Kubernetes Admission Control - Student Guide

Learning Objectives

By the end of this lesson, you will be able to:

- Understand what admission control is in Kubernetes
- Configure and use built-in admission controllers
- Implement resource limits using LimitRange
- Set up and use OPA Gatekeeper for policy enforcement
- Troubleshoot common admission control issues

Prerequisites

- Basic knowledge of Kubernetes concepts (Pods, Namespaces, Resources)
- kubectl CLI installed and configured
- minikube installed
- Understanding of YAML manifests

What is Admission Control?

Admission control is a phase in the Kubernetes API request lifecycle that occurs after authentication and authorization but before the object is persisted to etcd. It consists of two types of controllers:

1. **Mutating Admission Controllers**: Modify objects before they are stored
2. **Validating Admission Controllers**: Validate objects and can reject requests

Lab Environment Setup

Start your minikube cluster with specific admission plugins enabled:

```bash
minikube start --extra-config=apiserver.enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,D
```

Admission Plugins Explained

- **NamespaceLifecycle**: Ensures objects cannot be created in non-existent or terminating namespaces
- **LimitRanger**: Enforces resource limits on containers and pods

- **ServiceAccount**: Automatically assigns service accounts to pods
- **DefaultStorageClass**: Assigns default storage class to PVCs
- **NodeRestriction**: Restricts what nodes can modify about themselves
- **MutatingAdmissionWebhook**: Enables custom mutating webhooks
- **ValidatingAdmissionWebhook**: Enables custom validating webhooks

Exercise 1: Resource Limits with LimitRange

Step 1: Create a LimitRange

Create a file called `limit.yaml`:

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default: # Default limits applied if not specified
      cpu: 500m
    defaultRequest: # Default requests applied if not specified
      cpu: 500m
    max: # Maximum allowed values
      cpu: "1"
    min: # Minimum required values
      cpu: 100m
    type: Container
```

Step 2: Apply the LimitRange

```bash
kubectl apply -f limit.yaml
```

Expected output:

```
limitrange/cpu-resource-constraint created
```

Step 3: Test Resource Limits

Create a pod that exceeds the limits in `resource-exceeding-pod.yaml`:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-exceeding-pod
spec:
  containers:
  - name: test-container
    image: nginx
    resources:
      requests:
        cpu: "12"  # This exceeds our limit of 1 CPU
```

Try to apply it:

```bash
kubectl apply -f resource-exceeding-pod.yaml
```

Expected error:

```
The Pod "resource-exceeding-pod" is invalid: spec.containers[0].resources.requests: Invalid value: "12": must be less than or equal to cpu limit of 500m
```

Understanding the Error

The LimitRanger admission controller rejected the pod because:

- The requested CPU (12 cores) exceeds the maximum allowed (1 core)
- The default limit (500m) is applied when no limit is specified

Exercise 2: Namespace Validation

Test Invalid Namespace Names

Try creating a namespace with an invalid name:

```bash
kubectl create namespace InvalidNamespaceName
```

Expected error:

```
The Namespace "InvalidNamespaceName" is invalid: metadata.name: Invalid value: "InvalidNamespaceName": a lowercase RFC 1123 label must consist of lower case alphanumeric characters or '-', and must start and end with an alphanumeric character (e.g. 'my-name', or '123-abc', regex used for validation is '[a-z0-9]([-a-z0-9]*[a-z0-9])?')
```

Understanding Namespace Validation

The NamespaceLifecycle admission controller enforces RFC 1123 naming conventions:

- Must be lowercase
- Can contain alphanumeric characters and hyphens
- Must start and end with alphanumeric characters
- Maximum length of 63 characters

Create a Valid Namespace

```bash
kubectl create namespace valid-namespace-name
```

Exercise 3: OPA Gatekeeper Setup

Step 1: Reset the Environment

```bash
minikube delete
minikube start --extra-config=apiserver.enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,D
```

Step 2: Install OPA Gatekeeper

```bash
kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/master/deploy/gatekeeper.yaml
```

Wait for Gatekeeper to be ready:

```bash
kubectl wait --for=condition=Ready pod -l control-plane=controller-manager -n gatekeeper-system --timeout=90s
```

Step 3: Apply HTTPS-Only Policy

Apply the constraint template:

```bash
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/
```

Apply the constraint:

```bash
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/
```

Step 4: Test Policy Enforcement

Try to create an ingress without HTTPS:

```bash
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/
```

Expected error:

```
Error from server (Forbidden): error when creating "...": admission webhook "validation.gatekeeper.sh" denied the request: [ingress-https-only] Ingress must be https. tls configuration is required for test-ingress
```

Step 5: Create a Valid HTTPS Ingress

```bash
kubectl apply -f https://raw.githubusercontent.com/citrix/citrix-k8s-ingress-controller/master/docs/how-to/webhook/
```

Expected output:

```
ingress.networking.k8s.io/test-ingress created
ingressclass.networking.k8s.io/citrix-ingress created
```

Key Concepts Summary

Built-in Admission Controllers

- **LimitRanger**: Enforces resource constraints
- **NamespaceLifecycle**: Validates namespace operations
- **ResourceQuota**: Enforces resource quotas per namespace
- **PodSecurityPolicy**: Enforces pod security policies (deprecated in favor of Pod Security Standards)

Custom Admission Controllers

- **OPA Gatekeeper**: Policy-as-code using Rego language
- **Custom Webhooks**: Implement custom business logic

Best Practices

1. **Always test policies in development** before applying to production
2. **Use dry-run mode** to validate changes: `kubectl apply --dry-run=server`
3. **Monitor admission controller performance** as they add latency to API requests
4. **Implement gradual rollouts** for new policies
5. **Have rollback plans** for admission controller changes

Troubleshooting Common Issues

Policy Not Taking Effect

```bash
# Check if Gatekeeper is running
kubectl get pods -n gatekeeper-system

# Verify constraint template is applied
kubectl get constrainttemplates

# Check constraint status
kubectl get constraints
```

Resource Limit Issues

```bash
```

```
# View current limit ranges
kubectl get limitrange

# Describe limit range for details
kubectl describe limitrange cpu-resource-constraint
```

Webhook Failures

```bash
# Check webhook configurations
kubectl get validatingadmissionwebhooks
kubectl get mutatingadmissionwebhooks

# View admission controller logs
kubectl logs -n gatekeeper-system -l control-plane=controller-manager
```

Hands-On Exercises

Exercise A: Create Custom Resource Limits

1. Create a LimitRange that limits memory to 512Mi max and 128Mi default
2. Test with a pod that exceeds this limit
3. Create a valid pod within the limits

Exercise B: Namespace Policies

1. Create a Gatekeeper policy that requires all namespaces to have a "team" label
2. Test creating namespaces with and without the required label

Exercise C: Security Policies

1. Implement a policy that prevents containers from running as root
2. Test with privileged and unprivileged containers

Additional Resources

- Kubernetes Admission Controllers Documentation
- OPA Gatekeeper Library
- Pod Security Standards

Assessment Questions

1. What is the difference between mutating and validating admission controllers?
2. In what order are admission controllers executed?
3. How would you debug a failing admission webhook?
4. What happens if an admission controller is misconfigured?
5. How do you temporarily bypass admission control for emergency situations?

Lab Cleanup

When finished with the lab:

```bash
minikube delete
```