Kubernetes kubectl Command Study Guide Structure

1. Introduction

- What is Kubernetes and kubectl
- Setting up kubectl
- Kubectl syntax and command structure
- Kubectl autocomplete setup

2. Cluster Management and Configuration

- Checking cluster status
- Managing contexts and configurations
- Switching between clusters and namespaces
- Viewing and managing kubeconfig

3. Basic Resource Operations

- Creating resources
- Viewing and finding resources
- Describing resources in detail
- Deleting resources

4. Deployment Management

- Creating and updating deployments
- Scaling deployments
- Rolling updates and rollbacks
- Checking deployment status

5. Configuration Management

- Working with ConfigMaps
- Managing Secrets
- Setting environment variables

- Resource quotas and limits

6. Pod Operations

- Creating and running pods
- Accessing pod logs
- Executing commands in containers
- Port forwarding to pods

7. Service Management

- Creating and exposing services
- Service discovery
- Managing ingress
- Checking service endpoints

8. Troubleshooting

- Debugging pods and containers
- Viewing logs and events
- Checking resource status
- Common error patterns and solutions

9. Advanced Operations

- Using labels and selectors
- Working with annotations
- Using jsonpath for filtering and formatting
- Patching resources

10. Best Practices

- Namespace organization
- Resource naming conventions
- Using apply vs create
- Declarative vs imperative approaches

Introduction to kubectl

What is Kubernetes and kubectl?

Kubernetes (often abbreviated as K8s) is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery.

`kubectl` is the official command-line tool for Kubernetes. It allows you to run commands against Kubernetes clusters to: - Deploy applications - Inspect and manage cluster resources - View logs - Execute commands in containers - And much more

As a Kubernetes engineer, `kubectl` will be your primary interface for interacting with Kubernetes clusters on a day-to-day basis.

Setting up kubectl

Before you can use kubectl, you need to install it and configure it to communicate with your Kubernetes cluster.

Installation

Here's how to install kubectl on different operating systems:

Linux:

```
 # Using curl
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/
bin/linux/amd64/kubectl"
chmod +x kubectl
sudo mv kubectl /usr/local/bin/

# Using package manager (Ubuntu/Debian)
sudo apt-get update && sudo apt-get install -y kubectl

# Using package manager (CentOS/RHEL)
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
```

```
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://
packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF
sudo yum install -y kubectl
```

macOS:

```
 # Using Homebrew
brew install kubectl

# Using curl
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/
bin/darwin/amd64/kubectl"
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

Windows:

```
 # Using curl
curl -LO "https://dl.k8s.io/release/v1.26.0/bin/windows/amd64/kubectl.exe"
# Add the binary to your PATH

# Using Chocolatey
choco install kubernetes-cli
```

Configuration

After installation, kubectl needs to know which Kubernetes cluster to communicate with and how to authenticate to that cluster. This information is stored in a configuration file called `kubeconfig`.

The default location for the kubeconfig file is `~/.kube/config` on Linux and macOS, and `%USERPROFILE%\.kube\config` on Windows.

To check if your kubectl is properly configured:

```
kubectl cluster-info
```

If kubectl is configured correctly, you should see the URL of your Kubernetes master and other services.

Kubectl Syntax and Command Structure

The basic syntax for kubectl commands is:

```
kubectl [command] [TYPE] [NAME] [flags]
```

Where: - **command**: Specifies the operation you want to perform (e.g., create, get, describe, delete) - **TYPE**: Specifies the resource type (e.g., pods, deployments, services) - **NAME**: Specifies the name of the resource (optional for some commands) - **flags**: Specifies optional flags

Common Command Types

1. **Imperative commands**: Direct actions like `run`, `create`, `expose` `bash kubectl create deployment nginx --image=nginx`

2. **Imperative object configuration**: Create objects using configuration files `bash kubectl create -f nginx.yaml`

3. **Declarative object configuration**: Apply configurations to objects `bash kubectl apply -f nginx.yaml`

Basic Command Examples

```
# Get information about the cluster
kubectl cluster-info

# Get a list of all resources in the current namespace
kubectl get all

# Get a list of pods
kubectl get pods

# Get detailed information about a specific pod
kubectl describe pod <pod-name>

# Create a resource from a file
kubectl apply -f <filename.yaml>

# Delete a resource
kubectl delete <resource-type> <resource-name>

# Get logs from a pod
kubectl logs <pod-name>
```

```
# Execute a command in a container
kubectl exec -it <pod-name> -- <command>
```

Kubectl Autocomplete Setup

Setting up autocomplete for kubectl can significantly improve your productivity by reducing typing and helping you discover available commands and options.

Bash Autocomplete

```
 # Set up autocomplete in bash
source <(kubectl completion bash)

# Add autocomplete permanently to your bash shell
echo "source <(kubectl completion bash)" >> ~/.bashrc

# Create a kubectl alias with autocomplete
echo 'alias k=kubectl' >> ~/.bashrc
echo 'complete -o default -F __start_kubectl k' >> ~/.bashrc
```

Zsh Autocomplete

```
 # Set up autocomplete in zsh
source <(kubectl completion zsh)

# Add autocomplete permanently to your zsh shell
echo '[[ $commands[kubectl] ]] && source <(kubectl completion zsh)' >> ~/.zshrc

# Create a kubectl alias with autocomplete
echo 'alias k=kubectl' >> ~/.zshrc
echo 'compdef __start_kubectl k' >> ~/.zshrc
```

Fish Autocomplete

```
 # Set up autocomplete in fish
kubectl completion fish | source

# Add autocomplete permanently to your fish shell
echo 'kubectl completion fish | source' > ~/.config/fish/completions/kubectl.fish
```

Helpful Kubectl Aliases

Creating aliases for commonly used kubectl commands can save you a lot of typing:

```
# Common aliases
alias k='kubectl'
alias kg='kubectl get'
alias kgp='kubectl get pods'
alias kgd='kubectl get deployments'
alias kgs='kubectl get services'
alias kgn='kubectl get nodes'
alias kd='kubectl describe'
alias kdp='kubectl describe pod'
alias kdd='kubectl describe deployment'
alias kds='kubectl describe service'

# Namespace shortcuts
alias kn='kubectl config set-context --current --namespace'
alias kgpa='kubectl get pods --all-namespaces'
alias kgda='kubectl get deployments --all-namespaces'
alias kgsa='kubectl get services --all-namespaces'

# Context management
alias kx='kubectl config use-context'
alias kgc='kubectl config get-contexts'
```

In the next section, we'll dive deeper into cluster management and configuration commands that you'll use regularly as a Kubernetes engineer.

Cluster Management and Configuration

As a Kubernetes engineer, managing cluster connections and configurations is a fundamental skill. This section covers the essential commands for working with clusters, contexts, and configurations.

Checking Cluster Status

Before performing any operations, it's important to verify your connection to the cluster and check its status.

```
# Display cluster information
kubectl cluster-info

# Check the health of cluster components
```

```
kubectl get componentstatuses

# View all nodes in the cluster
kubectl get nodes

# Get detailed information about a specific node
kubectl describe node <node-name>

# Check the version of the client and server
kubectl version

# Get a short version output
kubectl version --short
```

Managing Contexts and Configurations

Kubernetes uses contexts to determine which cluster you're communicating with and which user credentials to use. The configuration is stored in the kubeconfig file.

```
 # List all available contexts
kubectl config get-contexts

# Display the current context
kubectl config current-context

# Switch to a different context
kubectl config use-context <context-name>

# View the full kubeconfig
kubectl config view

# View the full kubeconfig with sensitive data
kubectl config view --raw
```

Switching Between Clusters and Namespaces

Namespaces provide a way to divide cluster resources between multiple users or projects.

```
 # List all namespaces
kubectl get namespaces

# Set the default namespace for the current context
kubectl config set-context --current --namespace=<namespace-name>

# Run a command in a specific namespace
```

```
kubectl get pods --namespace=<namespace-name>

# Use the shorthand -n flag for namespace
kubectl get pods -n <namespace-name>

# Get resources across all namespaces
kubectl get pods --all-namespaces
# Or use the shorthand
kubectl get pods -A
```

Viewing and Managing Kubeconfig

The kubeconfig file contains clusters, users, and contexts that kubectl uses for communication.

```
 # Merge multiple kubeconfig files
KUBECONFIG=~/.kube/config:~/.kube/another-config kubectl config view --merge

# Add a new cluster to your kubeconfig
kubectl config set-cluster <cluster-name> --server=<server-url> --certificate-authority=<ca-file>

# Add user credentials
kubectl config set-credentials <user-name> --client-certificate=<cert-file> --client-key=<key-file>

# Create a new context
kubectl config set-context <context-name> --cluster=<cluster-name> --user=<user-name> --namespace=<namespace-name>

# Delete a context from kubeconfig
kubectl config delete-context <context-name>

# Delete a cluster from kubeconfig
kubectl config delete-cluster <cluster-name>

# Delete a user from kubeconfig
kubectl config unset users.<user-name>
```

Useful Context Management Aliases

These aliases can help you manage contexts and namespaces more efficiently:

```
 # Switch context
alias kx='kubectl config use-context'
```

```
# Set namespace for current context
alias kn='kubectl config set-context --current --namespace'

# Display current context and namespace
alias kc='kubectl config current-context'
alias kns='kubectl config view --minify | grep namespace | cut -d" " -f6'
```

Basic Resource Operations

This section covers the fundamental operations for working with Kubernetes resources, which you'll use daily as a Kubernetes engineer.

Creating Resources

There are multiple ways to create resources in Kubernetes:

```
 # Create a resource from a YAML or JSON file
kubectl create -f <filename.yaml>

# Create a deployment
kubectl create deployment <name> --image=<image>

# Create a service
kubectl create service <type> <name> --tcp=<port>:<target-port>

# Create a namespace
kubectl create namespace <name>

# Create a configmap
kubectl create configmap <name> --from-file=<path> --from-literal=<key>=<value>

# Create a secret
kubectl create secret generic <name> --from-file=<path> --from-
literal=<key>=<value>

# Create a job
kubectl create job <name> --image=<image> -- <command>

# Create a cronjob
kubectl create cronjob <name> --image=<image> --schedule="*/1 * * * *" --
<command>
```

Applying Configuration (Declarative Approach)

The `apply` command is the recommended way to manage Kubernetes resources in production:

```
 # Apply a configuration file
kubectl apply -f <filename.yaml>

# Apply multiple files
kubectl apply -f <file1.yaml> -f <file2.yaml>

# Apply all files in a directory
kubectl apply -f <directory>/

# Apply configurations from a URL
kubectl apply -f https://example.com/manifest.yaml

# Apply with record flag to record the command in the resource annotation
kubectl apply -f <filename.yaml> --record
```

Viewing and Finding Resources

These commands help you list and find resources in your cluster:

```
 # List all resources in the current namespace
kubectl get all

# List specific resource types
kubectl get pods
kubectl get deployments
kubectl get services
kubectl get configmaps
kubectl get secrets
kubectl get nodes
kubectl get namespaces

# Get resources with more details
kubectl get pods -o wide
kubectl get deployments -o wide

# Format output as YAML
kubectl get pod <pod-name> -o yaml

# Format output as JSON
kubectl get pod <pod-name> -o json

# List resources with custom columns
```

```
kubectl get pods -o custom-
columns=NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName

# Sort resources by a field
kubectl get pods --sort-by=.metadata.creationTimestamp

# Filter resources using a label selector
kubectl get pods -l app=nginx

# List resources across all namespaces
kubectl get pods --all-namespaces
```

Describing Resources in Detail

The `describe` command provides detailed information about a resource:

```
 # Describe a specific pod
kubectl describe pod <pod-name>

# Describe all pods
kubectl describe pods

# Describe a deployment
kubectl describe deployment <deployment-name>

# Describe a service
kubectl describe service <service-name>

# Describe a node
kubectl describe node <node-name>

# Describe resources selected by label
kubectl describe pods -l app=nginx
```

Deleting Resources

Remove resources from your cluster:

```
 # Delete a specific resource
kubectl delete pod <pod-name>
kubectl delete deployment <deployment-name>
kubectl delete service <service-name>

# Delete using a file
kubectl delete -f <filename.yaml>
```

```
# Delete all pods in the current namespace
kubectl delete pods --all

# Delete resources using label selectors
kubectl delete pods -l app=nginx

# Delete a namespace and all its resources
kubectl delete namespace <namespace-name>

# Force delete a pod without waiting for confirmation from the API server
kubectl delete pod <pod-name> --grace-period=0 --force

# Delete all resources in the current namespace
kubectl delete all --all
```

Explaining Resources

The `explain` command helps you understand resource definitions:

```
 # Get documentation for pod resource
kubectl explain pods

# Get documentation for a specific field
kubectl explain pods.spec.containers

# Get recursive documentation
kubectl explain pods --recursive

# Get documentation for a specific API version
kubectl explain pods --api-version=v1
```

Working with Labels and Annotations

Labels and annotations help organize and select your resources:

```
 # Add a label to a resource
kubectl label pods <pod-name> environment=production

# Update an existing label
kubectl label pods <pod-name> environment=development --overwrite

# Remove a label
kubectl label pods <pod-name> environment-

# Add an annotation
kubectl annotate pods <pod-name> description="My description"
```

```
# Show labels for resources
kubectl get pods --show-labels

# Filter resources by label
kubectl get pods -l environment=production
kubectl get pods -l 'environment in (production,development)'
kubectl get pods -l 'environment notin (production,development)'
```

These basic operations form the foundation of your daily work with Kubernetes. In the next sections, we'll explore more specific operations for deployments, configurations, and troubleshooting.

Deployment Management

Deployments are one of the most commonly used resources in Kubernetes, allowing you to declaratively manage application updates. This section covers essential commands for working with deployments in your day-to-day operations.

Creating and Updating Deployments

```
 # Create a deployment
kubectl create deployment nginx --image=nginx

# Create a deployment with a specific number of replicas
kubectl create deployment nginx --image=nginx --replicas=3

# Create a deployment and expose it as a service
kubectl create deployment nginx --image=nginx --port=80

# Apply a deployment from a YAML file
kubectl apply -f deployment.yaml

# Update a deployment's image
kubectl set image deployment/nginx nginx=nginx:1.19

# Edit a deployment directly
kubectl edit deployment nginx
```

Deployment YAML Example

Here's a basic deployment YAML file for reference:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: "500m"
            memory: "512Mi"
          requests:
            cpu: "100m"
            memory: "128Mi"
```

Scaling Deployments

```bash
# Scale a deployment to a specific number of replicas
kubectl scale deployment nginx --replicas=5

# Scale multiple deployments
kubectl scale deployment nginx1 nginx2 --replicas=0

# Scale based on a file
kubectl scale --replicas=3 -f deployment.yaml

# Autoscale a deployment (create a Horizontal Pod Autoscaler)
kubectl autoscale deployment nginx --min=2 --max=5 --cpu-percent=80
```

Rolling Updates and Rollbacks

```
 # Update a deployment with a new image (triggers a rolling update)
kubectl set image deployment/nginx nginx=nginx:1.20 --record

# Check the status of a rolling update
kubectl rollout status deployment/nginx

# Pause a rolling update
kubectl rollout pause deployment/nginx

# Resume a paused rolling update
kubectl rollout resume deployment/nginx

# View rollout history
kubectl rollout history deployment/nginx

# View details of a specific revision
kubectl rollout history deployment/nginx --revision=2

# Rollback to the previous revision
kubectl rollout undo deployment/nginx

# Rollback to a specific revision
kubectl rollout undo deployment/nginx --to-revision=2

# Restart a deployment (rolling restart of all pods)
kubectl rollout restart deployment/nginx
```

Checking Deployment Status

```
 # Get basic deployment information
kubectl get deployments

# Get detailed deployment information
kubectl describe deployment nginx

# Check the rollout status
kubectl rollout status deployment/nginx

# Get the YAML representation of the current deployment
kubectl get deployment nginx -o yaml

# Check ReplicaSets created by the deployment
kubectl get replicasets -l app=nginx

# Get pods managed by the deployment
kubectl get pods -l app=nginx
```

```
# Watch deployment changes in real-time
kubectl get deployment nginx -w
```

Deployment Strategies

Kubernetes supports different deployment strategies:

1. **RollingUpdate (default)**: Gradually replaces old pods with new ones `yaml spec: strategy: type: RollingUpdate rollingUpdate: maxUnavailable: 25% maxSurge: 25%`

2. **Recreate**: Terminates all existing pods before creating new ones `yaml spec: strategy: type: Recreate`

Deployment Troubleshooting

```
 # Check deployment events
kubectl describe deployment nginx

# Check ReplicaSet events
kubectl describe rs <replicaset-name>

# Check pod events
kubectl describe pod <pod-name>

# Check pod logs
kubectl logs <pod-name>

# Check previous container logs if a container has restarted
kubectl logs <pod-name> --previous

# Check container status and reasons for crashes
kubectl get pod <pod-name> -o yaml
```

Configuration Management

Configuration management is crucial for maintaining application settings and secrets in Kubernetes. This section covers commands for working with ConfigMaps, Secrets, and other configuration resources.

Working with ConfigMaps

ConfigMaps allow you to decouple configuration from container images.

```
 # Create a ConfigMap from literal values
kubectl create configmap app-config --from-literal=key1=value1 --from-
literal=key2=value2

# Create a ConfigMap from a file
kubectl create configmap app-config --from-file=config.properties

# Create a ConfigMap from a directory
kubectl create configmap app-config --from-file=config-dir/

# Get ConfigMaps
kubectl get configmaps

# Describe a ConfigMap
kubectl describe configmap app-config

# Edit a ConfigMap
kubectl edit configmap app-config

# Delete a ConfigMap
kubectl delete configmap app-config
```

ConfigMap YAML Example

```
 apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  database_url: "mysql://mysql:3306/mydb"
  cache_url: "redis://redis:6379/0"
  ui_properties: |
    color=blue
    size=medium
    shape=round
```

Managing Secrets

Secrets are similar to ConfigMaps but are intended for sensitive data.

```
# Create a Secret from literal values
kubectl create secret generic db-credentials --from-literal=username=admin --from-literal=password=secret

# Create a Secret from files
kubectl create secret generic tls-certs --from-file=cert.pem --from-file=key.pem

# Create a TLS Secret
kubectl create secret tls tls-secret --cert=cert.pem --key=key.pem

# Get Secrets
kubectl get secrets

# Describe a Secret (note that values are not shown)
kubectl describe secret db-credentials

# Get Secret values (base64 encoded)
kubectl get secret db-credentials -o yaml

# Decode a Secret value
kubectl get secret db-credentials -o jsonpath='{.data.password}' | base64 --decode

# Edit a Secret
kubectl edit secret db-credentials

# Delete a Secret
kubectl delete secret db-credentials
```

Secret YAML Example

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  username: YWRtaW4=  # base64 encoded "admin"
  password: c2VjcmV0  # base64 encoded "secret"
```

Setting Environment Variables

```
# Set environment variables in a deployment from a ConfigMap
kubectl set env deployment/nginx --from=configmap/app-config

# Set environment variables in a deployment from a Secret
kubectl set env deployment/nginx --from=secret/db-credentials
```

```
# Set a specific environment variable
kubectl set env deployment/nginx DB_HOST=mysql

# Remove an environment variable
kubectl set env deployment/nginx DB_HOST-

# View environment variables in a pod
kubectl exec <pod-name> -- env
```

Resource Quotas and Limits

```
 # Create a ResourceQuota
kubectl create quota my-quota --hard=cpu=1,memory=1G,pods=10

# Get ResourceQuotas
kubectl get resourcequotas

# Describe a ResourceQuota
kubectl describe resourcequota my-quota

# Create a LimitRange
kubectl create -f limit-range.yaml

# Get LimitRanges
kubectl get limitranges

# Describe a LimitRange
kubectl describe limitrange cpu-limit-range
```

LimitRange YAML Example

```yaml
 apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: "1"
      memory: "512Mi"
    defaultRequest:
      cpu: "0.5"
      memory: "256Mi"
    type: Container
```

Pod Operations

Pods are the smallest deployable units in Kubernetes. This section covers essential commands for working with pods.

Creating and Running Pods

```
# Create a pod directly (not recommended for production)
kubectl run nginx --image=nginx

# Create a pod with specific resource requests and limits
kubectl run nginx --image=nginx --requests=cpu=100m,memory=128Mi --
limits=cpu=500m,memory=512Mi

# Create a pod and expose it as a service
kubectl run nginx --image=nginx --port=80 --expose

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

Pod YAML Example

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
    resources:
      limits:
        cpu: "500m"
        memory: "512Mi"
      requests:
        cpu: "100m"
        memory: "128Mi"
```

Accessing Pod Logs

```
 # Get logs from a pod
kubectl logs <pod-name>

# Get logs from a specific container in a multi-container pod
kubectl logs <pod-name> -c <container-name>

# Stream logs in real-time
kubectl logs -f <pod-name>

# Get logs with timestamps
kubectl logs <pod-name> --timestamps

# Get logs from the previous instance of a container
kubectl logs <pod-name> --previous

# Get a specific number of lines from the log
kubectl logs <pod-name> --tail=100

# Get logs since a specific time
kubectl logs <pod-name> --since=1h
```

Executing Commands in Containers

```
 # Execute a command in a pod
kubectl exec <pod-name> -- ls -la

# Execute a command in a specific container of a multi-container pod
kubectl exec <pod-name> -c <container-name> -- ls -la

# Start an interactive shell in a pod
kubectl exec -it <pod-name> -- /bin/bash

# Copy files from a pod to local machine
kubectl cp <pod-name>:/path/to/file /local/path

# Copy files from local machine to a pod
kubectl cp /local/path <pod-name>:/path/in/pod
```

Port Forwarding to Pods

```
 # Forward a local port to a port on the pod
kubectl port-forward <pod-name> 8080:80
```

```
# Forward multiple ports
kubectl port-forward <pod-name> 8080:80 8443:443

# Forward to a deployment, service, or replicaset
kubectl port-forward deployment/nginx 8080:80
kubectl port-forward service/nginx 8080:80
kubectl port-forward rs/nginx 8080:80
```

Troubleshooting

Effective troubleshooting is essential for maintaining a healthy Kubernetes cluster. This section covers commands and techniques for diagnosing and resolving common issues.

Debugging Pods and Containers

```
 # Check pod status
kubectl get pods

# Get detailed information about a pod
kubectl describe pod <pod-name>

# Check pod logs
kubectl logs <pod-name>

# Check previous container logs if a container has restarted
kubectl logs <pod-name> --previous

# Check events in the namespace
kubectl get events

# Sort events by timestamp
kubectl get events --sort-by=.metadata.creationTimestamp

# Create a debugging container
kubectl run debug --image=busybox --rm -it --restart=Never -- sh

# Create a debugging container in the same namespace as a problematic pod
kubectl run debug --image=busybox --rm -it --restart=Never --
namespace=<namespace> -- sh

# Create an ephemeral debug container in an existing pod (Kubernetes v1.18+)
kubectl debug -it <pod-name> --image=busybox --target=<container-name>
```

Viewing Logs and Events

```
 # View cluster-wide events
kubectl get events --all-namespaces

# View events for a specific resource
kubectl describe pod <pod-name> | grep -A 10 Events:

# View events for a specific namespace
kubectl get events -n <namespace>

# View warning events only
kubectl get events --field-selector type=Warning

# View events related to a specific object
kubectl get events --field-selector involvedObject.name=<pod-name>

# Stream logs from all containers in a deployment
kubectl logs -f deployment/<deployment-name> --all-containers=true

# Get logs from all pods with a specific label
kubectl logs -l app=nginx
```

Checking Resource Status

```
 # Check node status
kubectl get nodes
kubectl describe node <node-name>

# Check pod resource usage
kubectl top pods

# Check node resource usage
kubectl top nodes

# Check pod status with wide output
kubectl get pods -o wide

# Check deployment status
kubectl rollout status deployment/<deployment-name>

# Check service endpoints
kubectl get endpoints <service-name>

# Check if a service is properly connected to pods
kubectl describe service <service-name>
```

Common Error Patterns and Solutions

Pod in Pending State

```
# Check if there are resource constraints
kubectl describe pod <pod-name> | grep -A 10 Events:
```

Common causes: - Insufficient resources (CPU, memory) - PersistentVolumeClaim not bound - Node selector or affinity rules can't be satisfied

Pod in CrashLoopBackOff State

```
# Check container logs
kubectl logs <pod-name>

# Check previous container logs
kubectl logs <pod-name> --previous
```

Common causes: - Application error - Missing configuration - Resource limits too low

Pod in ImagePullBackOff State

```
# Check pod events
kubectl describe pod <pod-name> | grep -A 10 Events:
```

Common causes: - Image doesn't exist - Private registry authentication issues - Network connectivity problems

Service Not Routing Traffic

```
# Check service and endpoints
kubectl get service <service-name>
kubectl get endpoints <service-name>

# Check if pods are running and ready
kubectl get pods -l <selector-used-by-service>

# Check if service selector matches pod labels
kubectl describe service <service-name>
kubectl get pods --show-labels
```

Useful Debugging Commands

```
 # Check DNS resolution inside a pod
kubectl exec -it <pod-name> -- nslookup kubernetes.default

# Check network connectivity from a pod
kubectl exec -it <pod-name> -- wget -O- http://service-name:port

# Check API server connectivity
kubectl exec -it <pod-name> -- curl -k https://kubernetes.default.svc

# Get a shell to a node (requires SSH access)
ssh <node-name>

# Check kubelet logs on a node
ssh <node-name> 'sudo journalctl -u kubelet'

# Check container runtime logs
ssh <node-name> 'sudo journalctl -u docker' # for Docker
ssh <node-name> 'sudo journalctl -u containerd' # for containerd
```

These advanced operations will help you manage deployments, handle configurations, work with pods, and troubleshoot issues in your Kubernetes clusters. As you become more familiar with these commands, you'll be able to manage your Kubernetes resources more efficiently and effectively.

Expanded Examples for `kubectl run` Command

The `kubectl run` command is a versatile tool for quickly creating and running pods in your Kubernetes cluster. Here's a comprehensive set of examples to help you understand its capabilities:

Basic Usage

```
 # Run a simple nginx pod
kubectl run nginx --image=nginx

# Run a pod with a specific version of an image
kubectl run nginx --image=nginx:1.19.0

# Run a pod and expose it on a specific port
```

```
kubectl run nginx --image=nginx --port=80

# Run a pod with a specific label
kubectl run nginx --image=nginx --labels="app=web,tier=frontend"

# Run a pod in a specific namespace
kubectl run nginx --image=nginx --namespace=development
```

Resource Management

```
 # Run a pod with specific resource requests
kubectl run resource-demo --image=nginx --requests=cpu=100m,memory=128Mi

# Run a pod with both resource requests and limits
kubectl run resource-demo --image=nginx --requests=cpu=100m,memory=128Mi --
limits=cpu=500m,memory=256Mi

# Run a pod with a specific service account
kubectl run sa-demo --image=nginx --serviceaccount=custom-sa
```

Command and Arguments

```
 # Run a pod with a specific command
kubectl run command-demo --image=busybox --command -- sleep 3600

# Run a pod with command arguments
kubectl run command-demo --image=busybox -- echo "Hello Kubernetes!"

# Run a pod with multiple command arguments
kubectl run command-demo --image=busybox -- /bin/sh -c "while true; do echo
hello; sleep 10; done"
```

Environment Variables

```
 # Run a pod with environment variables
kubectl run env-demo --image=nginx --env="DB_HOST=mysql" --
env="DB_PORT=3306"

# Run a pod with environment variables from a ConfigMap
kubectl run env-demo --image=nginx --env="DB_HOST=$(kubectl get configmap
db-config -o jsonpath='{.data.host}')"
```

Restart Policies

```
 # Run a pod with a specific restart policy
kubectl run restart-demo --image=busybox --restart=Never -- echo "This pod will
not restart"

# Run a pod that will restart on failure
kubectl run restart-demo --image=busybox --restart=OnFailure -- /bin/sh -c "exit 1"

# Run a pod that will always restart (default behavior)
kubectl run restart-demo --image=busybox --restart=Always -- sleep 3600
```

Temporary Pods for Debugging

```
 # Run a temporary interactive pod for debugging
kubectl run debug --image=busybox --rm -it --restart=Never -- sh

# Run a temporary pod to test network connectivity
kubectl run test-connectivity --image=busybox --rm -it --restart=Never -- wget -O-
http://nginx-service:80

# Run a temporary pod to test DNS resolution
kubectl run dns-test --image=busybox --rm -it --restart=Never -- nslookup
kubernetes.default

# Run a temporary pod in a specific namespace for debugging
kubectl run debug --image=busybox --rm -it --restart=Never --
namespace=production -- sh
```

Advanced Options

```
 # Run a pod with a specific node selector
kubectl run node-selector-demo --image=nginx --overrides='{"spec":
{"nodeSelector": {"disktype": "ssd"}}}'

# Run a pod with tolerations
kubectl run toleration-demo --image=nginx --overrides='{"spec": {"tolerations":
[{"key": "example-key", "operator": "Exists", "effect": "NoSchedule"}]}}'

# Run a pod with a specific priority class
kubectl run priority-demo --image=nginx --overrides='{"spec":
{"priorityClassName": "high-priority"}}'

# Run a pod with a specific security context
```

```
kubectl run security-demo --image=nginx --overrides='{"spec": {"securityContext":
{"runAsUser": 1000, "runAsGroup": 3000}}}'
```

Generating YAML Instead of Creating a Pod

```
 # Generate YAML for a pod without creating it
kubectl run nginx --image=nginx --dry-run=client -o yaml > nginx-pod.yaml

# Generate YAML with multiple options
kubectl run complex-pod --image=nginx --port=80 --labels="app=web" --
requests=cpu=100m,memory=128Mi --limits=cpu=500m,memory=256Mi --
env="ENV=production" --dry-run=client -o yaml > complex-pod.yaml
```

Combining with Other Commands

```
 # Create a pod and immediately get its details
kubectl run nginx --image=nginx && kubectl describe pod nginx

# Create a pod and watch its status
kubectl run nginx --image=nginx && kubectl get pod nginx -w

# Create a pod and immediately get its logs
kubectl run log-demo --image=busybox -- echo "Hello logs" && sleep 5 && kubectl
logs log-demo
```

Practical Use Cases

```
 # Run a database migration job
kubectl run db-migration --image=flyway --restart=Never -- flyway migrate

# Run a data backup job
kubectl run backup --image=backup-tool --restart=Never -- backup --source=/data --
destination=s3://backup

# Run a pod for load testing
kubectl run load-test --image=locust --restart=Never -- locust -f /tests/locustfile.py --
host=http://target-service

# Run a pod for health checking
kubectl run health-check --image=curlimages/curl --restart=Never -- curl -f http://
service:8080/health
```

Best Practices

1. **Use Deployments for long-running applications**: While `kubectl run` is convenient for quick tasks, use Deployments for production workloads: `bash kubectl create deployment nginx --image=nginx --replicas=3`

2. **Use Jobs for batch processes**: For completion tasks, use Jobs instead of pods with `restart=Never`: `bash kubectl create job backup --image=backup-tool -- perform-backup`

3. **Use CronJobs for scheduled tasks**: For recurring tasks, use CronJobs: `bash kubectl create cronjob cleanup --image=cleanup-tool --schedule="0 2 * * *" -- cleanup-old-data`

4. **Generate YAML for version control**: Use the `--dry-run=client -o yaml` flags to generate YAML that can be stored in version control: `bash kubectl run nginx --image=nginx --dry-run=client -o yaml > nginx-pod.yaml`

5. **Use labels for organization**: Always add meaningful labels to your pods: `bash kubectl run nginx --image=nginx --labels="app=web,environment=dev,team=frontend"`

These examples demonstrate the flexibility and power of the `kubectl run` command for various scenarios you'll encounter as a Kubernetes engineer.

Expanded Examples for `kubectl create` Command

The `kubectl create` command is essential for creating Kubernetes resources. Here's a comprehensive set of examples to help you understand its capabilities:

Basic Resource Creation

```
 # Create a namespace
kubectl create namespace development

# Create a simple pod (though kubectl run or apply is typically preferred for pods)
kubectl create -f pod.yaml

# Create a deployment
```

```
kubectl create deployment nginx --image=nginx

# Create a job
kubectl create job backup-job --image=backup-tool

# Create a cronjob
kubectl create cronjob cleanup --image=cleanup-tool --schedule="0 2 * * *"
```

ConfigMaps and Secrets

```
 # Create a ConfigMap from literal values
kubectl create configmap app-config --from-literal=API_URL=https://
api.example.com --from-literal=MAX_CONNECTIONS=100

# Create a ConfigMap from a file
kubectl create configmap app-config --from-file=config.properties

# Create a ConfigMap from a directory of files
kubectl create configmap app-config --from-file=config-dir/

# Create a ConfigMap from an env file
kubectl create configmap app-config --from-env-file=.env

# Create a generic Secret from literal values
kubectl create secret generic db-credentials --from-literal=username=admin --from-
literal=password=s3cr3t

# Create a Secret from files
kubectl create secret generic tls-certs --from-file=cert.pem --from-file=key.pem

# Create a TLS Secret
kubectl create secret tls tls-secret --cert=path/to/tls.cert --key=path/to/tls.key

# Create a docker-registry Secret for private registry authentication
kubectl create secret docker-registry regcred --docker-server=<your-registry-
server> --docker-username=<your-name> --docker-password=<your-password> --
docker-email=<your-email>
```

Service Resources

```
 # Create a ClusterIP service
kubectl create service clusterip nginx --tcp=80:80

# Create a NodePort service
kubectl create service nodeport nginx --tcp=80:80
```

```
# Create a LoadBalancer service
kubectl create service loadbalancer nginx --tcp=80:80

# Create a ExternalName service
kubectl create service externalname my-service --external-name example.com
```

RBAC Resources

```
 # Create a ServiceAccount
kubectl create serviceaccount jenkins

# Create a Role
kubectl create role pod-reader --verb=get,list,watch --resource=pods

# Create a ClusterRole
kubectl create clusterrole pod-reader --verb=get,list,watch --resource=pods

# Create a RoleBinding
kubectl create rolebinding read-pods --role=pod-reader --
serviceaccount=default:default

# Create a ClusterRoleBinding
kubectl create clusterrolebinding read-pods --clusterrole=pod-reader --
serviceaccount=default:default

# Create a RoleBinding for a user
kubectl create rolebinding admin --role=admin --user=user1
```

Resource Quotas and Limits

```
 # Create a ResourceQuota
kubectl create quota my-quota --hard=cpu=1,memory=1G,pods=10

# Create a ResourceQuota with scopes
kubectl create quota my-quota --hard=cpu=1,memory=1G --scopes=BestEffort
```

Generating Resource YAML

```
 # Generate a deployment YAML without creating it
kubectl create deployment nginx --image=nginx --dry-run=client -o yaml >
deployment.yaml

# Generate a service YAML without creating it
kubectl create service clusterip nginx --tcp=80:80 --dry-run=client -o yaml >
```

```
service.yaml

# Generate a namespace YAML without creating it
kubectl create namespace development --dry-run=client -o yaml > namespace.yaml

# Generate a configmap YAML without creating it
kubectl create configmap app-config --from-literal=API_URL=https://
api.example.com --dry-run=client -o yaml > configmap.yaml

# Generate a secret YAML without creating it
kubectl create secret generic db-credentials --from-literal=username=admin --from-
literal=password=s3cr3t --dry-run=client -o yaml > secret.yaml
```

Creating Multiple Resources

```
 # Create multiple resources from a directory
kubectl create -f ./resources/

# Create multiple resources from URLs
kubectl create -f https://raw.githubusercontent.com/kubernetes/examples/master/
guestbook/redis-master-deployment.yaml -f https://raw.githubusercontent.com/
kubernetes/examples/master/guestbook/redis-master-service.yaml

# Create multiple resources from a combination of files, directories, and URLs
kubectl create -f ./deployment.yaml -f ./service/ -f https://example.com/
configmap.yaml
```

Creating Resources with Additional Options

```
 # Create a deployment with a specific number of replicas
kubectl create deployment nginx --image=nginx --replicas=3

# Create a deployment and expose it as a service
kubectl create deployment nginx --image=nginx --port=80

# Create a job with a specific command
kubectl create job pi --image=perl -- perl -Mbignum=bpi -wle 'print bpi(2000)'

# Create a cronjob with specific command arguments
kubectl create cronjob date-printer --image=busybox --schedule="*/1 * * * *" -- /
bin/sh -c "date; echo Hello from Kubernetes"
```

Creating Resources in Different Namespaces

```
 # Create a deployment in a specific namespace
kubectl create deployment nginx --image=nginx --namespace=development

# Create a service in a specific namespace
kubectl create service clusterip nginx --tcp=80:80 --namespace=development

# Create a configmap in a specific namespace
kubectl create configmap app-config --from-literal=ENV=dev --
namespace=development
```

Creating Resources with Labels

```
 # Create a deployment with labels
kubectl create deployment nginx --image=nginx --labels="app=web,tier=frontend"

# Create a service with labels
kubectl create service clusterip nginx --tcp=80:80 --labels="app=web,tier=frontend"
```

Practical Use Cases

```
 # Create a complete application stack
kubectl create namespace my-app
kubectl create deployment frontend --image=frontend:v1 --namespace=my-app
kubectl create deployment backend --image=backend:v1 --namespace=my-app
kubectl create service clusterip frontend --tcp=80:80 --namespace=my-app
kubectl create service clusterip backend --tcp=8080:8080 --namespace=my-app
kubectl create configmap app-config --from-file=config.json --namespace=my-app
kubectl create secret generic app-secrets --from-file=secrets.json --namespace=my-
app

# Create a database with persistent storage
kubectl create -f pvc.yaml
kubectl create deployment postgres --image=postgres:13 --namespace=database
kubectl set env deployment/postgres POSTGRES_PASSWORD=secretpassword
kubectl create service clusterip postgres --tcp=5432:5432 --namespace=database

# Create a scheduled backup job
kubectl create namespace backup
kubectl create serviceaccount backup-sa --namespace=backup
kubectl create rolebinding backup-rb --role=backup-role --
serviceaccount=backup:backup-sa --namespace=backup
```

```
kubectl create cronjob db-backup --image=backup-tool --schedule="0 2 * * *" --
namespace=backup
```

Best Practices

1. **Use declarative files for production**: While `kubectl create` is convenient for quick tasks, use YAML files with `kubectl apply` for production: `bash # Generate the YAML first kubectl create deployment nginx --image=nginx --dry-run=client -o yaml > deployment.yaml # Edit if needed, then apply kubectl apply -f deployment.yaml`

2. **Use namespaces for organization**: `bash kubectl create namespace development kubectl create namespace staging kubectl create namespace production`

3. **Add meaningful labels**: `bash kubectl create deployment app --image=app:v1 --labels="app=myapp,environment=dev,team=frontend"`

4. **Use resource quotas for governance**: `bash kubectl create quota dev-quota --hard=cpu=4,memory=8G,pods=10 --namespace=development`

5. **Create resources with proper RBAC**: `bash kubectl create serviceaccount app-sa kubectl create role app-role --verb=get,list,watch --resource=pods,services kubectl create rolebinding app-rb --role=app-role --serviceaccount=default:app-sa`

These examples demonstrate the versatility of the `kubectl create` command for various scenarios you'll encounter as a Kubernetes engineer.