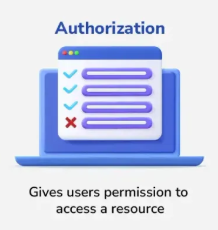
[**Authorization in Kafka**](https://inmorphis.sharepoint.com/sites/Confluent/Conlfuent%20Platform/Authorization%20in%20Kafka.aspx)



Authorization is the process of determining whether a user, service, or system has the necessary permissions to perform a specific action on a resource. It occurs after authentication, where the identity of the user or system is verified. Authorization ensures that only entities with the appropriate permissions can access certain data or perform specific operations.

**Authorization in Apache Kafka**

In Kafka, authorization controls which users or clients can access specific topics, consumer groups, or perform administrative operations. Kafka implements **Access Control Lists (ACLs)** to enforce authorization. ACLs define rules that allow or deny access to resources based on the user's identity and the action being requested.

**Key Concepts of Kafka Authorization:**

Resources: These include:

**Topic:** Messages in Kafka are organized into topics.

**Consumer Group:** A set of consumers sharing a subscription.

**Cluster:** Administrative operations like creating topics or modifying configurations.

**Transactional IDs:** For managing transactions in Kafka.

**Operations:** Actions that can be performed on Kafka resources, such as:

**Read:** Consume messages from a topic.

**Write:** Publish messages to a topic.

**Describe:** View metadata of a resource.

**Alter:** Modify configurations.

**Delete:** Remove a resource.

**Principals:** Entities (users or services) that interact with Kafka. Kafka uses the SASL (Simple Authentication and Security Layer) mechanism to identify principals.

**ACLs:** Rules that specify:

The resource type **(e.g., topic, consumer group).**

The principal **(e.g., User:producer1).**

The operation **(e.g., Write).**

The permission type **(Allow or Deny).**

**Use Cases of Authorization in Kafka**

**Secure Multi-Tenant Environments:** Ensure tenants or users only access their respective topics.

**Data Protection:** Prevent unauthorized users from consuming or altering sensitive data.

**Operational Boundaries:**Restrict administrative operations to designated users (e.g., only admins can create or delete topics).

**Least Privilege Principle:**Assign only the necessary permissions to users, reducing the risk of misconfigurations or data breaches.

**Best Practices for Kafka Authorization**

**Enable Authentication Before Authorization:** Without authentication, you can't effectively enforce authorization.

**Minimize Default Permissions:**Set allow.everyone.if.no.acl.found to false to ensure explicit permissions.

**Audit ACLs Regularly:** Periodically review ACL configurations to ensure compliance and security.

**Use Secure Protocols:** Always enable encryption (e.g., SSL/TLS) for secure communication.

**Combine Authorization with Monitoring**: Use tools like Kafka's audit logs or third-party monitoring systems to track access and detect anomalies.

By carefully configuring and enforcing authorization, Kafka administrators can maintain robust security while supporting diverse workloads and user requirements.

**Apache Kafka supports two primary types of authorization mechanisms:**

[**Access Control Lists (ACLs)**](https://inmorphis.sharepoint.com/sites/Confluent/Conlfuent%20Platform/Access%20Control%20Lists%20(ACLs).aspx) are the traditional authorization mechanism in Kafka. They define fine-grained permissions for specific users or services (principals) to access Kafka resources.

[**Role-Based Access Control (RBAC)**](https://inmorphis.sharepoint.com/sites/Confluent/Conlfuent%20Platform/Role-Based%20Access%20Control%20(RBAC).aspx) is a more modern authorization mechanism introduced in Confluent Platform (a commercial distribution of Kafka). RBAC simplifies permission management by assigning users to predefined roles with specific permissions

**Comparison of ACL (Access Control Lists) and RBAC (Role-Based Access Control)**

|  |  |  |
| --- | --- | --- |
| Aspect | ACL (Access Control Lists) | RBAC (Role-Based Access Control) |
| Definition | Grants or denies specific permissions to individual users or services for specific resources. | Assigns roles to users or services, and roles define the permissions. |
| Granularity | Highly fine-grained, controlling access at the resource level (e.g., individual topics, groups). | Coarser-grained, permissions are assigned at the role level rather than per resource. |
| Management | Decentralized; permissions are defined manually and separately for each user and resource. | Centralized; roles and their permissions are managed through a unified interface. |
| Ease of Use | Complex to configure and maintain at scale, especially for large deployments. | Simplified and scalable due to role-based abstraction. |
| Scalability | Challenging in multi-tenant environments due to the need to manage permissions individually. | Designed for scalability, suitable for multi-tenant or enterprise setups. |
| Availability | Available in open-source Apache Kafka. | Only available in Confluent Platform (commercial Kafka distribution). |
| Dynamic Updates | Changes require direct updates to ACLs, which can be cumbersome | Roles can be adjusted dynamically without modifying individual user permissions. |
| Resource Scope | Resource-specific; can specify permissions for individual topics, groups, or other resources. | Role-specific; applies predefined sets of permissions to all resources that a role covers. |
| Predefined Permissions | Not predefined; you must explicitly set the permissions for each resource. | Includes predefined roles like DeveloperRead, DeveloperWrite, ClusterAdmin, etc. |
| Use Case | Best for smaller deployments or single-tenant environments requiring fine-grained control | Ideal for enterprise-level, large-scale, or multi-tenant environments. |

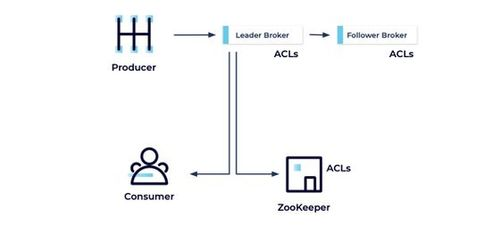
**Key Takeaways:**

**ACL:**Best for environments that require fine-grained, low-level control over specific resources. However, it can become unwieldy in large or dynamic environments.

**RBAC:** Simplifies authorization management, making it suitable for enterprise or multi-tenant setups where scalability and ease of management are critical. However, it is only available in the Confluent Platform.

Choose based on your environment size, management capabilities, and whether you are using open-source Kafka or the Confluent Platform.

[Access Control Lists (ACLs)](https://inmorphis.sharepoint.com/sites/Confluent/Conlfuent%20Platform/Access%20Control%20Lists%20(ACLs).aspx)



An **Access Control List (ACL)** in Kafka is a security feature used to manage authorization for operations performed on Kafka resources such as **topics**, **consumer groups**, and **Kafka cluster** itself. ACLs define which users (principals) are allowed or denied access to perform specific actions (operations) on various resources.

Kafka ACLs are used for **authentication** and **authorization** to ensure that only authorized users can produce or consume data, manage topics, or modify configurations in the Kafka cluster.

**Components of an ACL:**

1. **Principal**: The user or service that interacts with Kafka. In Kafka, this is typically a user defined via SASL authentication (e.g., User:producer, User:consumer).
2. **Operation**: The type of action the principal is allowed to perform, such as Write, Read, Describe, Delete, etc.
3. **Resource**: The Kafka entity that the principal is interacting with, such as Topic, Consumer Group, Cluster, etc.
4. **Permission**: Whether the action is **allowed** or **denied** for the given principal.

**ACL Evaluation Workflow**

When a user or client attempts to perform an operation, Kafka evaluates the ACLs associated with the resource.

**Authentication:**

The client authenticates with Kafka, and its principal is identified (e.g., User:user1).

**Authorization Check:**

Kafka checks the ACLs for the requested resource (e.g., topic, consumer group).

Operations are allowed if there is a matching Allow rule for the principal.

If no matching rule is found and allow.everyone.if.no.acl.found is set to false, the operation is denied.

**Decision**:

The operation is either permitted or denied based on the ACLs.

**How to Implement ACLs in Kafka (Using --bootstrap-server)**

Kafka's kafka-acls.sh tool is used to manage ACLs, and you can interact with Kafka directly using the **bootstrap server** (--bootstrap-server) option without requiring Zookeeper.

**Prerequisites:**

1. Kafka is configured to use **authentication** (SASL) and **authorization** (ACLs).
2. SASL-based authentication is enabled for client interactions, and Kafka brokers should be set up to authorize the requests.

**Steps to Configure ACLs Using --bootstrap-server in Kafka**

**1. Configure Kafka Brokers to Use ACLs (authorization)**

First, enable ACLs on the Kafka brokers by modifying the server.properties file on each broker. These settings ensure the broker enforces ACLs.

**Enable the ACL authorizer**:

|  |
| --- |
| authorizer.class.name=kafka.security.auth.SimpleAclAuthorizer |

**Enable security protocol (e.g., SASL\_PLAINTEXT or SASL\_SSL)**:

|  |
| --- |
| security.inter.broker.protocol=SASL\_PLAINTEXT  # or SASL\_SSL  sasl.enabled.mechanisms=PLAIN  # For plain authentication, or GSSAPI for Kerberos  sasl.mechanism.inter.broker.protocol=PLAIN |

**Define superusers (users with full access)**:

|  |
| --- |
| super.users=User:admin |

After modifying server.properties, restart the Kafka brokers.

**2. Kafka ACL Commands Using --bootstrap-server**

Now, you can use the kafka-acls.sh tool with the --bootstrap-server option to create, list, and remove ACLs.

**Basic Command Syntax:**

|  |
| --- |
| kafka-acls --bootstrap-server <broker\_list> --add|--remove|--list --allow-principal User:<username> --operation <operation> --topic <topic\_name> --group <group\_name> --cluster |

**3. Add ACL for a User**

These commands allow you to define which operations a user can perform on different Kafka resources.

**Add Permission to Write to a Topic**

This command grants a user the ability to write (produce) messages to a topic.

|  |
| --- |
| kafka-acls --bootstrap-server <broker\_list> --add|--remove|--list --allow-principal User:<username> --operation <operation> --topic <topic\_name> --group <group\_name> --cluster |

**Allow the producer user to write to the my\_topic:**

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:producer --operation Write --topic my\_topic |

**Add Permission to Read from a Topic**

This command grants a user the ability to read (consume) messages from a topic.

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:<username> --operation Read --topic <topic\_name> |

Allow the consumer user to read from my\_topic:

|  |
| --- |
| kafka-acls.sh --bootstrap-server localhost:9093 --add --allow-principal User:consumer --operation Read --topic my\_topic |

**Add Permission to Join a Consumer Group**

This command grants a user the ability to consume messages from a consumer group.

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:<username> --operation Read --group <consumer\_group\_name> |

Allow the consumer user to read from the my\_consumer\_group:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:consumer --operation Read --group my\_consumer\_group |

**Add Permission to Describe Cluster Metadata**

This command grants a user the ability to describe cluster metadata (e.g., list topics, view topic configurations).

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:consumer --operation Read --group my\_consumer\_group |

Allow the admin user to describe the Kafka cluster:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:admin --operation Describe --cluster |

**Add Permission to Delete Topics**

This command grants a user permission to delete Kafka topics.

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:<username> --operation Delete --topic <topic\_name> |

Allow the admin user to delete my\_topic:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:admin --operation Delete --topic my\_topic |

**4. Remove ACL for a User**

You can remove an existing ACL by using the --remove flag. This removes the specific permission granted to a user on a resource.

**Remove Write Permission for a User on a Topic**

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --remove --allow-principal User:<username> --operation Write --topic <topic\_name> |

Remove write permission for the producer on my\_topic:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --remove --allow-principal User:producer --operation Write --topic my\_topic |

**Remove Read Permission for a User on a Topic**

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --remove --allow-principal User:<username> --operation Read --topic <topic\_name> |

Remove read permission for the consumer on my\_topic:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --remove --allow-principal User:consumer --operation Read --topic my\_topic |

**Remove Permission for a User from a Consumer Group**

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --remove --allow-principal User:<username> --operation Read --group <consumer\_group\_name> |

Example: Remove read permission for the consumer from the my\_consumer\_group:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --remove --allow-principal User:consumer --operation Read --group my\_consumer\_group |

**5. List Existing ACLs**

You can list all the ACLs that are currently active in the Kafka cluster to check the permissions granted to users.

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --list |

This will show you all the ACLs applied to various resources (topics, consumer groups, cluster) and the associated users and operations.

**6. Use Wildcards for Topics or Consumer Groups**

You can use wildcards (\*) in ACLs to apply permissions to all topics or consumer groups.

**Allow User to Write to All Topics**

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:<username> --operation Write --topic '\*' |

Allow the producer to write to **all topics**:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:producer --operation Write --topic '\*' |

**Allow User to Read from All Topics**

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:<username> --operation Read --topic '\*' |

Allow the consumer to read from **all topics**:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:<username> --operation Read --topic '\*' |

**Allow User to Read from All Consumer Groups**

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:<username> --operation Read --group '\*' |

Allow the consumer to read from **all consumer groups**:

|  |
| --- |
| kafka-acls --bootstrap-server localhost:9093 --add --allow-principal User:consumer --operation Read --group '\*' |

**7. Testing ACLs**

Once ACLs are configured, you can test if the permissions are applied correctly by attempting to produce or consume messages with a user's credentials.

**Test Producing to a Topic**

To test producing to a topic as a producer:

|  |
| --- |
| kafka-console-producer --broker-list localhost:9093 --topic my\_topic --producer.config producer.properties |

In producer.properties, specify the authentication details:

|  |
| --- |
| security.protocol=SASL\_PLAINTEXT  sasl.mechanism=PLAIN  sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="producer" password="producer-password"; |

**Test Consuming from a Topic**

To test consuming from a topic as a consumer:

|  |
| --- |
| kafka-console-consumer --bootstrap-server localhost:9093 --topic my\_topic --from-beginning --consumer.config consumer.properties |

In consumer.properties, specify the authentication details:

|  |
| --- |
| security.protocol=SASL\_PLAINTEXT  sasl.mechanism=PLAIN  sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required username="consumer" password="consumer-password"; |

Role-based access control (RBAC) is a method for controlling system access based on roles assigned to users within an organization. RBAC is defined around predefined roles and the privileges associated with those roles (also known as role bindings). Roles are a collection of permissions that you can bind to a resource; this binding allows the privileges associated with that role to be performed on that resource. You must grant the role to a principal at the time you bind a resource to the role.

Using RBAC, you can manage who has access to specific Confluent Platform resources, and the actions a user can perform within that resource. RBAC leverages the Confluent Platform Metadata Service to configure and manage your RBAC implementation from a centralized configuration context, thereby simplifying access management across Confluent Platform resources.

Before implementing RBAC you should evaluate the security needs of the users in your organization and, based on the resources they require to perform their duties, group users into roles that satisfy those requirements

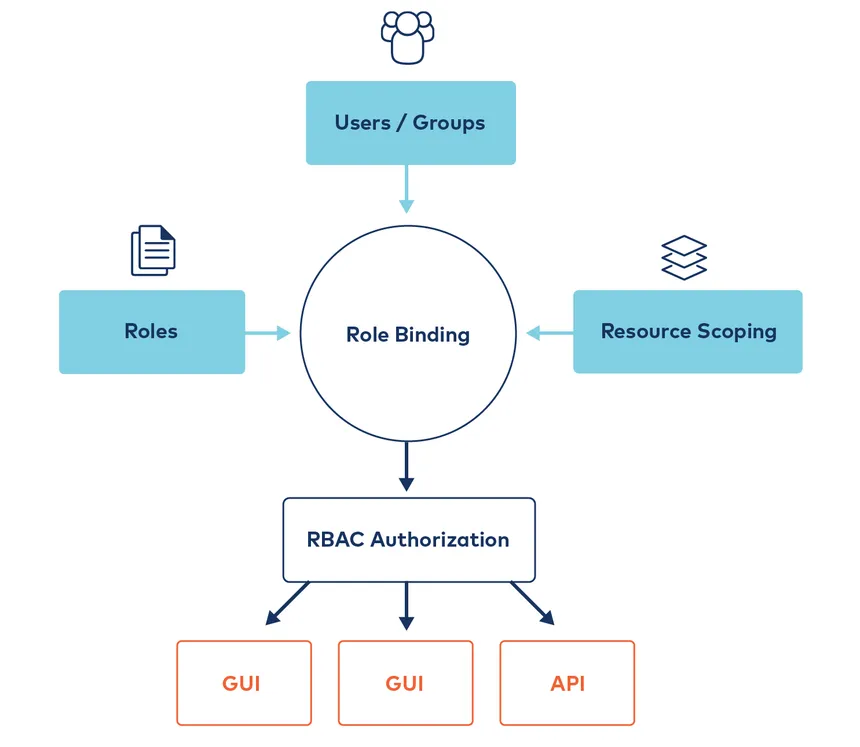
**RBAC benefits**

* Manage security access across the Confluent Platform (Kafka, ksqlDB, Connect, Schema Registry, Confluent Control Center) using granular permissions to control user and group access. For example, with RBAC you can specify permissions for each connector in a cluster, making it easier and quicker to get multiple connectors up and running.
* Manage authorization at scale. Administrators can centrally manage the assignment of predefined roles, and also delegate the responsibility of managing access and permissions to the different departments or business units who are the true owners and most familiar with those resources.
* Centrally manage authentication and authorization for multiple clusters, which includes: the MDS, a Kafka cluster, Connect, ksqlDB, Schema Registry clusters, and a single instance of Confluent Control Center.

**How RBAC works**

Predefined role assignments determine who can access specific Confluent Platform resources, and what actions an individual user can perform within that resource. An administrator assigns predefined roles to users and groups on various resources; each user can be assigned multiple roles on each resource. Certain privileged users (such as the UserAdmin or SystemAdmin) assign roles to users and groups, and then map specific resources to those user roles. For example, a ResourceOwner in the finance department can grant department members access to all topics that use the prefix finance\_, which makes it easier for them to manage the resources with which they are most familiar.

User administrators can add LDAP users and groups, making it quicker and easier to centrally configure authentication and authorization for the various Confluent Platform resources used in an organization.



With RBAC, the user administrator can map roles to LDAP users and groups that are scoped to specific resources (role binding). After a role binding is set using the Confluent CLI confluent iam rbac role-binding create command, users can’t go to an API or Confluent Control Center to bypass and get access to resources. Binding roles to groups enables client administrators to avoid having to grant explicit access to each user across every component. For details about viewing role bindings, refer to Confluent CLI confluent iam rbac role-binding list command. Note that role binding does not support wildcard matching in principal names.

**Terminology**

**Access control**

Access is the ability of an individual user or application to perform a specific task, such as view, create or modify a resource (e.g. topics). Access control enables secure access to Confluent Platform services and resources.

**Principal**

The identity of a user or software requesting permission to perform a specific action on a specific resource. Principals can be authenticated or non-authenticated (ANONYMOUS).

**User principal**

A single identity tied to a specific user or piece of software.

**Group principal**

A shared identity that groups together a list of user principals and/or other group principals.

**Role**

A Confluent-defined job function that is assigned a set of permissions required to perform specific actions or operations on Confluent resources. Each role is bound to a principal and Confluent resources.

**Resource**

A resource can be an Apache Kafka® topic, consumer group, transactionalID, cluster, Schema Registry, ksqlDB, and any other Confluent Platform component.

**Role binding**

A principal-role-resource combination that allows a principal to perform operations on a resource or set of resources as defined by the role.

**Role-based access control (RBAC)**

With RBAC, permissions are associated with roles, and users or groups are assigned to appropriate roles. Roles are defined according to job competency, authority, and responsibility within the enterprise. Users and groups are easily reassigned from one role to another. Permissions assigned to roles tend to change relatively slowly compared with changes in user membership of roles.

**Confluent Platform Metadata Service**

MDS is the primary mechanism by which RBAC is implemented, and offers a single, centralized configuration context that once set up for a cluster, saves administrators from the complex and time-consuming task of defining and assigning roles for each resource on an individual basis. The Confluent Platform MDS binds and enforces a Kafka cluster configuration across different resources (such as topics, connectors, and schemas). The Metadata Service (MDS) acts as the central authority for all authorization and authentication data. You must configure each Kafka broker in the MDS cluster with MDS.

MDS provides ease of use and convenience in the implementation of role-based access control (RBAC); it can also scale so that you can use this same non-binding permissions model to offer other types of security.

Running on a Confluent Server broker, MDS is also integrated with LDAP to provide authentication and refreshable bearer tokens for impersonation. The MDS is also the master of record for role binding

**RBAC and ACLs**

RBAC serves as an additional authorization enforcement layer on top of ACLs, and does not change the way ACLs are created or managed. When considering whether to use RBAC or ACLs for access control, it is suggested you use RBAC as the default because of its ease of use and manageability at scale, but for edge cases where you need to have more granular access control, or wish to explicitly deny access, ACLs may make more sense. For example, you could use RBAC to allow access for a group of users, but an ACL to deny access for a particular member of that group.

RBAC adds an additional authorization mechanism that addresses the following authorization challenges when using ACLs:

Without RBAC, you cannot use ACLs to grant access to connectors. With RBAC, each connector has its own principal that identifies access to resources. Users have access only to connectors on which they have explicitly been granted permission. If you require connector access control, RBAC is essential.

RBAC provides the ability to offer Confluent Control Center users granular access to resources; prior to RBAC, any user with access to Confluent Control Center had full or read-only access to topics and resources. If granular access control in Confluent Control Center is a requirement, RBAC is recommended.

RBAC provides a consistent authentication and authorization mechanism for users access across the entire Confluent Platform, which is not possible if solely using ACLs.

Prior to RBAC, the creation and management of ACLs could be difficult to manage and maintain, and in organizations with thousands of resources and users, ACL setup could take a long time. With RBAC, the delegation of responsibility to various resources is managed using the ResourceOwner role.

For example, say you are responsible for managing user access to 1000 topics. Using RBAC, you could grant ResourceOwner to other users to manage the topics owned by specific business units, and in turn, those users could manage access for others within their own teams. Using ACLs in this scenario, you would need to centrally manage access to all the topics, which would be a time and resource-intensive task.

[**Predefined RBAC Roles in Confluent Platform**](https://inmorphis.sharepoint.com/sites/Confluent/Conlfuent%20Platform/Predefined%20RBAC%20Roles%20in%20Confluent%20Platform.aspx)

Confluent Platform provides predefined roles to help implement granular permissions for specific resources and to simplify access control across the Confluent Platform.

[**RBAC Implementation**](https://inmorphis.sharepoint.com/sites/Confluent/Conlfuent%20Platform/RBAC%20Implementation.aspx)

Role-based access control (RBAC) is administered by a super user using the Confluent CLI and distributed across an organization.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Confluent Platform provides predefined roles to help implement granular permissions for specific resources and to simplify access control across the Confluent Platform. A predefined role is a Confluent-defined job function that is assigned a set of permissions required to perform specific actions or operations on Confluent resources. Each role is bound to a principal and Confluent resources. Users can have multiple roles assigned to them. You cannot use a predefined role to override denial-of-access (DENY) that is configured in an ACL.  When a role is assigned at the cluster-level (Kafka cluster, Schema Registry cluster, ksqlDB cluster, or Connect cluster) it means that users who are assigned this role have access to all resources in a cluster. For example, the ClusterAdmin of a Kafka cluster has access to Confluent Control Center alerts. There are corresponding resource types for each cluster type. For example, you can assign the ResourceOwner role to the resource types KsqlCluster:ksql-cluster or Cluster:kafka-cluster to provide a user all the ResourceOwner privileges for a ksqlDB or Kafka cluster.  When a role is assigned at the resource-level it means that users assigned this role only have access to specific resources as defined in the role binding.  The resource types for which you can assign RBAC roles and role bindings are:   * Kafka cluster * Topic * Consumer group * TransactionalID * Schema Registry cluster * Schema Registry subject * ksqlDB cluster * Connect cluster * Connector   Confluent Platform provides the following predefined roles:   | **Role Name** | **Role Scope** | **View Role Bindings of Others** | **Manage Role Bindings** | **Monitor** | **Resource Read** | **Resource Write** | **Resource Manage** | | --- | --- | --- | --- | --- | --- | --- | --- | | super.user | Cluster-level | Yes | Yes | Yes | Yes | Yes | Yes | | SystemAdmin | Cluster-level | Yes | Yes | Yes | Yes | Yes | Yes | | ClusterAdmin | Cluster-level | No | No | Yes | Yes | Yes | Yes | | UserAdmin | Cluster-level | Yes | Yes | No | No | No | No | | SecurityAdmin | Cluster-level | Yes | No | No | No | No | No | | AuditAdmin | Cluster-level | No | No | No | No | No | Yes | | Operator | Cluster-level | No | No | Yes | No | No | Yes | | ResourceOwner | Resource-level | Yes | Yes | No | Yes | Yes | Yes | | DeveloperRead | Resource-level | No | No | No | Yes | No | No | | DeveloperWrite | Resource-level | No | No | No | No | Yes | No | | DeveloperManage | Resource-level | No | No | No | No | No | Yes |   **super.user**  The purpose of super.user is to have a bootstrap user who can initially grant another user the SystemAdmin role.  Technically speaking, super.user is not a predefined role. It is a server.properties attribute that defines a user who has full access to all resources within a Metadata Service (MDS) cluster. A super.user has no access to resources in other clusters (unless also configured as a super.user on other clusters). The primary use of super.user is to bootstrap Confluent Platform and assign a SystemAdmin. On MDS clusters, super.user can create role bindings for all other clusters. Permissions granted by super.user apply only to the broker where the super.user attribute is specified, and not to other brokers, clusters, or Confluent Platform components. No authorization is enforced on users defined as super.user. It is strongly recommended that this role is assigned only to a limited number of users (for example, 1-2 users who are responsible for bootstrapping).  **SystemAdmin**  Provides full access to all scoped resources in the cluster (ksqlDB cluster, Kafka cluster, or Schema Registry cluster).  It is strongly recommended that this role is assigned only to a limited number of users (one or two per cluster) who need full permission for initial setup or to address urgent issues when absolutely necessary in production instances. You may wish to assign this role more liberally in small test and development use cases, or when working in ksqlDB clusters that are primarily single tenant. Otherwise, it is recommended that you do not assign this role.  **ClusterAdmin**  Sets up clusters (ksqlDB cluster, Kafka cluster, or Schema Registry cluster).  Responsible for setting up and managing Kafka clusters, brokers, networking, ksqlDB clusters, Connect clusters, and adding or removing nodes and performing upgrades. The ClusterAdmin typically creates topics and sets the properties of those topics, for example performance and capacity, but cannot read or write to topics, and has no access to data. For monitoring applications, it is recommended that this role is delegated to the operator who monitors your applications. Typically, the ClusterAdmin user does not possess knowledge about the content of the cluster data and he/she delegates the ownership responsibility of those resources to users assigned the ResourceOwner role. For example, after creating topics the ClusterAdmin can set ownership to a specific user familiar with the topic data.  **UserAdmin**  Manages role bindings for users and groups in all clusters managed by MDS.  Manages users and groups in a cluster, including the mapping of users and groups to roles. Has no access to any other resources. Typically, users with the UserAdmin role are tasked with setting up access to resources. Users granted this role should be extremely trustworthy because they can grant roles to themselves and others. You can monitor the actions of the UserAdmin using audit logs.  **SecurityAdmin**  Enables management of platform-wide security initiatives.  Sets up security-related features (for example, encryption, tracking of audit logs, and watching for abnormal behavior). Provides a dedicated set of users for the initial setup and ongoing management of security functions.  **AuditAdmin**  Users or groups assigned this role on the MDS cluster and every registered Kafka cluster can manage the audit log configuration using the Confluent Metadata API.  Operator  Provides operational management of clusters and scale applications as needed.  Monitors the health of applications and clusters, including monitoring uptime. This role cannot create applications, nor does it allow you to view or edit the content of the topics. However, you can view what topics and partitions exist.  **ResourceOwner**  Transfers the ownership of critical resources and to scale the ability to manage authorizations for those resources.  Owns the resource and has full access to it, including read, write, and list. ResourceOwner can grant permission to others who need access to resources. Owner cannot change some of the configurations, for example the number of partitions. Must own the resource to grant others access to it. Enables scaling of authorization for critical resources.  **DeveloperRead, DeveloperWrite, DeveloperManage**  Allows developers to drive the implementation of applications they are working on and manage the content.  **RBAC role use cases**  These use cases are based on a new project where security is managed using RBAC predefined roles as follows:   | **Predefined Role** | **Plan** | | --- | --- | | super.user | Sam is granted full access to all project resources and operations. He will create the initial set of roles for the project. | | ResourceOwner | Ryan will own all topics with the prefix finance\_. He can grant others permission to access and use this resource. In this use case, he is the ResourceOwner for the finance topics. | | UserAdmin | Uri will manage the users and groups for the project. | | Operator | Olivia will be responsible for the operational and health management of the platform and applications. | | ClusterAdmin | Cindy is a member of the Kafka cluster central team. | | DeveloperRead, DeveloperWrite, DeveloperManage | David will be responsible for developing and managing the application. | |