Cloudera Streams Messaging Operator for Kubernetes 1.6.0

Kafka Connect Operations

Date published: 2024-06-11
Date modified: 2026-01-27

https://docs.cloudera.com/

Legal Notice

Contents

Managing connectors

Learn about deploying and managing Kafka Connect connectors using KafkaConnector resources. Deploying and managing connectors with KafkaConnector resources is the recommended method by Cloudera for managing connectors.

To deploy and manage connectors in Cloudera Streams Messaging Operator for Kubernetes, you use `KafkaConnector` resources. `KafkaConnector` resources describe instances of connectors and offer a Kubernetes-native approach to connector management. You create a `KafkaConnector` resource to deploy a connector and manage it by updating the `KafkaConnector` resource. The Strimzi Cluster Operator updates configurations and manages the lifecycle of the connectors.

Enabling KafkaConnector resources

`KafkaConnector` resources are not enabled by default for Kafka Connect clusters. To use `KafkaConnector` resources, the `KafkaConnect` resource used for deploying your Kafka Connect cluster must have the strimzi.io/use-connector-resources annotation set to true.

Full resource examples provided by Cloudera in this documentation as well as on the Cloudera Archive have the strimzi.io/use-connector-resources annotation set to true.

Rest API usage

Kafka Connect offers a REST API which is also available for use when you deploy a cluster in Cloudera Streams Messaging Operator for Kubernetes. However, usage of the API is not recommended by Cloudera, and should be limited to select use cases.

Related Information

Enabling KafkaConnector resources

Using the Kafka Connect REST API

Deploying connectors

Learn how to deploy Kafka Connect connectors using KafkaConnector resources.

Before you begin

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.
- Ensure that a Kafka Connect cluster is available and running. See Deploying Kafka Connect clusters .
- Ensure that the connectors you plan to deploy are installed in the Kafka Connect cluster. That is, the Kafka image used by your Kafka Connect cluster includes the required plugin artifacts. See Installing Kafka Connect connector plugins.
- Ensure that the strimzi.io/use-connector-resources annotation is set to true in the Kafka Connect cluster. See Enabling KafkaConnector resources.
- Ensure that you know the namespace where the Kafka Connect cluster is deployed. Connectors must be deployed in the same namespace as the Kafka Connect cluster they are deployed in.
- The example resource in these steps demonstrates deployment of the `MirrorHeartbeatConnector`, which is installed by default in all Kafka Connect clusters.

Procedure

1. Create a YAML configuration containing the manifest for your `KafkaConnector` resource.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
```

```
metadata:
  name: my-heartbeats-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.mirror.MirrorHeartbeatConnector
  tasksMax: 1
  config:
    source.cluster.alias: source
    target.cluster.bootstrap.servers: my-cluster-kafka-bootstrap:9092
```

- metadata.name specifies the name of the connector.
- labels.strimzi.io/cluster specifies the name of the Kafka Connect cluster that this connector is deployed in.
- spec.class specifies the fully qualified name of the connector plugin implementation. The connector plugin must be installed in Kafka Connect.
- spec.taskMax specifies the maximum number of tasks this connector is allowed to create. This is an upper limit. The connector might not create the maximum number of allowed tasks.
- spec.config includes the configuration of the connector.

You can find additional information about supported properties in the Strimzi API Reference.

2. Deploy the resource.

```
kubectl apply --filename [***YAML CONFIG***] --namespace [***NAMESPACE***]
```

Ensure that you deploy the connector in the same namespace where the Kafka Connect cluster is running.

3. Validate that the `KafkaConnector` resource is created and ready.

```
kubectl get kafkaconnectors --namespace [***NAMESPACE***]
```

The output should list your `KafkaConnector` resource.

```
NAME                         CLUSTER              CONNECTOR CLASS
                                   MAX TASKS   READY
#...
my-heartbeats-connector   my-connect-cluster   org.apache.kafka.connect
.mirror.MirrorHeartbeatConnector   2           True
```

Related Information

KafkaConnectorSpec schema reference | Strimzi API Reference

Deleting connectors

You can delete a connector by deleting its corresponding KafkaConnector resource with kubectl delete.

```
kubectl delete kafkaconnector [***CONNECTOR NAME***] \
  --namespace [***NAMESPACE***]
```

Stopping, pausing, and resuming connectors

You can stop, pause, and resume connectors by configuring the spec.state property in the KafkaConnector resource.

Before you begin

```
#...
kind: KafkaConnector
spec:
```

```
  state: stopped
```

You can set spec.state to running, paused, or stopped. You resume stopped or paused connectors by setting their state to running. The default value is running.

Restarting connectors

You can restart a connector by annotating the KafkaConnector resource with strimzi.io/restart=”true”.

```
kubectl annotate kafkaconnector [***CONNECTOR NAME***] \
  --namespace [***NAMESPACE***] \
  strimzi.io/restart="true"
```

If the annotation is added, the Strimzi Cluster Operator immediately restarts the connector. If the initial restart fails for any reason, the Strimzi Cluster Operator attempts to restart the connector once per reconciliation loop. The annotation is automatically removed if the restart is successful.

Checking connector task IDs

You can check the task IDs of a connector by describing the KafkaConnector resource.

```
kubectl describe kafkaconnector [***CONNECTOR NAME***] \
  --namespace [***NAMESPACE***]
```

You can find the task IDs in Connector Status. Connector Status also includes the task state and the worker ID.

```
#...
Connector Status:
    Connector:
      State:      RUNNING
      worker_id:  my-connect-cluster-connect-0.my-connect-cluster-connect.
connect.svc:8083
    Name:         my-source-connector
    Tasks:
      Id:               0
      State:            RUNNING
      worker_id:        my-connect-cluster-connect-0.my-connect-cluster-conn
ect.connect.svc:8083
```

Restarting connector tasks

You can restart a connector task by annotating the KafkaConnector resource with strimzi.io/restart-task=”[***TASK ID***]”.

```
kubectl annotate KafkaConnector [***CONNECTOR NAME***] \
  --namespace [***NAMESPACE***] \
  strimzi.io/restart-task="[***TASK ID***]"
```

If the annotation is added, the Strimzi Cluster Operator immediately restarts the connector task. If the initial restart fails for any reason, the Strimzi Cluster Operator attempts to restart the connector task once per reconciliation loop. The annotation is automatically removed if the restart is successful.

Listing connector offsets

You can list connector offsets by adding the spec.listOffsets property to the KafkaConnector resource. After the property is added, you annotate the resource with strimzi.io/connector-offsets="list". The annotation triggers the Strimzi Cluster Operator to write connector offsets to the ConfigMap specified in spec.listOffsets.

Procedure

1. Configure your `KafkaConnector` resource to include the spec.listOffsets property.

```
#...
kind: KafkaConnector
spec:
  listOffsets:
    toConfigMap:
      name: [***CONFIGMAP NAME***]
```

If the `ConfigMap` you specify does not exist, the Strimzi Cluster Operator creates it when you list connector offsets using the strimzi.io/connector-offsets="list" annotation.

Note: If the `ConfigMap` is created by the Strimzi Cluster Operator, the metadata.ownerReferences property is set to point to the `KafkaConnector` resource. This means that if at a later point you delete the `KafkaConnector` resource, the `ConfigMap` is deleted as well. If you create the `ConfigMap` in advance and set metadata.ownerReferences to a custom value, the Strimzi Cluster Operator does not override the value when it writes the connector offsets to the `ConfigMap`.

2. List connector offsets by annotating your `KafkaConnector` resource with strimzi.io/connector-offsets="list".

```
kubectl annotate kafkaconnector [***CONNECTOR NAME***] \
  --namespace [***NAMESPACE***] \
  strimzi.io/connector-offsets="list"
```

Once the annotation is applied, the connector offsets are written to the `ConfigMap` specified in the spec.listOff sets property of the `KafkaConnector` resource. Afterward, the Strimzi Cluster Operator automatically removes the annotation.

3. View connector offsets.

To do this, view the `ConfigMap` specified in the spec.listOffsets property of the `KafkaConnector` resource.

```
kubectl get configmap [***CONFIGMAP NAME***] \
  --namespace [***NAMESPACE***] \
  --output yaml
```

The offsets are written to data[“offsets.json”] in the `ConfigMap`. For example:

For Sink connector

```
#...
kind: ConfigMap
metadata:
  labels:
    strimzi.io/cluster: my-connect-cluster
  ownerReferences:
    - apiVersion: kafka.strimzi.io/v1
      blockOwnerDeletion: false
      controller: false
      kind: KafkaConnector
      name: my-sink-connector
      uid: 76273d69-ba9a-4b60-a532-c546a44dcdc5
  resourceVersion: "12345"
```

```
  uid: aad947b4-f10f-4a65-a671-2e1e51cb29ad
data:
  offsets.json: |-
    {
      "offsets": [
        {
          "partition": {
            "kafka_topic": "my-topic",
            "kafka_partition": 4
          },
          "offset": {
            "kafka_offset": 6
          }
        }
      ]
    }
```

For Source connector

```
#...
kind: ConfigMap
metadata:
  labels:
    strimzi.io/cluster: my-connect-cluster
  ownerReferences:
    - apiVersion: kafka.strimzi.io/v1
      blockOwnerDeletion: false
      controller: false
      kind: KafkaConnector
      name: my-source-connector
      uid: 2298f93d-5c3d-4a8f-b611-e27b08164a73
  resourceVersion: "54321"
  uid: 6eeeeb55-d593-4f08-be25-f3ca480f41c2
data:
  offsets.json: |-
    {
      "offsets" : [ {
        "partition" : {
          "filename" : "/data/myfile.txt"
        },
        "offset" : {
          "position" : 23582
        }
      } ]
    }
```

Note: Sink connectors use the standard consumer offset mechanism of Kafka, while source connectors store offsets in a custom format within a Kafka topic. This means that for sink connectors the format will always be the same, while for source connectors it will vary from connector to connector.

Related Information

ListOffsets schema reference | Strimzi API reference

Listing connector offsets | Strimzi

Altering connector offsets

You can alter connector offsets by stopping your connector and adding the spec.alterOffsets property to the KafkaConnector resource. The alterOffsets property specifies a ConfigMap that includes your offset changes. Following configuration, you annotate the resource with strimzi.io/connector-offsets=”alter”. The annotation triggers

the Strimzi Cluster Operator to update the connector offsets. You alter connector offsets if you want to skip or reprocess certain records.

Before you begin

List the offsets of the connector. Listing offsets generates a `ConfigMap`, which you use to input your changes. Note down the name of the `ConfigMap` as you will need to edit its contents and specify it in the `KafkaConnector` resource.

Procedure

1. Configure your `KafkaConnector` resource to include the spec.alterOffsets property. In addition, stop the connector by setting spec.state to stopped.

```
#...
kind: KafkaConnector
spec:
  state: stopped
  alterOffsets:
    fromConfigMap:
      name: [***CONFIGMAP NAME***]
```

   The `ConfigMap` you specify in alterOffsets is the `ConfigMap` that you generated by listing connector offsets.

2. Edit the `ConfigMap` to include your connector offset changes.

   Make your changes in data[“offsets.json”]. For example, assume you want to reprocess some records, in this case you reset the offset position to an appropriate value.

   For Current offset position

```
#...
kind: ConfigMap
metadata:
  ownerReferences:
    - apiVersion: kafka.strimzi.io/v1
      blockOwnerDeletion: false
      controller: false
      kind: KafkaConnector
      name: my-sink-connector
      uid: 8b6d745f-5137-4524-9cc0-c67847f319d8
  resourceVersion: "19343"
  uid: da5268a3-9b94-4752-a997-15883fee105c
data:
  offsets.json: |-
    {
      "offsets": [
        {
          "partition": {
            "kafka_topic": "test-topic",
            "kafka_partition": 2
          },
          "offset": {
            "kafka_offset": 10
          }
        }
      ]
    }
```

   For Altered offset position

```
#...
kind: ConfigMap
```

```
metadata:
  ownerReferences:
    - apiVersion: kafka.strimzi.io/v1
      blockOwnerDeletion: false
      controller: false
      kind: KafkaConnector
      name: my-sink-connector
      uid: 8b6d745f-5137-4524-9cc0-c67847f319d8
  resourceVersion: "19343"
  uid: da5268a3-9b94-4752-a997-15883fee105c
data:
  offsets.json: |-
    {
      "offsets": [
        {
          "partition": {
            "kafka_topic": "test-topic",
            "kafka_partition": 2
          },
          "offset": {
            "kafka_offset": 6
          }
        }
      ]
    }
```

Note: Sink connectors use the standard consumer offset mechanism of Kafka, while source connectors store offsets in a custom format within a Kafka topic. This means that for sink connectors the format will always be the same, while for source connectors it will vary from connector to connector.

3. Alter connector offsets by annotating your `KafkaConnector` resource with strimzi.io/connector-offsets="alter".

```
kubectl annotate kafkaconnector [***CONNECTOR NAME***] \
  --namespace [***NAMESPACE***] \
  strimzi.io/connector-offsets="alter"
```

The annotation is automatically removed by the Strimzi Cluster Operator after connector offsets are successfully updated.

4. Verify your changes by listing connector offsets.
5. Start the connector by setting spec.state to running in the `KafkaConnector` resource.

```
#...
kind: KafkaConnector
spec:
  state: running
```

Related Information

AlterOffsets schema reference | Strimzi API reference

Altering connector offsets | Strimzi

Resetting connector offsets

You can reset connector offsets by stopping your connector and annotating the KafkaConnector resource with strimzi.io/connector-offsets="reset". After offsets are reset, you restart your connector.

Before you begin

Review Listing connector offsets on page 7. In the following steps, you list connector offsets to verify changes.

Procedure

1. Stop the connector by setting spec.state in the `KafkaConnector` resource to stopped.

```
#...
kind: KafkaConnector
spec:
  state: stopped
```

2. Reset connector offsets by annotating the `KafkaConnector` resource with strimzi.io/connector-offsets="reset".

```
kubectl annotate kafkaconnector [***CONNECTOR NAME***] \
  --namespace [***NAMESPACE***] \
  strimzi.io/connector-offsets="reset"
```

   The annotation is automatically removed by the Strimzi Cluster Operator after connector offsets are successfully reset.

3. Verify your changes by listing connector offsets.

   If connector offsets are successfully reset, data[“offsets.json”] will be empty. For example.

```
#...
kind: ConfigMap
metadata:
data:
  offsets.json: |-
    {
      "offsets" : []
    }
```

4. Start the connector by setting spec.state to running in the `KafkaConnector` resource.

```
#...
kind: KafkaConnector
spec:
  state: running
```

Related Information

Resetting connector offsets | Strimzi

Configuring connectors

Learn how you configure connectors with KafkaConnector resources. Configuring connectors with Kafkaconnector resources is the recommended method by Cloudera for configuring connectors.

Connectors that you deploy using `KafkaConnector` resources are configured with their corresponding `KafkaConnector` resource. When you make a configuration update, the Strimzi Cluster Operator, which manages the lifecycle of connectors, updates configurations.

Connector properties that you configure in your `KafkaConnector` resources largely depend on the specific connector you are using. This is because most configuration properties are connector specific. Always consult the documentation of the specific connector that you want to configure.

Related Information

Managing connectors

Enabling KafkaConnector resources

Using the Kafka Connect REST API

Configuring automatic restarts for connectors

You can enable the automatic restart of failed connectors and tasks with spec.autoRestart.enabled. Additionally, you can configure the maximum number of allowed automatic restarts with spec.autoRestart.maxRestarts. Both properties are configured in the KafkaConnector resource.

```
#...
kind: KafkaConnector
spec:
  autoRestart:
    enabled: true
    maxRestarts: 10
```

Note: By default the number of allowed restarts is infinite.

Configuring connector properties

You configure a connector by specifying connector properties in spec.config of the KafkaConnector resource.

This is an example configuration for a `FilestreamSourceConnector` that reads the Apache Kafka license file and produces the contents of the file to a specified topic.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
    file: "/opt/kafka/LICENSE"
    topic: my-topic
```

Connector configurations you specify in spec.config will largely depend on the connector itself. Consult the documentation for your specific connector to learn what properties it supports.

Note: You can load external configuration values from external sources by using configuration providers. This can be useful for loading credentials like passwords, access keys, or any other sensitive information.

Related Information

Configuration providers

Loading configuration values from external sources | Strimzi

Configuring client overrides in connectors

Learn how to configure client configuration overrides in connectors. Configuring overrides enables you to fine-tune your connector configuration.

Important: Connector configuration overrides must be enabled in your Kafka Connect cluster (`KafkaConnector` resource). Otherwise, the client overrides you specify in your connector are disregarded. See Configuring connector configuration override policy.

The Kafka Connect framework manages Kafka clients (producers, consumers, and admin clients) used by connectors and tasks. By default, these clients use worker-level properties. In some use-cases, you might want to fine-tune these properties with overrides.

For example, you can use overrides to configure unique authentication credentials for your connectors. Or you can use overrides to fine-tune connector performance.

Overrides take precedence over worker-level properties. You configure overrides with the following prefixes.

- producer.override. – used for overriding producer properties.
- consumer.override. – used for overriding consumer properties.
- admin.override. – used for overriding admin client properties.

To configure an override, add any supported consumer, producer, or admin client property with the corresponding prefix to spec.config in your `KafkaConnector` resource.

```
#...
kind: KafkaConnector
spec:
  config:
    producer.override.batch.size: 1234
```

This example configures the producer batch size, which is a typical property you tweak when fine-tuning connectors and clients for performance.

Related Information

Producer Configs | Kafka

Consumer Configs | Kafka

Admin Configs | Kafka

Using the Kafka Connect REST API

Kafka Connect offers a REST API that you can use to manage and monitor connectors. Learn about the REST API, available endpoints, and recommended use. Additionally learn about connect_shell.sh, which is a command line tool that you can use to establish quick access to the REST API.

The Kafka Connect REST API is available as a `ClusterIP` type Kubernetes service. The service is named *[***CONNECT CLUSTER NAME***]*-connect-api. Its default port is 8083.

*[***CONNECT CLUSTER NAME***]* is the name of your Kafka Connect cluster. The name is specified in the meta data.name property of the `KafkaConnect` resource used to deploy the cluster. The service is created when you deploy the cluster.

The REST API offers various endpoints and operations that you can use to manage (create, update, delete) as well as to monitor the connectors running in your Kafka Connect cluster. You can find a comprehensive reference in the *Kafka Connect Rest API reference*.

API access and security

By default the Kafka Connect API is only accessible from within the Kubernetes Cluster. Additionally, the default network policies only allow access by the Strimzi Cluster Operator and Kafka Connect pods. This is done because the REST API is insecure by default and it cannot be secured. As a result Cloudera recommends the following:

- Do not expose the REST API to applications running outside the Kubernetes cluster.
- Use `KafkaConnector` resources to manage connectors instead of the REST API.

Recommended use

Cloudera recommends that you use the API selectively for specific use-cases. In general for any connector management operations, use `KafkaConnector` resources. However, you can use any endpoints or operations that return information about the cluster and connectors For example, you can use the `GET /connector-plugins` endpoint with connectorsOnly set to false to list all plugins that are installed in the Kafka Connect cluster.

If you want to query the REST API, Cloudera recommends that you use the connect_shell.sh tool.

Related Information

Kafka Connect REST API

Using connect_shell.sh

Use connect_shell.sh to set up a pod that allows easy access to the Kafka Connect REST API. The pod created with this tool includes preset configurations, such as the $CONNECT_REST_URL environment variable, which is set to the base URL of the API.

Before you begin

- Ensure that you have access to your Cloudera credentials (username and password).
- Ensure that the environment where you run the tool has the following:
  - Bash 4 or higher.
  - GNU utilities:
    - echo
    - grep
    - sed
    - head
  - kubectl or oc
  - kubeconfig configured to target Kubernetes cluster

Procedure

1. Download the tool.

```
curl --user [***USERNAME***] \
  https://archive.cloudera.com/p/csm-operator/1.6/tools/connect_shell.sh \
  --output connect_shell.sh --location \
&& chmod +x connect_shell.sh
```

   Replace *[***USERNAME***]* with your Cloudera username. Enter your Cloudera password when prompted.

2. Use the tool.

   You have two choices. You can either use the tool interactively. In this case, you run the tool which opens an interactive shell window where you run queries. Alternatively, you can use pipe (|) to run queries one at a time.

   For Interactive

   a. Run the tool.

```
./connect_shell.sh \
  --namespace=[***CONNECT CLUSTER NAMESPACE***] \
```

```
  --cluster=[***CONNECT CLUSTER NAME***]
```

b. Query the REST API.

For example, you can list your topics with the following command.

```
curl $CONNECT_REST_URL/connector-plugins
```

This example queries /connector-plugins endpoint which returns available connector plugins in the cluster.

The pod is deleted when you exit the interactive shell.

For Pipe

To run one-off queries, pipe them into connect_shell.sh.

```
echo 'curl $CONNECT_REST_URL/connector-plugins' \
| ./connect_shell.sh --namespace=[***CONNECT CLUSTER NAMESPACE***] \
  --cluster=[***CONNECT CLUSTER NAME***]
```

Tip: Use the --help option to view additional options and information on tool usage.

Related Information

Kafka Connect REST API

Rolling restart Kafka Connect workers

You can initiate a rolling restart of your Kafka Connect workers by annotating the StrimziPodSet resource or the individual Kafka Connect pods with the strimzi.io/manual-rolling-update="true" annotation. You annotate the StrimziPodSet if you want to restart all workers in your cluster. You annotate individual pods if you want to restart specific workers.

Annotating the StrimziPodSet resource

Each Kafka Connect cluster has their own `StrimziPodSet` resource. This resource manages all the pods related to the Kafka Connect cluster. Annotating this resource restarts all workers in your cluster.

```
kubectl annotate strimzipodset [***CONNECT CLUSTER NAME***]-connect strimzi
.io/manual-rolling-update="true" --namespace [***NAMESPACE***]
```

In the next reconciliation loop, the Strimzi Cluster Operator initiates a rolling restart of all pods. After the pods are restarted, the annotation is automatically removed from the `StrimziPodSet`.

Annotating Pod resources

If you want to restart a specific pod or a set of specific pods, you annotate each `Pod` resource individually.

```
kubectl annotate pod [***KAFKA CONNECT POD***] strimzi.io/manual-rolling-u
pdate="true" --namespace [***NAMESPACE***]
```

The annotated pod is rolling restarted with the next reconciliation loop. The annotation is automatically removed from the pod after the pod is restarted.

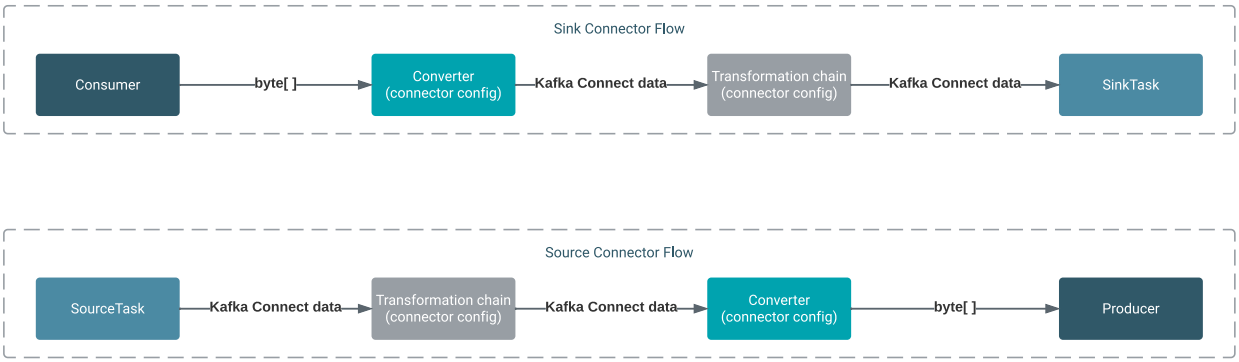
Single Message Transforms

Single Message Transforms (SMT) is a message transformation framework that you can deploy on top of Kafka Connect connectors to apply message transformations and filtering. Learn about the SMT framework as well as the transformation plugins available in Cloudera Streams Messaging Operator for Kubernetes.

Kafka Connect connectors provide ready-to-use tools to integrate between Kafka and external data stores. Still, in many use cases, the data moved by the connectors require some sanitization and transformation. To provide extra flexibility built on top of connectors, Kafka Connect also supports an SMT framework.

The SMT framework installs a transformation chain on top of connectors that modifies and filters the data on a single message basis. An SMT chain consists of transform and predicate plugins. Transform plugins are used to modify the data. For example, you can insert, replace, mask as well as perform various other modifications on the messages moved by connectors. Predicate plugins are used to add additional logic to your chain so that the transformation chain is only applied to messages which satisfy specified conditions.

The SMT framework requires that data is converted to the Kafka Connect internal data format. This data format is specific to Kafka Connect and consists of a structure and schema descriptor (SchemaAndValue) specific to Connect.



Supported SMT plugins

Cloudera Streams Messaging Operator for Kubernetes ships with and supports the following SMT plugins:

- All Apache Kafka plugins. For more information, see Transformations in the Apache Kafka documentation.
- The following Cloudera specific plugins:
  - ConvertFromBytes on page 18
  - ConvertToBytes on page 20

Note: The ConverFromBytes and ConvertToBytes transformation plugins transform binary data to and from the Kafka Connect internal data format. These plugins are specifically developed to enable the use of the SMT framework with connectors that only support binary data.

Configuring an SMT chain

Learn how to configure a Single Message Transformation (SMT) chain for Kafka Connect connectors.

SMT chains can be configured within the configuration of a Kafka Connect connector using SMT specific configuration properties. To set up a chain, you first define your transformation chain with the transforms property and optionally define your predicates using the predicates property. Afterward, you use transforms.* and predicates.* to configure the plugins in the chain. For example, the following configuration snippet sets up a transformation chain that filters messages based on their header and removes a specified field from messages.

```
#...
```

```
kind: KafkaConnector
spec:
  config:
    transforms: FilterAudit,MaskField
    transforms.MaskField.type: org.apache.kafka.connect.transforms.MaskField
$Value
    transforms.MaskField.fields: CreditCardNumber

    transforms.FilterAudit.type: org.apache.kafka.connect.transforms.Filter
    transforms.FilterAudit.predicate: IsAudit
    transforms.FilterAudit.negate: false

    predicates: IsAudit
    predicates.IsAudit.type: org.apache.kafka.connect.transforms.predicate
s.HasHeaderKey
    predicates.IsAudit.name: Audit
```

The following sections go through the properties in this example and give an overview on how to set up a transformation chain.

Configuring transforms

The transforms property contains a comma-separated list of transformation aliases. Each alias represents one step in the transformation chain. The aliases you add to the property are arbitrary names, they are used in other properties to configure that particular transformation step. For example, the following defines a two step transformation chain.

```
transforms: FilterAudit,MaskField
```

The transforms.*[***ALIAS***]*.type property specifies which transformation plugin should be used in a transformation step. *[***ALIAS***]* is one of the aliases that you specified in transforms. The value of the property is the fully qualified name of the transformation plugin that should be used in the step. For example, the following line specifies org.apache.kafka.connect.transforms.MaskField$Value as the plugin for the MaskField step.

```
transforms.MaskField.type: org.apache.kafka.connect.transforms.MaskField$Val
ue
```

Many transformation plugins support changing both the key and the value of a record. For these plugins, typically, a nested value or key class can be referenced as the type.

The transforms.*[***ALIAS***].[***KEY***]* property is used to configure the transformation plugins in your chain. This property is passed to the transformation plugin itself with transforms.*[***ALIAS***]* stripped from the property key. *[***ALIAS***]* is the alias of a plugin you specified in transforms. *[***KEY***]* is a property key that the plugin accepts. For example, the MaskField plugin has a fields property that specifies which fields should be removed from the structure.

```
transforms.MaskField.fields: CreditCardNumber
```

Configuring predicates

Predicates are a separate set of plugins. You use them to conditionally enable certain steps in the transformation chain. Predicates are configured in a similar way to transforms. You must specify the predicate aliases, associate the aliases with a plugin, and set plugin specific properties.

```
predicates: IsAudit
predicates.IsAudit.type: org.apache.kafka.connect.transforms.predicates.HasH
eaderKey
predicates.IsAudit.name: Audit
```

In this example the IsAudit predicate is an instance of the HasHeaderKey predicate plugin. This predicate returns true for records where a specific header key is present. predicates.IsAudit.name=Audit configures the predicate to look for the Audit header in the records.

After a predicate is set up, you can associate the predicate with any number of transformation steps using the pred icate property. If a predicate is associated with a transformation, that transformation step is only applied to the messages that match the condition specified in the predicate.

A good example for using a predicate is the Filter transformation plugin. This is because Filter filters (drops) all messages by default. Therefore, it must be used together with predicates to specify filtering logic. For example, the following configuration instructs the SMT framework that the FilterAudit step should only be invoked for messages where the IsAudit predicate returns true. That is, all messages with the Audit header will be filtered and will not be transformed by any subsequent steps in the transformation chain.

```
transforms.FilterAudit.predicate: IsAudit
transforms.FilterAudit.negate: false
```

The condition of a predicate can be inverted using negate. If negate is set to true, the SMT framework applies the transformation to any record that does **not** match the condition. For example, the following configuration instructs the SMT framework that the FilterAudit step should only be invoked for messages where the IsAudit predicate returns false.

```
transforms.FilterAudit.predicate: IsAudit
transforms.FilterAudit.negate: true
```

ConvertFromBytes

ConvertFromBytes is a Cloudera specific transformation plugin that converts binary data to the Kafka Connect internal data format. You can use this plugin to make connectors that only support binary data compatible with the Single Message Transforms (SMT) framework.

Fully qualified names

- `com.cloudera.dim.kafka.connect.transforms.ConvertFromBytes$Key`
- `com.cloudera.dim.kafka.connect.transforms.ConvertFromBytes$Value`

Description

Important: Ensure that you have an in-depth understanding about the following aspects of the connector that you plan on using with ConvertFromBytes.

- The type of the connector
- The converter type used by the connector
- The header converter type used by the connector

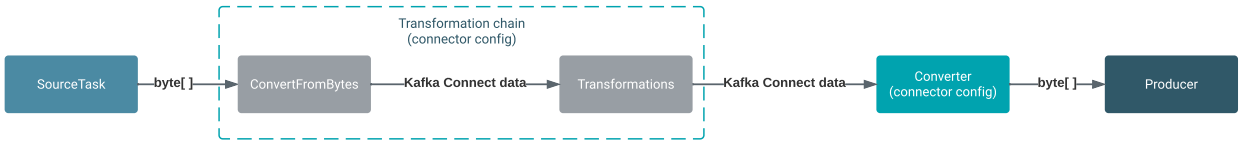
These aspects of a connector heavily influence how the plugin and how the transformation chain that includes this plugin must be configured.

The ConvertFromBytes transformation plugin accepts binary data and converts it into the Kafka Connect internal data format with a nested converter that transforms binary data. To support header based converter logic, the plugin requires a header converter to correctly transform record headers when interacting with the converter. This plugin supports both key and value conversion.

Using this plugin with connectors that only support binary data makes the connector fully compatible with the SMT framework. On their own, connectors that only support binary data have limited compatibility with transformations even if the binary data is structured. This is because transformations are only fully supported on data that is in the Kafka Connect internal data format. Binary only connectors, for example MirrorSourceConnector, emit data that has the BYTES schema and do not provide conversion to the Kafka Connect internal data format by default. When you

use a binary only connector with the ConvertFromBytes plugin, the binary data is parsed into a compatible structure, which can then be further processed with the transformation chain.

Figure 1: Source connector example flow with ConvertFromBytes



Example

The following configuration example adds a ConvertFromBytes transformation as a first step of the transformation chain. The conversion uses a schemaless JSON transformation to parse the binary data. The transformation steps, the connector, or the converter, whichever comes directly after FromBytes, receives a properly structured record instead of binary data.

```
#...
kind: KafkaConnector
spec:
  config:
    transforms: FromBytes,...
    transforms.FromBytes.type: com.cloudera.dim.kafka.connect.transformation
s.convert.ConvertFromBytes$Value
    transforms.FromBytes.converter: org.apache.kafka.connect.json.JsonConve
rter
    transforms.FromBytes.converter.schemals.enable: false
```

Configuration properties

Table 1: ConvertFromBytes properties reference

Property	Default Value	Required	Description
converter		True	The fully qualified name of the converter implementation to use. For example: org.apache.kafka .connect.json.JsonConverter
header.converter	org.apache.kafka.connect.storage .SimpleHeaderConverter	True	The fully qualified name of the header converter implementation to use. This converter must match the header converter of the connector.

Property	Default Value	Required	Description
converter.		False	A configuration prefix. Use this prefix to configure the properties of the converter specified in converter. Property keys and values specified with the prefix are passed directly to the converter with the prefix stripped. For example: `transforms.[***TRANSFORM ALIAS***].converter.[***CONVERTER PROPERTY KEY***]:[***CONVERTER PROPERTY VALUE***]`
header.converter.		False	A configuration prefix. Use this prefix to configure the properties of the header converter specified in header.converter. Property keys and values specified with the prefix are passed directly to the header converter with the prefix stripped. For example: `transforms.[***TRANSFORM ALIAS***].converter.[***HEADER CONVERTER PROPERTY KEY***]:[***HEADER CONVERTER PROPERTY VALUE***]`

ConvertToBytes

ConvertToBytes is a Cloudera specific transformation plugin that converts Kafka Connect internal data to binary data. You can use this plugin to make connectors that only support binary data compatible with the Single Message Transforms (SMT) framework.

Fully qualified names

- `com.cloudera.dim.kafka.connect.transforms.ConvertFromBytes$Key`
- `com.cloudera.dim.kafka.connect.transforms.ConvertFromBytes$Value`

Description

Important: Ensure that you have an in-depth understanding about the following aspects of the connector that you plan on using with ConvertToBytes.

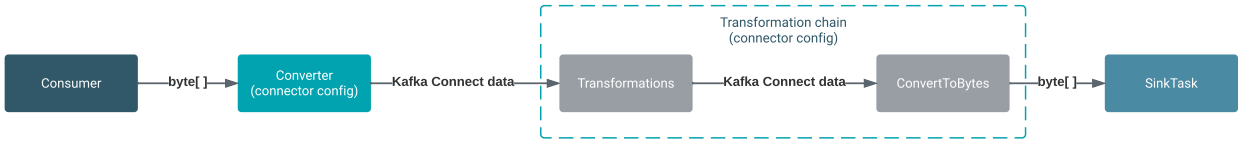
- The type of the connector
- The converter type used by the connector
- The header converter type used by the connector

These aspects of a connector heavily influence how the plugin and how the transformation chain that includes this plugin must be configured.

The ConvertToBytes transformation plugin accepts data in the Kafka Connect internal data format and converts it to binary data with a nested converter. To support header based converter logic, the plugin requires a header converter to correctly transform record headers when interacting with the converter. This plugin supports both key and value conversion.

Using this plugin with connectors that only support binary data makes the connector fully compatible with the SMT framework. On their own, connectors that only support binary data have limited compatibility with transformations even if the binary data is structured. This is because the format of the data after transformations are carried out is normally the Kafka Connect internal data format. Binary only connectors, however, expect data that has the BYTES schema and do not provide conversion from the Kafka Connect internal data format by default. When you use the ConvertToBytes plugin with a binary only connector, the structured data is converted to binary format, which can then be picked up by the connector.

Figure 2: Sink connector example flow with ConvertToBytes



Example

The following configuration example adds a `ConvertToBytes` transformation as the last step of the transformation chain. The conversion uses a schemaless JSON transformation to serialize the structured data. The transformation steps, the connector, or the converter, whichever comes directly after ToBytes, receives a properly structured record instead of binary data.

```
#...
kind: KafkaConnector
spec:
  config:
    transforms: ...,ToBytes
    transforms.ToBytes.type: com.cloudera.dim.kafka.connect.transformation
s.convert.ConvertToBytes$Value
    transforms.ToBytes.converter: org.apache.kafka.connect.json.JsonConve
rter
    transforms.ToBytes.converter.schemals.enable: false
```

Configuration properties

Table 2: ConvertToBytes properties reference

Property	Default Value	Required	Description
converter		True	The fully qualified name of the converter implementation to use. For example: org.apache.kafka .connect.json.JsonConverter
header.converter	org.apache.kafka.connect.storage .SimpleHeaderConverter	True	The fully qualified name of the header converter implementation to use. This converter must match the header converter of the connector.
converter.		False	A configuration prefix. Use this prefix to configure the properties of the converter specified in conv erter. Property keys and values specified with the prefix are passed directly to the converter with the prefix stripped. For example: `transformers.[***TRANSFORM ALIAS***].converter.[***CONVERTER PROPERTY KEY***]:[***CONVERTER PROPERTY VALUE***]`
header.converter.		False	A configuration prefix. Use this prefix to configure the properties of the header converter specified in header.converter. Property keys and values specified with the prefix are passed directly to the header converter with the prefix stripped. For example: `transformers.[***TRANSFORM ALIAS***].converter.[***HEADER CONVERTER PROPERTY KEY***]:[***HEADER CONVERTER PROPERTY VALUE***]`