Cloudera Streams Messaging Operator for Kubernetes 1.6.0

Kafka Connect Deployment and Configuration

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

https://docs.cloudera.com/

Legal Notice

Contents

Deploying Kafka Connect clusters

You can deploy a Kafka Connect cluster by creating a KafkaConnect resource. The Kafka Connect workers are automatically configured to run in distributed mode. You can configure the number of workers. Each worker is a separate pod.

Before you begin

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.
- Ensure that you have a working Kafka cluster. The Kafka cluster does not need to be managed by Strimzi, and it does not need to run on Kubernetes.
- Ensure that a namespace is available where you can deploy your cluster. If not, create one.

```
kubectl create namespace [***NAMESPACE***]
```

- Ensure that the `Secret` containing credentials for the Docker registry where Cloudera Streams Messaging Operator for Kubernetes artifacts are hosted is available in the namespace where you plan on deploying your cluster. If the `Secret` is not available, create it.

```
kubectl create secret docker-registry [***REGISTRY CREDENTIALS SECRET***]
 \
  --namespace [***NAMESPACE***] \
  --docker-server [***YOUR REGISTRY***] \
  --docker-username [***USERNAME***] \
  --docker-password "$(echo -n 'Enter your password: ' >&2; read -s passw
ord; echo >&2; echo $password)"
```

  - *[***REGISTRY CREDENTIALS SECRET***]* must be the same as the name of the `Secret` containing registry credentials that you created during Strimzi installation.
  - Replace *[***YOUR REGISTRY***]* with the server location of the Docker registry where Cloudera Streams Messaging Operator for Kubernetes artifacts are hosted. If your Kubernetes cluster has internet access, use container.repository.cloudera.com. Otherwise, enter the server location of your self-hosted registry.
  - Replace *[***USERNAME***]* with a username that provides access to the registry, and enter the corresponding password when prompted. If you are using container.repository.cloudera.com, enter your Cloudera credentials. Otherwise, enter credentials providing access to your self-hosted registry.
- The following steps walk you through a basic cluster deployment example. If you want to deploy a Kafka Connect cluster that has third-party connectors or other types of plugins installed, see Installing Kafka Connect connector plugins.

Procedure

1. Create a YAML configuration that contains your `KafkaConnect` resource.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  version: 4.1.1.1.6
  replicas: 3
  bootstrapServers: my-cluster-kafka-bootstrap.kafka:9092
```

   - The spec.version property specifies the Kafka version to use. The property must specify a Cloudera Kafka version supported by Cloudera Streams Messaging Operator for Kubernetes. For example, 4.1.1.1.6. Do not

add Apache Kafka versions, they are not supported. You can find a list of supported Kafka versions in the Release Notes.

- The bootstrapServers property specifies the Kafka brokers to which to connect. Cloudera recommends providing multiple brokers to handle broker failures and enable connecting to another instance.

  The my-cluster-kafka-bootstrap.kafka:9092 value is the bootstrap of a Kafka cluster that is deployed on Kubernetes with Cloudera Streams Messaging Operator for Kubernetes. my-cluster is the name of the cluster specified in metadata.name of the `Kafka` resource. The kafka-bootstrap string is fixed. The string kafka after the dot is the namespace where the cluster is deployed.

  Note: You can deploy multiple connect clusters, as long as their name, group ID, and internal topic name is different. Strimzi provides a default group ID and internal topic name, but you can only use those if you run a single Kafka Connect cluster.

2. Deploy the resource.

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

   The namespace where you deploy Kafka Connect must be watched by the Strimzi Cluster Operator.

3. Verify that the `KafkaConnect` resource is ready.

```
kubectl get kafkaconnect [***CONNECT CLUSTER NAME***] --names
pace [***NAMESPACE***] --watch
```

Results

If cluster deployment is successful, you should see an output similar to the following.

```
NAME                 DESIRED REPLICAS   READY
my-connect-cluster   3                  True
```

What to do next

- Learn more about configuring your Kafka Connect cluster. See Configuring Kafka Connect clusters.
- Install third-party connectors. See Installing Kafka Connect connector plugins.
- Deploy connectors. See Deploying connectors.

Related Information

Configuring group IDs
Configuring internal topics
Deploying Kafka Connect | Strimzi
KafkaConnect schema reference | Strimzi API Reference

Configuring Kafka Connect clusters

Learn how you can update Kafka Connect properties in your KafkaConnect resource.

Updating Kafka Connect configurations

You update Kafka Connect configuration by editing your KafkaConnect resources.

Procedure

1. Run the following command.

```
kubectl edit kafkaconnect [***CONNECT CLUSTER NAME***]  --namespac
e [***NAMESPACE***]
```

   Running kubectl edit opens the resource manifest in an editor.
2. Make your changes.
3. Save the file.
   Once the changes are saved, a rolling update is triggered and the workers restart one after the other with the applied changes.

Related Information

Kafka Connect Configs | Apache Kafka

KafkaConnectSpec schema properties | Strimzi

Configurable Kafka Connect properties and exceptions

Learn which Kafka Connect properties you can configure in the KafkaConnect resource and what default values some properties have, as well as which properties are managed by Strimzi.

Kafka Connect properties are configured by adding as keys to config in your `KafkaConnect` resource. The values can be on of the following JSON types:

- String
- Number
- Boolean

You can find a full reference of the available Kafka Connect properties in the Apache Kafka documentation. All properties can be specified, however, some properties are automatically configured with a default value if they are not specified in spec.config. Also, some properties are managed by Strimzi, and cannot be changed.

Properties with default values

The group ID, internal topic names, as well key and value converters get the following default values.

```
#...
kind: KafkaConnect
spec:
  groupId: connect-cluster
  configStorageTopic: connect-cluster-configs
  offsetStorageTopic: connect-cluster-offsets
  statusStorageTopic: connect-cluster-status
  config:
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
```

Note: The ID and topic names are configured in spec while converters are configured in spec.config.

Exceptions

Strimzi takes care of configuring and managing certain properties. The values of these properties cannot be changed.

The properties Strimzi takes care of are related to the following.

- Kafka cluster bootstrap address
- Security (encryption, authentication, and authorization)

- Listener and REST interface configuration
- Plugin path configuration

This means that properties with the following prefixes cannot be set.

- bootstrap.servers
- consumer.interceptor.classes
- listeners.
- plugin.path
- producer.interceptor.classes
- rest.
- sasl.
- security.
- ssl.

If the config property contains an option that cannot be changed, it is disregarded, and a warning message is logged in the Strimzi Cluster Operator log. All other supported properties are forwarded to Kafka, including the following exceptions to the options configured by Strimzi:

- Any SSL configuration for supported TLS versions and cipher suites

Related Information

Supported TLS versions and cipher suites | Strimzi
Kafka Connect Configs | Apache Kafka
KafkaConnectSpec schema properties | Strimzi

Configuring group IDs

Kafka Connect workers use a group ID for coordinating the cluster. All Connect workers use the same group ID inside a cluster.

About this task

Make sure to choose the group ID carefully, especially if you run multiple Kafka Connect clusters using the same Kafka cluster, because the group IDs must not clash with each other.

Configure the group ID by setting the value of the spec.groupId property.

```
#...
kind: KafkaConnect
spec:
  groupId: my-connect-cluster
```

Configuring internal topics

Kafka Connect uses three internal Kafka topics to store connector and task configurations, offsets, and status.

About this task

Configure the internal Kafka topic names in spec. Additionally, configure topic properties in spec.config using the appropriate prefixes.

```
#...
kind: KafkaConnect
spec:
  offsetStorageTopic: my-connect-cluster-offsets
  configStorageTopic: my-connect-cluster-configs
```

```
  statusStorageTopic: my-connect-cluster-status
  config:
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
```

Note:

Make sure to choose the internal topic names carefully, especially if you run multiple `KafkaConnect` clusters using the same Kafka clusters, because their internal topic names must not clash with each other.

Cloudera recommends setting the replication factor to at least 3 in a production environment. If you set the replication factor to -1, the default replication factor of the Kafka cluster will be used.

Configuring worker replica count

You can configure the number of worker pods created in the Kafka Connect cluster. Cloudera recommends having more than one worker pod for high availability.

Configure the number of worker pods created by setting the value of the spec.replicas property.

```
#...
kind: KafkaConnect
spec:
  replicas: 3
```

Note: Changing the number of replicas in the Kafka Connect cluster is how you scale your cluster. Unlike with Kafka brokers, no additional preparation, steps, or other configuration is needed for scaling. Simply increasing or decreasing the replica count is sufficient.

Configuring the Kafka bootstrap

You can configure the bootstrap servers of a Kafka cluster. Cloudera recommends providing multiple brokers, as this makes the connection between Kafka and Kafka Connect clusters highly available.

Configure the bootstrap servers of the Kafka cluster by setting the value of the spec.bootstrapServers property.

You can provide additional security configurations in spec.authentication and spec.tls.

```
#...
kind: KafkaConnect
spec:
  bootstrapServers: my-cluster-kafka-bootstrap.kafka:9092
  authentication:
    type: tls
    certificateAndKey:
      certificate: user.crt
      key: user.key
      secretName: connect-user
  tls:
    trustedCertificates:
      - certificate: ca.crt
        secretName: my-cluster-kafka-cluster-ca-cert
```

This example specifies a Kafka cluster that has TLS encryption and authentication.

Enabling KafkaConnector resources

Kafka Connect connectors are managed either using KafkaConnector resources or with the Kafka Connect REST API. If you want to manage connectors using KafkaConnector resources, you must enable them in the KafkaConnect resource. Managing connectors with KafkaConnector resources is the method recommended by Cloudera.

You enable `KafkaConnector` resource by setting the strimzi.io/use-connector-resources annotation to true.

```
#...
kind: KafkaConnect
metadata:
  annotations:
    strimzi.io/use-connector-resources: "true"
```

Managing connectors with `KafkaConnector` resources or the Kafka Connect REST API are mutually exclusive connector management methods.

- If you do not enable the use of `KafkaConnector` resources, you can only use the REST API to manage connectors in this Kafka Connect cluster.
- If you enable the use of `KafkaConnector` resources, you can only manage connectors using `KafkaConnector` resources. If you make any changes over the REST API, the changes are reverted by the Strimzi Cluster Operator.

Cloudera recommends creating connectors using `KafkaConnector` resources, that is, enabling this annotation for all your Kafka Connect clusters. Cloudera does not recommend exposing and using Kafka Connect REST API externally, because the REST API is insecure.

Tip: Even if you enable `KafkaConnector` resources, you can still use some API endpoints to query information about the connector.

Related Information

Using the Kafka Connect REST API

Configuration providers

Learn how to provide configuration that is not in plain text key-value pair format.

Connectors connect to external systems, requiring additional connection and security configurations. The connector configurations only contain key-value pairs. For some use-cases and configurations, this is not viable. For example, credentials like passwords, access keys, or any other sensitive information should not be added as plain text to the configuration of a connector.

To support these use-cases, connectors can use `ConfigProviders` and configuration references. `ConfigProviders` are responsible for pulling configurations from external configuration stores. Kafka Connect in Cloudera Streams Messaging Operator for Kubernetes ships with various `ConfigProviders` that are available by default. Cloudera recommends using the following `ConfigProviders`.

- `KubernetesSecretConfigProvider` - Loads configurations from Kubernetes secrets. You can use it to store sensitive configurations securely.
- `KubernetesConfigMapConfigProvider` - Loads configurations from Kubernetes `ConfigMaps`. You can use it to group and centralize reusable configurations across multiple connectors.
- `FileConfigProvider` - Loads configurations from a property file. You can use it to reference properties from files available in the Kafka Connect worker file system.

`ConfigProviders` must be enabled in the `KafkaConnect` resource if you want to use them in your connector configuration. You enable `ConfigProviders` in spec.config.

This example enables the `ConfigProvider`s recommended by Cloudera.

```
#...
kind: KafkaConnect
spec:
  config:
    config.providers: cfmap,secret,file
    config.providers.cfmap.class: io.strimzi.kafka.KubernetesConfigMapConfig
Provider
    config.providers.secret.class: io.strimzi.kafka.KubernetesSecretConfigP
rovider
    config.providers.file.class: org.apache.kafka.common.config.provider.Fil
eConfigProvider
```

Important: Configurations referenced through config providers do not get automatically updated when the underlying configmap/secret is updated. Connectors referring to these resources need a manual restart to get the configuration updates.

Example config provider usage of a secret, where the secret resource called connect-secrets is located in the connect-ns namespace, and contains a sasl.jaas.config key:

```
#...
kind: KafkaConnector
spec:
  config:
    producer.override.sasl.jaas.config: ${secret:connect-ns/connect-secret
s:sasl.jaas.config}
```

Kubernetes*ConfigProviders

When using the Kubernetes*`ConfigProviders`, the Kafka Connect workers require permissions to access the configuration maps and secrets in the Kubernetes cluster. Strimzi automatically creates a `ServiceAccount` for the Kafka Connect worker pods. An additional `Role` and `RoleBinding` is required to make the configuration providers work.

For example, assume you have a db-credentials Secret that contains credentials for a database to which that your connector will connect. To establish access to this secret through a `ConfigProvider` you need to create the following `Role` and `Rolebinding`.

For example, the following role grants access to the db-credentials secret in the database namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: connector-configuration-role
  namespace: database
rules:
  - apiGroups: [""]
    resources: ["secrets"]
    resourceNames: ["db-credentials"]
    verbs: ["get"]
```

The following `RoleBinding` binds the new Role to the Connect `ServiceAccount`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: connector-configuration-role-binding
subjects:
  - kind: ServiceAccount
    name: my-connect-connect
```

```
    namespace: my-project
roleRef:
  kind: Role
  name: connector-configuration-role
  apiGroup: rbac.authorization.k8s.io
```

The service account name is always generated with the *[***CLUSTER NAME***]*-connect pattern.

Related Information

Loading configuration values from external sources | Strimzi

Adding external configuration to Kafka Connect worker pods

Depending on the connector plugins and the connection settings of the external system, you might need to configure additional external configurations for the Kafka Connect workers that are stored in environment variables or in additional volumes. Environment variables and volumes are specified in your KafkaConnect resource.

Adding environment variables

Environment variables are added using spec.externalConfiguration.env. For example, connectors may need a topic prefix which is stored in an environment variable.

The alias for the configuration provider is used to define other configuration parameters. The provider parameters use the alias from spec.config.providers, taking the form spec.config.provider s.*[***ALIAS***]*.class.

```
#...
kind: KafkaConnect
spec:
  config:
    config.providers: env
    config.providers.env.class: org.apache.kafka.common.config
.provider.EnvVarConfigProvider
  externalConfiguration:
    env:
      - name: TOPIC_PREFIX
        value: prefix-text
```

Mounting additional volumes

Additional volumes are mounted using pod and container templates (spec.template.*) properties. For example, connectors might require an additional TLS truststore or keystore.

Specify additional volumes for Kafka Connect workers in the spec.template.pod.volumes property of the `KafkaConnect` resource. Attach volumes to the Kafka Connect container with the spec.tem plate.connectContainer.volumeMounts property.

The volumes you specify are mounted under the path you specified in mountPath. In the following example, this is /mnt/dbkeystore/*[***A FILE NAME FROM THE VOLUME***]*.

Important: All additional mounted paths must be located inside the /mnt path. If you mount a volume outside of this path, the `Kafka` resource remains in a NotReady state, the Kafka pods are not created, and a related warning is logged in the Strimzi Cluster Operator log.

```
#...
kind: KafkaConnect
spec:
  template:
    pod:
      volumes:
        - name: dbkeystore
```

```
          secret:
            secretName: dbkeystore
    connectContainer:
      volumeMounts:
        - name: dbkeystore
          mountPath: /mnt/dbkeystore
```

Related Information

ExternalConfiguration schema reference | Strimzi

Additional Volumes | Strimzi

AdditionalVolume schema reference | Strimzi API reference

ContainerTemplate schmea properties | Strimzi API reference

VolumeMount v1 core | Kubernetes

Configuring connector configuration override policy

Learn how to configure the connector configuration override policy. The override policy controls what client properties can be overridden by connectors.

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. You can fine-tune worker-level properties with connector configuration overrides. Properties specified with configuration overrides take priority over worker-level properties. Additionally, they can be applied on a per connector basis.

What configuration properties can be overridden is controlled by a configuration override policy. The policy is specified for Kafka Connect workers (`KafkaConnect` resource).

Kafka Connect includes the following policies by default.

- All - Allow overrides for all client properties (default).
- None - Do not allow any overrides.
- Principal - Only allow overriding JAAS configurations.

To configure the policy, set connector.client.config.override.policy in spec.config of your `KafkaConnect` resource. You can set the value of this property to the fully qualified name or the standard service name (for example, None) of the chosen policy.

```
#...
kind: KafkaConnect
spec:
  config:
    connector.client.config.override.policy: None
```

Tip: Policies use the plugin framework of Kafka Connect and can be extended. You can create a custom policy by developing your own implementation of `ConnectorClientConfigOverridePolicy`. You can install your custom policy plugin the same way you install connector plugins.

Related Information

Configuring client overrides in connectors

Installing Kafka Connect connector plugins

client.config.override.policy | Kafka

Configuring delayed rebalance

Learn how to disable or configure the delayed rebalance of Kafka Connect workers.

By default, Kafka Connect workers operate with a delayed rebalance. This means that if a worker stops for any reason, its resources (tasks and connectors) are not immediately reassigned to a different worker.

Instead, by default, a five minute grace period is in effect. During this time, the resource assigned to the stopped worker remains unassigned. This allows the worker to restart and rejoin its group. Worker resources are only reassigned to a different worker after the five minute period is up.

This is useful if the tasks and connectors are heavy operations and you do not want them to be rescheduled immediately. However, this also means that in case of a stoppage, some worker resources might be in an idle state of up to five minutes, which leads to a temporary service outage. This is true even if the stopped worker restarts and rejoins its group before the five minutes is up.

You can configure the delay to be shorter, or disable it altogether. To do this, configure the scheduled.rebalance.max.delay.ms Kafka Connect property in your `KafkaConnect` resource.

```
#...
kind: KafkaConnect
spec:
  config:
    scheduled.rebalance.max.delay.ms: 0
```

Note: When the Connect group leader is restarted, an immediate rebalance is triggered. This cancels the delayed rebalance.

Related Information

scheduled.rebalance.max.delay.ms | Kafka

Installing Kafka Connect connector plugins

Learn how to install third-party connectors in Kafka Connect. Third-party connectors are installed by building a new Kafka image that includes the connector artifacts. In Cloudera Streams Messaging Operator for Kubernetes, you build new images with Strimzi by configuring the KafkaConnect resource.

Note: In addition to connector plugins, Kafka Connect supports various other types of plugins, like data converters and transforms. While the following instructions are focused on connectors, you can follow them to install any other type of third-party plugin. The installation process is exactly the same.

By default the Strimzi Cluster Operator deploys a Kafka Connect cluster using the Kafka image shipped in Cloudera Streams Messaging Operator for Kubernetes. The Kafka image contains the connector plugins that are included by default in Apache Kafka.

Additional, third-party connectors are not included. If you want to deploy and use a third-party connector, you must build a new Kafka image that includes the connector plugins that you want to use. Your new image will be based on the default Kafka image that is shipped in Cloudera Streams Messaging Operator for Kubernetes. If the connector plugins are included in the image, you will be able to deploy instances of these connectors using `KafkaConnector` resources.

To build a new image, you add various properties to your `KafkaConnect` resource. These properties specify what connector plugin artifacts to include in the image as well the target registry where the image is pushed.

If valid configuration is included in the resource, Strimzi automatically builds a new Kafka image that includes the specified connector plugins. The image is built when you deploy your `KafkaConnect` resource. Specifically, Strimzi downloads the artifacts, builds the image, uploads it to the specified container registry, and then deploys Kafka Connect cluster.

The images built by Strimzi must be pushed to a container registry. Otherwise, they cannot be used to deploy Kafka Connect. You can use a public registry like quay.io or Docker Hub. Alternatively, you can push to your self-hosted registry. What registry you use will depend on your operational requirements and best practices.

If you are deploying multiple Kafka Connect clusters, Cloudera recommends using a unique image (different tag) for each of your clusters. Images behind tags can change and a change in an image should not affect more than a single cluster.

Building a new Kafka image automatically with Strimzi

You can configure your KafkaConnect resource so that Strimzi automatically builds a new container image that includes your third-party connector plugins. Configuration is done in spec.build.

About this task

When you specify spec.build.plugins properties in your `KafkaConnect` resource, Strimzi automatically builds a new Kafka image that contains the specified connector plugins. The image is pushed to the container registry specified in spec.build.output. The newly built image is automatically used in the Kafka Connect cluster that is deployed by the resource.

Before you begin

Important: Ensure that the JDK version of your platform is compatible with the Java version that the connector was compiled with. Mismatched versions prevent connectors from loading properly.

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.
- Ensure that a namespace is available where you can deploy your Kafka Connect cluster. If not, create one.

```
kubectl create namespace [***KAFKA CONNECT NAMESPACE***]
```

- A container registry is available where you can upload the container image.
- These steps demonstrate a basic configuration and deployment example. You can find additional information regarding spec.build.output and spec.build.plugin in Configuring the target registry and Configuring connectors to add. Alternatively, see Build schema reference in the Strimzi API documentation.

Procedure

1. Create a Docker configuration JSON file named docker_secret.json that contains your credentials to both the Cloudera container repository and your own repository where the images will be pushed.

```
{
    "auths": {
        "container.repository.cloudera.com": {
            "username": "[***CLOUDERA USERNAME***]",
            "password": "[***CLOUDERA PASSWORD***]"
        },
        "[***YOUR REGISTRY***]": {
            "username": "[***USERNAME***]",
            "password": "[***PASSWORD***]"
        }
    }
}
```

Note: If you installed Cloudera Streams Messaging Operator for Kubernetes from a self-hosted registry (air-gapped installation), replace container.repository.cloudera.com with the location of your self-hosted registry and use the corresponding credentials.

2. Create a Kubernetes `Secret` from the Docker configuration file.

```
kubectl create secret docker-registry [***SECRET NAME***] \
  --from-file=.dockerconfigjson=docker_secret.json \
  --namespace [***KAFKA CONNECT NAMESPACE***]
```

3. Configure your `KafkaConnect` resource.

   The resource configuration has to specify a container registry in spec.build.output. Third-party connector plugins are added to spec.build.plugins

   The following example adds the Kafka `FileStreamSource` and `FileStreamSink` example connectors and uploads the newly built image to a secured registry of your choosing.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  version: 4.1.1.1.6
  replicas: 3
  bootstrapServers: my-cluster-kafka-bootstrap.kafka:9092
  groupId: my-connect-cluster
  offsetStorageTopic: my-connect-cluster-offsets
  configStorageTopic: my-connect-cluster-configs
  statusStorageTopic: my-connect-cluster-status
  build:
    output:
      type: docker
      image: [***YOUR REGISTRY***]/[***IMAGE***]:[***TAG***]
      pushSecret: [***SECRET NAME***]
    plugins:
      - name: kafka-connect-file
        artifacts:
          - type: maven
            group: org.apache.kafka
            artifact: connect-file
            version: 3.7.0
```

4. Deploy the resource.

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

5. Wait until images are built and pushed. The Kafka Connect cluster is automatically deployed afterwards.

   While you wait, you can monitor the deployment process with kubectl get and kubectl logs.

```
kubectl get pods --namespace [***KAFKA CONNECT NAMESPACE***]
```

   The output lists a `Pod` called *[***CONNECT CLUSTER NAME***]*-connect-build. This is a build `Pod` responsible for constructing and pushing your image.

```
NAME                                                    READY     STATUS
   RESTARTS
#...
[***CONNECT CLUSTER NAME***]-connect-build              1/1       Running
   0
```

   You can get additional information by checking `Pod` logs.

```
kubectl logs [***CONNECT CLUSTER NAME***]-connect-build --namespa
ce [***KAFKA CONNECT NAMESPACE***]
```

   The log will contain various INFO entries related to building and pushing the image. After the image is successfully built and pushed, the build `Pod` is deleted and the Kafka Connect cluster is deployed.

6. Verify that the Kafka Connect cluster is deployed.

```
kubectl get kafkaconnect [***CONNECT CLUSTER NAME***] --names
pace [***KAFKA CONNECT NAMESPACE***]
```

The output is expected to show the cluster as ready.

```
NAME                            DESIRED REPLICAS   READY
#...
[***CONNECT CLUSTER NAME***]   3                   True
```

7. Verify that connector plugins are available.

You can do this by listing the contents of /opt/kafka/plugins in any Kafka Connect pod.

```
kubectl exec -it \
  --namespace [***KAFKA CONNECT NAMESPACE***] \
  [***CONNECT CLUSTER NAME***]-connect-[***ID***] \
  --container [***CONNECT CLUSTER NAME***]-connect \
  -- /bin/bash -c "ls /opt/kafka/plugins"
```

Tip: You can list available connectors with the GET /connector-plugins endpoint of the Kafka Connect API as well.

Results

Kafka Connect is deployed with an image that contains third-party connectors. Deploying the third-party connectors you added is now possible with `KafkaConnector` resources.

What to do next

Deploy a connector using a `KafkaConnector` resource. See, Deploying connectors.

Related Information

Using the Kafka Connect REST API

Configuring the target registry

The Kafka image built by Strimzi is uploaded to a container registry of your choosing. The target registry where the image is uploaded is configured in your KafkaConnect resource with spec.build.output.

```
#...
kind: KafkaConnect
spec:
  build:
    output:
      type: docker
      image: [***YOUR REGISTRY***]/[***IMAGE***]:[***TAG***]
      pushSecret: [***SECRET NAME***]
```

- type - specifies the type of image Strimzi outputs. The value you specify is decided by the type of your target registry. The property accepts docker or imagestream as valid values.
- image - specifies the full name of the image. The name includes the registry, image name, as well as tags.
- pushSecret - specifies the name of the secret that contains the credentials required to connect to the registry specified in image. This property is optional and required only if the registry requires credentials for access.

Related Information

output | Strimzi API reference

Configuring connector plugins to add

The Kafka image built by Strimzi includes the connector plugins that you reference in the spec.build.plugin property of your KafkaConnect resource.

Each connector plugin is specified as an array.

```
#...
spec:
  build:
    plugins:
      - name: kafka-connect-file
        artifacts:
          - type: maven
            group: org.apache.kafka
            artifact: connect-file
            version: 3.7.0
```

Each connector plugin must have a name and a type. The name must be unique in the Kafka Connect deployment.

Various artifact types are supported including jar, tgz, zip, maven, and other.

The type of the artifact defines what required and optional properties are supported. At minimum, for all types, you must specify a location where the artifact is downloaded from. For example, with maven type artifacts, you specify the Maven group and artifact. For jar type artifacts you specify a URL.

You can specify artifacts for other types of plugins, like data converters or transforms, not just connectors.

Note: For maven type artifacts, you can configure the repository property, which specifies the Maven repository where the artifact is downloaded from. If the repository property is omitted,the artifact is downloaded from Maven Central.

Related Information

Maven Central | Maven

plugins | Strimzi API reference

Rebuilding a Kafka image

It is possible that the base image or the plugin behind the URL changed over time. You can trigger Strimzi to rebuild the image by applying the strimzi.io/force-rebuild=true annotation on the Kafka Connect StrimziPodSet resource.

```
kubectl annotate strimzipodsets.core.strimzi.io --namesp
ace [***NAMESPACE***] \
  [***CONNECT CLUSTER NAME***]-connect \
  strimzi.io/force-rebuild=true
```

Exactly-once semantics

Exactly-once semantics (EOS) is a feature that enables Kafka and Kafka applications to guarantee that each message is delivered precisely once without it being duplicated or lost. EOS can be enabled for Kafka Connect and Kafka Connect source connectors.

Source connectors progress is tracked by periodically committing the offsets of the processed messages. If the connector fails, uncommitted messages are reprocessed after the connector starts running again.

Using EOS, source connectors are able to handle offset commits and message produces in a single transaction. This either results in a successful operation where messages are produced to the target topic along with offset commits, or

a rollback of the whole operation. EOS is enabled in the `KafkaConnect` resource. Additionally you can fine-tune EOS related properties in the configuration of connector instances.

Note: Consumers consuming the target topic should have isolation.level set to read_committed to avoid reading uncommitted data.

Enabling exactly-once semantics

You enable EOS for source connectors by configuring exactly.once.source.support in the KafkaConnect resource.

Configuration differs for newly deployed resources and existing resources.

For New resources

Set exactly.once.source.support to enabled.

```
#...
kind: KafkaConnect
spec:
  config:
    exactly.once.source.support: enabled
```

For Existing resources

1. Set exactly.once.source.support to preparing.

   ```
   #...
   kind: KafkaConnect
   spec:
     config:
       exactly.once.source.support: preparing
   ```

2. Wait until configuration changes are applied. This happens in the next reconciliation loop.
3. Set exactly.once.source.support to enabled.

Disabling exactly-once semantics

You disable EOS for source connectors by configuring exactly.once.source.support in the KafkaConnect resource.

Procedure

1. Set exactly.once.source.support to preparing.

   ```
   #...
   kind: KafkaConnect
   spec:
     config:
       exactly.once.source.support: preparing
   ```

2. Wait until configuration changes are applied.

   This happens in the next reconciliation loop.
3. Set exactly.once.source.support to disabled.

Source connector properties for exactly-once semantics

After enabling EOS for source connectors in the KafkaConnect resource, you can fine-tune EOS by configuring your connector instances (KafkaConnector resources).

Use the following source connector properties to configure EOS. Cloudera recommends that you use the default values.

Name	Default value	Description
exactly.once.support	requested	Permitted values are requested and forces a preflight check for the con provide exactly-once delivery with Some connectors may be capable o delivery but not signal to Kafka Co this. In this case, review the docum before connector deployment and s ested. Additionally, if the value is worker that performs preflight vali exactly-once support enabled for s to create or validate the connector
transaction.boundary	poll	Permitted values are poll, connecto poll, a new producer transaction is for every batch of records that each provides to Kafka Connect. If set t connector-defined transaction bou connectors are capable of defining boundaries, and in that case, attem property set to connector will fail. transactions only after a user-defin
offsets.storage.topic	null	The name of a separate offsets topi If left empty or not specified, the w name is used. If specified, the offs does not already exist on the Kafka connector (which may be different worker's global offsets topic if the of the connector's producer has bee worker's).
transaction.boundary.interval.ms	null	If transaction.boundary is set to int interval for producer transaction co If unset, defaults to the value of the .interval.ms property.

Configuring Kafka Connect for Prometheus monitoring

To monitor Kafka Connect with Prometheus, you must configure your Kafka Connect cluster to expose the necessary metric endpoints that integrate with your Prometheus deployment. This is done by configuring the metricsConfig property in your KafkaConnect resource.

About this task

By default a Kafka Connect cluster you deploy with a `KafkaConnect` resource does not expose metrics that Prometheus can scrape. In order to use Prometheus to monitor your Kafka Connect cluster, you must enable and expose these metrics. This is done by adding a metricsConfig property to the spec of your `KafkaConnect` resource.

Specifying metricsConfig in the `KafkaConnect` resource enables the Prometheus JMX Exporter which exposes metrics through a HTTP endpoint. The metrics are exposed on port 9094. The metricsConfig property can reference a `ConfigMap` that holds your JMX metrics configuration or will include the metrics configurations in-line. The following steps demonstrate the configuration by referencing a `ConfigMap`.

Before you begin

A Prometheus deployment that can connect to the metric endpoints of the Kafka connect cluster running in the Kubernetes environment is required. Any properly configured Prometheus deployment can be used to monitor Kafka Connect. You can find additional information and examples on Prometheus setup in the Strimzi documentation.

Procedure

1. Create a `ConfigMap` with JMX metrics configuration for Kafka Connect.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: connect-metrics
  labels:
    app: strimzi
data:
  metrics-config.yml: |
    [***KAFKA-CONNECT-METRICS-CONFIGURATION***]
```

   Replace *[***KAFKA-CONNECT-METRICS-CONFIGURATION***]* with your JMX Prometheus metrics configuration.

2. Update your `KafkaConnect` resource with a metricsConfig property. The property needs to reference the `ConfigMap` you created in 1 on page 20.

```
#...
kind: KafkaConnect
spec:
  metricsConfig:
    type: jmxPrometheusExporter
    valueFrom:
      configMapKeyRef:
        name: connect-metrics
        key: metrics-config.yml
```

What to do next

- Configure Prometheus and specify alert rules to start scraping metrics from the Kafka Connect pods. You can find an example rules file (prometheus-rules.yaml) as well as various other configuration examples on the Cloudera Archive. Examples related to Prometheus are located in the /csm-operator/1.2/examples/metrics directory.
- Review Cloudera recommendations on what alerts and metrics to configure. See Monitoring with Prometheus.

Related Information

Cloudera Archive

Prometheus JMX Exporter | GitHub

Configuring the security context of Kafka Connect

Learn how to configure the security context of Kafka Connect pods

The Kafka Connect resource allows users to specify the security context at the pod and/or the container level with template properties.

```
#...
kind: KafkaConnect
spec:
  template:
    pod:
```

```
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
        runAsNonRoot: true
        seccompProfile:
          type: RuntimeDefault
```

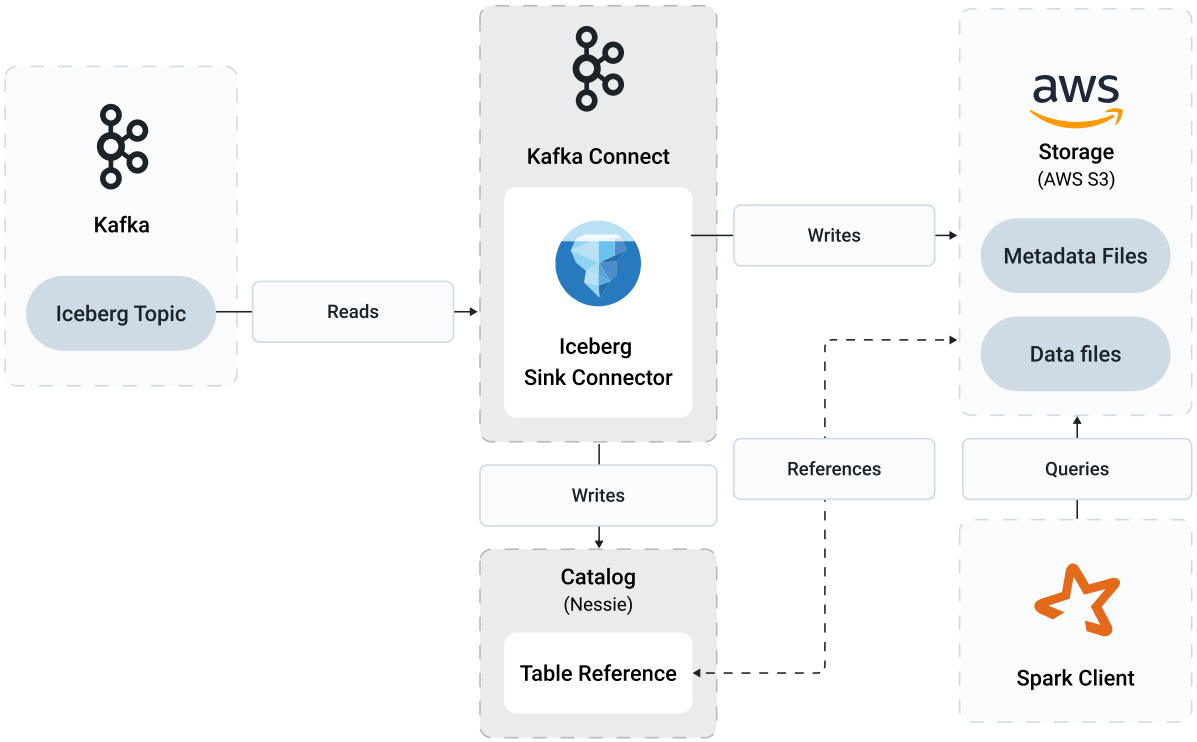
Example: Deploying the Apache Iceberg Sink Connector for Kafka Connect

If you need to make your Kafka streaming data accessible for analytics, you can deploy the Apache Iceberg Sink Connector to write data into Iceberg table format. This example shows you how to configure the connector with a Nessie catalog and S3 storage, enabling you to query your real-time Kafka data using analytics engines such as Spark.

About this task

This example demonstrates how to set up the following end-to-end Apache Iceberg Sink Connector deployment:

Figure 1: Apache Iceberg Sink Connector example deployment



In this example setup, the connector reads records from a Kafka topic and writes them to S3 storage as Parquet files with enhanced metadata including schema and partitioning information. The connector also updates a Nessie data catalog with references to the current metadata and schema.

After the connector is deployed and data from the Kafka topic is written to S3, you retrieve the data by querying the table with Apache Spark.

About connector dependencies

The Apache Iceberg Sink Connector is part of a large ecosystem with many storage and catalog options. The required dependencies vary depending on your specific use case, including the target storage system, such as S3, ADLS, or HDFS, the data catalog implementation, such as Nessie, AWS Glue, Hive, or JDBC, and the data serialization format, such as Parquet, ORC, or Avro. Each combination requires a different set of artifacts to function correctly.

For this example, which uses Nessie as the data catalog, S3 as the storage backend, and Parquet as the data format, the following artifacts are required:

- iceberg-kafka-connect – The connector plugin itself
- hadoop-common – Core Hadoop libraries
- iceberg-parquet – Parquet file format support
- iceberg-nessie – Nessie catalog client libraries
- iceberg-aws-bundle – AWS SDK bundle for S3 integration
- iceberg-aws – Iceberg AWS integration libraries

These artifacts are specified in your `KafkaConnect` resource and will be downloaded from Maven.

Before you begin

Important: Ensure that the JDK version of your platform is compatible with the Java version that the connector was compiled with. Mismatched versions prevent connectors from loading properly.

- Ensure that the Strimzi Cluster Operator is installed and running. See Installation.
- Ensure that you have a Kafka cluster deployed and running on Kubernetes. If not, deploy one, see Deploying Kafka.

  This example assumes a Kafka cluster deployed with Cloudera Streams Messaging Operator for Kubernetes. The name of the cluster is referred to as *[***KAFKA CLUSTER NAME***]*. The namespace is referred to as *[***KAFKA NAMESPACE***]*.
- Ensure that a namespace is available where you can deploy your Kafka Connect cluster. If not, create one.

  ```
  kubectl create namespace [***KAFKA CONNECT NAMESPACE***]
  ```
- Ensure that you have access to a container registry where you can upload a Kafka Connect container image.

  The registry is required as you will be building your own custom Kafka Connect image that includes the Apache Iceberg Sink Connector and its dependencies. You can use your own private registry or a public registry such as Quay.io or Docker Hub. The registry is referred to as *[***YOUR REGISTRY***]* in this example.
- Ensure that you have an S3 bucket available for storing Iceberg table data.

  You will need the name and AWS region of the bucket as well access credentials (access key ID and secret access key). These are referred to as *[***S3 BUCKET***]*, *[***S3 REGION***]*, *[***AWS ACCESS KEY ID***]*, and *[***AWS SECRET ACCESS KEY***]* in this example.
- Download the kafka_shell.sh tool from the Cloudera Archive.

  You will use this tool to create topics and to produce data for testing.

Procedure

1. Deploy the Nessie data catalog.

   The catalog is required to track table metadata and schemas.

   a) Create a YAML configuration file for the Nessie `Deployment` and `Service`.

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: nessie
   spec:
     selector:
   ```

```
    matchLabels:
      app: nessie
  template:
    metadata:
      labels:
        app: nessie
    spec:
      containers:
      - name: nessie
        image: projectnessie/nessie:latest
        env:
          - name: NESSIE_VERSION_STORE_TYPE
            value: IN_MEMORY
        ports:
        - containerPort: 19120
---
apiVersion: v1
kind: Service
metadata:
  name: nessie
spec:
  type: ClusterIP
  ports:
  - port: 19120
    name: http
  selector:
    app: nessie
```

b) Deploy Nessie.

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

Take note of the namespace where you deploy Nessie, as you will need it when configuring the connector.

2. Create a Docker configuration `Secret` for image registry authentication.

Strimzi needs credentials to pull base images from the Cloudera repository and push the built Kafka Connect image to your registry.

a) Create a Docker configuration JSON file named docker_secret.json that contains your credentials to both the Cloudera container repository and your own repository where the images will be pushed.

```
{
    "auths": {
        "container.repository.cloudera.com": {
            "username": "[***CLOUDERA USERNAME***]",
            "password": "[***CLOUDERA PASSWORD***]"
        },
        "[***YOUR REGISTRY***]": {
            "username": "[***USERNAME***]",
            "password": "[***PASSWORD***]"
        }
    }
}
```

Note: If you installed Cloudera Streams Messaging Operator for Kubernetes from a self-hosted registry (air-gapped installation), replace container.repository.cloudera.com with the location of your self-hosted registry and use the corresponding credentials.

b) Create a Kubernetes `Secret` from the Docker configuration file.

```
kubectl create secret docker-registry [***SECRET NAME***] \
  --from-file=.dockerconfigjson=docker_secret.json \
```

```
  --namespace [***KAFKA CONNECT NAMESPACE***]
```

3. Deploy a Kafka Connect cluster that includes the connector and its dependencies.

   The Kafka Connect cluster must include the connector plugin and its dependencies. Strimzi automatically builds a custom image and uploads it to the specified registry when the build.output and build.plugins properties are configured.

   a) Create a YAML configuration file for the `KafkaConnect` resource.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnect
metadata:
  name: [***CONNECT CLUSTER NAME***]
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  version: 4.1.1.1.6
  replicas: 3
  bootstrapServers: [***KAFKA CLUSTER NAME***]-kafka-bootstrap.[***KAFKA
 NAMESPACE***]:9092
  build:
    output:
      type: docker
      image: [***YOUR REGISTRY***]/iceberg/kafka-connect-iceberg:latest
      pushSecret: [***SECRET NAME***]
    plugins:
      - name: iceberg-kafka-connect
        artifacts:
          - type: maven
            group: org.apache.iceberg
            artifact: iceberg-kafka-connect
            version: 1.10.0
          - type: maven
            group: org.apache.iceberg
            artifact: iceberg-nessie
            version: 1.10.0
          - type: maven
            group: org.apache.iceberg
            artifact: iceberg-parquet
            version: 1.10.0
          - type: maven
            group: org.apache.hadoop
            artifact: hadoop-common
            version: 3.4.1
          - type: maven
            group: org.apache.iceberg
            artifact: iceberg-aws-bundle
            version: 1.10.0
          - type: maven
            group: org.apache.iceberg
            artifact: iceberg-aws
            version: 1.10.0
  groupId: connect-cluster
  offsetStorageTopic: connect-cluster-offsets
  configStorageTopic: connect-cluster-configs
  statusStorageTopic: connect-cluster-status
  config:
    config.providers: secrets
```

```
    config.providers.secrets.class: io.strimzi.kafka.KubernetesSecretCo
nfigProvider
```

Tip: Strimzi automatically resolves and downloads transitive dependencies when you use the maven artifact type. This simplifies the configuration process as you only need to specify the primary dependencies. If you use other artifact types, such as jar, zip, or tgz, you must manually specify all dependencies including transitive ones.

b) Deploy the `KafkaConnect` resource.

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

c) Wait until images are built and pushed. The Kafka Connect cluster is automatically deployed afterwards.

While you wait, you can monitor the deployment process with kubectl get and kubectl logs.

```
kubectl get pods --namespace [***KAFKA CONNECT NAMESPACE***]
```

The output lists a `Pod` called *[***CONNECT CLUSTER NAME***]*-connect-build. This is a build `Pod` responsible for constructing and pushing your image.

```
NAME                                                    READY      STATUS
     RESTARTS
#...
[***CONNECT CLUSTER NAME***]-connect-build              1/1        Running
     0
```

You can get additional information by checking `Pod` logs.

```
kubectl logs [***CONNECT CLUSTER NAME***]-connect-build --namespa
ce [***KAFKA CONNECT NAMESPACE***]
```

The log will contain various INFO entries related to building and pushing the image. After the image is successfully built and pushed, the build `Pod` is deleted and the Kafka Connect cluster is deployed.

d) Verify that the Kafka Connect cluster is deployed.

```
kubectl get kafkaconnect [***CONNECT CLUSTER NAME***] --names
pace [***KAFKA CONNECT NAMESPACE***]
```

The output is expected to show the cluster as ready.

```
NAME                            DESIRED REPLICAS   READY
#...
[***CONNECT CLUSTER NAME***]   3                   True
```

4. Configure AWS credentials for S3 access.

The Iceberg connector needs credentials to write data to S3. You store these credentials in a Kubernetes `Secret` and configure Role-Based Access Control (RBAC) to allow the Kafka Connect service account to access the `Secret`.

a) Create a Kubernetes `Secret` for AWS credentials.

```
kubectl create secret generic iceberg-aws-credentials \
  --namespace [***KAFKA CONNECT NAMESPACE***] \
  --from-literal=access_key_id=[***AWS ACCESS KEY ID***] \
  --from-literal=secret_access_key="$(echo -n 'Enter your AWS secret
access key: ' >&2; read -s key; echo >&2; echo $key)"
```

Enter your AWS secret access key when prompted.

b) Create a YAML configuration file for RBAC.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: iceberg-connector-role
rules:
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["iceberg-aws-credentials"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: iceberg-connector-role-binding
subjects:
- kind: ServiceAccount
  name: [***CONNECT CLUSTER NAME***]-connect
roleRef:
  kind: Role
  name: iceberg-connector-role
  apiGroup: rbac.authorization.k8s.io
```

c) Deploy the RBAC resources.

```
kubectl apply --filename [***AWS RBAC YAML CONFIG***] \
  --namespace [***KAFKA CONNECT NAMESPACE***]
```

5. Deploy the Apache Iceberg Sink Connector.

a) Create a YAML configuration file for the `KafkaConnector` resource.

```
apiVersion: kafka.strimzi.io/v1
kind: KafkaConnector
metadata:
  name: iceberg-sink
  labels:
    strimzi.io/cluster: [***CONNECT CLUSTER NAME***]
spec:
  class: org.apache.iceberg.connect.IcebergSinkConnector
  tasksMax: 1
  config:
    value.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter.schemas.enable: false
    topics: "data-topic"
    iceberg.catalog.type: "nessie"
    iceberg.catalog.uri: "http://nessie.[***NESSIE NAMESPACE***]:19120/a
pi/v2"
    iceberg.catalog.ref: "main"
    iceberg.catalog.warehouse: "s3a://[***S3 BUCKET***]"
    iceberg.catalog.io-impl: "org.apache.iceberg.aws.s3.S3FileIO"
    iceberg.catalog.s3.region: "[***S3 REGION***]"
    iceberg.catalog.s3.access-key-id: "${secrets:iceberg-aws-credentia
ls:access_key_id}"
    iceberg.catalog.s3.secret-access-key: "${secrets:iceberg-aws-cred
entials:secret_access_key}"
    iceberg.catalog.s3.path-style-access: "true"
    iceberg.tables.auto-create-enabled: "true"
    iceberg.tables: "data-topic"
    iceberg.control.commit.interval-ms: 60000
```

b) Deploy the `KafkaConnector` resource.

```
kubectl apply --filename [***CONNECTOR YAML CONFIG***] \
  --namespace [***KAFKA CONNECT NAMESPACE***]
```

c) Verify that the connector is running.

```
kubectl get kafkaconnectors \
  --namespace [***KAFKA CONNECT NAMESPACE***]
```

The output is expected to show the connector as ready.

```
NAME            CLUSTER              CONNECTOR CLASS
              MAX TASKS   READY
iceberg-sink    [***CONNECT CLUSTER NAME***]   org.apache.iceberg.connec
t.IcebergSinkConnector  1           True
```

6. Test the deployment with Apache Spark.

You can verify that the connector is writing data correctly by creating a Kafka topic, producing messages to the topic, and querying the Iceberg table using Spark.

a) Run kafka_shell.sh.

```
./kafka_shell.sh \
  --namespace=[***KAFKA NAMESPACE***] \
  --cluster=[***KAFKA CLUSTER NAME***]
```

This tool opens an interactive shell with configuration presets that enable you to quickly run Kafka command line tools.

b) Create a Kafka topic.

```
bin/kafka-topics.sh \
  --command-config /tmp/client.properties \
  --bootstrap-server $BOOTSTRAP_SERVERS \
  --create --topic data-topic \
  --partitions 1 \
  --replication-factor 1
```

c) Start a console producer.

```
bin/kafka-console-producer.sh \
  --producer.config /tmp/client.properties \
  --bootstrap-server $BOOTSTRAP_SERVERS \
  --topic data-topic
```

d) Enter the following JSON records:

```
{"id": 203, "user_name": "Bob", "department": "Sales", "event_ts": "2023
-10-27T11:00:00Z"}
{"id": 204, "user_name": "Charlie", "department": "Marketing", "event
_ts": "2023-10-27T11:05:00Z"}
```

e) Exit the console producer and the interactive shell.

f) Create a YAML configuration file for a Spark `Pod`.

```
apiVersion: v1
kind: Pod
metadata:
  name: spark-client
spec:
  containers:
```

```
  - name: spark
    image: apache/spark:3.5.0
    securityContext:
      runAsUser: 0
    command: ["/bin/bash", "-c", "sleep infinity"]
    env:
      - name: HOME
        value: "/root"
```

g) Deploy the Spark `Pod`.

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

h) Start a Spark shell with the Iceberg extension.

```
kubectl exec -it spark-client --namespace [***SPARK NAMESPACE***] -- /
opt/spark/bin/spark-shell \
  --packages "org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.10.0,
org.apache.hadoop:hadoop-aws:3.4.2" \
  --conf spark.sql.extensions="org.apache.iceberg.spark.extensions.Ice
bergSparkSessionExtensions" \
  --conf spark.sql.catalog.nessie="org.apache.iceberg.spark.SparkCatal
og" \
  --conf spark.sql.catalog.nessie.catalog-impl="org.apache.iceberg.nes
sie.NessieCatalog" \
  --conf spark.sql.catalog.nessie.uri="http://nessie.[***NESSIE
 NAMESPACE***]:19120/api/v1" \
  --conf spark.sql.catalog.nessie.ref="main" \
  --conf spark.sql.catalog.nessie.authentication.type="NONE" \
  --conf spark.sql.catalog.nessie.warehouse="s3://[***S3 BUCKET***]" \
  --conf spark.sql.catalog.nessie.io-impl="org.apache.iceberg.aws.s3.S3F
ileIO" \
  --conf spark.sql.catalog.nessie.s3.path-style-access="true" \
  --conf spark.sql.catalog.nessie.s3.access-key-id="[***AWS ACCESS KEY
 ID***]" \
  --conf spark.sql.catalog.nessie.s3.secret-access-key="$(echo -n 'Enter
 your AWS secret access key: ' >&2; read -s key; echo >&2; echo $key)"
```

Enter your AWS secret access key when prompted.

i) Query the Iceberg table using the Spark shell.

```
spark.sql("USE nessie")
spark.sql("SELECT * FROM `data-topic` LIMIT 10").show()
```

Note: It might take some time for the connector to commit data to the Iceberg table. The iceberg.control.commit.interval-ms property controls the commit interval that is 60 seconds in this example.

Results

The Iceberg Sink Connector is deployed and writing data from Kafka topics to Iceberg tables stored in S3. The Nessie catalog tracks references to the table metadata and schemas, allowing analytics engines such as Spark to query the data.

Related Information

Installing Kafka Connect connector plugins

Using kafka_shell.sh

Kafka Connect | Apache Iceberg

Connector configuration reference | Apache Iceberg

Catalog configuration | Apache Iceberg

Table format specification | Apache Iceberg
Project Nessie documentation | Project Nessie