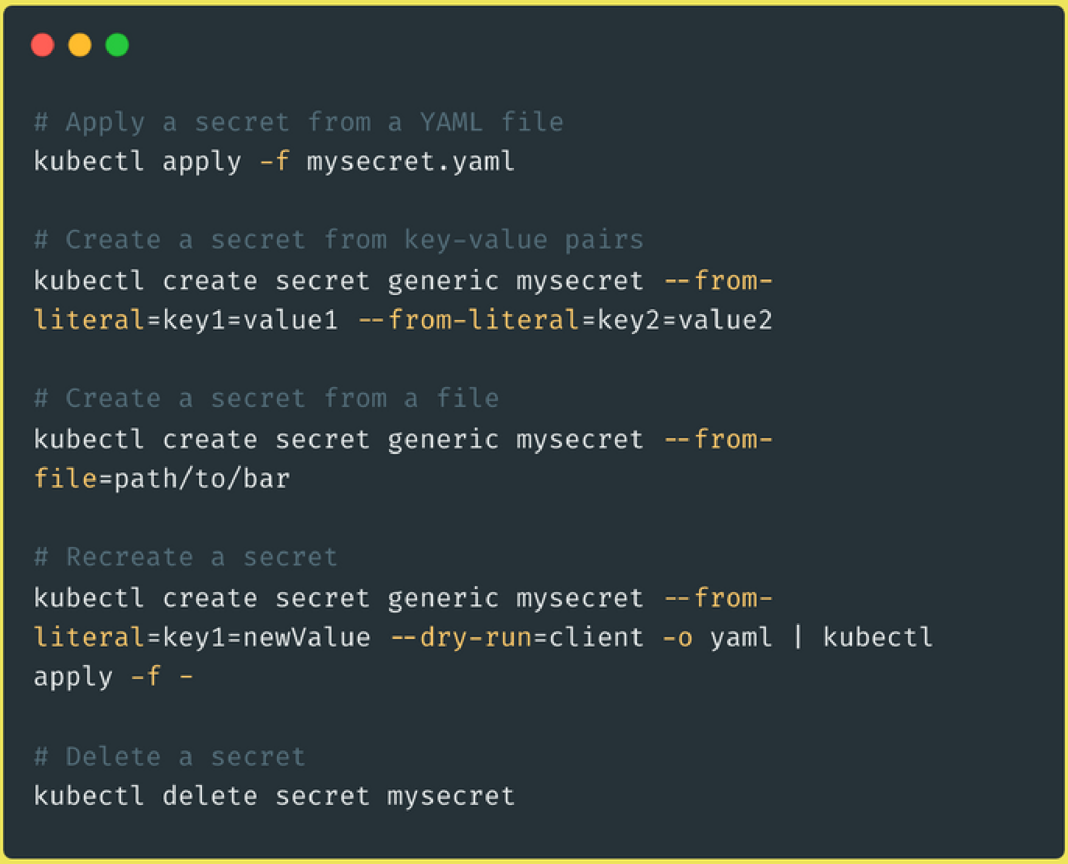
```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  key1: dmFsdWUx==   # The base64 encoded value for "value1"
  key2: dmFsdWUy==   # The base64 encoded value for "value2"
```

Secret



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mycontainer
    image: nginx
    env:
      - name: SECRET_KEY
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: key1
```

Managing Secrets

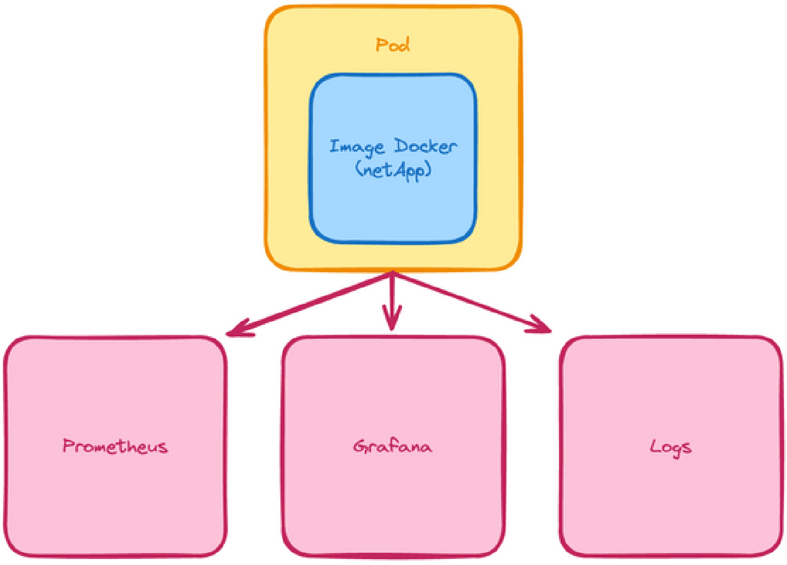


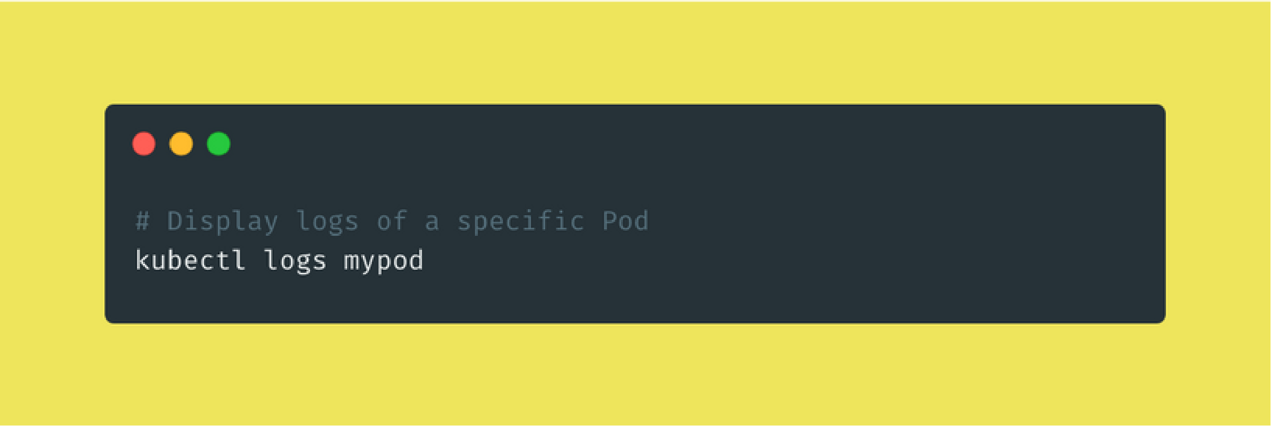
```
# Apply a secret from a YAML file
kubectl apply -f mysecret.yaml

# Create a secret from key-value pairs
kubectl create secret generic mysecret --from-literal=key1=value1 --from-literal=key2=value2

# Create a secret from a file
kubectl create secret generic mysecret --from-file=path/to/bar

# Recreate a secret
kubectl create secret generic mysecret --from-literal=key1=newValue --dry-run=client -o yaml | kubectl apply -f -

# Delete a secret
kubectl delete secret mysecret
```

Managing Logs

Monitoring is crucial for maintaining the health and performance of your Kubernetes cluster. It involves monitoring resources, performance, and the health of Pods, nodes, and other components.

Kubernetes can be integrated with various monitoring tools such as Prometheus, Grafana, etc.





```
# Display logs of a specific Pod
kubectl logs mypod
```