CONFLUENT

Role-Based Access Control (RBAC) for Kafka Connect

Table of Contents

Introduction

Role-Based Access Control (RBAC) ensures that only authorized clients have appropriate access to system resources. These resources include those available across services in Confluent Platform:

- Kafka brokers
- Kafka Connect
- KSQL
- Confluent Schema Registry
- Confluent Control Center
- Confluent REST Proxy

RBAC defines granular privileges for users and service accounts to different resources. We will review basic RBAC concepts and then dive into using RBAC specifically with Kafka Connect and connectors.

Target Audience

The reader should understand basic principles of Apache Kafka® and Kafka Connect, and understand how to deploy security across the services in Confluent Platform. Refer to the Connect and security documentation for prerequisite reading.

Role-Based Access Control

Overview

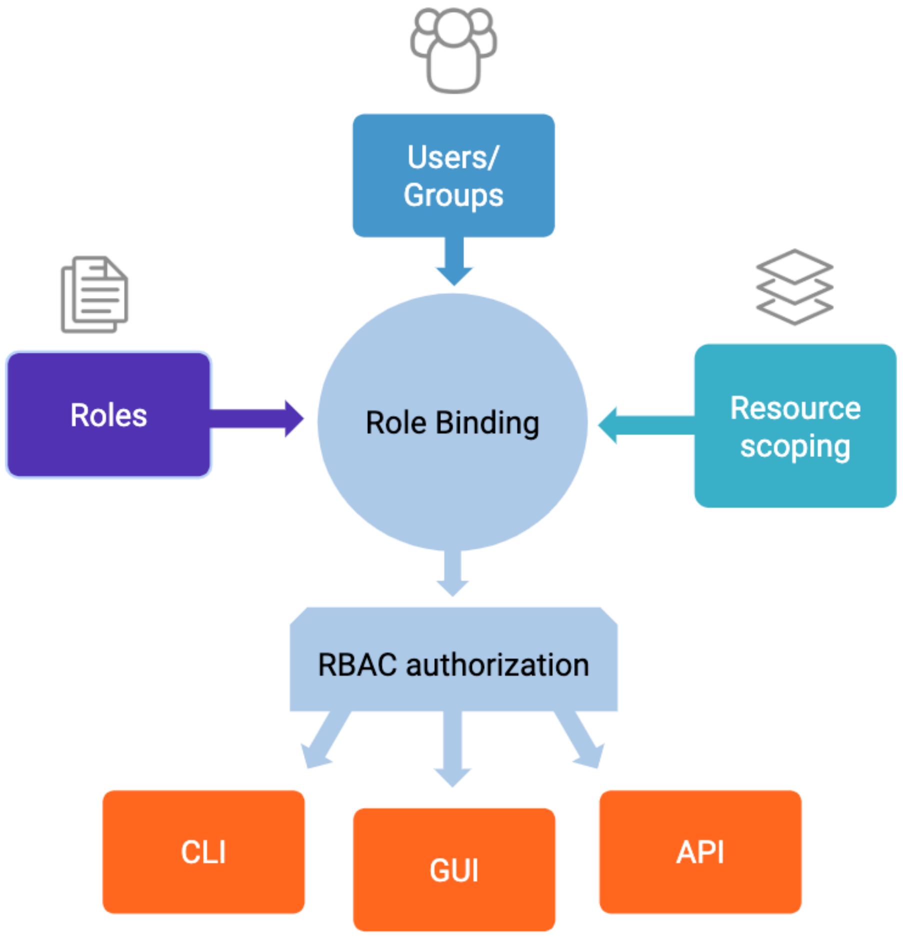
Because Kafka streams events that may contain extremely sensitive data, customers often want to implement very strict rules that control who has access to this data and the services in Confluent Platform.

For basic, simple authorization, a user could define ACLs to allow or deny specific users access to certain resources. However, these ACLs have tangible limitations:

- How do you efficiently manage privileges across an organization with hundreds of users?
- What if you wanted to allow a user to configure a sink connector that would consume from a topic and send the messages to an end system?
- Without RBAC, a user with Confluent Control Center UI access is either a `superUser` or a `readOnly` user with no middle ground for access—how do you restrict a user to read from some topics but not others?
- What if you wanted to reduce your system dependencies on ZooKeeper, which is used to store the ACLs?
- How do you control granular access across all services in the Confluent Platform, including connectors, KSQL, REST Proxy, etc.?
- How do you match the identities of a business workflow that includes both users (humans) and service accounts (applications)?

RBAC leverages predefined role assignments to determine who can access specific resources and what actions an individual user can perform within those resources. An administrator assigns predefined roles to users and groups; each user or group can be assigned multiple roles. Certain privileged users, such as the `UserAdmin` or `SystemAdmin`, assign roles to users and groups, and then map specific resources to those user roles.

The `ResourceOwner` role also has `AlterAccess` permissions on the resources to which they are bound, allowing them to delegate management of permissions to other users. Consequently, a `ResourceOwner` in a finance department can grant department members access to resources, perhaps to topics that use the prefix `finance_`, for example.



User administrators can add LDAP users and groups, making it quicker and easier to configure authentication and authorization centrally 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 granted access to specific resources, via a "role binding." These role bindings can be at the user level or group level. Group-level bindings enable administrators to avoid having to grant explicit access to individual users across every component.

Benefits

RBAC benefits include:

- Robust framework that centralizes authentication and authorization in the Confluent Metadata Service (MDS)
- Consistent behavior across the Confluent Platform such that all services in the event streaming platform can authorize users with the same mechanism
- KSQL supports impersonation for Interactive Queries so that it passes user credentials transparently from the end user to the cluster
- REST Proxy supports impersonation so that it passes user credentials transparently from the end user to the cluster
- Administrators can differentiate and authorize individual roles
- With a unified security CLI, administrators can define RBAC role bindings across the entire Confluent Platform

Confluent Metadata Service

Confluent Metadata Service (MDS) offers a single, centralized configuration context that binds and enforces a Kafka cluster configuration across different resources, such as topics, connectors, and Schema Registry subjects. MDS acts as the central authority for all authorization, and it saves administrators from the complex and time-consuming task of defining and assigning roles for each resource on an individual basis. It can be integrated with LDAP to provide authentication and refreshable bearer tokens for impersonation. MDS is the master record for these role bindings, and all components in the Confluent Platform communicate with MDS to ensure that after a role binding is set, users can't gain access via another API or Confluent Control Center to gain unauthorized access to resources.

Connect Configuration

Before the introduction of RBAC, any user that could authenticate with Kafka Connect could take any action on the connectors or Kafka topic data. However, with RBAC, Connect administrators can grant granular access to users and service accounts, with connector-based authorization and role-based access. They can create multi-tenant Connect clusters that are shared between many departments in an enterprise. Sharing a Connect cluster and scaling it is especially compelling with the improvements made in Confluent Platform 5.3 that enable 10s to 100s of connectors per Kafka Connect cluster.

The rest of this paper describes the workflow for enabling RBAC on Connect. Before proceeding to the sections below, ensure that your Kafka cluster is properly configured for RBAC, and refer to the RBAC documentation as needed.

Connect Worker

The Connect cluster administrator will need to configure all the Kafka Connect workers and start the Connect cluster. There are many configuration parameters that can be set, and this paper focuses on a subset of those required for RBAC.

For consistency with the RBAC demo that uses Hash Login Service with users, the examples here do not have the configuration required to integrate with LDAP. If you need more information on required LDAP configuration, refer to the LDAP Authorizer documentation.

After configuring the Connect worker, create the appropriate role bindings described in the section Personas and Roles.

Refer to an example of the delta Connect configuration required to be added to your existing Connect configuration file. This configuration is part of a demo all running on a localhost (e.g., bootstrap server at `localhost:9092` and MDS at `localhost:8090`), so you'll need to adapt it to your specific environment.

- RBAC authentication and authorization: enable communication between the Connect worker and the Kafka cluster, basic token authentication between the Connect worker and MDS, and authentication for the Connect REST API

```
# Configuration required for communication with the Kafka cluster
bootstrap.servers=localhost:9092
security.protocol=SASL_PLAINTEXT
sasl.mechanism=OAUTHBEARER
sasl.login.callback.handler.class=io.confluent.kafka.clients.plugins.a
uth.token.TokenUserLoginCallbackHandler
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBea
rerLoginModule required username="connect" password="connect1"
metadataServerUrls="http://localhost:8090";

# Enables basic and bearer authentication for requests made to the
worker
rest.servlet.initializor.classes=io.confluent.common.security.jetty.in
itializer.InstallBearerOrBasicSecurityHandler

# The path to a directory containing public keys that should be used
to verify json web tokens during authentication
public.key.path=/tmp/tokenPublicKey.pem

# The location of a running metadata service; used to verify that
requests are authorized by the users that make them
confluent.metadata.bootstrap.server.urls=http://localhost:8090

# Credentials to use when communicating with the MDS; these should
usually match the ones used for communicating with Kafka
confluent.metadata.basic.auth.user.info=connect:connect1
confluent.metadata.http.auth.credentials.provider=BASIC

## REST extensions: RBAC and Secret Registry ##

# Installs the RBAC and Secret Registry REST extensions
rest.extension.classes=io.confluent.connect.security.ConnectSecurityEx
tension,io.confluent.connect.secretregistry.ConnectSecretRegistryExten
sion
```

- Clients embedded within each Connect worker: producer, consumer, and AdminClient

```
producer.security.protocol=SASL_PLAINTEXT
producer.sasl.mechanism=OAUTHBEARER
producer.sasl.login.callback.handler.class=io.confluent.kafka.clients.
plugins.auth.token.TokenUserLoginCallbackHandler
# Intentionally omitting `producer.sasl.jaas.config` to force
connectors to use their own

consumer.security.protocol=SASL_PLAINTEXT
consumer.sasl.mechanism=OAUTHBEARER
consumer.sasl.login.callback.handler.class=io.confluent.kafka.clients.
plugins.auth.token.TokenUserLoginCallbackHandler
# Intentionally omitting `consumer.sasl.jaas.config` to force
connectors to use their own

admin.security.protocol=SASL_PLAINTEXT
admin.sasl.mechanism=OAUTHBEARER
admin.sasl.login.callback.handler.class=io.confluent.kafka.clients.plu
gins.auth.token.TokenUserLoginCallbackHandler
# Intentionally omitting `admin.sasl.jaas.config` to force connectors
to use their own

# Allow producer/consumer/admin client overrides (this enables per-
connector principals)
connector.client.config.override.policy=All
```

- Connect Secret Registry: enable Connect to store encrypted Connect credentials in a topic exposed through a REST API

```
config.providers=secret
config.providers.secret.class=io.confluent.connect.secretregistry.rbac
.config.provider.InternalSecretConfigProvider
config.providers.secret.param.master.encryption.key=password1234

config.providers.secret.param.kafkastore.topic=_secrets
config.providers.secret.param.kafkastore.bootstrap.servers=localhost:9
092
config.providers.secret.param.kafkastore.security.protocol=SASL_PLAINT
EXT
config.providers.secret.param.kafkastore.sasl.mechanism=OAUTHBEARER
config.providers.secret.param.kafkastore.sasl.login.callback.handler.c
lass=io.confluent.kafka.clients.plugins.auth.token.TokenUserLoginCallb
ackHandler
config.providers.secret.param.kafkastore.sasl.jaas.config=org.apache.k
afka.common.security.oauthbearer.OAuthBearerLoginModule required
username="connect" password="connect1"
metadataServerUrls="http://localhost:8090";
```

Connector

Additional connector configuration includes:

- RBAC authentication and authorization: enable basic authentication between the connector and Kafka

Refer to an example of the source connector or sink connector.

```
# Source connector
producer.override.sasl.jaas.config=org.apache.kafka.common.security.oa
uthbearer.OAuthBearerLoginModule required username="connector"
password="connector1" metadataServerUrls="http://localhost:8090";

# Sink connector
consumer.override.sasl.jaas.config=org.apache.kafka.common.security.oa
uthbearer.OAuthBearerLoginModule required username="connector"
password="connector1" metadataServerUrls="http://localhost:8090";
```

This configuration is part of a demo all running on a local host, so you'll need to adapt it to your specific environment.

The connector should be submitted after Kafka Connect has successfully started and with the appropriate role bindings described in the section Personas and Roles.

Personas and Roles

With RBAC, administrators can authorize users or groups to access resources on a per-role basis. A user or group who is assigned a role receives all the privileges of that role, and a user or group can be assigned to multiple roles. Creating role bindings for a group, i.e., using a principal that starts with `Group:`, enables administrators to scale and operationally manage access for large organizations, while role bindings for a user, i.e., using a principal that starts with `User:`, may be useful for specific cases. For consistency in this paper, particularly with the RBAC demo which uses Hash Login with users, the examples here refer to users.

Let's consider the following three Connect personas:

- Connect Cluster Administrator: a principal that represents a Connect worker and is used by that worker to access the Connect cluster group and internal Connect topics
- Connector Submitter: a principal that submits the connector to the Connect cluster
- Connector: a connector principal that needs access to the relevant topics and related resources, e.g., Schema Registry subjects

These personas will need appropriate role bindings in order to work with RBAC, to give access to different resources as described in the following sections. The examples provided in this white paper are general guidelines that you will need to adapt to your specific environment. You should develop a custom plan to address how these personas are appropriated to different groups or users, whether these are all different principals or not, who does the privilege assignment, etc. Review your plan with your RBAC system administrator before creating a Connect cluster or running connectors.

Refer to the Appendix section Cluster IDs to obtain the values for `$KAFKA_CLUSTER_ID`, `$CONNECT_CLUSTER_ID`, and `$SCHEMA_REGISTRY_CLUSTER_ID`, used in the examples below.

Confluent CLI for Role Bindings

To manage RBAC and role bindings, you must use the unified Confluent CLI.

1. Use CLI version v0.128.0 or later. Check the version of the CLI installed by executing the command `confluent version`.

```
confluent - Confluent CLI

Version:     v0.128.0
Git Ref:     0e56172115bf3c2ba7dda6920fb205c9803efa81
Build Date:  2019-07-11T20:51:02Z
Build Host:  semaphore@semaphore-vm
Go Version:  go1.12.5 (darwin/amd64)
Development: false
```

2. Assuming MDS is already running, log in to the MDS endpoint with the appropriate credentials by executing the command `confluent login`. Log in as a user with the `SystemAdmin` role (the configured `super.user`), the `ResourceOwner` role, or the `UserAdmin` role, basically any user with privileges to create the subsequent role bindings shown in this paper.

```
$ confluent login --url http://localhost:8090

Enter your Confluent credentials:
Username: mds
Password:

Logged in as mds
```

3. List the available roles and the associated privileges by executing the command

`confluent iam role list`. For any given role, it shows the permitted operations on the respective resources. (The output is too long to include in the paper).

Connect Cluster Administrator

Role Bindings

The role bindings for the Connect cluster administrator, i.e., `$USER_ADMIN_CONNECT`, are shown below.

1. Grant `$USER_ADMIN_CONNECT` the `ResourceOwner` role to the three "internal" Kafka topics that Connect uses to store information about connector tasks, offsets, and statuses. With this role, the user owns those topics and can also grant other users access to those topics.

    - `config.storage.topic`: a topic for storing connector and task configurations, in which the default value is `connect-configs`
    - `offset.storage.topic`: a topic for storing offsets, in which the default value is `connect-offsets`
    - `status.storage.topic`: a topic for storing connector statuses, in which the default value is `connect-statuses`

```
# connect-configs
confluent iam rolebinding create --principal
User:$USER_ADMIN_CONNECT --role ResourceOwner --resource
Topic:connect-configs --kafka-cluster-id $KAFKA_CLUSTER_ID

# connect-offsets
confluent iam rolebinding create --principal
User:$USER_ADMIN_CONNECT --role ResourceOwner --resource
Topic:connect-offsets --kafka-cluster-id $KAFKA_CLUSTER_ID

# connect-statuses
confluent iam rolebinding create --principal
User:$USER_ADMIN_CONNECT --role ResourceOwner --resource
Topic:connect-statuses --kafka-cluster-id $KAFKA_CLUSTER_ID
```

2. Grant `$USER_ADMIN_CONNECT` the `ResourceOwner` role for the Connect cluster group, defined by the parameter `group.id`, which by default is configured to `connect-cluster`. With this role binding, the user is the owner for that group.

   ```
   confluent iam rolebinding create --principal
   User:$USER_ADMIN_CONNECT --role ResourceOwner --resource
   Group:connect-cluster --kafka-cluster-id $KAFKA_CLUSTER_ID
   ```

3. Optional if using Connect Secret Registry): grant `$USER_ADMIN_CONNECT` the `ResourceOwner` role to the resources required for Connect Secret Registry.

   - Grant `ResourceOwner` for the topic defined by `config.providers.secret.param.kafkastore.topic`, which by default is configured to `_secrets`. With this role binding, the user owns that topic and can create it.

     ```
     confluent iam rolebinding create --principal
     User:$USER_ADMIN_CONNECT --role ResourceOwner --resource
     Topic:_secrets --kafka-cluster-id $KAFKA_CLUSTER_ID
     ```

   - Grant `ResourceOwner` for the consumer group defined by `config.providers.secret.param.secret.registry.group.id`, which by default is configured to `secret-registry`. With this role binding, the user is the owner for that consumer group, which is used to read from the topic above.

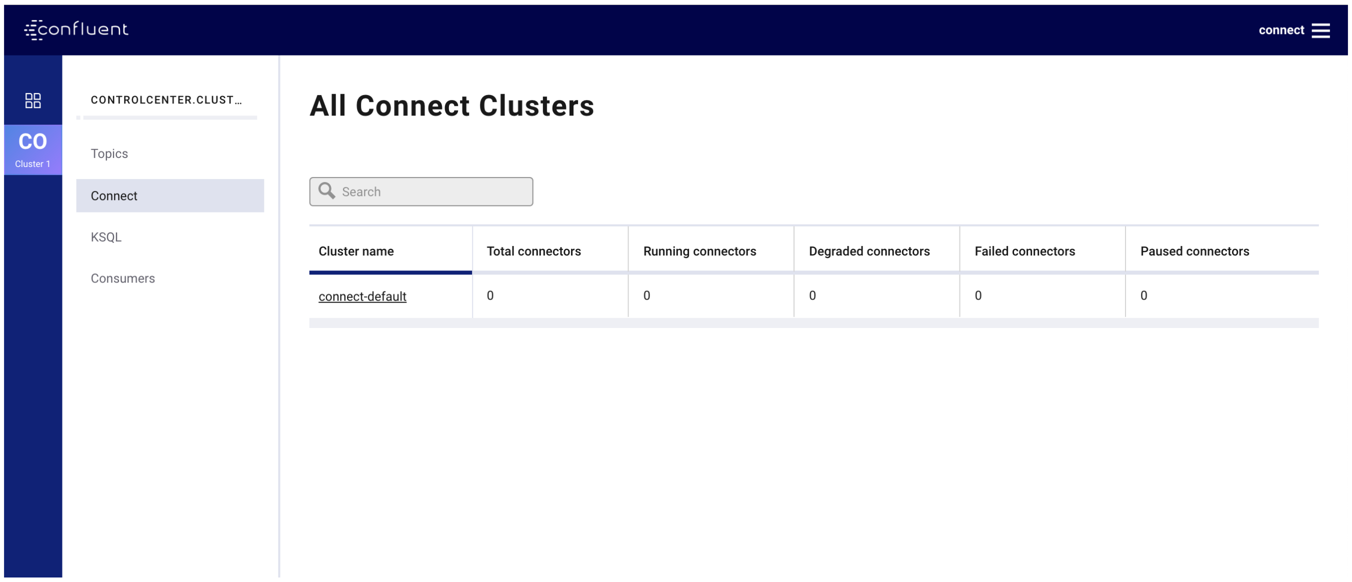
     ```
     confluent iam rolebinding create --principal
     User:$USER_ADMIN_CONNECT --role ResourceOwner --resource
     Group:secret-registry --kafka-cluster-id $KAFKA_CLUSTER_ID
     ```

4. Grant `$USER_ADMIN_CONNECT` the `SecurityAdmin` role to the Connect cluster. With this role binding, the user can make requests to MDS to determine if the connector user is authorized to perform required operations. Notice that the following command does not require a `--resource` argument, because the `SecurityAdmin` role has a cluster-level scope (the roles `SystemAdmin`, `UserAdmin`, `ClusterAdmin`, and `Operator` also have cluster-level scope).

```
confluent iam rolebinding create --principal
User:$USER_ADMIN_CONNECT --role SecurityAdmin --kafka-cluster-id
$KAFKA_CLUSTER_ID --connect-cluster-id $CONNECT_CLUSTER_ID
```

RBAC and Control Center

Connectors can now be managed through Confluent Control Center, while honoring access that has been granted. When logging into the Control Center UI with appropriate credentials, `$USER_ADMIN_CONNECT` has access to the connector cluster in the UI. This user has access only to the Connect worker information but not the connectors, topics, KSQL, consumers, etc. Other users without the appropriate role bindings do not have access to this Connect worker information.



All Connect Clusters

Cluster name	Total connectors	Running connectors	Degraded connectors	Failed connectors	Paused connectors
connect-default	0	0	0	0	0

Likewise, other users without the appropriate permissions do not have access to the Connect cluster.

Connector Submitter

Role Bindings

The role bindings for the person who submits a connector to the Connect cluster, i.e., `$USER_CONNECTOR_SUBMITTER`, are shown below.

5. Grant `$USER_CONNECTOR_SUBMITTER` the `ResourceOwner` role for the connector defined by `$CONNECTOR_NAME`. With this role binding, the user is the owner of the connector and can grant others access to this connector.

   ```
   confluent iam rolebinding create --principal
   User:$USER_CONNECTOR_SUBMITTER --role ResourceOwner --resource
   Connector:$CONNECTOR_NAME --kafka-cluster-id $KAFKA_CLUSTER_ID
   --connect-cluster-id $CONNECT_CLUSTER_ID
   ```

If a user who is not authorized tries to submit the connector, the Connect REST API will return an HTTP error `403 Forbidden` with the text `Unauthorized operation`.

Connector

Role Bindings

The role bindings for the connector, i.e., `$USER_CONNECTOR`, are shown below.

6. Grant `$USER_CONNECTOR` the `ResourceOwner` role for the data. Data means two resources: topic and Schema Registry subject.

   ◦ The topic in the Kafka cluster is defined by `$TOPIC2_AVRO`. With the

`ResourceOwner` role binding, the connector can create the topic. If the topic already exists, you may consider creating a more restricted privilege like `DeveloperWrite` for a source connector (or `DeveloperRead` for a sink connector).

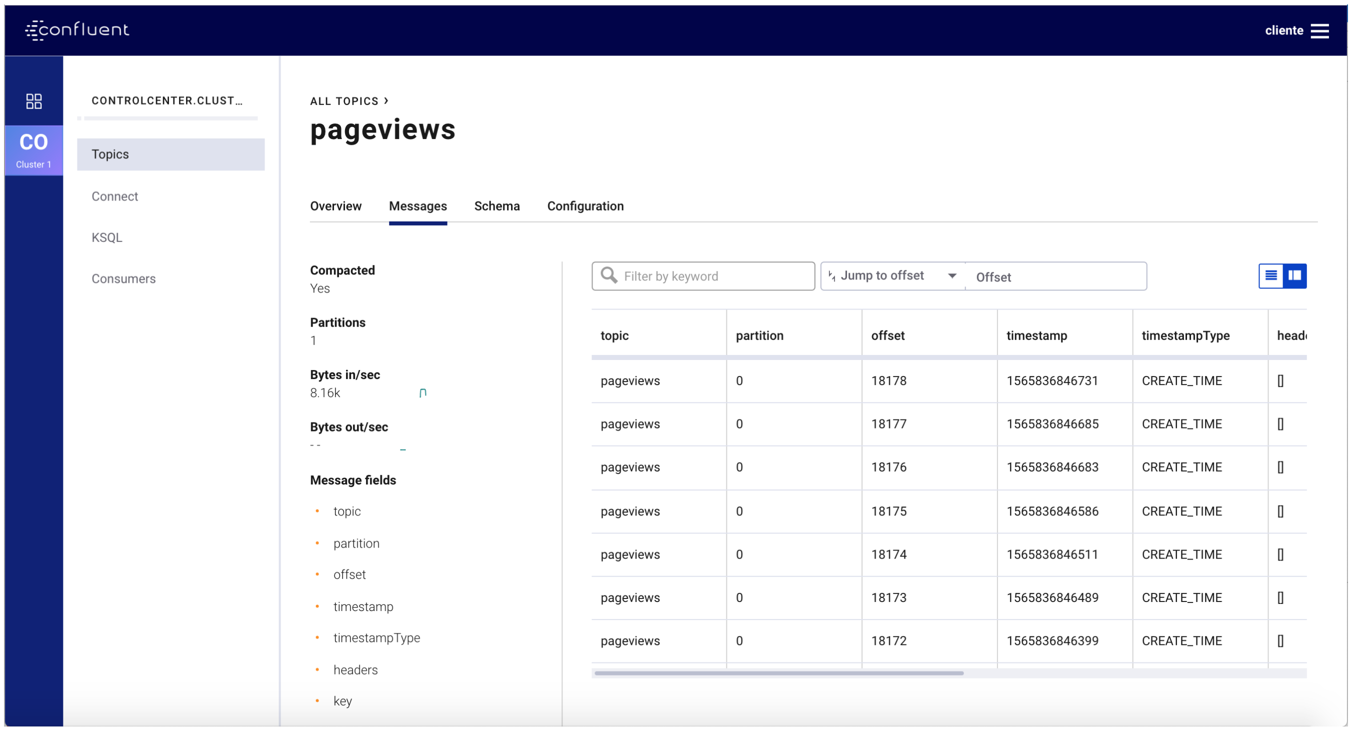
```
confluent iam rolebinding create --principal
User:$USER_CONNECTOR --role ResourceOwner --resource
Topic:$TOPIC2_AVRO --kafka-cluster-id $KAFKA_CLUSTER_ID
```

- Optional if using Confluent Schema Registry: The subject is defined by `${TOPIC2_AVRO}-key` and/or `${TOPIC2_AVRO}-value` (it assumes that the subject name strategy is `TopicNameStrategy`). With the `ResourceOwner` role binding, the connector can create the subject. If the topic already exists, you may consider creating a more restricted privilege like `DeveloperWrite` for a source connector (or `DeveloperRead` for a sink connector).

```
confluent iam rolebinding create --principal
User:$USER_CONNECTOR --role ResourceOwner --resource
Subject:${TOPIC2_AVRO}-value --kafka-cluster-id
$KAFKA_CLUSTER_ID --schema-registry-cluster-id
$SCHEMA_REGISTRY_CLUSTER_ID
```

RBAC and Control Center

The topic data and schema related to this connector can now be managed through Confluent Control Center, while honoring access that has been granted. When logging into the Control Center UI with appropriate credentials, `$USER_CONNECTOR` has access to the topic `$TOPIC2_AVRO` in the UI (in the diagram below, `$TOPIC2_AVRO=pageviews`). Other users without the appropriate role bindings do not have access to this topic data.



Compacted
Yes

Partitions
1

Bytes in/sec
8.16k

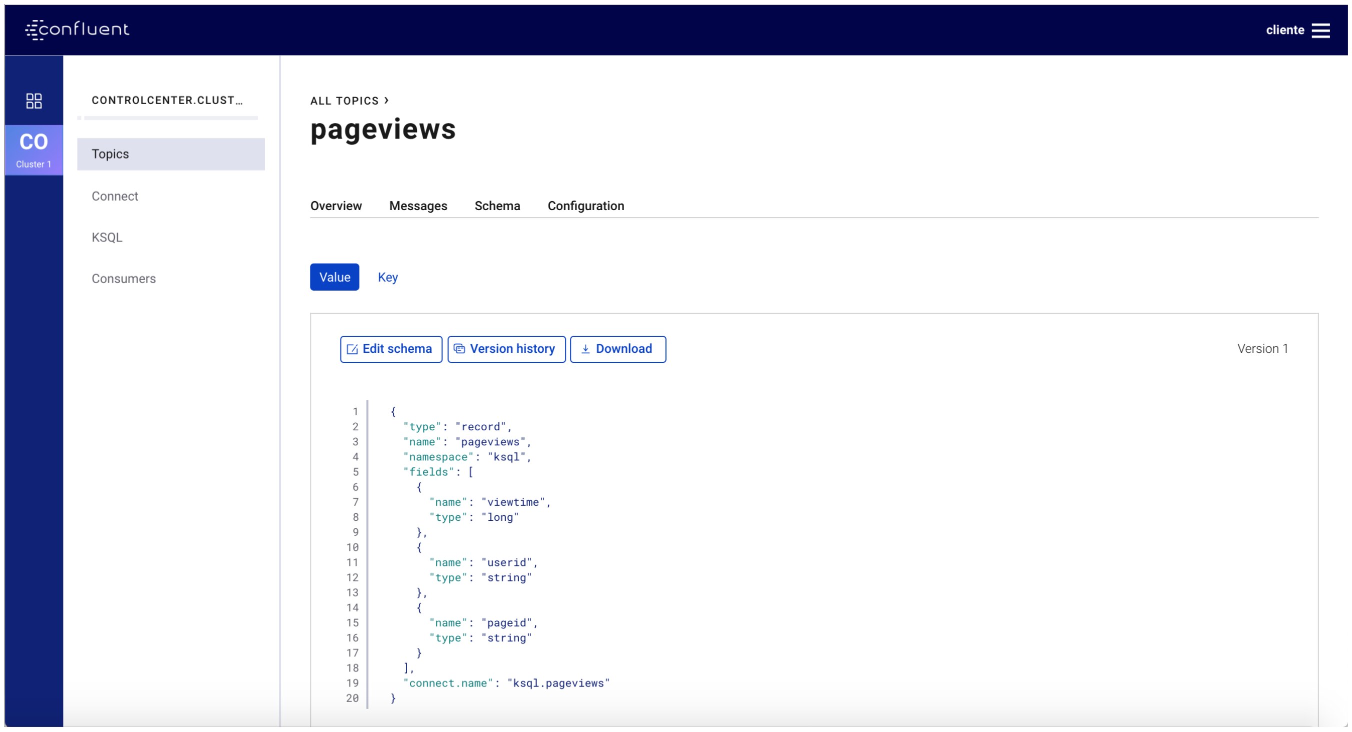
Bytes out/sec
--

Message fields

- topic
- partition
- offset
- timestamp
- timestampType
- headers
- key

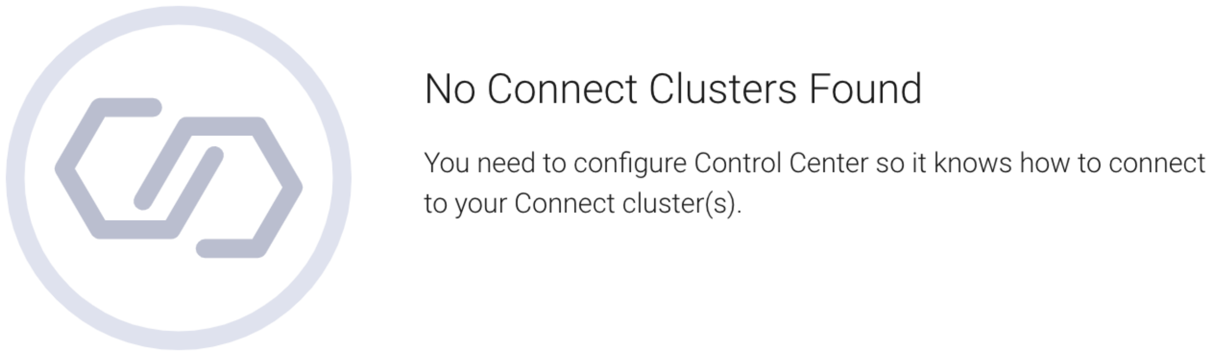
topic	partition	offset	timestamp	timestampType	head
pageviews	0	18178	1565836846731	CREATE_TIME	[]
pageviews	0	18177	1565836846685	CREATE_TIME	[]
pageviews	0	18176	1565836846683	CREATE_TIME	[]
pageviews	0	18175	1565836846586	CREATE_TIME	[]
pageviews	0	18174	1565836846511	CREATE_TIME	[]
pageviews	0	18173	1565836846489	CREATE_TIME	[]
pageviews	0	18172	1565836846399	CREATE_TIME	[]

The user also has access to the Schema Registry subject `${TOPIC2_AVRO}-value` (in the diagram below, the schema is shown for the topic value where `$TOPIC2_AVRO=pageviews`). Other users without the appropriate role bindings do not have access to this Schema Registry subject.



```
{
  "type": "record",
  "name": "pageviews",
  "namespace": "ksql",
  "fields": [
    {
      "name": "viewtime",
      "type": "long"
    },
    {
      "name": "userid",
      "type": "string"
    },
    {
      "name": "pageid",
      "type": "string"
    }
  ],
  "connect.name": "ksql.pageviews"
}
```

Finally, as expected, this user does not have access to the connector itself, KSQL, consumers, etc. For example, when trying to view connectors, this is what this user would see without appropriate role bindings.



No Connect Clusters Found

You need to configure Control Center so it knows how to connect to your Connect cluster(s).

Role Binding Summary

The summary of role bindings required to enable Connect for RBAC is shown below. Note that this is just a generic example that you should adapt for your specific environment.

You may configure more restricted privileges if a topic or subject already exists.

```
# Connect Admin
confluent iam rolebinding create --principal User:$USER_ADMIN_CONNECT
--role ResourceOwner --resource Topic:connect-configs --kafka-cluster
-id $KAFKA_CLUSTER_ID
confluent iam rolebinding create --principal User:$USER_ADMIN_CONNECT
--role ResourceOwner --resource Topic:connect-offsets --kafka-cluster
-id $KAFKA_CLUSTER_ID
confluent iam rolebinding create --principal User:$USER_ADMIN_CONNECT
--role ResourceOwner --resource Topic:connect-statuses --kafka-cluster
-id $KAFKA_CLUSTER_ID
confluent iam rolebinding create --principal User:$USER_ADMIN_CONNECT
--role ResourceOwner --resource Group:connect-cluster --kafka-cluster
-id $KAFKA_CLUSTER_ID
confluent iam rolebinding create --principal User:$USER_ADMIN_CONNECT
--role ResourceOwner --resource Topic:_secrets --kafka-cluster-id
$KAFKA_CLUSTER_ID
confluent iam rolebinding create --principal User:$USER_ADMIN_CONNECT
--role ResourceOwner --resource Group:secret-registry --kafka-cluster
-id $KAFKA_CLUSTER_ID
confluent iam rolebinding create --principal User:$USER_ADMIN_CONNECT
--role SecurityAdmin --kafka-cluster-id $KAFKA_CLUSTER_ID --connect
-cluster-id $CONNECT_CLUSTER_ID

# Connector Submitter
confluent iam rolebinding create --principal
User:$USER_CONNECTOR_SUBMITTER --role ResourceOwner --resource
Connector:$CONNECTOR_NAME --kafka-cluster-id $KAFKA_CLUSTER_ID
--connect-cluster-id $CONNECT_CLUSTER_ID

# Connector
confluent iam rolebinding create --principal User:$USER_CONNECTOR
--role ResourceOwner --resource Topic:$TOPIC2_AVRO --kafka-cluster-id
$KAFKA_CLUSTER_ID
confluent iam rolebinding create --principal User:$USER_CONNECTOR
--role ResourceOwner --resource Subject:${TOPIC2_AVRO}-value --kafka
-cluster-id $KAFKA_CLUSTER_ID --schema-registry-cluster-id
$SCHEMA_REGISTRY_CLUSTER_ID
```

Summary

This white paper applies basic RBAC concepts to Kafka Connect. You should now be able to apply these concepts to your deployment of the Confluent Platform.

As next steps, you can follow up on RBAC with these resources:

- RBAC demo
- Deploy RBAC across the entire Confluent Platform
- Best practices for taking Kafka Connect to production
- Download the Confluent Platform: RBAC and MDS are commercial features of the Confluent Platform
- Install Confluent CLI: the CLI is used for connecting to MDS and managing role bindings

---

Confluent Platform is the leading distribution of Apache Kafka, containing all of Kafka's capabilities and enhancing it with integrated, tested, and packaged features that make architecting and managing large-scale event streaming pipelines easier and more reliable.

Visit www.confluent.io/download to download the latest version of the Confluent Platform.

Appendix

Cluster IDs

Kafka Cluster ID

Get the Kafka cluster ID from ZooKeeper, which is running at `localhost:2181` in the example below. The user must have access to ZooKeeper to get the cluster ID.

```
zookeeper-shell localhost:2181 get /cluster/id | grep version | jq -r
.id
```

Connect Cluster ID

Get the Kafka Connect cluster ID from the Connect REST endpoint, which is running at `localhost:8083` in the example below. The user must be the Connect administrator (`$USER_ADMIN_CONNECT`) to get the cluster ID.

```
curl -u $USER_ADMIN_CONNECT:$USER_ADMIN_CONNECT_PASSWORD
http://localhost:8083/permissions | jq -r '.scope.clusters."connect-
cluster"'
```

Schema Registry Cluster ID

Get the Schema Registry cluster ID from the Schema Registry REST endpoint, which is running at `localhost:8081` in the example below. The user must be the Schema Registry administrator (`$USER_ADMIN_SR`) to get the cluster ID.

```
curl -u $USER_ADMIN_SR:$USER_ADMIN_SR_PASSWORD
http://localhost:8081/permissions | jq -r '.scope.clusters."schema-
registry-cluster"'
```