BY FENIL GAJJAR

KUBERNETES DAILY TASKS

RBAC

(ROLE BASED ACCESS CONTROL)

COMPLETE RBAC IN KUBERNETES DOCUMENTATION

- COMPREHENSIVE GUIDE
- THEORY + PRACTICAL TASKS
- REAL TIME SCENARIO TASKS
- END TO END DOC

FOLLOW FOR MORE TASKS

CONTACT US

- fenilgajjar.devops@gmail.com
- linkedin.com/in/fenil-gajjar
- github.com/Fenil-Gajjar

🔐 Mastering

Kubernetes RBAC:

Secure Your Cluster the

Right Way! 🚀

👋 Welcome to the Ultimate Guide on Kubernetes RBAC!

Hey everyone!
First of all, a **BIG THANK YOU** for joining me on this exciting journey of learning Kubernetes and DevOps. 🙌
Your continuous support, kind words, and engagement truly keep me motivated to dive deeper and share more every day. 💙

In this guide, we're going to explore one of the most crucial aspects of Kubernetes security — **RBAC (Role-Based Access Control)**.
Whether you're a beginner trying to understand how access control works in K8s or a DevOps enthusiast building secure clusters for production — this doc is **designed for you**! 🚀

🔐 What This Doc Covers

Here's what you can expect to learn:

✅ What RBAC is and why it's essential in Kubernetes
✅ Key components: Roles, RoleBindings, ClusterRoles, ClusterRoleBindings
✅ Different kinds of subjects: Users, Groups, ServiceAccounts
✅ Real-time scenarios and use cases with complete setups
✅ How to debug and audit RBAC issues effectively
✅ Practical tips and best practices for secure access management

I've kept the content friendly, detailed, and hands-on — so you can **understand RBAC deeply and apply it confidently** in your clusters.

📣 Let's Learn & Grow Together!

I'm incredibly grateful for all the support you've shown throughout my Kubernetes series. Your encouragement helps me stay consistent and keeps this momentum going strong! 🙏

So let's continue this learning journey together — step by step, one concept at a time.

Stay tuned, stay curious, and let's make Kubernetes simpler for everyone. 💪

Cheers,

~ Fenil Gajjar

> "Security is not a feature — it's a foundation. Let's build it right. 🔐🚀"

🔐 What is RBAC in Kubernetes?

RBAC (Role-Based Access Control) is a way to **control who can access and perform actions** on resources inside a Kubernetes cluster.

It helps you define:

- **Who** (user, group, or service account)
- **Can do what** (actions like get, create, delete, etc.)
- **On which resources** (like pods, deployments, secrets)
- **In which namespace** (or cluster-wide)

In simple terms, it helps you answer these questions:

- Who can **access** the cluster?
- What **actions** can they perform? (like create, read, update, delete)
- On which **resources**? (Pods, Deployments, Secrets, etc.)
- In which **namespace** or **scope**?

🌟 Why is RBAC Important?

Kubernetes clusters often host **multiple applications, teams, and environments**. Without proper access control:

- A user could accidentally delete resources they shouldn't touch.
- A compromised pod might access secrets from another namespace.
- Debugging access issues becomes chaotic.

RBAC helps solve all that by enforcing the **principle of least privilege**—only granting users and services the exact permissions they need, and nothing more.

🔙 What Was Used Before RBAC in Kubernetes?

Before Kubernetes adopted the widely used **RBAC (Role-Based Access Control)** model, access control was managed using a more primitive method called **ABAC (Attribute-Based Access Control)**.

While ABAC did the job in early Kubernetes days, it lacked the flexibility, scalability, and native integration needed for managing modern Kubernetes clusters securely.

📌 What is ABAC?

ABAC is a permission model where access is granted based on a set of user-defined policies using attributes such as:

- **User Identity**
- **Resource Type**
- **Namespace**
- **Action (Verb)**

These rules are written in **JSON** files and loaded into the Kubernetes API server at startup.

🔧 **Example ABAC Policy**:

```
{
  "user": "developer",
  "namespace": "dev",
  "resource": "pods",
  "verb": "create"
}
```

This allows the `developer` user to `create pods` only in the `dev` namespace.

⚠️ **Why ABAC Wasn't Ideal:**

Although ABAC was simple to get started with, it introduced several limitations that made it unfit for production-level systems:

- 🚫 **Not Dynamic**: Any change to ABAC policies required restarting the Kubernetes API server — which is disruptive.
- 📄 **Difficult to Manage at Scale**: Large static JSON policy files became hard to maintain and audit.
- 🧱 **No Role Concept**: ABAC didn't support reusable roles or grouping permissions — everything was user-specific.

- 🧩 **No Native Kubernetes Objects**: Unlike RBAC, it didn't use Kubernetes-native resources like `Roles` and `RoleBindings`.

- 🛡️ **Weaker Security Model**: Fine-grained access control was difficult to achieve.

🔁 The Shift to RBAC

As Kubernetes evolved, it needed a **more secure, scalable, and Kubernetes-native way to manage permissions**. That's when **RBAC** was introduced and became **generally available in Kubernetes v1.6**.

From that point onward:

- RBAC became the **default and recommended** authorization mechanism.

- ABAC is still available but is **discouraged** and rarely used — especially in production.

🚨 Why Do We Need RBAC in Kubernetes?

In any production-grade Kubernetes environment, **security and controlled access** are critical. As your cluster grows with more users, services, and teams working together — it becomes essential to control **who can do what** inside the cluster.

That's where **RBAC (Role-Based Access Control)** steps in. It helps you **secure, manage, and organize access** to Kubernetes resources in a scalable, maintainable way.

🔒 Here's Why RBAC is So Important:

1️⃣ Fine-Grained Access Control

RBAC lets you define **precise permissions**. Instead of giving users full admin access, you can allow:

- Only reading pods
- Only creating deployments
- Only viewing logs

  This reduces risk and enforces **least privilege access**.

2️⃣ Team-Based Role Assignment

With RBAC, you can assign **Roles to groups or teams** (like developers, DevOps, testers) rather than individual users.

- Developers ➜ Can deploy apps in the dev namespace

- DevOps ➜ Can manage nodes and networking

- QA ➜ Can read logs and run test jobs

  Makes management easier and avoids permission chaos.

3️⃣ Namespace-Level Isolation

Kubernetes is multi-tenant by nature. RBAC lets you **restrict actions within namespaces**, so:

- Devs can't touch production resources

- Testers can't modify deployments

  Perfect for separating teams, stages (dev/test/prod), or applications.

4️⃣ Improves Security & Compliance

RBAC enforces **who can access what**, which is crucial for:

- Auditing and logging activity

- Meeting compliance standards (ISO, HIPAA, SOC2, etc.)

- Reducing the blast radius of misconfigurations or attacks

5️⃣ Kubernetes-Native & Dynamic

RBAC is built into Kubernetes:

- You can **modify roles and bindings on the fly** — no need to restart anything
- It's declarative — you manage it like any other YAML resource
- Works seamlessly with Kubernetes tools, CLIs, and APIs

🚀 How RBAC Works in Kubernetes

RBAC (Role-Based Access Control) in Kubernetes allows you to control access to resources within the cluster by defining **Roles** and **RoleBindings** (or **ClusterRoles** and **ClusterRoleBindings** for cluster-wide access).

1️⃣ Roles & ClusterRoles

At the heart of RBAC are the **Roles** and **ClusterRoles**. These define the **permissions** (what actions can be performed) for a specific set of resources.

- **Role**: This defines permissions **within a namespace**. For example, you might want to allow users to **create and read pods** within a specific namespace but not in the entire cluster.

- **ClusterRole**: A ClusterRole is similar to a Role but can apply **across the entire cluster**, including **all namespaces**. It's used for cluster-wide resources like nodes or persistent volumes.

2️⃣ RoleBindings & ClusterRoleBindings

Once you have **Roles** or **ClusterRoles**, you then need to bind them to users, groups, or service accounts using **RoleBindings** and **ClusterRoleBindings**. These bindings grant the roles to a specific **user** or **service account**, giving them the defined permissions.

- **RoleBinding**: A RoleBinding grants the **permissions** of a **Role** within a specific namespace to a user or set of users.

- **ClusterRoleBinding**: A ClusterRoleBinding grants the **permissions** of a **ClusterRole** across the **entire cluster**.

This way, you can create flexible access control policies, whether you want to restrict actions to a particular namespace or give global access to cluster-wide resources.

3️⃣ Kubernetes Resources & Permissions

Roles define the types of actions that can be performed on **Kubernetes resources**. These resources could be things like **pods**, **deployments**, **services**, **configmaps**, etc. Permissions typically fall into **verbs** like:

- `get` (view)
- `list` (list multiple resources)
- `create` (create resources)
- `update` (update resources)
- `delete` (remove resources)

For example, a Role may allow a user to `get` and `list` pods but not `create` or `delete` them.

4️⃣ Service Accounts & RBAC

In addition to users, **service accounts** can be assigned roles. Service accounts are used by **pods** to access the Kubernetes API. This allows you to control and restrict **what the pod can do**. For example, you might have a pod that only needs to read from a specific set of resources and not modify them.

5️⃣ RBAC Workflow: Step-by-Step

Here's how RBAC works in Kubernetes:

1. **Define Roles (or ClusterRoles)**: Specify **what actions** can be performed on **which resources**.
    - Example: A Role allowing `create`, `list`, and `get` permissions on **pods** in a specific namespace.
2. **Create RoleBindings (or ClusterRoleBindings)**: Bind the defined **Roles** or **ClusterRoles** to specific **users**, **groups**, or **service accounts**.
    - Example: A RoleBinding assigns the Role created in step 1 to a developer, allowing them to create and manage pods in the development namespace.
3. **Access Control**: Now, when the user or service account makes requests to Kubernetes, RBAC determines whether they are allowed to perform the requested actions based on the **Roles** and **Bindings**.

6️⃣ RBAC Example in Action

Let's break down a simple example to understand how everything ties together:

Step 1: You define a **Role** (named `dev-namespace-admin`) that allows users to `create`, `get`, `list`, and `delete` pods in the `dev` namespace:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: dev
  name: dev-namespace-admin
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["create", "get", "list", "delete"]
```

Step 2: You then create a **RoleBinding** that binds this role to a specific user (`developer-user`) within the `dev` namespace:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
```

```
  name: dev-namespace-admin-binding
  namespace: dev
subjects:
- kind: User
  name: developer-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: dev-namespace-admin
  apiGroup: rbac.authorization.k8s.io
```

- **Step 3**: The `developer-user` is now able to perform the specified actions (create, get, list, delete) on pods in the `dev` namespace. If they attempt to access resources outside this namespace or try other actions (like `update`), they will be denied.

🚀 Key Components of RBAC in Kubernetes 🔐

RBAC (Role-Based Access Control) in Kubernetes provides a **powerful and flexible** way to manage access control within your cluster. It allows you to define **who can access what**, and **what actions** they can perform on specific resources. Let's explore the essential components that make up RBAC!

🛠️ 1. Role

A **Role** defines a set of permissions (like `get`, `create`, `list`, `delete`) for resources within a **specific namespace**. It is ideal for managing access to resources like **Pods**, **Deployments**, **ConfigMaps**, and more within that namespace.

Example Use Case:

- A **Role** could allow a user to `list` and `get` Pods in the `dev` namespace but **not** allow them to create or delete Pods.

🌍 2. ClusterRole

A **ClusterRole** is like a **Role**, but with permissions that apply **cluster-wide** (not just within a namespace). ClusterRoles are useful for managing access to resources that span across the entire Kubernetes cluster, such as **Nodes** or **PersistentVolumes**.

Example Use Case:

- A **ClusterRole** could allow a user to manage **all nodes** in the cluster or view **metrics** across multiple namespaces.

🔗 3. RoleBinding

A **RoleBinding** connects a **Role** to a specific **user**, **group**, or **service account** within a **namespace**. It ensures that the permissions granted by the Role apply only within that namespace.

Example Use Case:

- A **RoleBinding** can assign the `developer-role` to the `developer-user` in the `dev` namespace, enabling them to perform actions like `get` and `list` Pods.

🌐 4. ClusterRoleBinding

A **ClusterRoleBinding** connects a **ClusterRole** to a **user**, **group**, or **service account**, but applies across the **entire cluster**. This gives the subject access to resources and actions across all namespaces and cluster-wide resources.

Example Use Case:

- A **ClusterRoleBinding** can assign the `admin-cluster-role` to a `superuser`, granting full access to resources like **Nodes** and **Persistent Volumes**.

👫 5. Subjects

Subjects are the **users**, **groups**, or **service accounts** that are granted permissions through **RoleBindings** or **ClusterRoleBindings**. These are the entities that will have access to perform actions defined in the **Role** or **ClusterRole**.

Example Use Case:

- A subject could be a specific user (`developer-user`), a group (`dev-team`), or a service account (`dev-service-account`).

📌 6. RoleRef

The **RoleRef** in a **RoleBinding** or **ClusterRoleBinding** references the **Role** or **ClusterRole** that defines the permissions being granted. It's the link between the subject and the set of permissions they will have access to.

Example Use Case:

- A **RoleBinding** could reference a `developer-role` under `roleRef`, ensuring the assigned user has the correct permissions.

⚙️ 7. Verbs

Verbs define the **actions** that can be performed on resources. These actions are what you specify when creating a **Role** or **ClusterRole** to control what operations a subject can perform on specific resources.

Common Verbs include:

- `get`: Retrieve resource information

- `list`: List multiple resources
- `create`: Create a new resource
- `update`: Modify an existing resource
- `delete`: Remove a resource
- `patch`: Partially update a resource
- `watch`: Monitor resources for changes

🗂️ 8. Resources

Resources refer to the **types of objects** in Kubernetes that RBAC manages access to. Some examples of resources are:

- **Pods**
- **Deployments**
- **Services**
- **ConfigMaps**
- **Secrets**
- **Nodes** (for cluster-wide access)

These resources can have specific permissions associated with them, enabling fine-grained access control.

🔐 Deep Dive into Roles and RoleBindings in Kubernetes 🚀

In Kubernetes, **Role** and **RoleBinding** are core components of **Role-Based Access Control (RBAC)** that manage who can access what resources within a Kubernetes cluster, and what actions they can perform on those resources. Understanding how to configure **Roles** and **RoleBindings** is essential for secure and efficient management of your Kubernetes environment.

🎯 What is a Role?

A **Role** defines a set of **permissions** (known as **verbs**) that can be applied to **resources** within a specific **namespace**. The role allows administrators to control what actions users and services can take on resources within that namespace. A Role is **namespace-specific**, meaning it is only effective within the namespace where it is created.

🛠️ Key Elements of a Role

1. **Verbs**: These define the actions a subject can perform on the resource. Common verbs include:
    - `get`: Retrieve a resource.
    - `list`: List multiple resources.
    - `create`: Create a resource.

- `delete`: Remove a resource.
- `update`: Modify a resource.
- `patch`: Apply a partial update to a resource.
- `watch`: Monitor resources for changes.

2. **Resources**: These are the objects or entities within the cluster that users can interact with. Examples include:
   - `pods`
   - `services`
   - `deployments`
   - `configmaps`
   - `secrets`
   - `nodes` (if used in a ClusterRole)
3. **API Groups**: Resources in Kubernetes belong to different API groups. For example, `apps` for deployments, `core` for pods, and so on. The Role defines permissions for resources in a specific API group.

4. **Namespaces**: Roles are always bound to a **specific namespace** (except for ClusterRoles, which apply to the entire cluster). This means that a Role only governs permissions for resources within that namespace.

🎯 What is a RoleBinding?

A **RoleBinding** is used to **assign a Role** to a **user**, **group**, or **service account**. It defines which users or services should have the permissions defined in the **Role**. When a **RoleBinding** is created, the permissions specified in the Role are applied to the subject (user/group/service account).

A **RoleBinding** works within a **specific namespace**, meaning it is bound to the namespace where it is created. It ensures that the subject has access to the resources defined in the Role within that namespace.

🛠️ Key Elements of a RoleBinding

1. **RoleRef**:
    - This references the **Role** or **ClusterRole** that defines the set of permissions being granted. The **RoleRef** essentially links the Role to the user or service account being granted access.
2. **Subjects**:
    - The **subjects** are the entities (users, groups, or service accounts) to which the Role is granted. This is where the permissions from the

Role are applied.

🔑 How Role and RoleBinding Work Together

1. **Create the Role**:
    - First, a **Role** is created in a specific namespace, detailing the permissions (verbs) and resources (like pods, services) that a user or service can interact with.
2. **Bind the Role to Users**:
    - After the Role is created, a **RoleBinding** is created to bind that Role to a user or service account. The **RoleBinding** grants the user the permissions defined in the Role.
3. **Access Control**:
    - When a user or service account tries to access a resource in the namespace, the RoleBinding checks if they have the correct permissions. If the user has the necessary role-bound permissions, they are granted access; otherwise, access is denied.

🎓 Example of a Role and RoleBinding Configuration

Here's an example of creating a **Role** and **RoleBinding** to allow a user to **view** (but not modify) Pods in the **`dev`** namespace:

1️⃣ Role Definition:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  # Name of the Role
  name: pod-viewer
  # Specify the namespace where the role is applicable
  namespace: dev
rules:
  - verbs:
      - get
      - list
    resources:
      - pods
```

In this Role:

- The user can **get** and **list** Pods within the `dev` namespace.
- The Role doesn't allow the user to **create**, **delete**, or **update** Pods.

2️⃣ RoleBinding Definition:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  # Name of the RoleBinding
  name: pod-viewer-binding
  # Specify the namespace of the RoleBinding
  namespace: dev
subjects:
  - kind: User
    name: developer-user
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
```

```
  name: pod-viewer

  apiGroup: rbac.authorization.k8s.io
```

In this RoleBinding:

- The **`developer-user`** is granted the **`pod-viewer`** Role.
- The permissions to **`get`** and **`list`** Pods in the **`dev`** namespace are assigned to the user.

🌟 **Why Are Roles and RoleBindings Important?**

1. **Fine-Grained Access Control**:
    - Roles give you the flexibility to define precise permissions for users, ensuring that they only have access to the resources they need. By binding roles to specific namespaces, you ensure that users can only access the right resources.
2. **Least Privilege Principle**:
    - Roles and RoleBindings help enforce the **least privilege principle**, ensuring that users and service accounts only have access to the minimum resources necessary to do their jobs.

3. **Security**:

    - By using Roles and RoleBindings, you can limit the scope of access to Kubernetes resources, preventing unauthorized access or accidental misuse of sensitive resources like secrets and deployments.

4. **Namespace Isolation**:

    - Roles and RoleBindings support Kubernetes' **namespace isolation**, which helps segregate permissions across multiple environments, such as development, staging, and production.

⚡ Best Practices for Using Roles and RoleBindings

1. **Create Specific Roles for Specific Tasks**:

    - Avoid giving broad permissions (e.g., `admin`) unless absolutely necessary. Instead, create roles tailored to specific tasks.

2. **Use RoleBindings for Temporary Access**:

    - If a user only needs temporary access to certain resources, bind them to a role with the required permissions for a limited period.

3. **Avoid Overusing ClusterRoles in RoleBindings**:

- ClusterRoles are more general and apply across the entire cluster. Use them when needed, but try to restrict roles to specific namespaces whenever possible.

4. **Audit and Review Regularly**:

    - Regularly review the Roles and RoleBindings in your cluster to ensure they are still appropriate. Ensure no unnecessary permissions are granted, and revoke access when users no longer need it.

🚨 What Happens If a RoleBinding Refers to a Role in a Different Namespace? 🚨

In Kubernetes, **RoleBinding** is **namespace-specific** and can **only bind to Roles within the same namespace**. If you try to create a **RoleBinding** that references a **Role** in a different namespace, it will not work and will result in an **error**. This is because Kubernetes strictly enforces the namespace boundary for **Roles** and **RoleBindings**.

🔍 Why Is This the Case?

- **Roles** are **namespace-scoped**, meaning they define permissions for resources **only within the namespace** they are created in.

- **RoleBindings**, by design, must be **tied to the same namespace** as the **Role** they reference. The RoleBinding gives the **subject (user/service account)** the permissions granted by the **Role** within that same namespace.

Kubernetes enforces this rule to maintain clear **namespace isolation**, ensuring that resources and permissions are managed independently in each namespace.

💡 What Happens If You Do It Anyway?

If you attempt to create a **RoleBinding** that references a **Role** in a different namespace, Kubernetes will reject the request with an error like:

```
error: roles.rbac.authorization.k8s.io "role-name" is
forbidden: role "role-name" is in a different namespace
```

This error indicates that the Role and RoleBinding are not compatible because they exist in different namespaces.

✅ Correct Way to Handle Cross-Namespace Access:

If you need to **grant access to resources across multiple namespaces**, you can use **ClusterRoles** and **ClusterRoleBindings**.

1. **ClusterRole**: Unlike **Roles**, a **ClusterRole** is **cluster-wide** and can be used across multiple namespaces. It defines permissions for resources across the entire cluster, not bound to any particular namespace.

2. **ClusterRoleBinding**: If you want to grant a user access to resources across different namespaces (or even the entire cluster), you should use a **ClusterRoleBinding** instead of a RoleBinding.

🚀 Example of Correct Usage:

1️⃣ClusterRole Definition (Cluster-wide Permissions):

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: ClusterRole
metadata:
  name: pod-reader
rules:
  - verbs:
      - get
      - list
    resources:
      - pods
```

2️⃣ **ClusterRoleBinding Definition (Binding Across Namespaces):**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pod-reader-binding
subjects:
  - kind: User
    name: "developer-user"
    apiGroup: rbac.authorization.k8s.io
```

```
roleRef:
  kind: ClusterRole
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

In this example:

- The **ClusterRole** `pod-reader` grants permission to **view Pods** cluster-wide.

- The **ClusterRoleBinding** binds the **developer-user** to this ClusterRole, allowing them to view Pods across any namespace in the cluster.

🎯 Different Kinds of RoleBinding in Kubernetes 🎯

In Kubernetes, a **RoleBinding** is used to assign **roles** to **users**, **groups**, or **service accounts** within a specific **namespace**. RoleBinding essentially grants the permissions defined in a **Role** or **ClusterRole** to a set of subjects (users, groups, or service accounts).

While **RoleBinding** itself is quite specific to the namespace, there are a few different **subjects** and **role references** you can use to customize how you assign roles.

1️⃣RoleBinding with Role 🎭

This is the most common **RoleBinding** used in Kubernetes. It links a **Role** (which is namespace-scoped) to subjects such as users, service accounts, or groups.

Example:

Let's say you want to allow a **user** to list and get Pods in a specific namespace.

Role (within namespace `dev`):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: pod-reader
rules:
```

```
  - verbs:
      - get
      - list
    resources:
      - pods
```

RoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: dev
subjects:
  - kind: User
    name: "dev-user"
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
```

```
  apiGroup: rbac.authorization.k8s.io
```

In this example:

- The **Role** `pod-reader` in the `dev` namespace grants permission to get and list Pods.
- The **RoleBinding** binds this Role to the **user** `dev-user` in the `dev` namespace.

2️⃣RoleBinding with ClusterRole 🌍

In certain situations, you may want to grant **cluster-wide** permissions but within a specific namespace. For this, you can bind a **ClusterRole** to a subject using **RoleBinding**. This allows a user or service account to have permissions granted by a **ClusterRole** but confined to the namespace where the **RoleBinding** exists.

Example:

Let's say you want to allow a user to access Pods across the cluster, but in the **`dev`** namespace only. You can use a **ClusterRole** for cluster-wide permissions and bind it in a namespace.

ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
```

```
metadata:
  name: pod-reader-cluster
rules:
  - verbs:
      - get
      - list
    resources:
      - pods
```

RoleBinding (Namespace-specific):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: dev
subjects:
  - kind: User
    name: "dev-user"
    apiGroup: rbac.authorization.k8s.io
```

```
roleRef:
  kind: ClusterRole
  name: pod-reader-cluster
  apiGroup: rbac.authorization.k8s.io
```

In this case:

- The **ClusterRole** `pod-reader-cluster` grants permissions to get and list Pods across the entire cluster.
- The **RoleBinding** in the `dev` namespace grants `dev-user` the cluster-wide permissions within that namespace.

3️⃣ RoleBinding with ServiceAccount 👨‍💻

A **ServiceAccount** is often used in Kubernetes to allow applications to interact with the cluster. You can use a **RoleBinding** to assign roles to a **ServiceAccount**.

Example:

If you want to allow a **ServiceAccount** to get and list Pods in the `dev` namespace, you could do something like this:

Role (within namespace `dev`):

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: Role
metadata:
  namespace: dev
  name: pod-reader
rules:
  - verbs:
      - get
      - list
    resources:
      - pods
```

RoleBinding for ServiceAccount:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: dev
subjects:
```

```
  - kind: ServiceAccount
    name: "dev-service-account"
    namespace: dev
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

In this example:

- The **Role** grants permissions to get and list Pods in the `dev` namespace.
- The **RoleBinding** binds this **Role** to a **ServiceAccount** `dev-service-account` in the `dev` namespace.

4️⃣ RoleBinding with Group 👥

In Kubernetes, you can also bind a **Role** to a group of users. Groups are not directly created in Kubernetes, but they can be defined externally (e.g., through your identity provider or Kubernetes OIDC configuration).

Example:

If you want to allow a specific **group** of users to list Pods in the `dev` namespace:

Role (within namespace `dev`):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: dev
  name: pod-reader
rules:
  - verbs:
      - get
      - list
    resources:
      - pods
```

RoleBinding for Group:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: dev
```

```
subjects:
  - kind: Group
    name: "dev-group"
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

In this example:

- The **Role** grants permissions to get and list Pods in the `dev` namespace.
- The **RoleBinding** binds this **Role** to the **group** `dev-group`.

🔍 What is `kind: User` in Kubernetes RBAC?

When you define a subject in a `RoleBinding` or `ClusterRoleBinding`, you're specifying **who** should get the permissions. That `who` is called a **subject**, and it could be one of the following:

- `User`
- `Group`
- `ServiceAccount`

So in this example:

```
- kind: User
  name: "dev-user"
```

You're saying:
🧑‍💼 “I want to give access to a **user** named `dev-user`.”

💡 But what exactly is a "User" in Kubernetes?

Kubernetes **does not** manage user accounts by itself like traditional systems (e.g., Linux). Instead, **users are managed externally** – such as via:

- ✅ Certificates (for example, `kubectl` uses client certs to authenticate)
- ✅ Identity providers via OIDC (like Google, Azure AD, Okta)

- ✅ Authentication plugins
- ✅ Static username/password (for test setups, not recommended for production)

👤 **Examples of Kubernetes "User" identities:**

- A user authenticated with a **client certificate**:
  e.g. a `kubectl` user configured with `.kube/config` using certs.
- A user authenticated via **OIDC provider**:
  e.g. `fenil.gajjar@example.com` from your organization's SSO system.
- A static username configured in an **authentication webhook**.

🎯 Goal

- You have an IAM user (e.g., `arn:aws:iam::111122223333:user/dev-user`)
- You want to **grant them limited permissions** in EKS, such as:
  - Only read access to namespaces, pods, and deployments
- So you'll:

- Map the IAM user to a **custom Kubernetes group** in the `aws-auth` ConfigMap

- Then create a **Role** + **RoleBinding** (or ClusterRole + ClusterRoleBinding) in Kubernetes for that group

🔧 Step-by-Step Implementation

1️⃣ IAM User Mapping in `aws-auth` ConfigMap

Edit the `aws-auth` ConfigMap:

```
kubectl edit configmap aws-auth -n kube-system
```

Add the user under `mapUsers` like this:

```
mapUsers: |
  - userarn: arn:aws:iam::111122223333:user/dev-user
    username: dev-user
    groups:
      - eks-readonly-group
```

📝 Notes:

- We're assigning the IAM user to a custom group `eks-readonly-group`.
- This group name is arbitrary; just make sure it matches in your RBAC definition.

2️⃣ Create a Kubernetes `ClusterRole`

This defines what permissions the group has. Example: Read-only access to core resources.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: eks-readonly-role
rules:
  - apiGroups: [""]
    resources: ["pods", "services", "namespaces"]
    verbs: ["get", "list", "watch"]
  - apiGroups: ["apps"]
    resources: ["deployments", "replicasets"]
    verbs: ["get", "list", "watch"]
```

3️⃣ Bind the Role to the Group

Now bind the above ClusterRole to the group `eks-readonly-group`.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: eks-readonly-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: eks-readonly-role
subjects:
  - kind: Group
    name: eks-readonly-group
    apiGroup: rbac.authorization.k8s.io
```

✅ Now, any IAM user in that group (like `dev-user`) can only **view pods, deployments, namespaces, etc.**, and cannot create/update/delete anything.

✅ Test the IAM User

Log in as the IAM user, set credentials, then:

```
aws eks update-kubeconfig --region <region> --name
<cluster-name>

kubectl get pods -A

kubectl delete pod <pod-name>  # ❌ This should fail
```

🔐 Best Practice

Action	Recommended?
Give `system:masters`	❌ Avoid unless necessary
Use custom RBAC groups	✅ Yes, for least privilege
Use `ClusterRole` vs `Role`	`ClusterRole` for global access across namespaces

🧪 Real-world Scenario

Your DevOps team has multiple IAM users:

- `dev-user-1`, `dev-user-2`, `qa-user`

You assign all of them to `eks-readonly-group` in `aws-auth` and bind this group to a `ClusterRole` that provides read-only access.

This allows them to **monitor resources** without the risk of breaking things.

🧠 You don't create the group in Kubernetes directly.

In EKS, when you map an **IAM user or role** in `aws-auth` ConfigMap and assign a group (like `eks-readonly-group`), that group **doesn't need to exist** as a Kubernetes object. It's not like a Linux group or a K8s resource you create.

> 📌 The group is just a **label or string** that Kubernetes uses to match in RBAC bindings.

Let's explain this in detail 👇

🔍 Where Does `kind: Group` Come From?

In Kubernetes RBAC:

- A **Group** is a logical identity.
- It's not something you manually create.
- It's used to **match subjects** in `RoleBinding` or `ClusterRoleBinding`.

So, when you define this:

```
subjects:
  - kind: Group
    name: eks-readonly-group
```

Kubernetes checks:

> "Is the user (or role) calling me associated with this group?"

The answer depends on **how the IAM identity was mapped in `aws-auth`**.

🧩 Mapping in `aws-auth` Makes It Work

When you do this:

```
mapUsers: |
  - userarn: arn:aws:iam::111122223333:user/dev-user
    username: dev-user
    groups:
      - eks-readonly-group
```

You're telling Kubernetes:

> “When this IAM user calls `kubectl`, treat them as a user named `dev-user` and a member of the group `eks-readonly-group`.”

Now, if your RBAC bindings say “grant permissions to `eks-readonly-group`”, it matches and applies access.

🛠️ How This Works in Practice

Here's the full interaction:

1. IAM user `dev-user` runs `kubectl get pods`
2. `kubectl` signs the request with AWS credentials
3. EKS authenticates the request using IAM
4. It uses `aws-auth` to translate the IAM identity:
    - Maps to username: `dev-user`
    - Maps to group: `eks-readonly-group`
5. Kubernetes checks all RoleBindings/ClusterRoleBindings for `eks-readonly-group`
6. If a match is found, it allows/denies access accordingly

✅ No Need to Create the Group

You don't need to run `kubectl create group` or anything like that.

It's a virtual identity label.

Just make sure:

- The `group` name in `aws-auth` matches
- The `group` name in `RoleBinding`/`ClusterRoleBinding` matches

Then you're good!

🧪 Want to See This in Action?

Run this to see the user's identity and group used when making requests:

```
kubectl auth can-i get pods --as=dev-user
--as-group=eks-readonly-group
```

This simulates a request **as that user + group**, so you can verify your RBAC.

✅ What You Need to Configure for the IAM User

Let's say the IAM user is `dev-user`.

1️⃣ Ensure IAM User Has `eks:DescribeCluster` Permission

This is **required** to run `aws eks update-kubeconfig` — it fetches cluster endpoint & certificate.

🛡️ Add a policy to the IAM user like this:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "eks:DescribeCluster",
      "Resource": "*"
    }
  ]
}
```

You can also give broader EKS permissions depending on the role.

2️⃣ Install AWS CLI and Configure Credentials

On the IAM user's machine (or DevOps environment):

```
aws configure
```

✅ Enter:

- Access Key
- Secret Key
- Region
- Output format (e.g., `json`)

Or use environment variables:

```
export AWS_ACCESS_KEY_ID=xxxx
export AWS_SECRET_ACCESS_KEY=xxxx
export AWS_DEFAULT_REGION=us-east-1
```

3️⃣ Update kubeconfig for the EKS Cluster

This is a **must** so the IAM user can interact with the cluster using `kubectl`.

```
aws eks update-kubeconfig --region <region> --name
<cluster-name>
```

This command will:

- Add/update a context in `~/.kube/config`
- Set up the cluster endpoint, CA cert, and authentication

4️⃣ Test with kubectl

Once kubeconfig is updated, try this:

```
kubectl get pods --all-namespaces
```

✅ If the IAM user was added to a custom group like `eks-readonly-group`, and RBAC allows read access — they will succeed.

❌ If they try something not allowed (e.g., `kubectl delete pod`), RBAC will deny it.

✅ When Should You Configure the IAM User?

You're absolutely right:

> 🟢 **You only need to configure AWS credentials & run `aws eks update-kubeconfig` if the IAM user wants to interact with the EKS cluster using `kubectl`.**

🤔 What Happens If You Don't Configure the IAM User?

If you've done the following:

- ✅ Added IAM user to `aws-auth` ConfigMap
- ✅ Mapped to proper RBAC group
- ✅ Created appropriate `Role`/`RoleBinding` or `ClusterRoleBinding`

...but you **don't configure** `aws configure` or `update-kubeconfig` on their system — then:

> ❌ That IAM user won't be able to interact with the cluster — because they don't have any way to **authenticate** and **access the cluster endpoint**.

RBAC alone is not enough. RBAC is for **authorization**, not **authentication**.

🔍 What Happens When You Run `aws eks update-kubeconfig`?

This command **modifies (or creates)** the `~/.kube/config` file to allow your **IAM user** to interact with the EKS cluster.

✅ **It adds:**

1. A new **cluster entry**
2. A new **user entry**
3. A new **context** that links the two

🧠 Let's Look at a Real Example

After running the command, your kubeconfig may look like this:

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    server: https://<EKS-endpoint>
    certificate-authority-data: LS0tLS1...  # base64 encoded CA cert
```

```
  name: arn:aws:eks:us-east-1:111122223333:cluster/dev-cluster

users:
- name: arn:aws:eks:us-east-1:111122223333:cluster/dev-cluster
  user:
    exec:
      apiVersion: "client.authentication.k8s.io/v1beta1"
      command: "aws"
      args:
        - "eks"
        - "get-token"
        - "--cluster-name"
        - "dev-cluster"
      # This uses your current AWS IAM identity to fetch a temporary token

contexts:
- context:
```

```
    cluster:
arn:aws:eks:us-east-1:111122223333:cluster/dev-cluster
    user:
arn:aws:eks:us-east-1:111122223333:cluster/dev-cluster
  name:
arn:aws:eks:us-east-1:111122223333:cluster/dev-cluster


current-context:
arn:aws:eks:us-east-1:111122223333:cluster/dev-cluster
```

🧩 What Each Section Means

Section	Purpose
`clusters`	Contains cluster API endpoint and CA cert
`users`	Defines **how you authenticate** — in this case, using `aws eks get-token`
`contexts`	Combines a `cluster` and `user`
`current-context`	The default context used when you run `kubectl`

🔐 Authentication via `aws eks get-token`

- It uses your **IAM credentials** (`aws configure`) to generate a **temporary token**.
- The token is passed to the Kubernetes API server as a Bearer token.
- The cluster uses the `aws-auth` ConfigMap to map this IAM identity to a Kubernetes username/group.
- RBAC rules apply based on that mapping.

⚙️ So after configuring, what changes?

Before	After `update-kubeconfig`
No access to cluster	kubeconfig knows how to reach the cluster
No authentication	kubeconfig uses AWS CLI to generate token using IAM
No context	Now you can switch, use, and interact via `kubectl`

🧪 Bonus Tip: View the Config

To view your current kubeconfig:

```
kubectl config view --minify
```

To list all contexts:

```
kubectl config get-contexts
```

To switch:

```
kubectl config use-context <context-name>
```

🔄 Does `current-context` Change When I Run `aws eks update-kubeconfig`?

> **Yes**, by default it will **set the current context** to the new EKS cluster you're configuring.

So if you're using:

```
aws eks update-kubeconfig --region us-east-1 --name dev-cluster
```

It adds the new context and also sets:

```
current-context: arn:aws:eks:us-east-1:xxxxxxxxxxxx:cluster/dev-cluster
```

✅ So yes — if this IAM user is used to configure kubeconfig on a machine, the `current-context` becomes theirs by default.

🧠 Can kubeconfig hold multiple contexts?

> **Yes!** The kubeconfig file can hold **multiple clusters, users, and contexts**.

But...

⚠️ **Only one context can be "current" at a time**

The `current-context` is what `kubectl` uses when you don't specify `--context`.

You can view all contexts with:

```
kubectl config get-contexts
```

You can switch to another one using:

```
kubectl config use-context <context-name>
```

🔁 Example: Multiple Users or Clusters in One Kubeconfig

```
contexts:
- name: admin-context
  context:
    cluster: dev-cluster
    user: admin-user

- name: readonly-context
  context:
    cluster: dev-cluster
    user: readonly-user
```

```
current-context: admin-context
```

🧠 At any time, only one of these will be active (current).

🎯 So, in your scenario:

- You mapped IAM user to RBAC group
- You configured it with `aws eks update-kubeconfig`
- Now **this IAM user's context becomes current**
- But you can still switch back to your previous context if needed

🚀 Where IAM User/Role to EKS RBAC Mappings Are Used

Here are the most **realistic and practical scenarios** where this is implemented:

✅ 1. Secure Access Control for Teams

📌 Scenario:

You have multiple team members (Dev, QA, DevOps) who need access to the same EKS cluster, but with **different permissions**.

🎯 Solution:

- Map each **IAM user** or **IAM role** to a Kubernetes **group** via `aws-auth`
- Use **RBAC RoleBindings** to give scoped access

Example:

IAM User	Mapped Group	RBAC Role
dev_user	`eks-dev-group`	Can only `get`, `list`, `watch` pods in `dev` namespace
qa_user	`eks-qa-group`	Can only read ConfigMaps & secrets in `qa` namespace

admin_user	`eks-admins`	Full access (ClusterRoleBinding to `cluster-admin`)

✅ 2. Access Control for CI/CD Pipelines (IAM Roles)

📌 Scenario:

You're running **GitHub Actions**, **Jenkins**, or **GitLab CI** — and need your pipeline to deploy apps to EKS securely.

🎯 Solution:

- Use **IAM Role** for your CI/CD pipeline with `sts:AssumeRole` or **OIDC**
- Map that **IAM Role** in `aws-auth`
- Bind it to an RBAC `ClusterRole` with limited permissions (e.g., only in `prod` namespace)

✅ 3. Granular Namespace Isolation in Multi-Tenant EKS

📌 Scenario:

You run a **multi-tenant platform** (e.g., SaaS), where each customer has a namespace in the same EKS cluster.

🎯 Solution:

- Create a unique **IAM Role/User** per tenant
- Map them in `aws-auth` and to Kubernetes group
- Apply strict RBAC bindings to their namespace only
- Prevent cross-namespace access

✅ 4. Auditing and Least Privilege Enforcement

📌 Scenario:

Your security team wants **accountability and traceability** — they want to know *who did what* in the EKS cluster.

🎯 Solution:

- Use **individual IAM users** or **per-team IAM roles**
- Map those identities to unique Kubernetes users/groups
- Use CloudTrail for IAM actions + EKS audit logs for RBAC

✅ 5. Federated Access (SSO Integration)

📌 Scenario:

Your organization uses **AWS SSO**, **Okta**, or **Azure AD**, and developers log in via federated identities.

🎯 **Solution:**

- Federated login gives IAM Role via SAML/OIDC
- That IAM Role is mapped to EKS groups via `aws-auth`
- RBAC defines what they can do once logged in

✅ 6. External Dev Access in Dev/Test Clusters

📌 **Scenario:**

You give external consultants or interns temporary access to your dev EKS cluster.

🎯 **Solution:**

- Create IAM user with limited access
- Map to a low-privilege RBAC role (`view-only`)
- Rotate/remove when no longer needed

🆚 EKS vs Kops — Authentication & Access Control Differences

Feature	EKS	Kops
Authentication	IAM-based + `aws-auth` config	Mostly kubeconfig with **user/pass or token**
RBAC	Based on IAM ➜ Group ➜ RoleBinding	Based on Kubernetes-native **ServiceAccount**, user certs, or tokens
Best for	AWS-native integrations, federated access, CI/CD IAM roles	Custom Kubernetes clusters, self-managed auth, more flexibility

✅ In Kops — You Typically Use:

1. ServiceAccount + RBAC

This is best for **apps/pipelines/pods** needing in-cluster access.

- Create a ServiceAccount
- Bind it via Role or ClusterRoleBinding
- Generate token manually or via automation

- Use it in `kubeconfig` file

🔁 Common for **CI/CD, monitoring agents, etc.**

2. User Certificate Auth (for humans)

If a **human user** wants access:

- Generate client cert + private key
- Create a kubeconfig using that cert/key
- Use RBAC to bind to roles

🔐 Full control but **less dynamic** than IAM

3. Token Auth (Static or Dynamic)

- You can generate long-lived tokens for users manually
- Or use **OIDC/dex** for dynamic auth

“This IAM user/role mapping is useful in EKS, but in Kops, for scenarios like developer or pipeline access, I should use ServiceAccount + RBAC + token + kubeconfig.”

🔰 What is a ClusterRole in Kubernetes?

A **ClusterRole** is like a **Role**, but it is not limited to a single namespace.

It can grant permissions:

- **Across the entire cluster**
- Or for **non-namespaced resources** like nodes, persistent volumes, etc.

📌 Use `ClusterRole` when:

- You need to give access **across multiple namespaces**
- You're dealing with **cluster-wide resources**

🧱 What is a ClusterRoleBinding?

A **ClusterRoleBinding** connects:

- A `ClusterRole` ➜ to ➜ a **user**, **group**, or **service account**

It tells Kubernetes:

➡️ "This identity should have these cluster-wide permissions."

✅ **Example: Grant Cluster-Wide Read Access to All Resources**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: read-only
rules:
  - apiGroups: [""]
    resources: ["pods", "services", "configmaps"]
    verbs: ["get", "list", "watch"]
  - apiGroups: ["apps"]
    resources: ["deployments", "statefulsets"]
    verbs: ["get", "list", "watch"]
```

> This `ClusterRole` gives **read-only access** to core and apps resources.

🔗 **ClusterRoleBinding Example (for a User)**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
  name: read-access-binding
subjects:
  - kind: User
    name: dev-user
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: read-only
  apiGroup: rbac.authorization.k8s.io
```

This binds the `read-only` ClusterRole to a user named `dev-user`.

🧪 **Real-World Example: Give EKS IAM Group Read Access**

If you mapped a group in `aws-auth` like this:

```
groups:
  - eks:readonly
```

You can bind it like this:

```
subjects:
  - kind: Group
    name: eks:readonly
    apiGroup: rbac.authorization.k8s.io
```

🎭 Different Kinds of `subjects.kind` in RBAC:

Here are the valid values for `kind`:

1. User 👤

Represents an individual **user** identity.
In cloud setups like EKS, this often maps to an **IAM user**.

```
- kind: User
  name: "dev-user"
```

📌 Example use case: Giving a specific developer read access to resources.

2. Group 👥

Represents a **collection of users**.

If a user belongs to a group, they inherit the group's permissions.

```
- kind: Group
  name: "developers"
```

📌 Example use case: Granting access to all users in a group like `system:masters` or an IAM group in EKS.

3. ServiceAccount 🤖

Represents a **Kubernetes service account**, typically used by pods to interact with the API server.

```
- kind: ServiceAccount
  name: app-sa
  namespace: my-app
```

📌 Example use case: Allowing an app pod to read ConfigMaps or Secrets.

🔧 Scenario:

You have a user named `dev-user` (maybe an IAM user or a certificate-based user) who needs **read-only access** to Pods across **all namespaces**. This user should not be able to modify or delete anything — just **view** Pods.

1️⃣ Create a ClusterRole with read-only access to Pods

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

📝 This ClusterRole gives permission to view all Pods in any namespace.

2️⃣ **Create a ClusterRoleBinding to bind the ClusterRole to the user**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: read-pods-binding
subjects:
  - kind: User
    name: dev-user          # Must match the user identity
in the cluster auth system
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: read-pods
  apiGroup: rbac.authorization.k8s.io
```

⚠️ Note: The `name: dev-user` should match the user authenticated via certificate, IAM (in EKS), or your auth provider.

💡 Optional Namespace-Specific Setup (Using Role & RoleBinding)

If instead, you want to give the same read access **only in a specific namespace** (say `dev`), use a **Role** and **RoleBinding** instead:

Role (namespace-specific):

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

RoleBinding (namespace-specific):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
```

```
metadata:
  namespace: dev
  name: read-pods-binding
subjects:
  - kind: User
    name: dev-user
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: read-pods
  apiGroup: rbac.authorization.k8s.io
```

🔧 Scenario:

You have a group of developers (`dev-team`) who should have **read-only access** to all resources in the `dev` namespace. Instead of assigning roles user by user, you want to assign permissions to the **entire group**.

🧩 Step-by-step Setup Using Group-Based RBAC

🔹 1. Create a `Role` (namespace-scoped)

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: dev-team-read-access
  namespace: dev
rules:
  - apiGroups: [""]
    resources: ["pods", "services", "configmaps"]
    verbs: ["get", "list", "watch"]
```

🎯 This role gives `get`, `list`, and `watch` access to Pods, Services, and ConfigMaps in the `dev` namespace.

🔹 **2. Create a `RoleBinding` for the Group**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: dev-team-rolebinding
  namespace: dev
subjects:
  - kind: Group
    name: dev-team                    # Group name from IAM/OIDC/LDAP etc.
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: dev-team-read-access
  apiGroup: rbac.authorization.k8s.io
```

📌 This binds the `dev-team` group to the Role we just created in the `dev` namespace.

🏗️ Want to grant cluster-wide access instead?

Let's say the **same `dev-team` group** should be able to view **Nodes and PersistentVolumes** cluster-wide. For that, use `ClusterRole` and `ClusterRoleBinding`.

🔹 **3. Create a `ClusterRole` (cluster-scoped)**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: dev-team-cluster-read
rules:
  - apiGroups: [""]
    resources: ["nodes", "persistentvolumes"]
    verbs: ["get", "list", "watch"]
```

🔹 **4. Create a `ClusterRoleBinding` for the Group**

```
apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

  name: dev-team-clusterrolebinding

subjects:

  - kind: Group

    name: dev-team                        # Must match your
auth provider's group

    apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: ClusterRole

  name: dev-team-cluster-read

  apiGroup: rbac.authorization.k8s.io
```

🔧 Scenario:

You have a custom application running in a Pod that needs to **list and get ConfigMaps and Secrets** from the namespace it's deployed in. You don't want to give it too many permissions, just exactly what it needs. So, you'll create a dedicated **ServiceAccount** and use RBAC to allow access only to those resources.

🎯 Goal:

- Pod uses a specific **ServiceAccount**.
- That ServiceAccount has permissions to:
    - `get`, `list`, and `watch` ConfigMaps and Secrets.
- Scoped only to the **`app-namespace`**.

🧩 Step-by-Step Setup

🔹 1. Create the `ServiceAccount`

```
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: app-reader-sa
  namespace: app-namespace
```

🔐 This SA will be used by your application Pod.

🔹 **2. Create the `Role` (namespace-scoped)**

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

🗒️ This defines what the ServiceAccount is allowed to do in the namespace.

🔹 **3. Bind the Role to the ServiceAccount**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: bind-sa-to-role
  namespace: app-namespace
subjects:
  - kind: ServiceAccount
    name: app-reader-sa
    namespace: app-namespace
roleRef:
  kind: Role
  name: read-secrets-configmaps
  apiGroup: rbac.authorization.k8s.io
```

🔁 This links the permissions to the ServiceAccount.

🔹 **4. Use the ServiceAccount in your Pod/Deployment**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: config-reader
  namespace: app-namespace
spec:
  replicas: 1
  selector:
    matchLabels:
      app: config-reader
  template:
    metadata:
      labels:
        app: config-reader
    spec:
      serviceAccountName: app-reader-sa
      containers:
        - name: app
```

```
      image: my-app:latest
```

🚀 Now your app will only be able to read Secrets and ConfigMaps – **nothing else**.

When a **Kubernetes cluster** is created, some **default Roles, RoleBindings, ClusterRoles, ClusterRoleBindings, and ServiceAccounts** are automatically created by Kubernetes to ensure basic functionality and security of the control plane components and workloads.

⚙️ 1. Default ServiceAccounts

When you create a namespace (even `default`), Kubernetes **automatically creates a ServiceAccount** named `default`.

📌 **Example:**

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default
  namespace: default
```

🔹 **Purpose**:

Every Pod without an explicitly defined `serviceAccountName` automatically uses this `default` ServiceAccount.

It typically **has no permissions unless explicitly given via RBAC**.

🛡️ 2. Default Roles & RoleBindings (Namespace-scoped)

By default, Kubernetes does not create many namespace-scoped Roles unless an addon or component (like the Dashboard) requires it.

But you **can** create your own `Role` and `RoleBinding` scoped to a namespace.

➡️ These are not created unless some specific component or workload installs them.

🌐 3. Default ClusterRoles

Kubernetes provides several **predefined ClusterRoles** to help manage access and permissions.

✅ **Examples of built-in `ClusterRoles`:**

- `cluster-admin` – Full control over the entire cluster
- `admin` – Admin access within a namespace
- `edit` – Can edit most resources in a namespace (not RBAC)

- `view` – Read-only access in a namespace
- `system:discovery` – For unauthenticated users to perform discovery (like listing API versions)
- `system:node` – Used internally by kubelets
- `system:controller:*` – For internal controllers like deployment-controller, job-controller, etc.

These roles are defined **cluster-wide**, and you can use them for your users, groups, or ServiceAccounts via **ClusterRoleBindings** or **RoleBindings**.

🔗 4. Default ClusterRoleBindings

ClusterRoleBindings are automatically created for Kubernetes components to function properly.

✅ **Examples of default `ClusterRoleBindings`:**

- `cluster-admin` – Grants `cluster-admin` role to `system:masters` group (used by admins)
- `system:node` – Binds `system:node` role to `system:nodes` group (all kubelets)

- `system:public-info-viewer` – Lets unauthenticated users access basic cluster info

- `system:controller:...` – Binds appropriate roles to internal controllers

🔐 These are essential for Kubernetes components like:

- kubelets accessing the API server

- Controllers managing deployments

- Admin users performing cluster operations

🔍 Pro Tip:

You can **list** default roles and bindings in your cluster with:

```
kubectl get clusterrole

kubectl get clusterrolebinding

kubectl get role -A

kubectl get rolebinding -A

kubectl get serviceaccount -A
```

🔐 Real-Time RBAC Scenario: Dev & QA Access Management in a Kubernetes Cluster

🏢 Use Case:

You are managing a Kubernetes cluster for a company with multiple teams:

- **Dev Team**: Should have full access to all resources *only* in the `dev` namespace.
- **QA Team**: Should have read-only access to Pods and Services in the `qa` namespace.
- **A CI/CD tool (like Jenkins)**: Running in `cicd` namespace, needs permission to deploy apps in both `dev` and `qa` namespaces.

To implement this securely, you use **RBAC** with a combination of:

- Roles / ClusterRoles
- RoleBindings / ClusterRoleBindings
- Groups / Users / ServiceAccounts

🔹 Step-by-Step Implementation:

1️⃣ Dev Team Setup – Full Access in `dev` Namespace

✅ **Create a Role**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: dev-team-role
  namespace: dev
rules:
- apiGroups: ["", "apps", "batch"]
  resources: ["pods", "services", "deployments", "jobs"]
  verbs: ["get", "list", "create", "delete", "update",
"patch"]
```

✅ **Bind Role to the Dev Group**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
```

```
  name: dev-team-binding
  namespace: dev
subjects:
- kind: Group
  name: dev-team
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: dev-team-role
  apiGroup: rbac.authorization.k8s.io
```

📌 This grants full permissions to any user in the `dev-team` group within the `dev` namespace.

2️⃣ QA Team Setup – Read-Only in `qa` Namespace

✅ **Create a Role**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
```

```
metadata:
  name: qa-team-readonly
  namespace: qa
rules:
- apiGroups: [""]
  resources: ["pods", "services"]
  verbs: ["get", "list", "watch"]
```

✅ **Bind Role to QA Group**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: qa-team-binding
  namespace: qa
subjects:
- kind: Group
  name: qa-team
  apiGroup: rbac.authorization.k8s.io
```

```
roleRef:
  kind: Role
  name: qa-team-readonly
  apiGroup: rbac.authorization.k8s.io
```

🔒 This ensures the QA team can only view resources in the `qa` namespace.

3️⃣ CI/CD ServiceAccount Setup – Cluster-Wide Deployment Access

✅ **Create a ClusterRole**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cicd-deployer
rules:
- apiGroups: ["apps", ""]
  resources: ["deployments", "pods", "services"]
  verbs: ["create", "update", "delete", "get", "list"]
```

✅ **Create a ServiceAccount in `cicd` namespace**:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: jenkins-deployer
  namespace: cicd
```

✅ **Bind ClusterRole to ServiceAccount**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cicd-deployer-binding
subjects:
- kind: ServiceAccount
  name: jenkins-deployer
  namespace: cicd
roleRef:
  kind: ClusterRole
```

```
  name: cicd-deployer
  apiGroup: rbac.authorization.k8s.io
```

🔧 This allows Jenkins (via the `jenkins-deployer` ServiceAccount) to deploy apps in any namespace across the cluster.

🧠 Summary of RBAC Entities Used:

Entity	Kind	Scope	Description
`dev-team-role`	Role	Namespace	Full access to Dev team
`qa-team-readonly`	Role	Namespace	Read-only for QA team
`cicd-deployer`	ClusterRole	Cluster	Full deploy access
`RoleBinding`	RoleBinding	Namespace	Bind Role to user/group
`ClusterRoleBinding`	ClusterRoleBinding	Cluster	Bind ClusterRole to ServiceAccount

`jenkins-deployer`	ServiceAccount	Namespace	Used by CI/CD pipelines

✅ Real-World Benefits of This Setup:

🔐 Least privilege access — no team gets more permissions than required

🔄 Clear separation of environments (`dev`, `qa`, `prod`)

⚙️ Enables automation tools (like Jenkins) to safely deploy apps

🧩 Easy to audit and manage access per team/project

🔐 Kubernetes RBAC: ServiceAccount-Based Access Control Setup (Admin, General, Others)

🎯 Objective

Set up **three different ServiceAccounts** with distinct permissions in a Kubernetes cluster, generate tokens, create dedicated kubeconfig files, and verify access from separate VMs/EC2s.

🧱 1. Create ServiceAccounts

```
kubectl create serviceaccount admin-sa -n default

kubectl create serviceaccount general-sa -n default

kubectl create serviceaccount others-sa -n default
```

🔐 2. Create ClusterRoles & ClusterRoleBindings

✅ A. Admin ServiceAccount – Full Access

- 🔸 **ClusterRole**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: admin-cr
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

- **ClusterRoleBinding**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-crb
subjects:
- kind: ServiceAccount
  name: admin-sa
  namespace: default
roleRef:
```

```
  kind: ClusterRole
  name: admin-cr
  apiGroup: rbac.authorization.k8s.io
```

✅ B. General ServiceAccount – Read-Only for Core Resources

🔸 **ClusterRole**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: general-cr
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps", "secrets"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch"]
```

🔸 **ClusterRoleBinding**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: general-crb
subjects:
- kind: ServiceAccount
  name: general-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: general-cr
  apiGroup: rbac.authorization.k8s.io
```

✅ **C. Others ServiceAccount – Namespace-Level Read Access**

🔸 **ClusterRole**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
  name: others-cr
rules:
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["get", "list", "watch"]
```

🔸 **ClusterRoleBinding**:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: others-crb
subjects:
- kind: ServiceAccount
  name: others-sa
  namespace: default
roleRef:
  kind: ClusterRole
  name: others-cr
  apiGroup: rbac.authorization.k8s.io
```

🎟️ 3. Generate Tokens for ServiceAccounts

```
kubectl create token admin-sa -n default > admin.token

kubectl create token general-sa -n default > general.token

kubectl create token others-sa -n default > others.token
```

✅ Save these tokens securely — we'll use them in kubeconfig files.

📜 4. Get Cluster Info for Kubeconfig

```
CLUSTER_NAME=$(kubectl config view --minify -o
jsonpath='{.clusters[0].name}')

CLUSTER_SERVER=$(kubectl config view --minify -o
jsonpath='{.clusters[0].cluster.server}')

CA_DATA=$(kubectl config view --minify -o
jsonpath='{.clusters[0].cluster.certificate-authority-data}'
)
```

📁 5. Create Kubeconfig Files

✅ Admin

```
cat <<EOF > kubeconfig-admin.yaml

apiVersion: v1
```

```
kind: Config
clusters:
- cluster:
    certificate-authority-data: $CA_DATA
    server: $CLUSTER_SERVER
  name: $CLUSTER_NAME
contexts:
- context:
    cluster: $CLUSTER_NAME
    user: admin
  name: admin-context
current-context: admin-context
users:
- name: admin
  user:
    token: $(cat admin.token)
EOF
```

✅ **General**

```
cat <<EOF > kubeconfig-general.yaml
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: $CA_DATA
    server: $CLUSTER_SERVER
  name: $CLUSTER_NAME
contexts:
- context:
    cluster: $CLUSTER_NAME
    user: general
  name: general-context
current-context: general-context
users:
- name: general
  user:
```

```
    token: $(cat general.token)
EOF
```

✅ **Others**

```
cat <<EOF > kubeconfig-others.yaml
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority-data: $CA_DATA
    server: $CLUSTER_SERVER
  name: $CLUSTER_NAME
contexts:
- context:
    cluster: $CLUSTER_NAME
    user: others
  name: others-context
current-context: others-context
```

```
users:
- name: others
  user:
    token: $(cat others.token)
EOF
```

💻 6. Use from New EC2/VM

👉 In each target EC2/VM:

1. Install `kubectl`
2. Paste respective kubeconfig (`kubeconfig-admin.yaml`, etc.)

Export:

```
export KUBECONFIG=~/kubeconfig-admin.yaml  # or general/others
```

✅ 7. Verify Access in VM

Admin:

```
kubectl get pods -A
```

```
kubectl create ns test-ns

kubectl delete pod xyz -n dev
```

General:

```
kubectl get pods -n default

kubectl get configmaps

kubectl create ns test-ns    ❌ (should be forbidden)
```

Others:

```
kubectl get namespaces

kubectl get pods -n default  ❌ (should be forbidden)
```

✅ What This RBAC Scenario Does

The scenario sets up **three types of access levels** in a Kubernetes cluster using **ServiceAccounts**, **ClusterRoles**, **ClusterRoleBindings**, and **token-based authentication**. Here's what it achieves:

🔐 1. Implements Role-Based Access Control (RBAC)

You're assigning **specific permissions to specific ServiceAccounts** so that:

- 🛡️ Admin users can **do anything in the cluster** (create, delete, modify any resource).
- 📖 General users can only **view important resources** (pods, services, configmaps, secrets, deployments).
- 👀 Other users can **only view namespaces**, and nothing else.

This ensures **least privilege access** and prevents misuse or accidental modification of critical cluster resources.

🧑‍💼 2. Simulates Real-World Access Scenarios

Just like in a real company setup:

- **Admin team** manages the entire cluster.

- **Dev or QA team** only needs read-only access for debugging.
- **Auditors or compliance teams** just need to verify what namespaces exist.

This kind of separation is **very common in production environments** and supports real-world **team-based access models**.

🔗 3. Token-Based Authentication via ServiceAccounts

Instead of using kubeconfig generated for IAM or user credentials, you're using **ServiceAccount tokens** for authentication, which is:

- Lightweight
- Secure
- Works well for automation, bots, and internal tools

🗂️ 4. Custom Kubeconfig Files for External Access

You're generating **separate kubeconfig files** for each ServiceAccount and using them from **external EC2s or VMs**, simulating how:

- Devs connect from their own machines
- Bots or monitoring tools connect to the cluster

- Audit or Ops teams securely connect without sharing credentials

✅ 5. Complete Setup for Real Use Cases

This is a full-cycle implementation covering:

- Creation of ServiceAccounts
- Defining permissions (ClusterRole)
- Binding permissions (ClusterRoleBinding)
- Token generation
- Kubeconfig creation
- External machine access
- Access verification

It's like a **production-grade, best-practice example** of implementing secure and structured access to a Kubernetes cluster.

🎯 Why This Scenario Matters

- 🔒 **Security**: Avoid giving everyone cluster-admin access
- 👥 **Team-based access**: Dev, QA, Admin, Auditor all need different levels
- 💼 **Compliance**: Follows principle of least privilege
- ⚙️ **Automation ready**: Works great for bots and CI/CD tools
- 🚀 **Real-world experience**: Prepares you for interviews, real jobs, and handling production workloads

🎯 What's Different in EKS?

Unlike KOPS, **EKS uses AWS IAM for authentication**, and **RBAC for authorization**. That means:

- 👤 You **cannot directly use ServiceAccount tokens** for human users to log in via `kubectl`.
- Instead, **you authenticate using IAM identities (users or roles)**.
- Then, you **map those IAM identities to Kubernetes users/groups** using the `aws-auth` ConfigMap.

So, you achieve the **same separation of access and RBAC policies** — just with **IAM + RBAC combo** instead of pure ServiceAccount tokens.

✅ Real-Time Equivalent Scenario in EKS

Let's convert your 3-service-account scenario into EKS-style:

🔹 1. IAM Users Setup

Create 3 IAM users in AWS:

- `eks-admin`
- `eks-general`
- `eks-others`

Generate **access keys** for each user.

🔹 2. Update `aws-auth` ConfigMap in EKS

Use the following command to edit:

```
kubectl edit configmap aws-auth -n kube-system
```

Add each user with appropriate groups:

```
mapUsers: |
  - userarn: arn:aws:iam::<account-id>:user/eks-admin
    username: eks-admin
    groups:
      - system:masters # Full access

  - userarn: arn:aws:iam::<account-id>:user/eks-general
    username: eks-general
    groups:
      - view-group

  - userarn: arn:aws:iam::<account-id>:user/eks-others
    username: eks-others
    groups:
      - ns-read-group
```

🔹 **3. Create Corresponding ClusterRoles and ClusterRoleBindings**

🛡️ **ClusterRole for `view-group`**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: general-read-access
rules:
- apiGroups: [""]
  resources: ["pods", "services", "configmaps", "secrets"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch"]
```

ClusterRoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: general-read-binding
roleRef:
  kind: ClusterRole
  name: general-read-access
```

```
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: Group
  name: view-group
  apiGroup: rbac.authorization.k8s.io
```

🕶️ **ClusterRole for `ns-read-group`**

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

ClusterRoleBinding:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
```

```
  name: ns-read-binding
roleRef:
  kind: ClusterRole
  name: namespace-reader
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: Group
  name: ns-read-group
  apiGroup: rbac.authorization.k8s.io
```

🛑 You do **not** use tokens and kubeconfig manually here.
✅ Each IAM user runs `aws eks update-kubeconfig` with their credentials, and gets access as per RBAC.

🧪 Verifying Access from New EC2s/VMs

- On each EC2, configure credentials for respective IAM user using AWS CLI.
- Run:

```
aws eks update-kubeconfig --region <region> --name <cluster-name>
```

Now test access:

- `eks-admin` → Full access
- `eks-general` → Only view workloads
- `eks-others` → Can only list namespace

🔐 How is RBAC Useful in Kubernetes?

RBAC (**Role-Based Access Control**) is one of the **most powerful security features** in Kubernetes. It allows you to **control "who can do what"** in your cluster, ensuring that only the **right people or services** have access to the **right resources**.

Here's how RBAC helps:

🛡️ 1. Enhanced Security

RBAC lets you follow the **Principle of Least Privilege (PoLP)** – meaning each user/service only gets **the minimal permissions needed** to do their job.

🧠 Example:
Your developer only needs to view deployments, not delete them. RBAC ensures that!

👥 2. Team-Based Access Control

With RBAC, you can **define roles for different teams** — like developers, testers, admins, etc.

✅ Devs: Can view and edit resources in the `dev` namespace
✅ Testers: Can only view logs and pods
✅ Admins: Full access to everything

🔄 3. Centralized and Declarative Management

You manage access using YAML files, so it's **easy to review, audit, and version control** access policies — just like your code! 💻

🔗 4. Integration with IAM (in EKS)

In EKS, RBAC works seamlessly with **AWS IAM**, so you can bind IAM users and roles to Kubernetes permissions — best of both worlds! 🌍

🧩 5. Supports Multiple Subject Types

RBAC can be applied to:

- 👤 **Users**
- 👥 **Groups**
- 🛠️ **ServiceAccounts**

So whether it's a human, a bot, or a CI/CD tool — you've got control!

⚙️ 6. Granular Access Control

You can define permissions for:

- Specific resources (Pods, Services, Secrets, etc.)
- Specific verbs (`get`, `list`, `create`, `delete`, etc.)
- Specific namespaces or the entire cluster

🔍 Want a service to only `get` and `list` pods in one namespace? RBAC can do that.

🚨 7. Prevents Costly Mistakes

RBAC protects your cluster from **accidental deletions or misconfigurations** by unauthorized users.

🛑 Imagine a junior developer accidentally deletes a production pod. RBAC can prevent that.

📊 8. Auditing & Compliance

RBAC policies make it easier to **audit who has access to what**, which is super useful for:

- Security audits
- Compliance (HIPAA, GDPR, SOC 2)

🔍 How to Audit and Debug RBAC Issues in Kubernetes

When something doesn't work — like a user or service account can't access a resource — the problem is **often RBAC-related**. Here's how to **quickly identify and fix** those issues:

✅ 1. Use `kubectl auth can-i` – Your Best Friend

This command tells you **whether a user or service account** can perform a specific action.

🔧 Syntax:

```
kubectl auth can-i <verb> <resource> --as <user> -n <namespace>
```

📌 Example:

```
kubectl auth can-i get pods --as system:serviceaccount:default:read-only-sa -n default
```

🧠 Use it to **simulate access** from any identity and quickly validate if the permissions are working.

✅ 2. Describe RoleBindings & ClusterRoleBindings

To verify which Roles/ClusterRoles are bound to a subject:

```
kubectl describe rolebinding <name> -n <namespace>
```

```
kubectl describe clusterrolebinding <name>
```

🔍 Check:

- **Subjects**: Are you binding the right user/group/service account?
- **Roles**: Do they have the correct rules?

✅ 3. View Roles & ClusterRoles

Sometimes a Role may not have the expected permissions. Check the details like this:

```
kubectl get role <name> -n <namespace> -o yaml
```

```
kubectl get clusterrole <name> -o yaml
```

✅ 4. Inspect ServiceAccounts

If you're using ServiceAccounts, make sure you're referencing them correctly:

```
kubectl get serviceaccount <name> -n <namespace> -o yaml
```

🔑 Confirm:

- Tokens are mounted correctly
- Name and namespace match the binding

✅ 5. Check API Server Logs (Advanced)

If you have access to the control plane (like in Kops), you can check audit logs:

📂 Audit log location (self-managed cluster):

```
/var/log/kubernetes/audit.log
```

🔍 Look for:

- Denied requests
- Forbidden errors (`"reason":"Forbidden"`)

This is not possible in EKS unless CloudTrail or CloudWatch Audit Logging is enabled.

✅ 6. Enable Audit Logs in EKS (Optional for EKS)

If you're on **Amazon EKS**, enable Kubernetes audit logs in CloudWatch:

🔧 Go to:

`EKS > Cluster > Logging > Enable Audit Logs`

Use it to monitor all API requests and RBAC denials!

✅ **7. Common Issues to Watch For**

🚫 **RoleBinding in wrong namespace**

✅ Fix: Ensure RoleBinding is in the **same namespace** as Role and subject

🚫 **Using RoleBinding instead of ClusterRoleBinding**

✅ Fix: Use `ClusterRoleBinding` if you're working across namespaces or want cluster-wide access

🚫 **Incorrect service account name or namespace**

✅ Fix: Double-check both name and namespace when binding

🙌 Thank You for Being a Part of This RBAC Journey!

As we come to the end of this deep dive into **Kubernetes RBAC (Role-Based Access Control)**, I just want to take a moment to express my heartfelt gratitude to **everyone who supported, engaged, and followed along**. ❤️

Whether you're a beginner just getting started with Kubernetes or an experienced engineer refining your security practices — I hope this guide gave you the clarity and confidence to implement RBAC effectively and securely in your clusters.

🛡️ **What we've achieved together in this doc:**

- We broke down RBAC from the ground up — theory to practice.
- Explored real-world scenarios using `Users`, `Groups`, and `ServiceAccounts`.
- Set up complete role-based environments with scoped access control.
- Learned how to **debug and audit permissions** to stay in control.
- And finally, built a secure and scalable access management layer tailored for real Kubernetes clusters (Kops & EKS).

💬 What's Next?

The journey doesn't stop here!
There's **so much more in the Kubernetes world** that I can't wait to explore and share with you.

From **Network Policies**, **Pod Security**, **Helm**, **GitOps**, to **CI/CD in Kubernetes** — I'm just getting started.

So if you've enjoyed this doc:

- 🤝 **Let's connect** — I'd love to hear your feedback and suggestions.
- 🔁 **Share it with others** who might find this useful.
- ✅ **Follow me for more content** — daily Kubernetes learnings, hands-on projects, AWS tasks, and DevOps deep dives are on the way.

🌟 Thank You Again

Your support truly means the world to me — every comment, like, message, and reshare encourages me to keep going and keep learning.
Let's continue growing, building, and helping each other in this amazing DevOps & Cloud community. 💻☁️💙

With tons of gratitude,
~ Fenil Gajjar

> "Keep learning. Keep building. And always stay curious. 🚀"