Enhanced Kubernetes CLI Guide - Basic to Intermediate

August 2025 - Version 2.1 with Comprehensive Examples, Demos, and Mini-Projects

Table of Contents

1. Introduction

This guide serves as your comprehensive companion for mastering Kubernetes through the kubectl command-line interface. Think of kubectl as your Swiss Army knife for Kubernetes - it's the primary tool that translates your intentions into actions within the cluster.

Unlike simple tutorials that just show you commands, this guide explains the reasoning behind each operation, helping you develop the intuition needed to troubleshoot problems and make informed decisions in real production environments.

2. Prerequisites

Before diving into kubectl commands, ensure you have the following foundation:

- kubectl installed - Verify with `kubectl version --client` (this checks your local kubectl without needing cluster access)
- Cluster access - This could be Minikube for local development, or managed services like GKE, EKS, or AKS
- Command-line comfort - Basic terminal navigation and file editing skills
- YAML understanding - Kubernetes uses YAML extensively for configuration (see Appendix A for basics)

Demo: Setting Up Your Learning Environment

Let's start by setting up a proper learning environment that you can use throughout this guide.

```bash
# Check if kubectl is installed and working
kubectl version --client

# If you don't have a cluster yet, install Minikube for local development
# On macOS:
brew install minikube

# On Linux:
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube

# Start your local cluster
minikube start

# Verify your cluster is running
kubectl cluster-info
kubectl get nodes
```

Example: First Cluster Interaction

```bash
# This command gives you an overview of your cluster's health
kubectl get componentstatuses

# See what's running in the system namespace
kubectl get pods -n kube-system

# Check available resources in your cluster
kubectl api-resources | head -20
```

Mini-Project: Environment Validation Checklist

Create a simple script that validates your Kubernetes environment is ready for learning:

```bash
```

```bash
#!/bin/bash
# save as check-k8s-env.sh

echo "🔍 Checking Kubernetes Environment..."

# Check kubectl installation
if command -v kubectl &> /dev/null; then
    echo "✅ kubectl is installed"
    kubectl version --client --short
else
    echo "❌ kubectl not found"
    exit 1
fi

# Check cluster connectivity
if kubectl cluster-info &> /dev/null; then
    echo "✅ Cluster is accessible"
    kubectl get nodes --no-headers | wc -l | xargs echo "📊 Nodes available:"
else
    echo "❌ Cannot connect to cluster"
    exit 1
fi

# Check permissions
if kubectl auth can-i create pods &> /dev/null; then
    echo "✅ You have pod creation permissions"
else
    echo "⚠️  Limited permissions - some exercises may not work"
fi

echo "🎉 Environment ready for learning!"
```

Make it executable and run it: `chmod +x check-k8s-env.sh && ./check-k8s-env.sh`

3. Core Concepts

Understanding these fundamental concepts will help you grasp why certain kubectl commands exist and when to use them:

- **Pods**: The atomic unit of deployment - think of them as wrappers around your containers that provide shared networking and storage
- **ReplicaSets**: Ensure consistency by maintaining a desired number of identical pod replicas

- **Deployments**: Higher-level controllers that manage ReplicaSets, enabling smooth updates and rollbacks
- **Services**: Provide stable networking endpoints since pod IPs are ephemeral
- **ConfigMaps/Secrets**: Separate configuration from code, with Secrets specifically for sensitive data
- **Persistent Volumes**: Enable data persistence beyond pod lifecycles
- **Namespaces**: Provide logical isolation within clusters, similar to folders for organizing resources

Example: Understanding the Kubernetes Architecture

Let's explore your cluster's architecture to understand these concepts better:

```bash
# See the control plane components
kubectl get pods -n kube-system

# Understanding what each component does:
# - etcd: The database that stores all cluster state
# - kube-apiserver: The API that all components talk to
# - kube-scheduler: Decides which node to place pods on
# - kube-controller-manager: Runs various controllers

# Look at a node's details to see the worker components
kubectl describe node $(kubectl get nodes -o jsonpath='{.items[0].metadata.name}')
# You'll see kubelet and kube-proxy listed
```

Demo: The Pod Lifecycle

Understanding how pods work is fundamental to everything else in Kubernetes:

```bash
```

```
# Create a simple pod and watch its lifecycle
kubectl run lifecycle-demo --image=nginx --restart=Never

# Watch it start up (run this in another terminal)
kubectl get pods -w

# See detailed information about what happened
kubectl describe pod lifecycle-demo

# The Events section shows you the lifecycle:
# 1. Scheduled - assigned to a node
# 2. Pulling - downloading the container image
# 3. Pulled - image download complete
# 4. Created - container created
# 5. Started - container started
```

4. Essential kubectl Commands

4.1 Basic Setup

Setting up your kubectl environment properly saves time and reduces errors throughout your Kubernetes journey.

```bash
```

```bash
# Enable bash autocompletion - this dramatically speeds up command entry
# and helps prevent typos by showing available options as you type
source <(kubectl completion bash)
# Add to your shell profile to make permanent
echo "source <(kubectl completion bash)" >> ~/.bashrc

# Check cluster connection - essential first step to verify you're talking to the right cluster
kubectl cluster-info
# This shows the cluster's API server URL and other core services

# Get detailed version information for both client and server
kubectl version --output=yaml
# Useful for troubleshooting compatibility issues between kubectl and cluster versions

# List cluster nodes - gives you an overview of your cluster's compute resources
kubectl get nodes
# Shows basic node status (Ready/NotReady)
kubectl get nodes -o wide
# Extended information including IP addresses, OS versions, and container runtime details
# Critical for understanding your cluster's infrastructure
```

4.2 Help and Documentation

kubectl's built-in help system is incredibly comprehensive. Learning to navigate it effectively makes you self-sufficient.

```bash
```

```
# General help - your starting point for any kubectl exploration
kubectl -h
kubectl -h | less # Pipe through less for easier reading of long output

# Command-specific help - drill down into specific command details
kubectl get -h
# Shows all the options for the 'get' command, including output formats and selectors

kubectl create -h | less
# The 'create' command has many subcommands, this shows them all

kubectl create deploy -h | less
# Specific help for creating deployments imperatively

# Explain Kubernetes objects - this is like having the Kubernetes documentation at your fingertips
kubectl explain pod
# Shows the structure and purpose of Pod resources

kubectl explain pod.spec | less
# Drill down into specific sections - here, the pod specification

kubectl explain deployment.spec.template
# Navigate nested object structures to understand complex configurations
```

4.3 API Resources and Versions

Understanding what resources are available and their API versions helps you write correct YAML and troubleshoot issues.

```bash
# List all API resources - shows every type of object you can create
kubectl api-resources | less
# Notice the SHORTNAMES column - these are useful aliases (e.g., 'po' for 'pods')

# Filter by scope to understand resource organization
kubectl api-resources --namespaced=true # Resources that belong to namespaces
kubectl api-resources --namespaced=false # Cluster-wide resources

# List API versions - important for writing YAML files
kubectl api-versions | less
# Shows available API groups and versions (e.g., apps/v1, networking.k8s.io/v1)
```

Example: Exploring API Resources

Let's understand what resources are available in your cluster:

```bash
# Find all resources related to storage
kubectl api-resources | grep -i storage

# See what networking resources exist
kubectl api-resources | grep -i network

# Understand which resources are cluster-scoped vs namespace-scoped
echo "=== Cluster-scoped resources ==="
kubectl api-resources --namespaced=false | head -10
echo "=== Namespace-scoped resources ==="
kubectl api-resources --namespaced=true | head -10
```

Demo: Using kubectl explain Effectively

The `kubectl explain` command is like having interactive documentation:

```bash
# Start with a high-level view
kubectl explain deployment

# Drill down into specifications
kubectl explain deployment.spec

# Go deeper into the pod template
kubectl explain deployment.spec.template.spec

# Understand container specifications
kubectl explain deployment.spec.template.spec.containers

# This hierarchical exploration helps you understand how YAML is structured
```

Mini-Project: kubectl Command Reference Builder

Create your own quick reference sheet by exploring kubectl's capabilities:

```bash
```

```
#!/bin/bash
# save as build-kubectl-reference.sh

echo "# My kubectl Quick Reference" > kubectl-reference.md
echo "Generated on $(date)" >> kubectl-reference.md
echo "" >> kubectl-reference.md

echo "## Common Resource Shortcuts" >> kubectl-reference.md
kubectl api-resources | grep -E "(pods|services|deployments|configmaps|secrets)" | \
awk '{print "- " $1 " (" $2 ")"}' >> kubectl-reference.md

echo "" >> kubectl-reference.md
echo "## Available API Versions" >> kubectl-reference.md
kubectl api-versions | head -10 | awk '{print "- " $1}' >> kubectl-reference.md

echo "" >> kubectl-reference.md
echo "## Cluster Info" >> kubectl-reference.md
echo "- Cluster: $(kubectl config current-context)" >> kubectl-reference.md
echo "- Server Version: $(kubectl version --short | grep Server | cut -d' ' -f3)" >> kubectl-reference.md
echo "- Nodes: $(kubectl get nodes --no-headers | wc -l)" >> kubectl-reference.md

echo "Reference sheet created: kubectl-reference.md"
```

5. Working with Pods

Pods are the fundamental building blocks of Kubernetes applications. While you'll typically use higher-level controllers in production, understanding pods directly is crucial for debugging and learning.

5.1 Creating Pods

```bash
```

```bash
# Run a simple Pod - useful for quick testing and debugging
kubectl run nginx-pod --image=nginx --restart=Never
# --restart=Never creates a Pod instead of a Deployment
# This is perfect for one-off tasks or testing

# Run a Pod that sleeps - great for creating a persistent container for debugging
kubectl run busybox-pod --image=busybox --restart=Never -- sleep 3600
# The '--' separates kubectl arguments from container arguments
# sleep 3600 keeps the container running for an hour

# Run an interactive Pod - immediate access for troubleshooting
kubectl run -it busybox --image=busybox --restart=Never -- sh
# -it combines -i (interactive) and -t (TTY) for a proper shell experience
# This creates and immediately connects you to the container
```

5.2 Inspecting Pods

Visibility into pod status and configuration is essential for both operations and debugging.

```bash
# List Pods - your first step in checking application status
kubectl get pods
# Shows basic status: NAME, READY, STATUS, RESTARTS, AGE

kubectl get pods -o wide
# Extended view including IP addresses and node placement
# Critical for understanding network issues and resource distribution

kubectl get pods --show-labels
# Labels are key-value pairs used for organization and selection
# Essential for understanding how Services and other controllers select pods

kubectl get pods -A # All namespaces
# System-wide view - useful for cluster administration and troubleshooting

# Describe a Pod - detailed information including events and configuration
kubectl describe pod nginx-pod | less
# Events section at the bottom is crucial for debugging startup issues
# Shows the complete pod lifecycle from scheduling to running
```

5.3 Managing Pods

Interactive management of pods is essential for debugging and maintenance tasks.

```bash
# Execute commands in a Pod - run one-off commands without entering the container
kubectl exec nginx-pod -- ls /usr/share/nginx/html
# Useful for quick checks like verifying file existence or checking processes

# Interactive shell access - essential for debugging and exploration
kubectl exec -it nginx-pod -- /bin/bash
# -it provides interactive terminal access
# Choose the shell based on what's available in the container (/bin/sh for minimal containers)

# Port forwarding - access pod services from your local machine
kubectl port-forward nginx-pod 8080:80
# Maps local port 8080 to pod port 80
# Invaluable for testing services without exposing them externally

# Copy files to/from Pods - transfer configuration files or retrieve logs
kubectl cp local.txt nginx-pod:/tmp/
# Copy from local machine to pod
kubectl cp nginx-pod:/etc/hostname ./remote-hostname
# Copy from pod to local machine
# Useful for retrieving generated files or debugging information

# Delete Pods - clean up resources
kubectl delete pod nginx-pod
# Deletes specific pod
kubectl delete pod --all
# Deletes all pods in current namespace - use with caution!
```

5.4 Pod YAML Example

Declarative configuration is the preferred approach for production environments as it provides version control and repeatability.

```yaml
```

```yaml
# simple-pod.yaml
apiVersion: v1 # API version - v1 is the stable core API
kind: Pod # Resource type - fundamental unit of deployment
metadata: # Metadata section - information about the object
  name: simple-pod # Must be unique within namespace
  labels: # Key-value pairs for organization and selection
    app: web # Commonly used to group related components
    tier: frontend # Architectural layer designation
spec: # Specification - desired state of the pod
  containers: # List of containers - pods can have multiple containers
  - name: nginx # Container name within the pod
    image: nginx:1.20 # Specific version tag - avoid 'latest' in production
    ports: # Container ports - for documentation and service discovery
    - containerPort: 80 # Port the container listens on
    env: # Environment variables - configuration injection
    - name: ENVIRONMENT # Variable name
      value: "development" # Variable value
```

bash

```bash
# Create Pod from YAML - declarative approach
kubectl create -f simple-pod.yaml
# 'create' fails if resource already exists - good for initial creation

kubectl apply -f simple-pod.yaml
# 'apply' creates or updates - preferred for ongoing management
# Maintains configuration drift detection

# Export Pod YAML - useful for backing up configurations or creating templates
kubectl get pod simple-pod -o yaml > exported-pod.yaml
# Includes runtime information - you'll need to clean it up for reuse
```

Best Practice: Always use `kubectl apply` for declarative management as it maintains better state tracking and enables easier updates.

Example: Multi-Container Pod

Understanding multi-container pods is essential for advanced patterns like sidecars:

yaml

```yaml
# multi-container-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: web-server
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: shared-storage
      mountPath: /usr/share/nginx/html
  - name: content-updater
    image: busybox
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo $(date) > /shared/index.html; sleep 30; done"]
    volumeMounts:
    - name: shared-storage
      mountPath: /shared
  volumes:
  - name: shared-storage
    emptyDir: {}
```

bash

```bash
# Create the multi-container pod
kubectl apply -f multi-container-pod.yaml

# Check both containers are running
kubectl get pod multi-container-pod

# See logs from each container
kubectl logs multi-container-pod -c web-server
kubectl logs multi-container-pod -c content-updater

# Connect to specific containers
kubectl exec -it multi-container-pod -c web-server -- /bin/bash
kubectl exec -it multi-container-pod -c content-updater -- /bin/sh
```

Demo: Pod Lifecycle and Health Checks

Let's create a pod with health checks to understand how Kubernetes manages pod health:

```yaml
# healthy-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: healthy-pod
spec:
  containers:
  - name: web
    image: nginx
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 10
      periodSeconds: 5
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 3
```

```bash
# Create the pod and watch its status
kubectl apply -f healthy-pod.yaml
kubectl get pod healthy-pod -w

# Check the probe status
kubectl describe pod healthy-pod | grep -A 5 -B 5 "Liveness\|Readiness"

# Test what happens when health checks fail
kubectl exec healthy-pod -- rm /usr/share/nginx/html/index.html
# Watch the pod get restarted due to failed liveness probe
```

Mini-Project: Pod Debugging Toolkit

Create a comprehensive pod debugging session:

```bash
#!/bin/bash
# save as pod-debug-toolkit.sh

POD_NAME=${1:-debug-pod}

echo "🔧 Creating debugging pod: $POD_NAME"

# Create a feature-rich debugging pod
kubectl run $POD_NAME --image=nicolaka/netshoot --restart=Never -- sleep 3600

echo "⏳ Waiting for pod to be ready..."
kubectl wait --for=condition=Ready pod/$POD_NAME --timeout=60s

echo "🎯 Pod ready! Here's what you can do:"
echo "1. Network debugging:"
echo "   kubectl exec -it $POD_NAME -- nslookup kubernetes.default"
echo "   kubectl exec -it $POD_NAME -- curl -I https://httpbin.org/get"

echo "2. DNS testing:"
echo "   kubectl exec -it $POD_NAME -- dig @8.8.8.8 google.com"

echo "3. Interactive shell:"
echo "   kubectl exec -it $POD_NAME -- bash"

echo "4. Copy files:"
echo "   kubectl cp /path/to/local/file $POD_NAME:/tmp/"

echo "5. Port forwarding (if needed):"
echo "   kubectl port-forward $POD_NAME 8080:80"

echo "🧹 Cleanup when done:"
echo "   kubectl delete pod $POD_NAME"
```

6. Deployments and ReplicaSets

Deployments are the workhorses of Kubernetes, providing scalability, high availability, and smooth updates. They manage ReplicaSets, which in turn manage Pods.

6.1 Managing Deployments

```bash
# Create Deployment imperatively - quick for testing and learning
kubectl create deployment nginx-deploy --image=nginx --replicas=3
# Creates a deployment with 3 pod replicas automatically

# List Deployments - overview of your application deployments
kubectl get deployments
kubectl get deploy -o wide # 'deploy' is a shorthand alias
# Shows desired vs current replica counts and update status

# Scale Deployments - adjust capacity based on demand
kubectl scale deployment nginx-deploy --replicas=5
# Horizontal scaling - increases pod count
# Kubernetes automatically distributes pods across available nodes

kubectl scale deployment nginx-deploy --replicas=0
# Scale to zero - temporarily shut down application while keeping configuration
# Useful for maintenance or cost optimization in development environments

# Edit Deployment live - quick fixes in non-production environments
kubectl edit deployment nginx-deploy
# Opens deployment configuration in your default editor
# Changes are applied immediately - use carefully in production

# Update Deployment image - rolling update to new version
kubectl set image deployment nginx-deploy nginx=nginx:1.21
# Triggers rolling update - old pods replaced gradually
# Ensures zero-downtime deployment if health checks are configured

# Monitor rollout status - track deployment progress
kubectl rollout status deployment nginx-deploy
# Shows real-time update progress

# Rollback deployment - return to previous version
kubectl rollout undo deployment nginx-deploy
# Reverts to previous deployment revision
# Essential for quick recovery from problematic updates
```

6.2 Deployment YAML Example

Production deployments require careful resource management and health checking to ensure reliability.

```yaml
```

```yaml
# nginx-deployment.yaml
apiVersion: apps/v1 # Apps API group - for workload controllers
kind: Deployment # Deployment controller - manages replica sets
metadata:
  name: nginx-deployment
  labels:
    app: nginx # Labels for the deployment itself
spec:
  replicas: 3 # Desired number of pod instances
  selector: # How deployment finds pods to manage
    matchLabels:
      app: nginx # Must match template labels below
  template: # Pod template - blueprint for created pods
    metadata:
      labels:
        app: nginx # Labels applied to created pods
    spec:
      containers:
      - name: nginx
        image: nginx:1.20 # Specific version for predictability
        ports:
        - containerPort: 80 # Port container listens on
        resources: # Resource management - crucial for cluster stability
          requests: # Minimum resources guaranteed to container
            memory: "64Mi" # Memory request - used for scheduling decisions
            cpu: "250m" # CPU request - 250 millicores (0.25 cores)
          limits: # Maximum resources container can use
            memory: "128Mi" # Memory limit - container killed if exceeded
            cpu: "500m" # CPU limit - container throttled if exceeded
        livenessProbe: # Health check - restarts container if failing
          httpGet: # HTTP-based health check
            path: / # Health check endpoint
            port: 80 # Port to check
          initialDelaySeconds: 15 # Wait before first check
          periodSeconds: 10 # How often to check
        readinessProbe: # Readiness check - removes from service if failing
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 5
```

```bash
# Create Deployment from YAML - production approach
kubectl apply -f nginx-deployment.yaml
# Apply maintains configuration state and enables updates

# Generate Deployment YAML - quick template creation
kubectl create deployment myapp --image=nginx --replicas=3 --dry-run=client -o yaml > myapp.yaml
# --dry-run=client generates YAML without creating resources
# Excellent starting point for custom deployments
```

Example: Rolling Updates in Action

Let's see how Kubernetes handles updates without downtime:

```bash
# Create a deployment with multiple replicas
kubectl create deployment rolling-demo --image=nginx:1.19 --replicas=4

# Expose it so we can test connectivity during updates
kubectl expose deployment rolling-demo --port=80 --type=NodePort

# Check initial version in all pods
kubectl get pods -l app=rolling-demo -o jsonpath='{range .items[*]}{.metadata.name}{"\t"}{.spec.containers[0].image}{"

# Update to a new version and watch the rolling update
kubectl set image deployment rolling-demo nginx=nginx:1.20

# In another terminal, watch the rollout happen
kubectl rollout status deployment rolling-demo -w

# Check rollout history
kubectl rollout history deployment rolling-demo

# Rollback if needed
kubectl rollout undo deployment rolling-demo
```

Demo: Understanding ReplicaSets

Deployments create and manage ReplicaSets. Let's explore this relationship:

```bash
```

```bash
# Create a deployment
kubectl create deployment rs-demo --image=nginx --replicas=3

# See the ReplicaSet created by the deployment
kubectl get replicasets
kubectl get rs -l app=rs-demo

# Get the ReplicaSet name
RS_NAME=$(kubectl get rs -l app=rs-demo -o jsonpath='{.items[0].metadata.name}')
echo "ReplicaSet name: $RS_NAME"

# Describe the ReplicaSet to understand its role
kubectl describe rs $RS_NAME

# Delete a pod and watch ReplicaSet recreate it
POD_NAME=$(kubectl get pods -l app=rs-demo -o jsonpath='{.items[0].metadata.name}')
kubectl delete pod $POD_NAME

# Watch new pod being created
kubectl get pods -l app=rs-demo -w
```

Mini-Project: Blue-Green Deployment Simulation

Create a blue-green deployment pattern using two deployments:

```bash
```

```bash
#!/bin/bash
# save as blue-green-demo.sh

echo "🔵 Creating Blue deployment..."
kubectl create deployment blue-app --image=nginx:1.19 --replicas=3
kubectl label deployment blue-app version=blue

echo "🟢 Creating Green deployment..."
kubectl create deployment green-app --image=nginx:1.20 --replicas=3
kubectl label deployment green-app version=green

echo "🔗 Creating service pointing to Blue..."
kubectl create service clusterip app-service --tcp=80:80
kubectl patch service app-service -p '{"spec":{"selector":{"app":"blue-app"}}}'

echo "📊 Current setup:"
kubectl get deployments -l 'version in (blue,green)'
kubectl get service app-service -o yaml | grep -A 5 selector

echo "🔄 To switch to Green:"
echo "kubectl patch service app-service -p '{\"spec\":{\"selector\":{\"app\":\"green-app\"}}}'"

echo "🧹 Cleanup:"
echo "kubectl delete deployment blue-app green-app"
echo "kubectl delete service app-service"
```

7. Services and Networking

Services provide stable networking for ephemeral pods. Understanding service types is crucial for application connectivity.

7.1 Creating Services

```bash
```

```bash
# Expose as ClusterIP (default) - internal cluster communication only
kubectl expose deployment nginx-deploy --port=80 --target-port=80 --name=nginx-service
# --port: port exposed by service
# --target-port: port on pods that traffic is forwarded to
# ClusterIP is only accessible within cluster

# Expose as NodePort - accessible from outside cluster via node IPs
kubectl expose deployment nginx-deploy --port=80 --type=NodePort --name=nginx-nodeport
# Kubernetes assigns random high port (30000-32767) on each node
# Useful for development but not recommended for production

# Expose as LoadBalancer - cloud provider creates external load balancer
kubectl expose deployment nginx-deploy --port=80 --type=LoadBalancer
# Only works with cloud providers that support load balancer integration
# Provides external IP address for accessing service

# Create Service imperatively with specific options
kubectl create service clusterip nginx-svc --tcp=80:80
# More explicit than expose command - useful for specific configurations

kubectl create service nodeport nginx-nodeport --tcp=80:80
# Creates NodePort service with explicit port mapping
```

7.2 Managing Services

bash

```
# List Services - overview of cluster networking
kubectl get services
kubectl get svc -o wide # 'svc' is shorthand for services
kubectl get svc --show-labels # Shows service labels for debugging selectors

# Describe Service - detailed service configuration and endpoints
kubectl describe service nginx-service
# Shows service configuration, endpoints, and events
# Critical for debugging connectivity issues

# Check Endpoints - verify service is routing to healthy pods
kubectl get endpoints
# Shows IP addresses of pods behind each service
kubectl describe endpoints nginx-service
# Detailed endpoint information - empty endpoints indicate selection issues

# Edit Service - modify service configuration
kubectl edit service nginx-service
# Opens service configuration in editor for live changes
```

7.3 Service YAML Example

yaml

```yaml
# nginx-service.yaml
apiVersion: v1 # Core API - services are fundamental networking primitive
kind: Service # Service provides stable networking endpoint
metadata:
  name: nginx-service # Service name - used by other resources for discovery
spec:
  selector: # Pod selection criteria - must match pod labels
    app: nginx # Selects pods with label app=nginx
  ports: # Port configuration - can have multiple ports
  - protocol: TCP # Protocol - TCP or UDP
    port: 80 # Port exposed by service (what clients connect to)
    targetPort: 80 # Port on pods (where traffic is forwarded)
  type: ClusterIP # Service type - ClusterIP, NodePort, LoadBalancer, or ExternalName
  # ClusterIP: internal cluster access only
  # NodePort: accessible via node IPs on high ports
  # LoadBalancer: external load balancer (cloud provider dependent)
```

bash

```
# Create Service from YAML
kubectl apply -f nginx-service.yaml
# Declarative approach - recommended for production environments
```

Example: Service Discovery in Action

Let's explore how services enable pod-to-pod communication:

```bash
# Create two deployments - a web server and a client
kubectl create deployment web-server --image=nginx --replicas=2
kubectl create deployment client --image=busybox -- sleep 3600

# Expose the web server
kubectl expose deployment web-server --port=80 --name=web-service

# Test service discovery from the client pod
CLIENT_POD=$(kubectl get pod -l app=client -o jsonpath='{.items[0].metadata.name}')

# Test DNS resolution
kubectl exec $CLIENT_POD -- nslookup web-service

# Test HTTP connectivity
kubectl exec $CLIENT_POD -- wget -qO- web-service

# Show how service provides load balancing
for i in {1..5}; do
  echo "Request $i:"
  kubectl exec $CLIENT_POD -- wget -qO- web-service | grep "Welcome to nginx"
done
```

Demo: Service Types Comparison

Understanding the different service types and when to use each:

```yaml
```

```yaml
# service-types-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: service-demo
  template:
    metadata:
      labels:
        app: service-demo
    spec:
      containers:
      - name: web
        image: httpd:2.4
        ports:
        - containerPort: 80
---
# ClusterIP Service - internal only
apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  selector:
    app: service-demo
  ports:
  - port: 80
    targetPort: 80
---
# NodePort Service - external access via nodes
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  selector:
    app: service-demo
```

```
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30080
```

```bash
# Apply the demo
kubectl apply -f service-types-demo.yaml

# Compare the services
kubectl get services -l app=service-demo

# Test ClusterIP (only works from within cluster)
kubectl run test-pod --image=busybox --rm -it --restart=Never -- wget -qO- clusterip-service

# Test NodePort (accessible from outside cluster)
NODE_IP=$(kubectl get nodes -o jsonpath='{.items[0].status.addresses[?(@.type=="ExternalIP")].address}')
if [ -z "$NODE_IP" ]; then
  NODE_IP=$(kubectl get nodes -o jsonpath='{.items[0].status.addresses[?(@.type=="InternalIP")].address}')
fi
echo "Access via: http://$NODE_IP:30080"
```

Mini-Project: Service Mesh Exploration

Create a simple microservices application to understand service communication:

```yaml
```

```yaml
# microservices-demo.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: frontend
spec:
 replicas: 2
 selector:
  matchLabels:
   app: frontend
 template:
  metadata:
   labels:
    app: frontend
  spec:
   containers:
   - name: frontend
    image: nginx:alpine
    ports:
    - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: backend
spec:
 replicas: 3
 selector:
  matchLabels:
   app: backend
 template:
  metadata:
   labels:
    app: backend
  spec:
   containers:
   - name: backend
    image: httpd:alpine
    ports:
    - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
  name: database
spec:
  replicas: 1
  selector:
    matchLabels:
      app: database
  template:
    metadata:
      labels:
        app: database
    spec:
      containers:
      - name: database
        image: postgres:13
        env:
        - name: POSTGRES_DB
          value: "myapp"
        - name: POSTGRES_USER
          value: "user"
        - name: POSTGRES_PASSWORD
          value: "password"
        ports:
        - containerPort: 5432
---
# Services for each tier
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30081
---
apiVersion: v1
kind: Service
metadata:
  name: backend-service
spec:
```

```yaml
  selector:
    app: backend
  ports:
  - port: 80
    targetPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: database-service
spec:
  selector:
    app: database
  ports:
  - port: 5432
    targetPort: 5432
```

```bash
```

```bash
# Deploy the microservices
kubectl apply -f microservices-demo.yaml

# Create a script to test service connectivity
cat << 'EOF' > test-connectivity.sh
#!/bin/bash
echo "🌐 Testing Microservices Connectivity"

# Create a test pod
kubectl run connectivity-test --image=busybox --rm -it --restart=Never -- sh -c "
echo 'Testing DNS resolution:'
nslookup frontend-service
nslookup backend-service
nslookup database-service

echo 'Testing HTTP connectivity:'
wget -qO- frontend-service || echo 'Frontend not responding'
wget -qO- backend-service || echo 'Backend not responding'

echo 'Testing database connectivity:'
nc -zv database-service 5432 || echo 'Database not accessible'
"
EOF

chmod +x test-connectivity.sh
./test-connectivity.sh
```

8. Ingress

Ingress manages external HTTP/HTTPS access to services, providing features like SSL termination and path-based routing.

8.1 Creating Ingress

```bash
```

```
# Create Ingress imperatively - quick setup for testing
kubectl create ingress nginx-ingress --rule="example.com/=nginx-service:80"
# Maps example.com to nginx-service on port 80
# Requires ingress controller to be installed in cluster

# List Ingress resources - view external routing configuration
kubectl get ingress
kubectl describe ingress nginx-ingress
# Shows routing rules and backend service status
```

Best Practice: Ensure an Ingress controller (e.g., nginx-ingress, Traefik) is deployed in your cluster before creating Ingress resources.

```yaml
# nginx-ingress.yaml
apiVersion: networking.k8s.io/v1 # Networking API group
kind: Ingress # Ingress resource for HTTP routing
metadata:
  name: nginx-ingress
  annotations: # Ingress controller specific configuration
    nginx.ingress.kubernetes.io/rewrite-target: / # URL rewriting
spec:
  rules: # Routing rules based on host and path
  - host: example.com # Domain name for routing
    http:
      paths: # Path-based routing within domain
      - path: / # Root path
        pathType: Prefix # Path matching type (Prefix, Exact, ImplementationSpecific)
        backend: # Backend service configuration
          service:
            name: nginx-service # Service name to route to
            port:
              number: 80 # Service port
  tls: # HTTPS configuration (optional)
  - hosts:
    - example.com
    secretName: example-tls # Secret containing TLS certificate
```

```bash
```

```
# Apply Ingress
kubectl apply -f nginx-ingress.yaml
```

Example: Setting Up Ingress Controller

Before creating Ingress resources, you need an Ingress Controller:

```bash
# Install NGINX Ingress Controller (for minikube)
minikube addons enable ingress

# For other clusters, install using Helm or kubectl
kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.8.1/deploy/static/provider/

# Wait for the controller to be ready
kubectl wait --namespace ingress-nginx \
  --for=condition=ready pod \
  --selector=app.kubernetes.io/component=controller \
  --timeout=90s
```

Demo: Path-Based Routing

Create an ingress that routes different paths to different services:

```yaml
```

```yaml
# path-based-ingress.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: app1
spec:
 replicas: 2
 selector:
  matchLabels:
   app: app1
 template:
  metadata:
   labels:
    app: app1
  spec:
   containers:
   - name: app1
    image: nginx:alpine
    ports:
    - containerPort: 80
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: app2
spec:
 replicas: 2
 selector:
  matchLabels:
   app: app2
 template:
  metadata:
   labels:
    app: app2
  spec:
   containers:
   - name: app2
    image: httpd:alpine
    ports:
    - containerPort: 80
---
apiVersion: v1
kind: Service
```

```yaml
metadata:
  name: app1-service
spec:
  selector:
    app: app1
  ports:
  - port: 80
---
apiVersion: v1
kind: Service
metadata:
  name: app2-service
spec:
  selector:
    app: app2
  ports:
  - port: 80
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: path-based-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: myapp.local
    http:
      paths:
      - path: /app1
        pathType: Prefix
        backend:
          service:
            name: app1-service
            port:
              number: 80
      - path: /app2
        pathType: Prefix
        backend:
          service:
            name: app2-service
            port:
              number: 80
```

```bash
# Deploy the path-based routing demo
kubectl apply -f path-based-ingress.yaml

# Get the ingress IP
kubectl get ingress path-based-ingress

# Test the routing (you may need to add myapp.local to /etc/hosts)
# Add this line to /etc/hosts: <INGRESS_IP> myapp.local
curl http://myapp.local/app1
curl http://myapp.local/app2
```

Mini-Project: HTTPS Ingress with Self-Signed Certificate

Create a complete HTTPS setup with self-signed certificates:

```bash
```

```bash
#!/bin/bash
# save as https-ingress-setup.sh

echo "🔐 Creating HTTPS Ingress with Self-Signed Certificate"

# Generate self-signed certificate
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout tls.key -out tls.crt \
  -subj "/CN=secure-app.local/O=secure-app.local"

# Create TLS secret
kubectl create secret tls secure-app-tls --key tls.key --cert tls.crt

# Create application and service
kubectl create deployment secure-app --image=nginx
kubectl expose deployment secure-app --port=80

# Create HTTPS Ingress
cat << EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: secure-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  tls:
  - hosts:
    - secure-app.local
    secretName: secure-app-tls
  rules:
  - host: secure-app.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: secure-app
            port:
              number: 80
EOF
```

```
echo "✅ HTTPS Ingress created!"
echo "📝 Add to /etc/hosts: \$(kubectl get ingress secure-ingress -o jsonpath='{.status.loadBalancer.ingress[0].ip}') secu
echo "🌐 Test with: curl -k https://secure-app.local"

# Cleanup script
echo "🧹 Cleanup with:"
echo "kubectl delete ingress secure-ingress"
echo "kubectl delete secret secure-app-tls"
echo "kubectl delete service secure-app"
echo "kubectl delete deployment secure-app"
echo "rm tls.key tls.crt"
```

9. ConfigMaps

ConfigMaps decouple configuration from application code, enabling environment-specific configurations without rebuilding container images.

```bash
```

```
# Create ConfigMap from literal values - direct key-value pairs
kubectl create configmap app-config \
  --from-literal=database_url=localhost:5432 \
  --from-literal=debug_mode=true
# Useful for simple configuration values
# Each --from-literal creates one key-value pair

# Create ConfigMap from file - entire file becomes value
echo "Hello World" > index.html
kubectl create configmap web-config --from-file=index.html
# File name becomes the key, file contents become the value
# Useful for configuration files, web content, etc.

# Create ConfigMap from environment file - structured configuration
cat > app.env << EOF
DATABASE_URL=localhost:5432
DEBUG_MODE=true
API_KEY=dev-key-123
EOF
kubectl create configmap env-config --from-env-file=app.env
# Each line becomes a separate key-value pair
# Useful for application environment variables

# List ConfigMaps - view available configuration
kubectl get configmaps
kubectl get cm app-config -o yaml # 'cm' is shorthand
kubectl describe configmap app-config
# Shows all keys and values stored in ConfigMap
```

9.1 Using ConfigMaps in Pods

ConfigMaps can be consumed as environment variables or mounted as files, providing flexibility for different application needs.

yaml

```yaml
# pod-with-configmap.yaml
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
  - name: app
    image: nginx
    env: # Environment variable injection
    - name: DATABASE_URL # Environment variable name in container
      valueFrom: # Source of the value
        configMapKeyRef: # Reference to ConfigMap key
          name: app-config # ConfigMap name
          key: database_url # Key within ConfigMap
    volumeMounts: # File system mounts
    - name: config-volume # Volume name (must match below)
      mountPath: /usr/share/nginx/html # Where to mount in container
  volumes: # Volume definitions
  - name: config-volume # Volume name
    configMap: # ConfigMap volume source
      name: web-config # ConfigMap name
    # Files appear as individual files in mount path
    # Key names become file names, values become file contents
```

bash

```bash
# Apply ConfigMap Pod
kubectl apply -f pod-with-configmap.yaml

# Verify ConfigMap usage
kubectl exec app-pod -- env | grep DATABASE_URL # Check environment variable
kubectl exec app-pod -- ls /usr/share/nginx/html # Check mounted files
```

Example: Configuration Hot-Reload

Demonstrate how ConfigMap changes can be reflected in running pods:

yaml

```yaml
# hot-reload-demo.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-settings
data:
  config.json: |
    {
      "app_name": "My Application",
      "version": "1.0.0",
      "debug": false
    }
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: config-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: config-app
  template:
    metadata:
      labels:
        app: config-app
    spec:
      containers:
      - name: app
        image: nginx:alpine
        volumeMounts:
        - name: config-volume
          mountPath: /etc/config
      volumes:
      - name: config-volume
        configMap:
          name: app-settings
```

bash

```
# Deploy the configuration demo
kubectl apply -f hot-reload-demo.yaml

# Check initial configuration
kubectl exec deployment/config-app -- cat /etc/config/config.json

# Update the ConfigMap
kubectl patch configmap app-settings -p '{"data":{"config.json":"{\"app_name\":\"Updated Application\",\"version\":\"2.

# Wait a moment for the volume to update (can take up to 60 seconds)
sleep 10

# Check updated configuration
kubectl exec deployment/config-app -- cat /etc/config/config.json
```

Demo: Multiple Configuration Sources

Show how to combine multiple configuration sources:

```bash
# Create multiple ConfigMaps for different purposes
kubectl create configmap database-config --from-literal=host=db.example.com --from-literal=port=5432
kubectl create configmap app-features --from-literal=feature_x=enabled --from-literal=feature_y=disabled
kubectl create configmap logging-config --from-file=- <<EOF
level: info
format: json
output: stdout
EOF
```

```yaml
```

```yaml
# multi-config-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-config-pod
spec:
  containers:
  - name: app
    image: busybox
    command: ["sleep", "3600"]
    env:
    # Environment variables from multiple ConfigMaps
    - name: DB_HOST
      valueFrom:
        configMapKeyRef:
          name: database-config
          key: host
    - name: DB_PORT
      valueFrom:
        configMapKeyRef:
          name: database-config
          key: port
    envFrom:
    # Load all keys from a ConfigMap as environment variables
    - configMapRef:
        name: app-features
        prefix: FEATURE_
    volumeMounts:
    - name: logging-config
      mountPath: /etc/logging
  volumes:
  - name: logging-config
    configMap:
      name: logging-config
```

Mini-Project: Environment-Specific Configuration Manager

Create a system to manage configurations for different environments:

```bash
```

```bash
#!/bin/bash
# save as config-manager.sh

ENVIRONMENT=${1:-development}

echo "🔧 Setting up configuration for environment: $ENVIRONMENT"

case $ENVIRONMENT in
  "development")
    kubectl create configmap app-config-$ENVIRONMENT \
      --from-literal=database_url=dev-db:5432 \
      --from-literal=debug_mode=true \
      --from-literal=log_level=debug \
      --from-literal=cache_ttl=60
    ;;
  "staging")
    kubectl create configmap app-config-$ENVIRONMENT \
      --from-literal=database_url=staging-db:5432 \
      --from-literal=debug_mode=false \
      --from-literal=log_level=info \
      --from-literal=cache_ttl=300
    ;;
  "production")
    kubectl create configmap app-config-$ENVIRONMENT \
      --from-literal=database_url=prod-db:5432 \
      --from-literal=debug_mode=false \
      --from-literal=log_level=warn \
      --from-literal=cache_ttl=3600
    ;;
esac

# Create application configuration file
cat << EOF > app-$ENVIRONMENT.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-$ENVIRONMENT
spec:
  replicas: $([ "$ENVIRONMENT" = "production" ] && echo 3 || echo 1)
  selector:
    matchLabels:
      app: myapp
      env: $ENVIRONMENT
```

```
  template:
    metadata:
      labels:
        app: myapp
        env: $ENVIRONMENT
    spec:
      containers:
      - name: app
        image: nginx:alpine
        envFrom:
        - configMapRef:
            name: app-config-$ENVIRONMENT
EOF

echo "📋 Configuration created for $ENVIRONMENT"
echo "🚀 Deploy with: kubectl apply -f app-$ENVIRONMENT.yaml"
echo "🔍 Check config: kubectl exec deployment/myapp-$ENVIRONMENT -- env | grep -E '(database_url|debug_mode
```

10. Secrets

Secrets store sensitive data like passwords, tokens, and certificates. They're similar to ConfigMaps but designed for confidential information.

```bash
```

```bash
# Create Secret from literal values - sensitive configuration
kubectl create secret generic db-secret \
  --from-literal=username=admin \
  --from-literal=password=secret123
# 'generic' type for arbitrary key-value pairs
# Values are base64 encoded (not encrypted - use external secret management for encryption)

# Create Docker registry Secret - for pulling private images
kubectl create secret docker-registry regcred \
  --docker-server=registry.example.com \
  --docker-username=user \
  --docker-password=pass \
  --docker-email=user@example.com
# Used by pods to authenticate with private registries

# List Secrets - view available secrets (values are hidden)
kubectl get secrets
kubectl get secret db-secret -o yaml # Shows base64 encoded values
kubectl describe secret db-secret # Shows keys but not values

# Decode Secret for troubleshooting - DO NOT DO IN PRODUCTION LOGS
kubectl get secret db-secret -o jsonpath='{.data.password}' | base64 -d
# Decodes base64 encoded secret value
# Use only for debugging - never expose in logs or scripts
```

Best Practice: Use Secrets for sensitive data, not ConfigMaps. Consider external secret management solutions (HashiCorp Vault, AWS Secrets Manager) for production environments.

Example: TLS Secrets for HTTPS

Create and manage TLS certificates as secrets:

```bash
```

```
# Generate a self-signed certificate
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout server.key -out server.crt \
  -subj "/CN=myapp.example.com"

# Create TLS secret from certificate files
kubectl create secret tls myapp-tls --cert=server.crt --key=server.key

# View the TLS secret (certificates are also base64 encoded)
kubectl describe secret myapp-tls
```

Demo: Secret Consumption Patterns

Show different ways pods can consume secrets:

```yaml
```

```yaml
# secret-consumption-demo.yaml
apiVersion: v1
kind: Secret
metadata:
  name: app-secrets
type: Opaque
data:
  # Values must be base64 encoded
  database-password: cGFzc3dvcmQxMjM=  # password123
  api-key: YWJjZGVmZ2hpams=  # abcdefghijk
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-consumer
spec:
  containers:
  - name: app
    image: busybox
    command: ["sleep", "3600"]
    env:
    # Method 1: Individual environment variables
    - name: DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: app-secrets
          key: database-password
    # Method 2: All secret keys as environment variables
    envFrom:
    - secretRef:
        name: app-secrets
    volumeMounts:
    # Method 3: Mount secrets as files
    - name: secret-volume
      mountPath: /etc/secrets
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: app-secrets
      # Optional: Set file permissions
      defaultMode: 0400
```

```bash
# Apply the demo
kubectl apply -f secret-consumption-demo.yaml

# Test different consumption methods
kubectl exec secret-consumer -- env | grep -E "(DB_PASSWORD|database-password|api-key)"
kubectl exec secret-consumer -- ls -la /etc/secrets
kubectl exec secret-consumer -- cat /etc/secrets/database-password
```

Mini-Project: Secure Application Deployment

Create a complete secure application deployment with database credentials:

```bash
```

```bash
#!/bin/bash
# save as secure-app-deployment.sh

echo "🔐 Creating Secure Application Deployment"

# Generate random database credentials
DB_USER="appuser"
DB_PASS=$(openssl rand -base64 32 | tr -d "=+/" | cut -c1-25)
DB_NAME="myappdb"

echo "📝 Generated credentials (store these securely!):"
echo "Database User: $DB_USER"
echo "Database Password: $DB_PASS"

# Create secret with database credentials
kubectl create secret generic db-credentials \
  --from-literal=username="$DB_USER" \
  --from-literal=password="$DB_PASS" \
  --from-literal=database="$DB_NAME"

# Create TLS certificate for the application
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout app.key -out app.crt \
  -subj "/CN=secure-app.local/O=MyOrg"

kubectl create secret tls app-tls --cert=app.crt --key=app.key

# Deploy PostgreSQL database
cat << EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
```

```
      containers:
      - name: postgres
        image: postgres:13
        env:
        - name: POSTGRES_USER
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: username
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: password
        - name: POSTGRES_DB
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: database
        ports:
        - containerPort: 5432
---
apiVersion: v1
kind: Service
metadata:
  name: postgres-service
spec:
  selector:
    app: postgres
  ports:
  - port: 5432
EOF

# Deploy the application
cat << EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: secure-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: secure-app
```

```
  template:
    metadata:
      labels:
        app: secure-app
    spec:
      containers:
      - name: app
        image: nginx:alpine
        env:
        - name: DB_HOST
          value: "postgres-service"
        - name: DB_USER
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: username
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: password
        volumeMounts:
        - name: tls-certs
          mountPath: /etc/ssl/certs
          readOnly: true
        ports:
        - containerPort: 80
      volumes:
      - name: tls-certs
        secret:
          secretName: app-tls
---
apiVersion: v1
kind: Service
metadata:
  name: secure-app-service
spec:
  selector:
    app: secure-app
  ports:
  - port: 80
EOF

echo "✅ Secure application deployed!"
```

```
echo "🔍 Check deployment: kubectl get all -l app=secure-app"
echo "🔐 View secrets: kubectl get secrets | grep -E '(db-credentials|app-tls)'"
echo "🧹 Cleanup: kubectl delete secret db-credentials app-tls && kubectl delete all -l app=postgres && kubectl dele

# Cleanup files
rm app.key app.crt
```

11. Persistent Storage

Persistent Volumes (PV) and Persistent Volume Claims (PVC) provide storage that survives pod restarts and rescheduling.

11.1 Persistent Volumes and Claims

```yaml
# pv-example.yaml - Cluster administrator defines available storage
apiVersion: v1
kind: PersistentVolume # Cluster-wide storage resource
metadata:
  name: data-pv
spec:
  capacity: # Storage capacity
    storage: 1Gi # 1 Gigabyte of storage
  accessModes: # How storage can be accessed
  - ReadWriteOnce # RWO: single node read-write access
  persistentVolumeReclaimPolicy: Retain # What happens when PVC is deleted
  storageClassName: manual # Storage class for dynamic provisioning
  hostPath: # Storage backend - hostPath is for development only
    path: /data # Directory on node (not suitable for production)
```

```yaml
```

```yaml
# pvc-example.yaml - Application requests storage
apiVersion: v1
kind: PersistentVolumeClaim # Request for storage
metadata:
  name: data-pvc # PVC name used by pods
spec:
  accessModes: # Required access mode
  - ReadWriteOnce # Must match available PV
  resources: # Storage requirements
    requests:
      storage: 1Gi # Amount of storage requested
  storageClassName: manual # Which storage class to use
```

bash

```bash
# Create PV and PVC
kubectl create -f pv-example.yaml # Admin creates storage
kubectl create -f pvc-example.yaml # Application requests storage

# List PV and PVC
kubectl get pv # Cluster-wide view of storage
kubectl get pvc # Namespace view of storage claims
kubectl describe pv data-pv # Shows PV details and binding status
kubectl describe pvc data-pvc # Shows PVC details and which PV it's bound to
```

11.2 Using PVC in Pods

yaml

```yaml
# pod-with-storage.yaml
apiVersion: v1
kind: Pod
metadata:
  name: storage-pod
spec:
  containers:
  - name: app
    image: nginx
    volumeMounts: # Mount point in container
    - mountPath: "/usr/share/nginx/html" # Where to mount storage
      name: storage # Volume name (must match below)
  volumes: # Volume definitions
  - name: storage # Volume name
    persistentVolumeClaim: # PVC volume source
      claimName: data-pvc # PVC name to use
```

```bash
# Apply Pod with PVC
kubectl apply -f pod-with-storage.yaml

# Verify storage mounting
kubectl exec storage-pod -- df -h # Check mounted filesystems
kubectl exec storage-pod -- touch /usr/share/nginx/html/test-file # Create test file
kubectl delete pod storage-pod # Delete pod
kubectl apply -f pod-with-storage.yaml # Recreate pod
kubectl exec storage-pod -- ls /usr/share/nginx/html # Verify file persistence
```

Example: Data Persistence Demo

Demonstrate how data survives pod restarts:

```yaml
```

```yaml
# persistence-demo.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: demo-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: Pod
metadata:
  name: writer-pod
spec:
  containers:
  - name: writer
    image: busybox
    command: ["sh", "-c"]
    args:
    - |
      echo "Writing data at $(date)" > /data/timestamp.txt
      echo "Pod: $HOSTNAME" >> /data/timestamp.txt
      echo "Random data: $RANDOM" >> /data/timestamp.txt
      sleep 30
    volumeMounts:
    - name: shared-storage
      mountPath: /data
  volumes:
  - name: shared-storage
    persistentVolumeClaim:
      claimName: demo-pvc
  restartPolicy: Never
```

bash

```
# Run the persistence demo
kubectl apply -f persistence-demo.yaml

# Wait for the pod to complete
kubectl wait --for=condition=complete pod/writer-pod --timeout=60s

# Create a reader pod to verify data persistence
kubectl run reader-pod --image=busybox --rm -it --restart=Never \
  --overrides='{"spec":{"volumes":[{"name":"shared-storage","persistentVolumeClaim":{"claimName":"demo-pvc"}}],"con
  -- cat /data/timestamp.txt
```

Demo: Storage Class Exploration

Understanding different storage classes and their characteristics:

```bash
# List available storage classes
kubectl get storageclass
kubectl get sc -o wide

# Describe a storage class to understand its parameters
kubectl describe storageclass standard

# Create PVCs with different storage classes
kubectl create pvc demo-pvc-standard --storageclass=standard --size=1Gi
kubectl create pvc demo-pvc-fast --storageclass=fast --size=1Gi  # if available

# Watch PVCs get bound to PVs
kubectl get pvc -w
```

Mini-Project: Multi-Pod Shared Storage

Create a scenario where multiple pods share data through persistent storage:

```yaml
```

```yaml
# shared-storage-project.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: shared-data-pvc
spec:
  accessModes:
  - ReadWriteMany  # Multiple pods can read/write
  resources:
    requests:
      storage: 2Gi
---
# Data Producer Pod
apiVersion: v1
kind: Pod
metadata:
  name: data-producer
  labels:
    role: producer
spec:
  containers:
  - name: producer
    image: busybox
    command: ["sh", "-c"]
    args:
    - |
      while true; do
        echo "$(date): Data from producer pod" >> /shared/data.log
        echo "Producer active: $(date)" > /shared/producer-status.txt
        sleep 10
      done
    volumeMounts:
    - name: shared-volume
      mountPath: /shared
  volumes:
  - name: shared-volume
    persistentVolumeClaim:
      claimName: shared-data-pvc
---
# Data Consumer Pod 1
apiVersion: v1
kind: Pod
metadata:
```

```yaml
  name: data-consumer-1
  labels:
    role: consumer
spec:
  containers:
  - name: consumer
    image: busybox
    command: ["sh", "-c"]
    args:
    - |
      while true; do
        echo "Consumer 1 reading at $(date)"
        tail -5 /shared/data.log 2>/dev/null || echo "No data yet"
        sleep 15
      done
    volumeMounts:
    - name: shared-volume
      mountPath: /shared
  volumes:
  - name: shared-volume
    persistentVolumeClaim:
      claimName: shared-data-pvc
---
# Data Consumer Pod 2
apiVersion: v1
kind: Pod
metadata:
  name: data-consumer-2
  labels:
    role: consumer
spec:
  containers:
  - name: consumer
    image: busybox
    command: ["sh", "-c"]
    args:
    - |
      while true; do
        echo "Consumer 2 analyzing at $(date)"
        wc -l /shared/data.log 2>/dev/null || echo "No data file yet"
        cat /shared/producer-status.txt 2>/dev/null || echo "Producer not active"
        sleep 20
      done
    volumeMounts:
```

```yaml
    - name: shared-volume
      mountPath: /shared
  volumes:
  - name: shared-volume
    persistentVolumeClaim:
      claimName: shared-data-pvc
```

```bash
# Deploy the shared storage project
kubectl apply -f shared-storage-project.yaml

# Monitor the shared data scenario
echo "🔍 Monitoring shared storage scenario..."
echo "📊 Watch PVC status:"
kubectl get pvc shared-data-pvc -w &

echo "📝 Monitor producer logs:"
kubectl logs -f data-producer &

echo "👀 Monitor consumer 1 logs:"
kubectl logs -f data-consumer-1 &

echo "📈 Monitor consumer 2 logs:"
kubectl logs -f data-consumer-2 &

# Test data sharing
sleep 30
echo "🧪 Testing data sharing:"
kubectl exec data-consumer-1 -- ls -la /shared
kubectl exec data-consumer-2 -- head /shared/data.log

echo "🧹 Cleanup:"
echo "kubectl delete -f shared-storage-project.yaml"
```

12. Storage Classes

Storage Classes enable dynamic provisioning of storage, allowing PVCs to automatically create PVs.

```bash
```

```bash
# List Storage Classes - see available storage types
kubectl get storageclass
kubectl get sc # 'sc' is shorthand
kubectl describe storageclass standard # Details about specific storage class
```

```yaml
# dynamic-pvc.yaml - PVC that triggers dynamic provisioning
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard # References storage class for dynamic provisioning
  # When created, this triggers automatic PV creation by storage class provisioner
```

```bash
# Create PVC with Storage Class
kubectl apply -f dynamic-pvc.yaml

# Watch dynamic provisioning happen
kubectl get pvc dynamic-pvc -w # Watch PVC status change from Pending to Bound
kubectl get pv # See automatically created PV
```

Example: Custom Storage Class

Create a custom storage class for specific requirements:

```yaml
```

```yaml
# custom-storage-class.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast-ssd
provisioner: kubernetes.io/gce-pd  # Cloud provider specific
parameters:
  type: pd-ssd  # SSD disk type
  replication-type: regional-pd  # Regional replication
allowVolumeExpansion: true  # Allow PVC size increases
reclaimPolicy: Delete  # Delete PV when PVC is deleted
volumeBindingMode: WaitForFirstConsumer  # Delay binding until pod is scheduled
```

Demo: Storage Performance Comparison

Compare different storage classes performance characteristics:

```bash
```

```bash
#!/bin/bash
# save as storage-performance-test.sh

echo "🚀 Storage Performance Comparison Test"

# Create PVCs with different storage classes
kubectl create pvc test-standard --storageclass=standard --size=1Gi
kubectl create pvc test-fast --storageclass=fast --size=1Gi  # if available

# Wait for PVCs to be bound
kubectl wait --for=condition=bound pvc/test-standard --timeout=60s
kubectl wait --for=condition=bound pvc/test-fast --timeout=60s || echo "Fast storage not available"

# Performance test function
test_storage() {
  local pvc_name=$1
  local test_name=$2

  echo "🔬 Testing $test_name storage..."

  kubectl run storage-test-$test_name --image=busybox --rm -it --restart=Never \
    --overrides="{
      \"spec\": {
        \"containers\": [{
          \"name\": \"test\",
          \"image\": \"busybox\",
          \"command\": [\"sh\", \"-c\", \"dd if=/dev/zero of=/data/testfile bs=1M count=100 && sync && echo 'Write test
          \"volumeMounts\": [{\"name\": \"test-volume\", \"mountPath\": \"/data\"}]
        }],
        \"volumes\": [{\"name\": \"test-volume\", \"persistentVolumeClaim\": {\"claimName\": \"$pvc_name\"}}]
      }
    }" -- sh -c "dd if=/dev/zero of=/data/testfile bs=1M count=100 && sync && echo 'Write test completed'"
}

# Run performance tests
test_storage "test-standard" "standard"
test_storage "test-fast" "fast"

echo "🧹 Cleanup test PVCs:"
echo "kubectl delete pvc test-standard test-fast"
```

Mini-Project: Storage Lifecycle Management

Create a comprehensive storage management system:

```bash

```

```bash
#!/bin/bash
# save as storage-lifecycle-manager.sh

ENVIRONMENT=${1:-dev}
APP_NAME=${2:-myapp}

echo "📦 Storage Lifecycle Manager for $APP_NAME in $ENVIRONMENT"

# Define storage requirements based on environment
case $ENVIRONMENT in
  "dev")
    STORAGE_SIZE="1Gi"
    STORAGE_CLASS="standard"
    BACKUP_ENABLED="false"
    ;;
  "staging")
    STORAGE_SIZE="5Gi"
    STORAGE_CLASS="standard"
    BACKUP_ENABLED="true"
    ;;
  "production")
    STORAGE_SIZE="20Gi"
    STORAGE_CLASS="premium"  # Assuming premium class exists
    BACKUP_ENABLED="true"
    ;;
esac

echo "📋 Storage Configuration:"
echo "  Size: $STORAGE_SIZE"
echo "  Class: $STORAGE_CLASS"
echo "  Backup: $BACKUP_ENABLED"

# Create PVC
cat << EOF | kubectl apply -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ${APP_NAME}-storage-${ENVIRONMENT}
  labels:
    app: $APP_NAME
    environment: $ENVIRONMENT
    backup-enabled: $BACKUP_ENABLED
  annotations:
```

```
    created-by: storage-lifecycle-manager
    created-date: $(date -Iseconds)
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: $STORAGE_SIZE
  storageClassName: $STORAGE_CLASS
EOF

# Create application that uses the storage
cat << EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ${APP_NAME}-${ENVIRONMENT}
  labels:
    app: $APP_NAME
    environment: $ENVIRONMENT
spec:
  replicas: $([ "$ENVIRONMENT" = "production" ] && echo 3 || echo 1)
  selector:
    matchLabels:
      app: $APP_NAME
      environment: $ENVIRONMENT
  template:
    metadata:
      labels:
        app: $APP_NAME
        environment: $ENVIRONMENT
    spec:
      containers:
      - name: app
        image: nginx:alpine
        ports:
        - containerPort: 80
        volumeMounts:
        - name: app-storage
          mountPath: /var/www/html
        resources:
          requests:
            memory: $([ "$ENVIRONMENT" = "production" ] && echo "256Mi" || echo "128Mi")
            cpu: $([ "$ENVIRONMENT" = "production" ] && echo "200m" || echo "100m")
```

```
        limits:
          memory: $([ "$ENVIRONMENT" = "production" ] && echo "512Mi" || echo "256Mi")
          cpu: $([ "$ENVIRONMENT" = "production" ] && echo "500m" || echo "250m")
      volumes:
      - name: app-storage
        persistentVolumeClaim:
          claimName: ${APP_NAME}-storage-${ENVIRONMENT}
EOF

# Create backup CronJob if backup is enabled
if [ "$BACKUP_ENABLED" = "true" ]; then
  cat << EOF | kubectl apply -f -
apiVersion: batch/v1
kind: CronJob
metadata:
  name: ${APP_NAME}-backup-${ENVIRONMENT}
  labels:
    app: $APP_NAME
    environment: $ENVIRONMENT
    component: backup
spec:
  schedule: "0 2 * * *"  # Daily at 2 AM
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: backup
            image: busybox
            command:
            - /bin/sh
            - -c
            - |
              echo "Starting backup at \$(date)"
              tar -czf /backup/\${APP_NAME}-\${ENVIRONMENT}-\$(date +%Y%m%d-%H%M%S).tar.gz -C /data .
              echo "Backup completed at \$(date)"
              # Keep only last 7 backups
              ls -t /backup/*.tar.gz | tail -n +8 | xargs -r rm
            volumeMounts:
            - name: app-data
              mountPath: /data
            - name: backup-storage
              mountPath: /backup
            env:
```

```
          - name: APP_NAME
            value: "$APP_NAME"
          - name: ENVIRONMENT
            value: "$ENVIRONMENT"
        volumes:
        - name: app-data
          persistentVolumeClaim:
            claimName: ${APP_NAME}-storage-${ENVIRONMENT}
        - name: backup-storage
          persistentVolumeClaim:
            claimName: ${APP_NAME}-backup-${ENVIRONMENT}
        restartPolicy: OnFailure
EOF

  # Create backup PVC
  cat << EOF | kubectl apply -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ${APP_NAME}-backup-${ENVIRONMENT}
  labels:
    app: $APP_NAME
    environment: $ENVIRONMENT
    component: backup
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: $(echo "$STORAGE_SIZE" | sed 's/Gi/*2Gi/' | bc 2>/dev/null || echo "2Gi")
  storageClassName: $STORAGE_CLASS
EOF

fi

echo "✅ Storage lifecycle setup complete!"
echo "🔍 Check resources:"
echo "  kubectl get pvc -l app=$APP_NAME,environment=$ENVIRONMENT"
echo "  kubectl get deployment -l app=$APP_NAME,environment=$ENVIRONMENT"
[ "$BACKUP_ENABLED" = "true" ] && echo "  kubectl get cronjob -l app=$APP_NAME,environment=$ENVIRONMENT"

echo "🧹 Cleanup with:"
echo "  kubectl delete all,pvc -l app=$APP_NAME,environment=$ENVIRONMENT"
```

13. Namespaces

Namespaces provide logical isolation within clusters, enabling multi-tenancy and resource organization.

```bash
# Create Namespaces - logical resource separation
kubectl create namespace development # Environment-based namespace
kubectl create namespace production # Production isolation
kubectl create ns testing # 'ns' is shorthand

# List Namespaces - see all logical partitions
kubectl get namespaces
kubectl get ns # Shows all namespaces and status

# Set default Namespace - avoid repeating -n flag
kubectl config set-context --current --namespace=development
# Changes default namespace for current context
kubectl config view --minify | grep namespace # Verify current default

# Work with specific Namespace - explicit namespace selection
kubectl get pods -n kube-system # System pods
kubectl get all -n development # All resources in development namespace

# Delete Namespace - removes all resources within it
kubectl delete namespace testing # USE WITH EXTREME CAUTION
# Deletes namespace and ALL resources within it - irreversible
```

Best Practice: Use namespaces to separate environments (dev/staging/prod) or teams. Always verify which namespace you're working in before making changes.

Example: Namespace Resource Quotas

Control resource consumption per namespace:

```yaml
```

```yaml
# namespace-with-quota.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: limited-namespace
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: limited-namespace
spec:
  hard:
    requests.cpu: "2"
    requests.memory: 4Gi
    limits.cpu: "4"
    limits.memory: 8Gi
    pods: "10"
    persistentvolumeclaims: "5"
---
apiVersion: v1
kind: LimitRange
metadata:
  name: default-limits
  namespace: limited-namespace
spec:
  limits:
  - default:
      memory: "256Mi"
      cpu: "200m"
    defaultRequest:
      memory: "128Mi"
      cpu: "100m"
    type: Container
```

Demo: Multi-Tenant Application Deployment

Deploy the same application in different namespaces with different configurations:

```bash
```

```
#!/bin/bash
# save as multi-tenant-demo.sh

TENANTS=("tenant-a" "tenant-b" "tenant-c")

echo "🏢 Setting up multi-tenant environment"

for tenant in "${TENANTS[@]}"; do
  echo "🔧 Setting up $tenant..."

  # Create namespace
  kubectl create namespace $tenant

  # Create tenant-specific ConfigMap
  kubectl create configmap app-config \
    --namespace=$tenant \
    --from-literal=tenant_name=$tenant \
    --from-literal=database_url=${tenant}-db:5432 \
    --from-literal=app_color=$([ "$tenant" = "tenant-a" ] && echo "blue" || [ "$tenant" = "tenant-b" ] && echo "green" |

  # Deploy application for tenant
  cat << EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  namespace: $tenant
spec:
  replicas: 2
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: web
        image: nginx:alpine
        envFrom:
        - configMapRef:
            name: app-config
```

```
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
  namespace: $tenant
spec:
  selector:
    app: webapp
  ports:
  - port: 80
    targetPort: 80
  type: ClusterIP
EOF

  # Create resource quota for tenant
  cat << EOF | kubectl apply -f -
apiVersion: v1
kind: ResourceQuota
metadata:
  name: ${tenant}-quota
  namespace: $tenant
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 2Gi
    limits.cpu: "2"
    limits.memory: 4Gi
    pods: "5"
EOF

done

echo "✅ Multi-tenant setup complete!"
echo "🔍 Check tenants:"
for tenant in "${TENANTS[@]}"; do
  echo "  $tenant: kubectl get all -n $tenant"
done

echo "📊 Check resource quotas:"
echo "  kubectl get resourcequota --all-namespaces"
```

```
echo "🧹 Cleanup:"
echo "  kubectl delete namespace ${TENANTS[*]}"
```

Mini-Project: Namespace Management System

Create a comprehensive namespace management system with RBAC:

```
bash
```

```
#!/bin/bash
# save as namespace-manager.sh

TEAM_NAME=${1}
ENVIRONMENT=${2:-development}

if [ -z "$TEAM_NAME" ]; then
  echo "Usage: $0 <team-name> [environment]"
  echo "Example: $0 frontend-team development"
  exit 1
fi

NAMESPACE="${TEAM_NAME}-${ENVIRONMENT}"

echo "🏗 Creating managed namespace: $NAMESPACE"

# Create namespace with labels
cat << EOF | kubectl apply -f -
apiVersion: v1
kind: Namespace
metadata:
  name: $NAMESPACE
  labels:
    team: $TEAM_NAME
    environment: $ENVIRONMENT
    managed-by: namespace-manager
    created-date: $(date +%Y%m%d)
  annotations:
    description: "Namespace for $TEAM_NAME team in $ENVIRONMENT environment"
    contact: "${TEAM_NAME}@company.com"
EOF

# Set resource quotas based on environment
case $ENVIRONMENT in
  "development")
    CPU_REQUEST="2"
    MEMORY_REQUEST="4Gi"
    CPU_LIMIT="4"
    MEMORY_LIMIT="8Gi"
    POD_LIMIT="20"
    ;;
  "staging")
    CPU_REQUEST="4"
```

```
    MEMORY_REQUEST="8Gi"
    CPU_LIMIT="8"
    MEMORY_LIMIT="16Gi"
    POD_LIMIT="30"
    ;;
  "production")
    CPU_REQUEST="8"
    MEMORY_REQUEST="16Gi"
    CPU_LIMIT="16"
    MEMORY_LIMIT="32Gi"
    POD_LIMIT="50"
    ;;
esac

# Create resource quota
cat << EOF | kubectl apply -f -
apiVersion: v1
kind: ResourceQuota
metadata:
  name: ${NAMESPACE}-quota
  namespace: $NAMESPACE
spec:
  hard:
    requests.cpu: "$CPU_REQUEST"
    requests.memory: $MEMORY_REQUEST
    limits.cpu: "$CPU_LIMIT"
    limits.memory: $MEMORY_LIMIT
    pods: "$POD_LIMIT"
    services: "10"
    secrets: "20"
    configmaps: "20"
    persistentvolumeclaims: "10"
EOF

# Create default limit range
cat << EOF | kubectl apply -f -
apiVersion: v1
kind: LimitRange
metadata:
  name: ${NAMESPACE}-limits
  namespace: $NAMESPACE
spec:
  limits:
  - default:
```

```
      memory: "512Mi"
      cpu: "500m"
    defaultRequest:
      memory: "256Mi"
      cpu: "250m"
    max:
      memory: "2Gi"
      cpu: "1"
    min:
      memory: "128Mi"
      cpu: "100m"
    type: Container
EOF

# Create service account for the team
kubectl create serviceaccount ${TEAM_NAME}-sa --namespace=$NAMESPACE

# Create role with appropriate permissions
cat << EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: $NAMESPACE
  name: ${TEAM_NAME}-role
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps", "secrets", "persistentvolumeclaims"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups: ["apps"]
  resources: ["deployments", "replicasets"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
- apiGroups: [""]
  resources: ["pods/log", "pods/exec"]
  verbs: ["get", "list", "create"]
EOF

# Create role binding
cat << EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: ${TEAM_NAME}-binding
  namespace: $NAMESPACE
subjects:
```

```
- kind: ServiceAccount
  name: ${TEAM_NAME}-sa
  namespace: $NAMESPACE
roleRef:
  kind: Role
  name: ${TEAM_NAME}-role
  apiGroup: rbac.authorization.k8s.io
EOF

# Create network policy (if network policies are supported)
cat << EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ${NAMESPACE}-netpol
  namespace: $NAMESPACE
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          team: $TEAM_NAME
  egress:
  - to: []  # Allow all egress for now
EOF

echo "✅ Namespace $NAMESPACE created successfully!"
echo ""
echo "📋 Summary:"
echo "  Namespace: $NAMESPACE"
echo "  Resource Quota: $CPU_REQUEST CPU, $MEMORY_REQUEST Memory"
echo "  Pod Limit: $POD_LIMIT"
echo "  Service Account: ${TEAM_NAME}-sa"
echo ""
echo "🔧 Usage:"
echo "  Set context: kubectl config set-context --current --namespace=$NAMESPACE"
echo "  View quota: kubectl describe quota -n $NAMESPACE"
echo "  View limits: kubectl describe limitrange -n $NAMESPACE"
echo ""
```

```
echo "🧹 Cleanup:"
echo "  kubectl delete namespace $NAMESPACE"
```

14. RBAC (Role-Based Access Control)

RBAC controls who can do what in your Kubernetes cluster, implementing the principle of least privilege.

```bash
```

```
# Create Service Account - identity for applications and users
kubectl create serviceaccount app-sa # Service account for applications
kubectl create sa monitoring-sa # 'sa' is shorthand

# List Service Accounts
kubectl get serviceaccounts
kubectl get sa
kubectl describe sa app-sa # Shows associated secrets and tokens

# Create Role - permissions within a namespace
kubectl create role pod-reader \
  --verb=get,list,watch \
  --resource=pods \
  --namespace=development
# Defines what actions can be performed on which resources

# Create RoleBinding - assigns role to service account
kubectl create rolebinding pod-reader-binding \
  --role=pod-reader \
  --serviceaccount=development:app-sa
# Links the role to the service account within namespace

# Create ClusterRole - cluster-wide permissions
kubectl create clusterrole cluster-reader \
  --verb=get,list,watch \
  --resource=nodes,namespaces
# Permissions that span multiple namespaces

# Create ClusterRoleBinding - cluster-wide role assignment
kubectl create clusterrolebinding cluster-reader-binding \
  --clusterrole=cluster-reader \
  --serviceaccount=development:app-sa
# Grants cluster-wide permissions to service account

# Test permissions - verify RBAC configuration
kubectl auth can-i get pods \
  --as=system:serviceaccount:development:app-sa \
  --namespace=development
# Returns 'yes' or 'no' - essential for debugging access issues
```

Best Practice: Always test RBAC policies with `kubectl auth can-i` before deploying applications. Use least-privilege principles and regularly audit permissions.

Example: Creating a Read-Only User

Set up a user that can only view resources:

```yaml
# readonly-rbac.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: readonly-user
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: readonly-role
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps", "persistentvolumeclaims"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["apps"]
  resources: ["deployments", "replicasets"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["pods/log"]
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: readonly-binding
subjects:
- kind: ServiceAccount
  name: readonly-user
  namespace: default
roleRef:
  kind: ClusterRole
  name: readonly-role
  apiGroup: rbac.authorization.k8s.io
```

```bash
```

```bash
# Test the readonly user permissions
kubectl auth can-i get pods --as=system:serviceaccount:default:readonly-user
kubectl auth can-i create pods --as=system:serviceaccount:default:readonly-user
kubectl auth can-i delete pods --as=system:serviceaccount:default:readonly-user
```

Demo: Progressive RBAC Implementation

Implement RBAC step by step, showing how permissions build up:

```bash
```

```bash
#!/bin/bash
# save as rbac-demo.sh

NAMESPACE="rbac-demo"
SA_NAME="demo-user"

echo "🔐 RBAC Progressive Demo"

# Create namespace and service account
kubectl create namespace $NAMESPACE
kubectl create serviceaccount $SA_NAME --namespace=$NAMESPACE

echo "🔍 Initial permissions (should be none):"
kubectl auth can-i get pods --as=system:serviceaccount:$NAMESPACE:$SA_NAME --namespace=$NAMESPACE

# Step 1: Basic pod read permissions
echo "📖 Step 1: Adding pod read permissions..."
kubectl create role pod-reader --verb=get,list,watch --resource=pods --namespace=$NAMESPACE
kubectl create rolebinding pod-reader-binding --role=pod-reader --serviceaccount=$NAMESPACE:$SA_NAME --name

kubectl auth can-i get pods --as=system:serviceaccount:$NAMESPACE:$SA_NAME --namespace=$NAMESPACE
kubectl auth can-i create pods --as=system:serviceaccount:$NAMESPACE:$SA_NAME --namespace=$NAMESPACE

# Step 2: Add deployment permissions
echo "🚀 Step 2: Adding deployment permissions..."
kubectl create role deployment-manager --verb=get,list,watch,create,update,patch --resource=deployments --namesp
kubectl create rolebinding deployment-manager-binding --role=deployment-manager --serviceaccount=$NAMESPAC

kubectl auth can-i create deployments --as=system:serviceaccount:$NAMESPACE:$SA_NAME --namespace=$NAMESP
kubectl auth can-i delete deployments --as=system:serviceaccount:$NAMESPACE:$SA_NAME --namespace=$NAMESP

# Step 3: Add log access
echo "📋 Step 3: Adding log access permissions..."
kubectl create role log-reader --verb=get,list --resource=pods/log --namespace=$NAMESPACE
kubectl create rolebinding log-reader-binding --role=log-reader --serviceaccount=$NAMESPACE:$SA_NAME --namesp

kubectl auth can-i get pods/log --as=system:serviceaccount:$NAMESPACE:$SA_NAME --namespace=$NAMESPACE

echo "✅ RBAC demo complete!"
echo "🧹 Cleanup: kubectl delete namespace $NAMESPACE"
```

Mini-Project: Team-Based RBAC System

Create a comprehensive RBAC system for different teams:

```bash
```

```bash
#!/bin/bash
# save as team-rbac-system.sh

echo "👥 Team-Based RBAC System Setup"

# Define teams and their roles
declare -A TEAMS
TEAMS[developers]="dev-role"
TEAMS[qa-engineers]="qa-role"
TEAMS[devops]="devops-role"
TEAMS[security]="security-role"

# Create namespaces for different environments
NAMESPACES=("development" "staging" "production")

for ns in "${NAMESPACES[@]}"; do
  kubectl create namespace $ns --dry-run=client -o yaml | kubectl apply -f -
done

# Create team-specific roles
create_team_roles() {
  # Developer role - full access in dev, read in staging
  cat << EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: development
  name: dev-role
rules:
- apiGroups: ["", "apps", "extensions"]
  resources: ["*"]
  verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: staging
  name: dev-role
rules:
- apiGroups: ["", "apps"]
  resources: ["pods", "services", "deployments", "configmaps"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
```

```
  resources: ["pods/log", "pods/exec"]
  verbs: ["get", "list", "create"]
EOF

  # QA role - read access everywhere, write access to test resources
  cat << EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: development
  name: qa-role
rules:
- apiGroups: ["", "apps"]
  resources: ["pods", "services", "deployments", "configmaps"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["pods/log", "pods/exec"]
  verbs: ["get", "list", "create"]
- apiGroups: ["batch"]
  resources: ["jobs"]
  verbs: ["get", "list", "watch", "create", "delete"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: staging
  name: qa-role
rules:
- apiGroups: ["", "apps"]
  resources: ["pods", "services", "deployments", "configmaps"]
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources: ["pods/log", "pods/exec"]
  verbs: ["get", "list", "create"]
- apiGroups: ["batch"]
  resources: ["jobs"]
  verbs: ["get", "list", "watch", "create", "delete"]
EOF

  # DevOps role - full cluster access
  cat << EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
  name: devops-role
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
EOF

  # Security role - read access everywhere, security resource management
  cat << EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: security-role
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["*"]
  verbs: ["*"]
- apiGroups: ["networking.k8s.io"]
  resources: ["networkpolicies"]
  verbs: ["*"]
EOF
}

create_team_roles

# Create service accounts and bindings for each team
for team in "${!TEAMS[@]}"; do
  role=${TEAMS[$team]}
  echo "👤 Setting up $team team..."

  # Create service accounts in each namespace
  for ns in "${NAMESPACES[@]}"; do
    kubectl create serviceaccount $team --namespace=$ns --dry-run=client -o yaml | kubectl apply -f -
  done

  # Create role bindings based on team role
  case $team in
    "developers")
      # Full access in development, read in staging
      kubectl create rolebinding dev-full-access --role=dev-role --serviceaccount=development:$team --namespace=dev
      kubectl create rolebinding dev-staging-read --role=dev-role --serviceaccount=staging:$team --namespace=staging
```

```
    ;;
  "qa-engineers")
    # QA access in development and staging
    kubectl create rolebinding qa-dev-access --role=qa-role --serviceaccount=development:$team --namespace=deve
    kubectl create rolebinding qa-staging-access --role=qa-role --serviceaccount=staging:$team --namespace=staging
    ;;
  "devops")
    # Cluster-wide access
    kubectl create clusterrolebinding devops-cluster-access --clusterrole=devops-role --serviceaccount=development:$
    ;;
  "security")
    # Security role cluster-wide
    kubectl create clusterrolebinding security-cluster-access --clusterrole=security-role --serviceaccount=development:
    ;;
 esac
done

echo "✅ Team-based RBAC system created!"
echo ""
echo "🧪 Test permissions:"
echo "# Developers can create pods in development:"
echo "kubectl auth can-i create pods --as=system:serviceaccount:development:developers --namespace=development
echo ""
echo "# Developers cannot create pods in production:"
echo "kubectl auth can-i create pods --as=system:serviceaccount:development:developers --namespace=production"
echo ""
echo "# QA can read logs in staging:"
echo "kubectl auth can-i get pods/log --as=system:serviceaccount:staging:qa-engineers --namespace=staging"
echo ""
echo "# DevOps can do everything:"
echo "kubectl auth can-i '*' '*' --as=system:serviceaccount:development:devops"
echo ""
echo "🔍 Audit permissions:"
echo "kubectl get rolebindings,clusterrolebindings --all-namespaces"
echo ""
echo "🧹 Cleanup:"
for ns in "${NAMESPACES[@]}"; do
 echo "kubectl delete namespace $ns"
done
echo "kubectl delete clusterrole devops-role security-role"
echo "kubectl delete clusterrolebinding devops-cluster-access security-cluster-access"
```

15. Taints and Tolerations

Taints and tolerations control pod scheduling, allowing you to reserve nodes for specific workloads or exclude problematic nodes.

```bash
# Apply taint to a node - prevents scheduling unless pods have matching toleration
kubectl taint nodes node1 key1=value1:NoSchedule
# NoSchedule: new pods won't be scheduled on this node
# NoExecute: existing pods will be evicted if they don't tolerate the taint
# PreferNoSchedule: scheduler will try to avoid this node but may still use it
```

```yaml
# toleration-pod.yaml - Pod that can be scheduled on tainted node
apiVersion: v1
kind: Pod
metadata:
  name: toleration-pod
spec:
  tolerations: # Tolerations allow scheduling on tainted nodes
  - key: "key1" # Must match taint key
    operator: "Equal" # Matching operator (Equal or Exists)
    value: "value1" # Must match taint value (when using Equal)
    effect: "NoSchedule" # Must match taint effect
  containers:
  - name: nginx
    image: nginx
```

```bash
# Apply Pod with toleration
kubectl apply -f toleration-pod.yaml

# Remove taint from node
kubectl taint nodes node1 key1=value1:NoSchedule-
# The trailing minus (-) removes the taint
```

Use Cases: Reserve nodes for specific applications, isolate problematic nodes, or create dedicated node pools for different workload types.

Example: GPU Node Dedication

Create a dedicated GPU node pool using taints and tolerations:

```bash
# Taint GPU nodes (assuming you have GPU nodes)
kubectl taint nodes gpu-node-1 nvidia.com/gpu=true:NoSchedule
kubectl taint nodes gpu-node-2 nvidia.com/gpu=true:NoSchedule

# Label GPU nodes for easy identification
kubectl label nodes gpu-node-1 accelerator=nvidia-tesla-k80
kubectl label nodes gpu-node-2 accelerator=nvidia-tesla-k80
```

```yaml
# gpu-workload.yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-workload
spec:
  tolerations:
  - key: nvidia.com/gpu
    operator: Equal
    value: "true"
    effect: NoSchedule
  nodeSelector:
    accelerator: nvidia-tesla-k80
  containers:
  - name: gpu-app
    image: tensorflow/tensorflow:latest-gpu
    resources:
      limits:
        nvidia.com/gpu: 1
```

Demo: Node Maintenance with Taints

Simulate node maintenance using taints and observe pod behavior:

```bash
```

```bash
#!/bin/bash
# save as node-maintenance-demo.sh

echo "🔧 Node Maintenance Demo with Taints"

# Get a node to work with (avoid master nodes)
NODE=$(kubectl get nodes --no-headers -o custom-columns=":metadata.name" | grep -v master | head -1)
echo "Using node: $NODE"

# Create some test pods
echo "🚀 Creating test workloads..."
kubectl create deployment normal-app --image=nginx --replicas=3
kubectl create deployment critical-app --image=nginx --replicas=2

# Wait for pods to be scheduled
kubectl wait --for=condition=available deployment/normal-app --timeout=60s
kubectl wait --for=condition=available deployment/critical-app --timeout=60s

echo "📊 Initial pod distribution:"
kubectl get pods -o wide | grep -E "(normal-app|critical-app)"

# Simulate maintenance preparation - taint node with NoSchedule
echo "🔒 Step 1: Preventing new pods (NoSchedule taint)..."
kubectl taint nodes $NODE maintenance=planned:NoSchedule

# Scale up to see new pods avoid the tainted node
kubectl scale deployment normal-app --replicas=5
sleep 10

echo "📊 After NoSchedule taint:"
kubectl get pods -o wide | grep normal-app

# Prepare for maintenance - cordon node
echo "🚫 Step 2: Cordoning node..."
kubectl cordon $NODE

# Drain node (this will evict pods)
echo "🌊 Step 3: Draining node..."
kubectl drain $NODE --ignore-daemonsets --delete-emptydir-data --force

echo "📊 After draining:"
kubectl get pods -o wide | grep -E "(normal-app|critical-app)"
```

```bash
# Maintenance complete - uncordon and remove taint
echo "✅ Step 4: Maintenance complete, bringing node back..."
kubectl uncordon $NODE
kubectl taint nodes $NODE maintenance=planned:NoSchedule-

echo "📊 Final pod distribution:"
kubectl get pods -o wide | grep -E "(normal-app|critical-app)"

echo "🧹 Cleanup:"
echo "kubectl delete deployment normal-app critical-app"
```

Mini-Project: Advanced Node Scheduling System

Create a comprehensive node scheduling system with multiple taint strategies:

```bash
```

```bash
#!/bin/bash
# save as advanced-scheduling-system.sh

echo "🎯 Advanced Node Scheduling System"

# Simulate different node types by applying labels and taints
NODES=($(kubectl get nodes --no-headers -o custom-columns=":metadata.name" | grep -v master))

if [ ${#NODES[@]} -lt 3 ]; then
  echo "❌ Need at least 3 worker nodes for this demo"
  exit 1
fi

# Configure node types
echo "🏷️ Configuring node types..."

# High-performance compute node
kubectl label nodes ${NODES[0]} node-type=compute workload=cpu-intensive --overwrite
kubectl taint nodes ${NODES[0]} compute=high-performance:NoSchedule

# Memory-optimized node
kubectl label nodes ${NODES[1]} node-type=memory workload=memory-intensive --overwrite
kubectl taint nodes ${NODES[1]} memory=optimized:NoSchedule

# General purpose node (no taint)
kubectl label nodes ${NODES[2]} node-type=general workload=mixed --overwrite

echo "📋 Node configuration:"
kubectl get nodes -L node-type,workload

# Create workloads for different node types
cat << 'EOF' | kubectl apply -f -
# CPU-intensive workload
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cpu-intensive-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cpu-intensive
  template:
```

```yaml
    metadata:
      labels:
        app: cpu-intensive
    spec:
      tolerations:
      - key: compute
        operator: Equal
        value: high-performance
        effect: NoSchedule
      nodeSelector:
        node-type: compute
      containers:
      - name: cpu-app
        image: busybox
        command: ["sh", "-c", "while true; do echo 'CPU intensive work'; sleep 30; done"]
        resources:
          requests:
            cpu: 500m
          limits:
            cpu: 1000m
---
# Memory-intensive workload
apiVersion: apps/v1
kind: Deployment
metadata:
  name: memory-intensive-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: memory-intensive
  template:
    metadata:
      labels:
        app: memory-intensive
    spec:
      tolerations:
      - key: memory
        operator: Equal
        value: optimized
        effect: NoSchedule
      nodeSelector:
        node-type: memory
      containers:
```

```
    - name: memory-app
      image: busybox
      command: ["sh", "-c", "while true; do echo 'Memory intensive work'; sleep 30; done"]
      resources:
        requests:
          memory: 256Mi
        limits:
          memory: 512Mi
---
# General workload (can run anywhere)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: general-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: general
  template:
    metadata:
      labels:
        app: general
    spec:
      containers:
      - name: general-app
        image: nginx:alpine
---
# Critical workload with special scheduling
apiVersion: apps/v1
kind: Deployment
metadata:
  name: critical-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: critical
  template:
    metadata:
      labels:
        app: critical
    spec:
      priorityClassName: high-priority
```

```
    tolerations:
    - key: compute
      operator: Equal
      value: high-performance
      effect: NoSchedule
    - key: memory
      operator: Equal
      value: optimized
      effect: NoSchedule
    nodeSelector:
      node-type: compute
    containers:
    - name: critical-app
      image: nginx:alpine
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
EOF

# Create priority class for critical workloads
cat << EOF | kubectl apply -f -
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000
globalDefault: false
description: "High priority class for critical applications"
EOF

echo "⏳ Waiting for deployments..."
kubectl wait --for=condition=available deployment --all --timeout=120s

echo "📊 Final scheduling results:"
echo "CPU-intensive pods:"
kubectl get pods -l app=cpu-intensive -o wide

echo "Memory-intensive pods:"
kubectl get pods -l app=memory-intensive -o wide

echo "General pods:"
kubectl get pods -l app=general -o wide
```

```
echo "Critical pods:"
kubectl get pods -l app=critical -o wide

echo "🔍 Node utilization:"
kubectl describe nodes | grep -A 5 "Allocated resources"

echo "🧹 Cleanup commands:"
echo "kubectl delete deployments --all"
echo "kubectl delete priorityclass high-priority"
echo "kubectl taint nodes ${NODES[0]} compute=high-performance:NoSchedule-"
echo "kubectl taint nodes ${NODES[1]} memory=optimized:NoSchedule-"
echo "kubectl label nodes ${NODES[0]} node-type- workload-"
echo "kubectl label nodes ${NODES[1]} node-type- workload-"
echo "kubectl label nodes ${NODES[2]} node-type- workload-"
```

16. Monitoring and Troubleshooting

Effective troubleshooting is crucial for maintaining healthy Kubernetes applications. This section covers essential debugging techniques and common problem patterns.

16.1 Basic Monitoring Commands

bash

```bash
# View Pod logs - first step in debugging application issues
kubectl logs nginx-pod
# Shows stdout/stderr from the main container process

kubectl logs -f nginx-pod # Follow logs in real-time
# -f flag streams logs continuously - essential for monitoring live issues

kubectl logs deployment/nginx-deploy --all-containers=true
# Get logs from all containers in all pods of a deployment
# Useful for debugging issues across multiple replicas

kubectl logs nginx-pod --previous # Logs from previous container instance
# Critical when containers are crash-looping - shows what happened before restart

kubectl logs nginx-pod -c sidecar-container # Logs from specific container
# When pods have multiple containers, specify which one you want

# Describe resources for detailed status and events
kubectl describe pod problematic-pod
# Events section at bottom shows pod lifecycle events
# Look for Failed, Warning, or Error events

kubectl get events --sort-by='.lastTimestamp'
# Cluster-wide events sorted by time - helps correlate issues

kubectl get events --field-selector involvedObject.name=nginx-pod
# Events specific to a particular object - focused troubleshooting

# Debug nodes - infrastructure-level troubleshooting
kubectl describe node node-name # Node resource usage and conditions
kubectl get events --field-selector involvedObject.kind=Node # Node-level events

# Create debug Pod - temporary container for network/storage testing
kubectl run debug --image=busybox --it --rm --restart=Never -- sh
# --rm automatically deletes pod when you exit
# --it provides interactive terminal

# Advanced debugging with specialized tools
kubectl run netshoot --image=nicolaka/netshoot --it --rm --restart=Never -- bash
# netshoot container has network debugging tools (dig, nslookup, curl, etc.)

# Debug existing Pod - attach debugging tools to running pod
kubectl debug nginx-pod --it --image=busybox --target=nginx
```

```
# Attaches debugging container to existing pod's namespaces
# Useful for inspecting running applications without modifying them
```

16.2 Common Errors and Systematic Troubleshooting

```bash
# Pod in CrashLoopBackOff - systematic debugging approach
kubectl describe pod problematic-pod | grep -A 10 Events
# Check events for specific error messages

kubectl logs problematic-pod --previous
# See what happened before the crash

kubectl get pod problematic-pod -o yaml | grep -A 5 -B 5 image
# Verify image name and tag are correct

# Service not reachable - networking troubleshooting
kubectl get endpoints service-name
# Empty endpoints = no healthy pods matching service selector

kubectl describe service service-name | grep Selector
# Compare service selector with pod labels

kubectl get pods --show-labels | grep app=your-app
# Verify pods have labels that match service selector

# Pod not scheduling - resource and constraint issues
kubectl describe pod pending-pod | grep -A 10 Events
# Look for scheduling failures in events

kubectl describe nodes | grep -A 5 "Allocated resources"
# Check if nodes have sufficient resources

kubectl get pods -o wide | grep pending-pod
# See if pod is assigned to a node

# Storage issues - persistent volume troubleshooting
kubectl get pvc # Check PVC status
kubectl describe pvc your-pvc | grep Events # Look for binding issues
kubectl get pv # Check available persistent volumes
```

16.3 Performance and Resource Monitoring

```bash
# Resource usage monitoring - requires metrics-server
kubectl top pods # Current CPU/memory usage by pod
kubectl top pods --sort-by=cpu # Sort by CPU usage
kubectl top pods --sort-by=memory # Sort by memory usage
kubectl top nodes # Node resource utilization
kubectl top nodes --sort-by=cpu # Identify resource-constrained nodes

# Resource quota and limits
kubectl describe quota --all-namespaces # Check resource quotas
kubectl describe limitrange --all-namespaces # Check default limits
```

Example: Comprehensive Health Check Script

Create a script that performs systematic health checks:

```bash
```

```bash
#!/bin/bash
# save as cluster-health-check.sh

echo "🏥 Kubernetes Cluster Health Check"
echo "======================================"

# Cluster connectivity
echo "🔗 Cluster Connectivity:"
if kubectl cluster-info >/dev/null 2>&1; then
  echo "✅ Cluster accessible"
  kubectl version --short
else
  echo "❌ Cannot connect to cluster"
  exit 1
fi

# Node health
echo -e "\n🖥️  Node Health:"
kubectl get nodes --no-headers | while read node status roles age version; do
  if [[ $status == "Ready" ]]; then
    echo "✅ $node: $status"
  else
    echo "❌ $node: $status"
    kubectl describe node $node | grep -A 5 Conditions
  fi
done

# System pods health
echo -e "\n🔧 System Pods Health:"
kubectl get pods -n kube-system --no-headers | while read name ready status restarts age; do
  if [[ $status == "Running" && $ready == *"/"* ]]; then
    ready_count=$(echo $ready | cut -d'/' -f1)
    total_count=$(echo $ready | cut -d'/' -f2)
    if [ "$ready_count" -eq "$total_count" ]; then
      echo "✅ $name: $status ($ready)"
    else
      echo "⚠️  $name: $status ($ready)"
    fi
  else
    echo "❌ $name: $status ($ready)"
  fi
done
```

```
# Resource utilization (if metrics-server is available)
echo -e "\n📊 Resource Utilization:"
if kubectl top nodes >/dev/null 2>&1; then
  kubectl top nodes
else
  echo "⚠️ Metrics server not available"
fi

# Recent events (errors and warnings)
echo -e "\n⚠️ Recent Events (Warnings/Errors):"
kubectl get events --all-namespaces --field-selector type!=Normal \
  --sort-by='.lastTimestamp' | tail -10

# Storage health
echo -e "\n💾 Storage Health:"
kubectl get pv --no-headers | while read name capacity access reclaim status claim storageclass reason age; do
  if [[ $status == "Bound" ]]; then
    echo "✅ PV $name: $status"
  else
    echo "⚠️ PV $name: $status"
  fi
done

# Networking health
echo -e "\n🌐 Networking Health:"
# Test DNS resolution
if kubectl run dns-test --image=busybox --rm -it --restart=Never -- nslookup kubernetes.default >/dev/null 2>&1; the
  echo "✅ DNS resolution working"
else
  echo "❌ DNS resolution issues"
fi

echo -e "\n✅ Health check complete!"
```

Demo: Troubleshooting Common Issues

Create and debug common Kubernetes problems:

```
bash
```

```bash
#!/bin/bash
# save as troubleshooting-scenarios.sh

echo "🔍 Kubernetes Troubleshooting Scenarios"

# Scenario 1: ImagePullBackOff
echo "📍 Scenario 1: ImagePullBackOff"
kubectl run broken-image --image=non-existent-image:latest --restart=Never

echo "Debug steps:"
echo "1. Check pod status:"
echo "   kubectl get pods broken-image"
echo "2. Describe pod for events:"
echo "   kubectl describe pod broken-image"
echo "3. Check image name spelling"

# Scenario 2: CrashLoopBackOff
echo -e "\n📍 Scenario 2: CrashLoopBackOff"
kubectl run crash-loop --image=busybox --restart=Never -- sh -c "exit 1"

echo "Debug steps:"
echo "1. Check previous logs:"
echo "   kubectl logs crash-loop --previous"
echo "2. Check container command and args"

# Scenario 3: Service connectivity issues
echo -e "\n📍 Scenario 3: Service Connectivity Issues"
kubectl create deployment web-app --image=nginx
kubectl expose deployment web-app --port=80 --name=broken-service
# Modify service selector to break connectivity
kubectl patch service broken-service -p '{"spec":{"selector":{"app":"wrong-label"}}}'

echo "Debug steps:"
echo "1. Check service endpoints:"
echo "   kubectl get endpoints broken-service"
echo "2. Compare service selector with pod labels:"
echo "   kubectl describe service broken-service | grep Selector"
echo "   kubectl get pods --show-labels"

# Scenario 4: Resource constraints
echo -e "\n📍 Scenario 4: Resource Constraints"
cat << EOF | kubectl apply -f -
apiVersion: v1
```

```
kind: Pod
metadata:
  name: resource-hungry
spec:
  containers:
  - name: app
    image: nginx
    resources:
      requests:
        memory: "10Gi"  # Unrealistic memory request
        cpu: "8"        # Unrealistic CPU request
EOF

echo "Debug steps:"
echo "1. Check pod status:"
echo "   kubectl get pods resource-hungry"
echo "2. Check scheduling events:"
echo "   kubectl describe pod resource-hungry | grep Events -A 10"
echo "3. Check node resources:"
echo "   kubectl describe nodes | grep -A 10 'Allocated resources'"

echo -e "\n🧹 Cleanup scenarios:"
echo "kubectl delete pod broken-image crash-loop resource-hungry"
echo "kubectl delete deployment web-app"
echo "kubectl delete service broken-service"
```

Mini-Project: Automated Troubleshooting System

Create an automated system that detects and suggests fixes for common issues:

```bash
```

```
#!/bin/bash
# save as auto-troubleshoot.sh

NAMESPACE=${1:-default}

echo "🤖 Automated Troubleshooting System"
echo "Analyzing namespace: $NAMESPACE"

# Function to check pod issues
check_pod_issues() {
  local pod=$1
  local status=$(kubectl get pod $pod -n $NAMESPACE -o jsonpath='{.status.phase}')
  local ready=$(kubectl get pod $pod -n $NAMESPACE -o jsonpath='{.status.conditions[?(@.type=="Ready")].status}')

  case $status in
    "Pending")
      echo "🔍 Pod $pod is Pending"
      # Check for scheduling issues
      kubectl describe pod $pod -n $NAMESPACE | grep -A 10 Events | grep -i "FailedScheduling" && {
        echo "💡 Suggestion: Check resource constraints and node availability"
        echo "   kubectl describe nodes | grep -A 5 'Allocated resources'"
      }
      ;;
    "Running")
      if [[ $ready != "True" ]]; then
        echo "⚠️  Pod $pod is Running but not Ready"
        # Check readiness probe
        kubectl get pod $pod -n $NAMESPACE -o jsonpath='{.spec.containers[*].readinessProbe}' | grep -q . && {
          echo "💡 Suggestion: Check readiness probe configuration and endpoint"
          echo "   kubectl logs $pod -n $NAMESPACE"
        }
      fi
      ;;
    "Failed"|"CrashLoopBackOff")
      echo "❌ Pod $pod has failed"
      echo "💡 Suggestions:"
      echo "   - Check previous logs: kubectl logs $pod -n $NAMESPACE --previous"
      echo "   - Verify image and command: kubectl describe pod $pod -n $NAMESPACE"
      echo "   - Check resource limits and requests"
      ;;
  esac
}
```

```
# Function to check service issues
check_service_issues() {
    local service=$1
    local endpoints=$(kubectl get endpoints $service -n $NAMESPACE -o jsonpath='{.subsets[*].addresses[*].ip}' 2>/dev/

    if [[ -z "$endpoints" ]]; then
        echo "🔍 Service $service has no endpoints"
        echo "💡 Suggestions:"
        echo "   - Check service selector: kubectl describe service $service -n $NAMESPACE"
        echo "   - Verify pod labels: kubectl get pods --show-labels -n $NAMESPACE"
        echo "   - Check if pods are ready and running"
    fi
}

# Main analysis
echo "🔎 Scanning for issues..."

# Check pods
kubectl get pods -n $NAMESPACE --no-headers | while read name ready status restarts age; do
    if [[ $status != "Running" ]] || [[ $ready != *"/"* ]] || [[ $ready == "0/"* ]]; then
        check_pod_issues $name
    fi
done

# Check services
kubectl get services -n $NAMESPACE --no-headers | while read name type cluster_ip external_ip port age; do
    check_service_issues $name
done

# Check resource quotas
echo -e "\n📊 Resource Quota Analysis:"
kubectl describe quota -n $NAMESPACE 2>/dev/null | grep -A 10 "Resource.*Used" || echo "No resource quotas found"

# Check recent events
echo -e "\n📋 Recent Warning/Error Events:"
kubectl get events -n $NAMESPACE --field-selector type!=Normal --sort-by='.lastTimestamp' | tail -5

# Performance suggestions
echo -e "\n🚀 Performance Suggestions:"
if kubectl top pods -n $NAMESPACE >/dev/null 2>&1; then
    echo "High CPU usage pods:"
    kubectl top pods -n $NAMESPACE --sort-by=cpu | head -5
    echo "High Memory usage pods:"
    kubectl top pods -n $NAMESPACE --sort-by=memory | head -5
```

```
else
  echo "Install metrics-server for resource usage monitoring"
fi

echo -e "\n✅ Troubleshooting analysis complete!"
```

17. StatefulSets

StatefulSets manage stateful applications that require stable network identities and persistent storage, such as databases and distributed systems.

```yaml
```

```yaml
# web-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet # StatefulSet for stateful applications
metadata:
  name: web
spec:
  serviceName: "nginx" # Headless service name for stable network identity
  replicas: 3 # Number of instances
  selector:
    matchLabels:
      app: nginx
  template: # Pod template - similar to Deployment
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        volumeMounts: # Each pod gets its own persistent volume
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates: # Template for creating PVCs per pod
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

bash

```
# Create StatefulSet
kubectl apply -f web-statefulset.yaml

# List StatefulSets
kubectl get statefulsets
kubectl get sts # 'sts' is shorthand

# StatefulSet provides ordered deployment and scaling
kubectl get pods -l app=nginx # Pods have predictable names: web-0, web-1, web-2
kubectl describe statefulset web # Shows rolling update strategy and volume claims

# Scale StatefulSet - scaling is ordered (highest ordinal first for scale down)
kubectl scale statefulset web --replicas=5
kubectl scale statefulset web --replicas=1 # Scales down from web-4 to web-1

# Delete StatefulSet - pods deleted in reverse order
kubectl delete statefulset web
# Note: PVCs are not automatically deleted - manual cleanup required
```

Key Differences from Deployments:

- Pods have stable, unique network identities (web-0, web-1, etc.)
- Pods are created and deleted in order
- Each pod can have its own persistent volume
- Rolling updates happen in order

Example: MySQL StatefulSet

Deploy a MySQL database using StatefulSet for persistent data:

```yaml
```

```yaml
# mysql-statefulset.yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-headless
  labels:
    app: mysql
spec:
  ports:
  - port: 3306
    name: mysql
  clusterIP: None
  selector:
    app: mysql
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  serviceName: mysql-headless
  replicas: 3
  selector:
    matchLabels:
      app: mysql
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - name: mysql
        image: mysql:8.0
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: rootpassword
        - name: MYSQL_DATABASE
          value: testdb
        - name: MYSQL_USER
          value: testuser
        - name: MYSQL_PASSWORD
          value: testpass
        ports:
```

```yaml
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-data
          mountPath: /var/lib/mysql
        resources:
          requests:
            memory: 256Mi
            cpu: 250m
          limits:
            memory: 512Mi
            cpu: 500m
  volumeClaimTemplates:
  - metadata:
      name: mysql-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 5Gi
```

Demo: StatefulSet Scaling and Persistence

Demonstrate StatefulSet unique characteristics:

```bash
```

```bash
#!/bin/bash
# save as statefulset-demo.sh

echo "📊 StatefulSet Demo: Scaling and Persistence"

# Deploy the MySQL StatefulSet
kubectl apply -f mysql-statefulset.yaml

echo "⏳ Waiting for StatefulSet to be ready..."
kubectl wait --for=condition=ready pod -l app=mysql --timeout=300s

echo "🔍 Initial StatefulSet status:"
kubectl get statefulset mysql
kubectl get pods -l app=mysql

echo "💾 Checking persistent volumes:"
kubectl get pvc -l app=mysql

# Test data persistence by connecting to each MySQL instance
echo "🧪 Testing individual pod identity and persistence:"
for i in {0..2}; do
  echo "--- mysql-$i ---"
  kubectl exec mysql-$i -- mysql -u root -prootpassword -e "
    CREATE DATABASE IF NOT EXISTS pod_$i;
    USE pod_$i;
    CREATE TABLE IF NOT EXISTS test_table (id INT, pod_name VARCHAR(50));
    INSERT INTO test_table VALUES ($i, 'mysql-$i');
    SELECT * FROM test_table;
  " 2>/dev/null || echo "Pod mysql-$i not ready yet"
done

# Demonstrate ordere
```