Module 5-7: Scheduling, Logging, Metrics, Dashboard, Ingress, RBAC, CRD

Master essential Kubernetes operations, covering advanced scheduling techniques like node labeling and taints/tolerations, robust logging and metrics collection for observability, secure Kubernetes Dashboard setup, efficient Ingress configuration, fine-grained Role-Based Access Control (RBAC), and extending Kubernetes with Custom Resource Definitions (CRDs). This comprehensive guide provides practical `kubectl` commands and real-world examples for production cluster management.

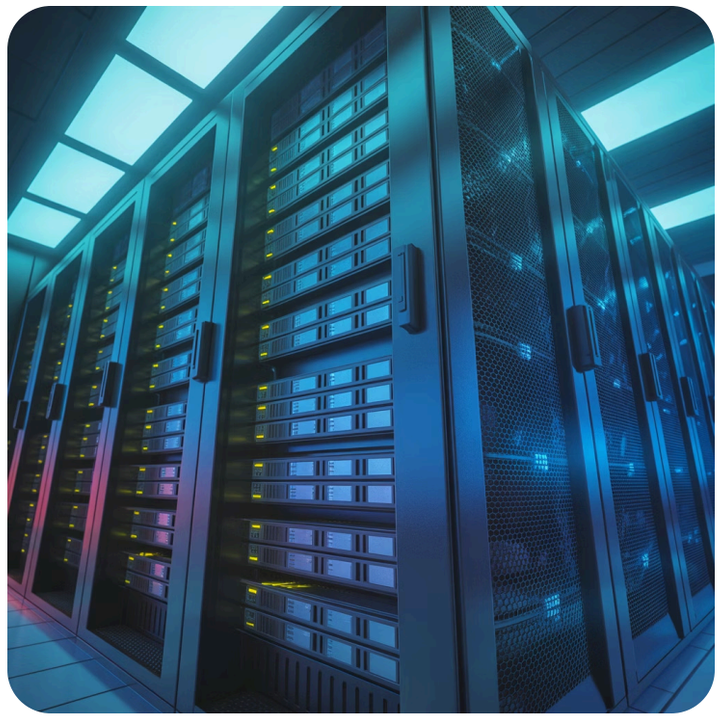
Node Labeling and Selection

Labels are key-value pairs attached to Kubernetes objects like nodes, enabling sophisticated workload placement strategies. By labeling nodes with attributes like disk type, you can ensure pods run on infrastructure that matches their requirements.

Start by applying meaningful labels to your nodes:

```
kubectl label node/master disk=ssd
kubectl label node/worker disk=sata
kubectl get nodes --show-labels
```

Then configure your deployments to target specific nodes using nodeSelector in the pod template specification. This ensures workloads land on appropriate hardware every time.



01

Label Nodes

Apply disk type labels to master and worker nodes for hardware-aware scheduling

02

Configure NodeSelector

Patch deployments to include nodeSelector specifications

03

Verify Placement

Check pod locations with kubectl get po -o wide

Deployment Scaling Strategy

Kubernetes scheduler intelligently distributes pods across available nodes when scaling deployments. This automatic spreading maximizes availability and resource utilization.

Scale your web deployment to observe scheduler behavior:

```
kubectl scale deploy/web --replicas 2
kubectl get po -o wide
kubectl scale deploy/web --replicas 1
```

Single Replica

Pod assigned to node with matching nodeSelector

Scale Up

Scheduler spreads pods across different nodes when possible

Optimal Distribution

Resources balanced, high availability achieved

Taints and Tolerations

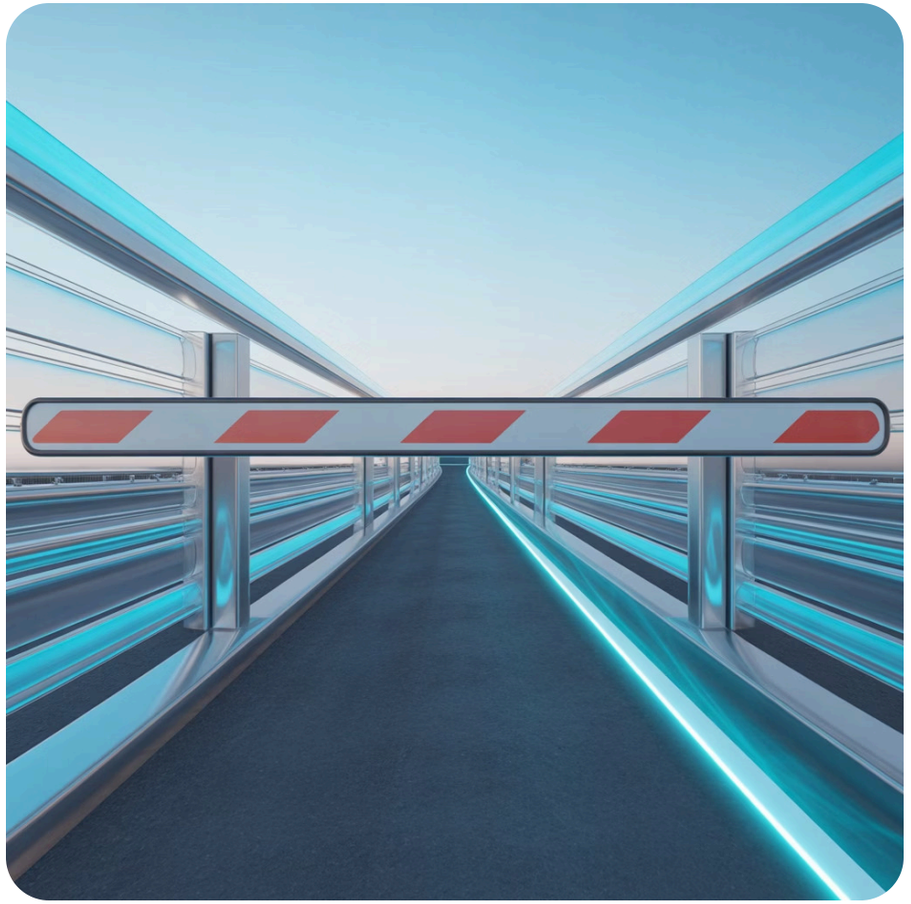
Understanding Taints

Taints prevent pods from being scheduled on specific nodes unless those pods have matching tolerations. This creates dedicated nodes for specialized workloads or maintenance scenarios.

Apply a taint to the master node:

```
kubectl taint node master
dedicated=special:NoSchedule
kubectl describe nodes | grep -i taint
```

After applying the taint, new pods without tolerations cannot schedule on the tainted node. Existing pods continue running unaffected.



NoSchedule

Hard restriction preventing new pod scheduling

PreferNoSchedule

Soft preference, scheduler avoids but allows if needed

NoExecute

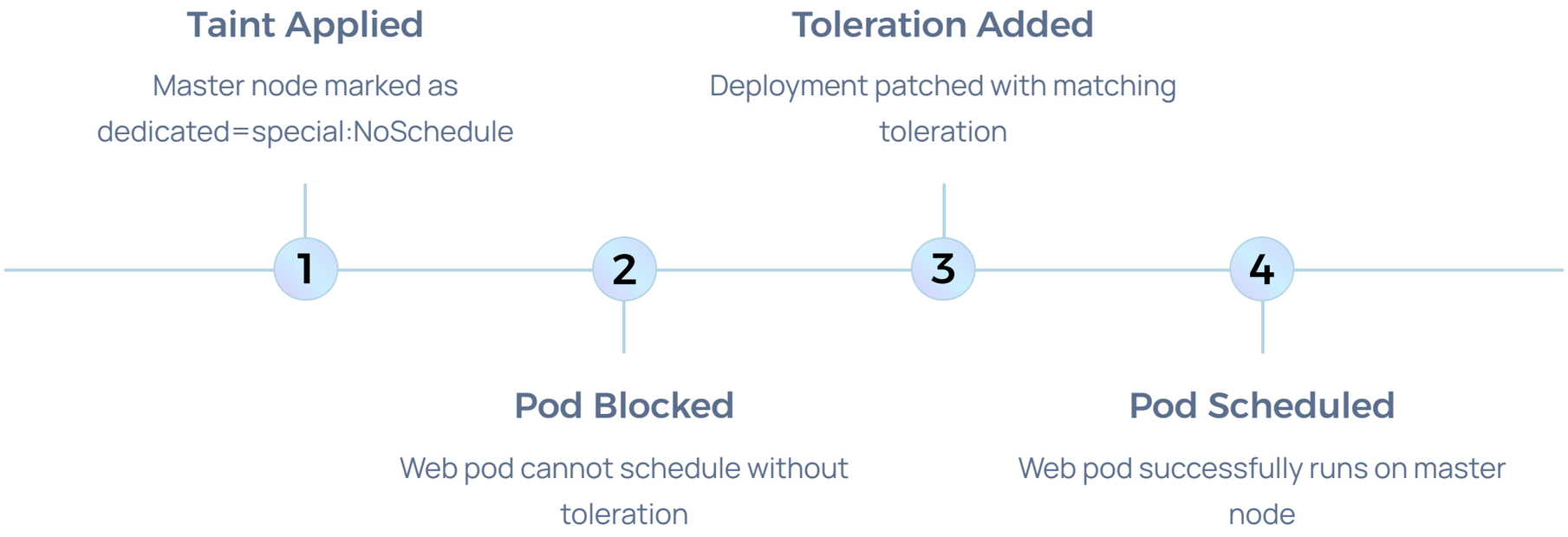
Evicts existing pods and blocks new ones immediately

Adding Tolerations to Deployments

Tolerations allow pods to schedule on tainted nodes, creating exceptions to taint rules. You can add tolerations via kubectl patch or by editing the deployment manifest directly.

Patch the web deployment with toleration:

```
kubectl patch deploy/web -p '{"spec": {"template": {"spec": {"tolerations": [{"key": "dedicated", "operator": "Equal", "value": "special", "effect": "NoSchedule"}]}}}}'
```



Verify the pod placement and clean up when finished:

```
kubectl get po -o wide
kubectl taint node master dedicated-
```

Taint Effect Comparison

NoSchedule

Hard Scheduling Restriction

- New pods blocked without toleration
- Existing pods remain running
- Use for dedicated workload nodes

PreferNoSchedule

Soft Scheduling Preference

- Scheduler tries to avoid the node
- May schedule if no alternatives exist
- Use for gentle workload separation

NoExecute

Immediate Eviction + Restriction

- New pods blocked completely
- Existing pods evicted immediately
- Use for maintenance or emergencies

Cluster Logging and Metrics

Accessing Logs

Kubernetes doesn't include cluster-wide logging by default, but you can access individual pod logs and consider external solutions like Fluentd for aggregation.

View API server logs:

```
kubectl -n kube-system logs kube-apiserver-master
```

Pod logs are also available on the filesystem:

```
ls /var/log/containers/kube-apiserver-master_kube-system_*
```

Installing Metrics Server

The Metrics Server provides resource usage data for nodes and pods, essential for monitoring and autoscaling.

```
kubectl create -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

Configure to accept self-signed certificates by adding the kubelet-insecure-tls flag to the metrics-server deployment.

Pod Logs

kubectl logs command for individual container output

Node Metrics

kubectl top nodes for CPU and memory usage

Pod Metrics

kubectl top pod for resource consumption data

Kubernetes Dashboard Setup

The Kubernetes Dashboard provides a web-based UI for cluster management, making it easier to visualize and interact with your cluster resources.

Add Helm Repository

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/
helm repo update
```

Install Dashboard

```
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-dashboard --create-namespace --namespace kubernetes-dashboard
```

Create Service Account

```
kubectl create sa dashboard-admin -n kubernetes-dashboard
kubectl create clusterrolebinding dashboard-admin --clusterrole=cluster-admin --serviceaccount=kubernetes-dashboard:dashboard-admin
```

Configure Ingress

Create ingress rule with HTTPS backend protocol and SSL passthrough annotations for secure access

Generate Token & Access

```
kubectl -n kube-system create token dashboard-admin
```

Update hosts file and browse to dash.master

Role-Based Access Control (RBAC)

RBAC controls who can access which Kubernetes resources and what actions they can perform. Implement fine-grained security by creating users with limited permissions.

Create a certificate-based user and configure access:

1. Generate private key and CSR for the user
2. Sign with cluster CA to create certificate
3. Add credentials and context to kubeconfig
4. Create Role defining allowed actions
5. Bind Role to user with RoleBinding

Create User Certificate

```
openssl genrsa -out student.key 2048
openssl req -new -key student.key -out student.csr -subj "/CN=student/O=development"
```

Define Role

```
kubectl create role developer --verb=* --resource=deployments,replicasets,pods -n testing
```

Bind Role to User

```
kubectl create rolebinding developer --role=developer --user=student -n testing
```

Custom Resource Definitions

CRDs extend Kubernetes by allowing you to define custom resource types. They enable you to store and retrieve structured data using the Kubernetes API, creating custom objects that behave like native resources.

1

View Existing CRDs

List all custom resource definitions: kubectl get crd --all-namespaces

2

Create CRD

Define your custom resource schema with shortcuts: kubectl apply -f 5s-crd.yaml

3

Create Custom Objects

Instantiate resources using your CRD: kubectl create -f 5s-crontab.yaml

4

Manage Resources

Use kubectl get/describe with custom resource types or shortcuts

Important: Deleting a CRD automatically removes all associated endpoints and objects. Always backup custom resources before removing their definitions.

> Custom Resource Definitions transform Kubernetes into a platform for building custom APIs, enabling you to extend cluster functionality beyond built-in resources.