Hand-Written Notes on
Kubernetes RBAC (Role-Based Access Control)
(A Simple Student Guide)

RBAC Introduction

Role-Based Access Control (RBAC) is a security mechanism in Kubernetes that helps manage permissions for users and applications. It ensures that only authorized users, groups, and services can perform specific actions on Kubernetes resources.

In this RBAC documentation, two key processes take place:

1. **Authentication –** Verifying the user's identity (Are you the right person?).
2. **Authorization –** Determining what actions or permissions (defined by Role or ClusterRole) the user is allowed to perform (What can the user do?).

Role and **ClusterRoles** are actually the permissions that you set for the User, Group, or Service Account, where Role provides namespace-based permissions and ClusterRole provides cluster-level permissions.

How RBAC Works Actually?

In Kubernetes **RBAC (Role-Based Access Control)**, access is controlled through **authentication and authorization**.

Authentication & Authorization in RBAC

1. **Authentication** – Verifying the identity of the object accessing Kubernetes (**Are you the right person/service?**).
2. **Authorization** – Determining what actions or permissions (**defined by Role or ClusterRole**) the object is allowed to perform (**What can you do?**).

RBAC Objects in Kubernetes

Kubernetes RBAC works with three main types of objects:

1. **User**
   - A real person (like a **developer, DevOps engineer, or security auditor**) who interacts with the Kubernetes cluster.

- Needs authentication and authorization before performing any actions.

2. **Group**
   - A **collection of users** with similar access needs.
   - Instead of assigning permissions to each user individually, permissions are assigned to a **Group**, making access management easier.
3. **Service (Service Account)**
   - An **automated process, application, or bot** that runs inside the cluster and needs access to Kubernetes resources.
   - Example: A **monitoring tool** reading logs from the cluster to check application health.

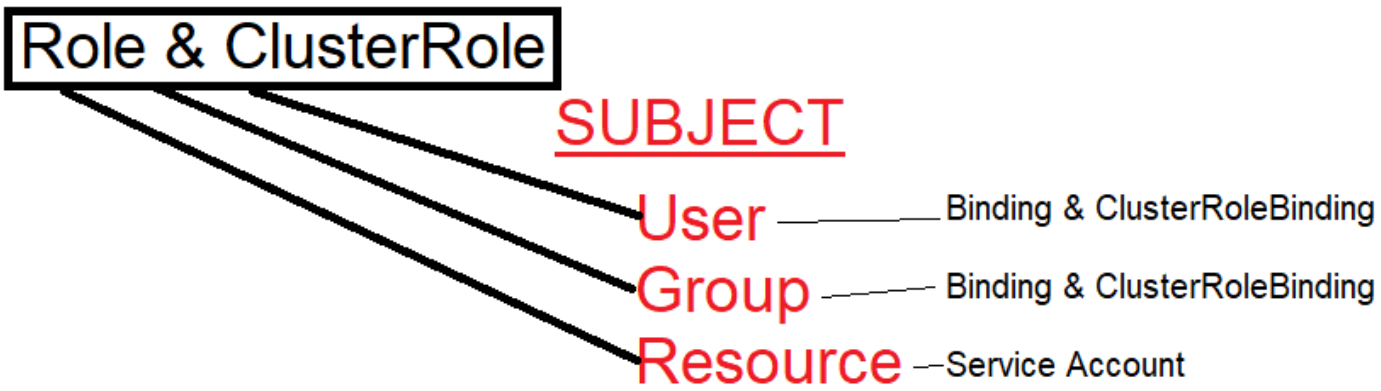
What Do You Mean by RBAC for Services (Service Accounts)?

- A **Service Account** is **not a user or group**, but an identity assigned to applications running inside Kubernetes.
- Just like users, a **Service Account must first authenticate** and then be **authorized** to perform specific actions.
- Service Accounts are assigned **Roles or ClusterRoles** to control their access to cluster resources.

Role vs. ClusterRole in RBAC

- **Role** → Provides **namespace-based** permissions (access limited to a specific namespace).
- **ClusterRole** → Provides **cluster-wide** permissions (access across all namespaces).

Roles and ClusterRoles are actually the permissions that you set for the User, Group, or Service Account to define what they can do in Kubernetes.



A Complete Example: (Simple IT Service Example to Understand RBAC)

Imagine a company that manages **web applications** for its clients. They use **Kubernetes** to run their applications, and different teams need different access levels.

Teams & Their Access

1. **Support Team (Basic Access)**

- Can view logs and check if apps are running.
- Cannot change or delete anything.

2. **Developers (Limited Access)**
   - Can deploy and update their applications.
   - Cannot touch other teams' apps or delete resources.
3. **DevOps Team (Full Access)**
   - Can manage everything – apps, networking, storage, and configurations.
4. **Security Team (Audit Access)**
   - Can only view logs and check security.
   - Cannot make any changes.

RBAC is enabled by default in Kubernetes and is managed via the rbac.authorization.k8s.io API group.

```
root@control:~# kubectl api-versions | grep rbac.authorization.k8s.io
rbac.authorization.k8s.io/v1
```

How RBAC Works in Kubernetes

RBAC works by assigning permissions using **Roles** and then linking them to users or groups through **RoleBindings**.

Enabling RBAC

To ensure RBAC is enabled, the Kubernetes API server should have the following flag:

kube-apiserver --authorization-mode=...,RBAC

```
root@control:~# kubectl get pod -n kube-system -l component=kube-apiserver -o yaml | grep authorization-mode
    - --authorization-mode=Node,RBAC
```

In short – (What is Role, ClusterRole, RoleBinding, and ClusterRoleBinding?)

Role & ClusterRole (Defining Permissions)

- **Role** → Gives **permissions inside a specific namespace** (limited access).
- **ClusterRole** → Gives **permissions across the entire cluster** (global access).
- These define **what actions a user, group, or service account can perform**.

RoleBinding & ClusterRoleBinding (Assigning Permissions)

- **RoleBinding** → Assigns a **Role** to a user, group, or service account **inside a namespace**.
- **ClusterRoleBinding** → Assigns a **ClusterRole** to a user, group, or service account **for the entire cluster**.

- These define **who gets the permissions** from Role or ClusterRole.

In Depth- (Understanding Role and ClusterRole)

Role (Namespace-Specific Access)

A **Role** defines what actions can be performed on resources within a single namespace. That can assign later to a user and group using role-binding.

Example: A Role that allows reading pods within the default namespace:

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

This Role allows users to **get, watch, and list** pods but not modify or delete them.

ClusterRole (Cluster-Wide Access)

A **ClusterRole** defines permissions at the cluster level and can be used across namespaces.

Example: A ClusterRole that allows reading secrets across the cluster:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

This means anyone assigned this role can view secrets **in all namespaces**.

RoleBinding vs. ClusterRoleBinding

RoleBinding (Namespace-Specific Role Assignment)

A **RoleBinding** assigns a Role to a user or group **within a specific namespace**.

Example: Assigning the pod-reader Role to a user named jane:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

Here, jane can **view pods** in the default namespace but cannot modify them.

ClusterRoleBinding (Cluster-Wide Role Assignment)

A **ClusterRoleBinding** assigns a ClusterRole to users across **all namespaces**.

Example: Assigning secret-reader ClusterRole to the manager group:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group
  name: manager
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

Now, the manager group can read secrets across all namespaces.

Managing RBAC with kubectl

Creating Roles and ClusterRoles

kubectl create role pod-reader --verb=get --verb=list --verb=watch --resource=pods --namespace=default

kubectl create clusterrole secret-reader --verb=get,list,watch --resource=secrets

Creating RoleBindings and ClusterRoleBindings

kubectl create rolebinding read-pods --role=pod-reader --user=jane --namespace=default

kubectl create clusterrolebinding read-secrets-global --clusterrole=secret-reader --group=manager

Viewing RBAC Resources

kubectl get roles --namespace=default

kubectl get clusterroles

kubectl get rolebindings --namespace=default

kubectl get clusterrolebindings

Special Kubernetes Roles

Role	Description
cluster-admin	Full access to the entire cluster.
admin	Read-write access within a namespace.
edit	Can modify most resources but cannot edit roles.
view	Read-only access to most resources.

Best Practices for RBAC

- **Follow the principle of least privilege** – Grant only the necessary permissions.
- **Use RoleBindings for namespace-specific permissions** instead of ClusterRoleBindings.
- **Use ClusterRoles only when absolutely necessary** to avoid excessive access.
- **Avoid using wildcards (*)** as they grant too many permissions.
- **Regularly review and update RBAC policies** to keep access secure.

RBAC Projects-1 (**Real-World Project Scenario: Implementing RBAC in an E-commerce Platform)**

Scenario: Imagine you are working as a **DevOps Engineer** for an **e-commerce company** that runs its platform on **Kubernetes**. The company has different teams managing various aspects of the

platform, and you need to ensure that each team has the right access to the Kubernetes cluster **without compromising security**.

Problem Statement: Currently, **all engineers have full access** to the Kubernetes cluster, which creates **security risks** and the potential for **accidental misconfigurations**. The company wants to **restrict access** based on job roles, ensuring that each team can only access the resources they need.

Teams & Required Access Levels

1. **Developers**
   - Can **deploy applications** but cannot delete resources.
   - Can read logs of their applications.
2. **QA Team (Quality Assurance)**
   - Can **test applications** by accessing running services.
   - Can check logs but **cannot modify or delete applications**.
3. **DevOps Team**
   - Full control over the **Kubernetes cluster** to manage deployments, networking, and storage.
4. **Security Team**
   - Read-only access to check compliance and security issues.

RBAC Implementation in Kubernetes

Step 1: Create Roles for Each Team

1. Developer Role (developer-role)

This role allows developers to **create, update, and view deployments**, but **not delete them**.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: ecommerce-app
  name: developer-role
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps"]
  verbs: ["get", "list", "watch", "create", "update"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update"]
```

---

2. QA Role (qa-role)

This role gives QA engineers **read-only access** to deployments and logs.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: ecommerce-app
  name: qa-role
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch"]
```

---

3. DevOps ClusterRole (devops-clusterrole)

The DevOps team needs **full control over the entire cluster**, so we use a **ClusterRole** instead of a Role.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: devops-clusterrole
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]  # Full access
```

---

4. Security Team Role (security-role)

The security team needs **read-only access** to everything for auditing.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: ecommerce-app
  name: security-role
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["get", "list", "watch"]
```

Step 2: Assigning Roles Using RoleBindings and ClusterRoleBindings

1. Assign developer-role to Developers

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: developer-rolebinding
  namespace: ecommerce-app
subjects:
```

```
- kind: User
  name: alice  # Example developer user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: developer-role
  apiGroup: rbac.authorization.k8s.io
```

---

2. Assign qa-role to QA Engineers

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: qa-rolebinding
  namespace: ecommerce-app
subjects:
- kind: Group
  name: qa-team  # Assigning to a group of QA engineers
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: qa-role
  apiGroup: rbac.authorization.k8s.io
```

---

3. Assign devops-clusterrole to the DevOps Team

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: devops-clusterrolebinding
subjects:
- kind: Group
  name: devops-team
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: devops-clusterrole
  apiGroup: rbac.authorization.k8s.io
```

---

4. Assign security-role to Security Team

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: security-rolebinding
  namespace: ecommerce-app
subjects:
- kind: Group
  name: security-team
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: security-role
  apiGroup: rbac.authorization.k8s.io
```

Step 3: Testing RBAC Permissions

After setting up RBAC, you can **test user permissions** using kubectl auth can-i commands.

Check if a Developer can create a deployment

kubectl auth can-i create deployments --namespace=ecommerce-app --as=alice

(Should return "**yes**")

Check if a QA Engineer can delete a pod

kubectl auth can-i delete pods --namespace=ecommerce-app --as=qa-user

(Should return "**no**")

Check if a Security Engineer can list all pods

kubectl auth can-i list pods --namespace=ecommerce-app --as=security-user

(Should return "**yes**")

Follow this channel for more: **https://www.linkedin.com/in/rakeshkumarjangid/**