Red Hat AMQ Streams 2.4

Deploying and Managing AMQ Streams on OpenShift

Deploy AMQ Streams 2.4 on OpenShift Container Platform

Last Updated: 2023-07-28

Red Hat AMQ Streams 2.4 Deploying and Managing AMQ Streams on OpenShift

Deploy AMQ Streams 2.4 on OpenShift Container Platform

Legal Notice

Abstract

Deploy AMQ Streams to an OpenShift cluster using the OperatorHub or installation artifacts. Use the AMQ Streams operators to deploy and manage Kafka components. Upgrade AMQ Streams to take advantage of new features. As part of the upgrade, upgrade Kafka to the latest supported version.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. DEPLOYMENT OVERVIEW

AMQ Streams simplifies the process of running Apache Kafka in an OpenShift cluster.

This guide provides instructions on all the options available for deploying and upgrading AMQ Streams, describing what is deployed, and the order of deployment required to run Apache Kafka in an OpenShift cluster.

As well as describing the deployment steps, the guide also provides pre- and post-deployment instructions to prepare for and verify a deployment. The guide also describes additional deployment options for introducing metrics.

Upgrade instructions are provided for AMQ Streams and Kafka upgrades.

AMQ Streams is designed to work on all types of OpenShift cluster regardless of distribution, from public and private clouds to local deployments intended for development.

1.1. CONFIGURING A DEPLOYMENT

The deployment procedures in this guide are designed to help you set up the initial structure of your deployment. After setting up the structure, you can use custom resources to configure the deployment to your precise needs. The deployment procedures use the example installation files provided with AMQ Streams. The procedures highlight any important configuration considerations, but they do not describe all the configuration options available.

You might want to review the configuration options available for Kafka components before you deploy AMQ Streams. For more information on the configuration options, see the Custom resource API reference.

1.1.1. Securing Kafka

On deployment, the Cluster Operator automatically sets up TLS certificates for data encryption and authentication within your cluster.

AMQ Streams provides additional configuration options for *encryption*, *authentication* and *authorization*:

- Secure data exchange between the Kafka cluster and clients by managing secure access to Kafka.
- Configure your deployment to use an authorization server to provide OAuth 2.0 authentication and OAuth 2.0 authorization.
- Secure Kafka using your own certificates .

1.1.2. Monitoring a deployment

AMQ Streams supports additional deployment options to monitor your deployment.

- Extract metrics and monitor Kafka components by deploying Prometheus and Grafana with your Kafka cluster.
- Extract additional metrics, particularly related to monitoring consumer lag, by deploying Kafka Exporter with your Kafka cluster.
- Track messages end-to-end by setting up distributed tracing.

1.1.3. CPU and memory resource limits and requests

By default, the AMQ Streams Cluster Operator does not specify requests and limits for CPU and memory resources for any operands it deploys.

Having sufficient resources is important for applications like Kafka to be stable and deliver good performance.

The right amount of resources you should use depends on the specific requirements and use-cases.

You should consider configuring the CPU and memory resources. You can set resource requests and limits for each container in the AMQ Streams custom resources.

1.2. AMQ STREAMS CUSTOM RESOURCES

A deployment of Kafka components to an OpenShift cluster using AMQ Streams is highly configurable through the application of custom resources. Custom resources are created as instances of APIs added by Custom resource definitions (CRDs) to extend OpenShift resources.

CRDs act as configuration instructions to describe the custom resources in an OpenShift cluster, and are provided with AMQ Streams for each Kafka component used in a deployment, as well as users and topics. CRDs and custom resources are defined as YAML files. Example YAML files are provided with the AMQ Streams distribution.

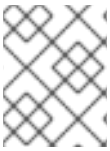
CRDs also allow AMQ Streams resources to benefit from native OpenShift features like CLI accessibility and configuration validation.

1.2.1. AMQ Streams custom resource example

CRDs require a one-time installation in a cluster to define the schemas used to instantiate and manage AMQ Streams-specific resources.

After a new custom resource type is added to your cluster by installing a CRD, you can create instances of the resource based on its specification.

Depending on the cluster setup, installation typically requires cluster admin privileges.



NOTE

Access to manage custom resources is limited to AMQ Streams administrators. For more information, see Section 4.6, “Designating AMQ Streams administrators” .

A CRD defines a new **kind** of resource, such as **kind:Kafka**, within an OpenShift cluster.

The Kubernetes API server allows custom resources to be created based on the **kind** and understands from the CRD how to validate and store the custom resource when it is added to the OpenShift cluster.

WARNING

When CRDs are deleted, custom resources of that type are also deleted. Additionally, the resources created by the custom resource, such as pods and statefulsets are also deleted.

Each AMQ Streams-specific custom resource conforms to the schema defined by the CRD for the resource's **kind**. The custom resources for AMQ Streams components have common configuration properties, which are defined under **spec**.

To understand the relationship between a CRD and a custom resource, let's look at a sample of the CRD for a Kafka topic.

Kafka topic CRD

```
apiVersion: kafka.strimzi.io/v1beta2
kind: CustomResourceDefinition
metadata: 1
  name: kafkatopics.kafka.strimzi.io
  labels:
    app: strimzi
spec: 2
  group: kafka.strimzi.io
  versions:
    v1beta2
  scope: Namespaced
  names:
    # ...
    singular: kafkatopic
    plural: kafkatopics
    shortNames:
    - kt 3
  additionalPrinterColumns: 4
      # ...
  subresources:
    status: {} 5
  validation: 6
    openAPIV3Schema:
      properties:
        spec:
          type: object
          properties:
            partitions:
              type: integer
              minimum: 1
            replicas:
              type: integer
              minimum: 1
              maximum: 32767
      # ...
```

1. The metadata for the topic CRD, its name and a label to identify the CRD.
2. The specification for this CRD, including the group (domain) name, the plural name and the supported schema version, which are used in the URL to access the API of the topic. The other names are used to identify instance resources in the CLI. For example, **oc get kafkatopic my-topic** or **oc get kafkatopics**.
3. The shortname can be used in CLI commands. For example, **oc get kt** can be used as an abbreviation instead of **oc get kafkatopic**.
4. The information presented when using a **get** command on the custom resource.
5. The current status of the CRD as described in the schema reference for the resource.
6. openAPIV3Schema validation provides validation for the creation of topic custom resources. For example, a topic requires at least one partition and one replica.

NOTE

You can identify the CRD YAML files supplied with the AMQ Streams installation files, because the file names contain an index number followed by 'Crd'.

Here is a corresponding example of a **KafkaTopic** custom resource.

Kafka topic custom resource

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic (1)
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster (2)
spec: (3)
  partitions: 1
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
status:
  conditions: (4)
    lastTransitionTime: "2019-08-20T11:37:00.706Z"
    status: "True"
    type: Ready
  observedGeneration: 1
  / ...
```

1. The **kind** and **apiVersion** identify the CRD of which the custom resource is an instance.
2. A label, applicable only to **KafkaTopic** and **KafkaUser** resources, that defines the name of the Kafka cluster (which is same as the name of the **Kafka** resource) to which a topic or user belongs.
3. The spec shows the number of partitions and replicas for the topic as well as the configuration parameters for the topic itself. In this example, the retention period for a message to remain in the topic and the segment file size for the log are specified.

4 Status conditions for the **KafkaTopic** resource. The **type** condition changed to **Ready** at the **lastTransitionTime**.

Custom resources can be applied to a cluster through the platform CLI. When the custom resource is created, it uses the same validation as the built-in resources of the Kubernetes API.

After a **KafkaTopic** custom resource is created, the Topic Operator is notified and corresponding Kafka topics are created in AMQ Streams.

Additional resources

- Extend the Kubernetes API with CustomResourceDefinitions
- Example configuration files provided with AMQ Streams

1.3. USING THE KAFKA BRIDGE TO CONNECT WITH A KAFKA CLUSTER

You can use the AMQ Streams Kafka Bridge API to create and manage consumers and send and receive records over HTTP rather than the native Kafka protocol.

When you set up the Kafka Bridge you configure HTTP access to the Kafka cluster. You can then use the Kafka Bridge to produce and consume messages from the cluster, as well as performing other operations through its REST interface.

Additional resources

- For information on installing and using the Kafka Bridge, see Using the AMQ Streams Kafka Bridge.

1.4. DOCUMENT CONVENTIONS

User-replaced values

User-replaced values, also known as *replaceables*, are shown in *italics* with angle brackets (< >). Underscores (_) are used for multi-word values. If the value refers to code or commands, **monospace** is also used.

For example, in the following code, you will want to replace ***<my_namespace>*** with the name of your namespace:

```
sed -i 's/namespace: .*/namespace: <my_namespace>/' install/cluster-operator/*RoleBinding*.yaml
```

1.5. ADDITIONAL RESOURCES

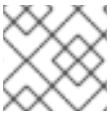
- AMQ Streams Overview
- Custom resource API reference
- Using the AMQ Streams Kafka Bridge

CHAPTER 2. AMQ STREAMS INSTALLATION METHODS

You can install AMQ Streams on OpenShift 4.10 to 4.13 in two ways.

Installation method	Description
Installation artifacts (YAML files)	Download *Red Hat AMQ Streams 2.4 OpenShift Installation and Example Files* from the AMQ Streams software downloads page. Deploy the YAML installation artifacts to your OpenShift cluster using **oc**. You start by deploying the Cluster Operator from **install/cluster-operator** to a single namespace, multiple namespaces, or all namespaces. You can also use the **install/** artifacts to deploy the following: ● AMQ Streams administrator roles (**strimzi-admin**) ● A standalone Topic Operator (**topic-operator**) ● A standalone User Operator (**user-operator**) ● AMQ Streams Drain Cleaner (**drain-cleaner**)
OperatorHub	Use the **AMQ Streams** operator in the OperatorHub to deploy AMQ Streams to a single namespace or all namespaces.

For the greatest flexibility, choose the installation artifacts method. The OperatorHub method provides a standard configuration and allows you to take advantage of automatic updates.



NOTE

Installation of AMQ Streams using Helm is not supported.

CHAPTER 3. WHAT IS DEPLOYED WITH AMQ STREAMS

Apache Kafka components are provided for deployment to OpenShift with the AMQ Streams distribution. The Kafka components are generally run as clusters for availability.

A typical deployment incorporating Kafka components might include:

- **Kafka** cluster of broker nodes
- **ZooKeeper** cluster of replicated ZooKeeper instances
- **Kafka Connect** cluster for external data connections
- **Kafka MirrorMaker** cluster to mirror the Kafka cluster in a secondary cluster
- **Kafka Exporter** to extract additional Kafka metrics data for monitoring
- **Kafka Bridge** to make HTTP-based requests to the Kafka cluster

Not all of these components are mandatory, though you need Kafka and ZooKeeper as a minimum. Some components can be deployed without Kafka, such as MirrorMaker or Kafka Connect.

3.1. ORDER OF DEPLOYMENT

The required order of deployment to an OpenShift cluster is as follows:

1. Deploy the Cluster Operator to manage your Kafka cluster
2. Deploy the Kafka cluster with the ZooKeeper cluster, and include the Topic Operator and User Operator in the deployment
3. Optionally deploy:
    - The Topic Operator and User Operator standalone if you did not deploy them with the Kafka cluster
    - Kafka Connect
    - Kafka MirrorMaker
    - Kafka Bridge
    - Components for the monitoring of metrics

The Cluster Operator creates OpenShift resources for the components, such as **Deployment**, **Service**, and **Pod** resources. The names of the OpenShift resources are appended with the name specified for a component when it's deployed. For example, a Kafka cluster named **my-kafka-cluster** has a service named **my-kafka-cluster-kafka**.

CHAPTER 4. PREPARING FOR YOUR AMQ STREAMS DEPLOYMENT

This section shows how you prepare for an AMQ Streams deployment, describing:

- The prerequisites you need before you can deploy AMQ Streams
- How to download the AMQ Streams release artifacts to use in your deployment
- How to push the AMQ Streams container images into your own registry (if required)
- How to set up *admin* roles for configuration of custom resources used in deployment

NOTE

To run the commands in this guide, your cluster user must have the rights to manage role-based access control (RBAC) and CRDs.

4.1. DEPLOYMENT PREREQUISITES

To deploy AMQ Streams, you will need the following:

- An OpenShift 4.10 to 4.13 cluster.
  AMQ Streams is based on Strimzi 0.34.x.
- The **oc** command-line tool is installed and configured to connect to the running cluster.

4.2. DOWNLOADING AMQ STREAMS RELEASE ARTIFACTS

To use deployment files to install AMQ Streams, download and extract the files from the AMQ Streams software downloads page.

AMQ Streams release artifacts include sample YAML files to help you deploy the components of AMQ Streams to OpenShift, perform common operations, and configure your Kafka cluster.

Use **oc** to deploy the Cluster Operator from the **install/cluster-operator** folder of the downloaded ZIP file. For more information about deploying and configuring the Cluster Operator, see Section 6.2, "Deploying the Cluster Operator".

In addition, if you want to use standalone installations of the Topic and User Operators with a Kafka cluster that is not managed by the AMQ Streams Cluster Operator, you can deploy them from the **install/topic-operator** and **install/user-operator** folders.

NOTE

Additionally, AMQ Streams container images are available through the Red Hat Ecosystem Catalog. However, we recommend that you use the YAML files provided to deploy AMQ Streams.

4.3. EXAMPLE CONFIGURATION AND DEPLOYMENT FILES

Use the example configuration and deployment files provided with AMQ Streams to deploy Kafka components with different configurations and monitor your deployment. Example configuration files for

custom resources contain important properties and values, which you can extend with additional supported configuration properties for your own deployment.

4.3.1. Example files location

The example files are provided with the downloadable release artifacts from the AMQ Streams software downloads page.

You can download and apply the examples using the **oc** command-line tool. The examples can serve as a starting point when building your own Kafka component configuration for deployment.

NOTE

If you installed AMQ Streams using the Operator, you can still download the example files and use them to upload configuration.

4.3.2. Example files provided with AMQ Streams

The release artifacts include an **examples** directory that contains the configuration examples.

Examples directory

```
examples
├── user 1
├── topic 2
├── security 3
│   ├── tls-auth
│   ├── scram-sha-512-auth
│   └── keycloak-authorization
├── mirror-maker 4
├── metrics 5
├── kafka 6
├── cruise-control 7
├── connect 8
└── bridge 9
```

1. **KafkaUser** custom resource configuration, which is managed by the User Operator.
2. **KafkaTopic** custom resource configuration, which is managed by Topic Operator.
3. Authentication and authorization configuration for Kafka components. Includes example configuration for TLS and SCRAM-SHA-512 authentication. The Red Hat Single Sign-On example includes **Kafka** custom resource configuration and a Red Hat Single Sign-On realm specification. You can use the example to try Red Hat Single Sign-On authorization services. There is also an example with enabled **oauth** authentication and **keycloak** authorization metrics.
4. **Kafka** custom resource configuration for a deployment of Mirror Maker. Includes example configuration for replication policy and synchronization frequency.
5. Metrics configuration, including Prometheus installation and Grafana dashboard files.
6. **Kafka** custom resource configuration for a deployment of Kafka. Includes example configuration for an ephemeral or persistent single or multi-node deployment.

7 **Kafka** custom resource with a deployment configuration for Cruise Control. Includes **KafkaRebalance** custom resources to generate optimizations proposals from Cruise Control, with

8 **KafkaConnect** and **KafkaConnector** custom resource configuration for a deployment of Kafka Connect. Includes example configuration for a single or multi-node deployment.

9 **KafkaBridge** custom resource configuration for a deployment of Kafka Bridge.

4.4. PUSHING CONTAINER IMAGES TO YOUR OWN REGISTRY

Container images for AMQ Streams are available in the Red Hat Ecosystem Catalog . The installation YAML files provided by AMQ Streams will pull the images directly from the Red Hat Ecosystem Catalog .

If you do not have access to the Red Hat Ecosystem Catalog or want to use your own container repository, do the following:

1. Pull **all** container images listed here
2. Push them into your own registry
3. Update the image names in the installation YAML files

NOTE

Each Kafka version supported for the release has a separate image.

Container image	Namespace/Repository	Description
Kafka	• registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 • registry.redhat.io/amq-streams/kafka-33-rhel8:2.4.0	AMQ Streams image for running Kafka, including: • Kafka Broker • Kafka Connect • Kafka MirrorMaker • ZooKeeper • TLS Sidecars
Operator	• registry.redhat.io/amq-streams/strimzi-rhel8-operator:2.4.0	AMQ Streams image for running the operators: • Cluster Operator • Topic Operator • User Operator • Kafka Initializer

Container image	Namespace/Repository	Description
Kafka Bridge	• registry.redhat.io/amq-streams/bridge-rhel8:2.4.0	AMQ Streams image for running the AMQ Streams Kafka Bridge
AMQ Streams Drain Cleaner	• registry.redhat.io/amq-streams/drain-cleaner-rhel8:2.4.0	AMQ Streams image for running the AMQ Streams Drain Cleaner

4.5. CREATING A PULL SECRET FOR AUTHENTICATION TO THE CONTAINER IMAGE REGISTRY

The installation YAML files provided by AMQ Streams pull container images directly from the Red Hat Ecosystem Catalog. If an AMQ Streams deployment requires authentication, configure authentication credentials in a secret and add it to the installation YAML.

NOTE

Authentication is not usually required, but might be requested on certain platforms.

Prerequisites

- You need your Red Hat username and password or the login details from your Red Hat registry service account.

NOTE

You can use your Red Hat subscription to create a registry service account from the Red Hat Customer Portal.

Procedure

1. Create a pull secret containing your login details and the container registry where the AMQ Streams image is pulled from:

```
oc create secret docker-registry <pull_secret_name> \
  --docker-server=registry.redhat.io \
  --docker-username=<user_name> \
  --docker-password=<password> \
  --docker-email=<email>
```

   Add your user name and password. The email address is optional.

2. Edit the **install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** deployment file to specify the pull secret using the **STRIMZI_IMAGE_PULL_SECRET** environment variable:

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: strimzi-cluster-operator
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
          - name: STRIMZI_IMAGE_PULL_SECRETS
            value: "<pull_secret_name>"
# ...
```

The secret applies to all pods created by the Cluster Operator.

4.6. DESIGNATING AMQ STREAMS ADMINISTRATORS

AMQ Streams provides custom resources for configuration of your deployment. By default, permission to view, create, edit, and delete these resources is limited to OpenShift cluster administrators. AMQ Streams provides two cluster roles that you can use to assign these rights to other users:

- **strimzi-view** allows users to view and list AMQ Streams resources.
- **strimzi-admin** allows users to also create, edit or delete AMQ Streams resources.

When you install these roles, they will automatically aggregate (add) these rights to the default OpenShift cluster roles. **strimzi-view** aggregates to the **view** role, and **strimzi-admin** aggregates to the **edit** and **admin** roles. Because of the aggregation, you might not need to assign these roles to users who already have similar rights.

The following procedure shows how to assign a **strimzi-admin** role that allows non-cluster administrators to manage AMQ Streams resources.

A system administrator can designate AMQ Streams administrators after the Cluster Operator is deployed.

Prerequisites

- The AMQ Streams Custom Resource Definitions (CRDs) and role-based access control (RBAC) resources to manage the CRDs have been deployed with the Cluster Operator.

Procedure

1. Create the **strimzi-view** and **strimzi-admin** cluster roles in OpenShift.

```
oc create -f install/strimzi-admin
```

2. If needed, assign the roles that provide access rights to users that require them.

```
oc create clusterrolebinding strimzi-admin --clusterrole=strimzi-admin --user=user1 --
user=user2
```

CHAPTER 5. INSTALLING AMQ STREAMS FROM THE OPERATORHUB USING THE WEB CONSOLE

Install the AMQ Streams operator from the OperatorHub in the OpenShift Container Platform web console.

The procedures in this section show how to:

- Install the AMQ Streams operator from the OperatorHub
- Deploy Kafka components using the AMQ Streams operator

5.1. INSTALLING THE AMQ STREAMS OPERATOR FROM THE OPERATORHUB

You can install and subscribe to the AMQ Streams operator using the OperatorHub in the OpenShift Container Platform web console.

This procedure describes how to create a project and install the AMQ Streams operator to that project. A project is a representation of a namespace. For manageability, it is a good practice to use namespaces to separate functions.

WARNING

Make sure you use the appropriate update channel. If you are on a supported version of OpenShift, installing AMQ Streams from the default stable channel is generally safe. However, we do not recommend enabling automatic updates on the stable channel. An automatic upgrade will skip any necessary steps prior to upgrade. Use automatic upgrades only on version-specific channels.

Prerequisites

- Access to an OpenShift Container Platform web console using an account with **cluster-admin** or **strimzi-admin** permissions.

Procedure

1. Navigate in the OpenShift web console to the **Home > Projects** page and create a project (namespace) for the installation.
   We use a project named **amq-streams-kafka** in this example.
2. Navigate to the **Operators > OperatorHub** page.
3. Scroll or type a keyword into the **Filter by keyword** box to find the **AMQ Streams** operator.
   The operator is located in the **Streaming & Messaging** category.
4. Click **AMQ Streams** to display the operator information.
5. Read the information about the operator and click **Install**.

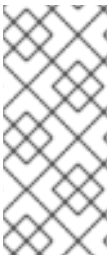
6. On the **Install Operator** page, choose from the following installation and update options:

   - **Update Channel**: Choose the update channel for the operator.
     - The (default) **stable** channel contains all the latest updates and releases, including major, minor, and micro releases, which are assumed to be well tested and stable.
     - An **amq-streams-*X*.x** channel contains the minor and micro release updates for a major release, where *X* is the major release version number.
     - An **amq-streams-*X.Y*.x** channel contains the micro release updates for a minor release, where *X* is the major release version number and *Y* is the minor release version number.
   - **Installation Mode**: Choose the project you created to install the operator on a specific namespace.
     You can install the AMQ Streams operator to all namespaces in the cluster (the default option) or a specific namespace. We recommend that you dedicate a specific namespace to the Kafka cluster and other AMQ Streams components.
   - **Update approval**: By default, the AMQ Streams operator is automatically upgraded to the latest AMQ Streams version by the Operator Lifecycle Manager (OLM). Optionally, select **Manual** if you want to manually approve future upgrades. For more information, see the Operators guide in the OpenShift documentation.

7. Click **Install** to install the operator to your selected namespace.
   The AMQ Streams operator deploys the Cluster Operator, CRDs, and role-based access control (RBAC) resources to the selected namespace.

8. After the operator is ready for use, navigate to **Operators > Installed Operators** to verify that the operator has installed to the selected namespace.
   The status will show as **Succeeded**.

   You can now use the AMQ Streams operator to deploy Kafka components, starting with a Kafka cluster.



NOTE

If you navigate to **Workloads > Deployments**, you can see the deployment details for the Cluster Operator and Entity Operator. The name of the Cluster Operator includes a version number: **amq-streams-cluster-operator-<version>**. The name is different when deploying the Cluster Operator using the AMQ Streams installation artifacts. In this case, the name is **strimzi-cluster-operator**.

5.2. DEPLOYING KAFKA COMPONENTS USING THE AMQ STREAMS OPERATOR

When installed on Openshift, the AMQ Streams operator makes Kafka components available for installation from the user interface.

The following Kafka components are available for installation:

- Kafka
- Kafka Connect
- Kafka MirrorMaker

- Kafka MirrorMaker 2
- Kafka Topic
- Kafka User
- Kafka Bridge
- Kafka Connector
- Kafka Rebalance

You select the component and create an instance. As a minimum, you create a Kafka instance. This procedure describes how to create a Kafka instance using the default settings. You can configure the default installation specification before you perform the installation.

The process is the same for creating instances of other Kafka components.

Prerequisites

- The AMQ Streams operator is installed on the OpenShift cluster.

Procedure

1. Navigate in the web console to the **Operators > Installed Operators** page and click **AMQ Streams** to display the operator details.
   From **Provided APIs**, you can create instances of Kafka components.
2. Click **Create instance** under **Kafka** to create a Kafka instance.
   By default, you'll create a Kafka cluster called **my-cluster** with three Kafka broker nodes and three ZooKeeper nodes. The cluster uses ephemeral storage.
3. Click **Create** to start the installation of Kafka.
   Wait until the status changes to **Ready**.

CHAPTER 6. DEPLOYING AMQ STREAMS USING INSTALLATION ARTIFACTS

Having prepared your environment for a deployment of AMQ Streams , you can deploy AMQ Streams to an OpenShift cluster. Use the installation files provided with the release artifacts.

AMQ Streams is based on Strimzi 0.34.x. You can deploy AMQ Streams 2.4 on OpenShift 4.10 to 4.13.

The steps to deploy AMQ Streams using the installation files are as follows:

1. Deploy the Cluster Operator
2. Use the Cluster Operator to deploy the following:
    a. Kafka cluster
    b. Topic Operator
    c. User Operator
3. Optionally, deploy the following Kafka components according to your requirements:
    - Kafka Connect
    - Kafka MirrorMaker
    - Kafka Bridge

NOTE

To run the commands in this guide, an OpenShift user must have the rights to manage role-based access control (RBAC) and CRDs.

6.1. BASIC DEPLOYMENT PATH

You can set up a deployment where AMQ Streams manages a single Kafka cluster in the same namespace. You might use this configuration for development or testing. Or you can use AMQ Streams in a production environment to manage a number of Kafka clusters in different namespaces.

The first step for any deployment of AMQ Streams is to install the Cluster Operator using the **install/cluster-operator** files.

A single command applies all the installation files in the **cluster-operator** folder: **oc apply -f ./install/cluster-operator**.

The command sets up everything you need to be able to create and manage a Kafka deployment, including the following:

- Cluster Operator (**Deployment**, **ConfigMap**)
- AMQ Streams CRDs (**CustomResourceDefinition**)
- RBAC resources (**ClusterRole**, **ClusterRoleBinding**, **RoleBinding**)
- Service account (**ServiceAccount**)

The basic deployment path is as follows:

1. Download the release artifacts
2. Create an OpenShift namespace in which to deploy the Cluster Operator
3. Deploy the Cluster Operator
    a. Update the **install/cluster-operator** files to use the namespace created for the Cluster Operator
    b. Install the Cluster Operator to watch one, multiple, or all namespaces
4. Create a Kafka cluster

After which, you can deploy other Kafka components and set up monitoring of your deployment.

6.2. DEPLOYING THE CLUSTER OPERATOR

The Cluster Operator is responsible for deploying and managing Kafka clusters within an OpenShift cluster.

When the Cluster Operator is running, it starts to watch for updates of Kafka resources.

By default, a single replica of the Cluster Operator is deployed. You can add replicas with leader election so that additional Cluster Operators are on standby in case of disruption. For more information, see Section 13.2.5, “Running multiple Cluster Operator replicas with leader election” .

6.2.1. Specifying the namespaces the Cluster Operator watches

The Cluster Operator watches for updates in the namespaces where the Kafka resources are deployed. When you deploy the Cluster Operator, you specify which namespaces to watch. You can specify the following namespaces:

- A single namespace (the same namespace containing the Cluster Operator)
- Multiple namespaces
- All namespaces

NOTE

The Cluster Operator can watch one, multiple, or all namespaces in an OpenShift cluster. The Topic Operator and User Operator watch for **KafkaTopic** and **KafkaUser** resources in a single namespace. For more information, see Section 13.1, “Watching namespaces with AMQ Streams operators”.

The Cluster Operator watches for changes to the following resources:

- **Kafka** for the Kafka cluster.
- **KafkaConnect** for the Kafka Connect cluster.
- **KafkaConnector** for creating and managing connectors in a Kafka Connect cluster.
- **KafkaMirrorMaker** for the Kafka MirrorMaker instance.

- **KafkaMirrorMaker2** for the Kafka MirrorMaker 2 instance.
- **KafkaBridge** for the Kafka Bridge instance.
- **KafkaRebalance** for the Cruise Control optimization requests.

When one of these resources is created in the OpenShift cluster, the operator gets the cluster description from the resource and starts creating a new cluster for the resource by creating the necessary OpenShift resources, such as StatefulSets, Services and ConfigMaps.

Each time a Kafka resource is updated, the operator performs corresponding updates on the OpenShift resources that make up the cluster for the resource.

Resources are either patched or deleted, and then recreated in order to make the cluster for the resource reflect the desired state of the cluster. This operation might cause a rolling update that might lead to service disruption.

When a resource is deleted, the operator undeploys the cluster and deletes all related OpenShift resources.

6.2.2. Deploying the Cluster Operator to watch a single namespace

This procedure shows how to deploy the Cluster Operator to watch AMQ Streams resources in a single namespace in your OpenShift cluster.

Prerequisites

- You need an account with permission to create and manage **CustomResourceDefinition** and RBAC (**ClusterRole**, and **RoleBinding**) resources.

Procedure

1. Edit the AMQ Streams installation files to use the namespace the Cluster Operator is going to be installed into.
   For example, in this procedure the Cluster Operator is installed into the namespace **my-cluster-operator-namespace**.

   On Linux, use:

   ```
   sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

   On MacOS, use:

   ```
   sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

2. Deploy the Cluster Operator:

   ```
   oc create -f install/cluster-operator -n my-cluster-operator-namespace
   ```

3. Check the status of the deployment:

   ```
   oc get deployments -n my-cluster-operator-namespace
   ```

Output shows the deployment name and readiness

```
NAME                      READY  UP-TO-DATE  AVAILABLE
strimzi-cluster-operator  1/1    1           1
```

READY shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

6.2.3. Deploying the Cluster Operator to watch multiple namespaces

This procedure shows how to deploy the Cluster Operator to watch AMQ Streams resources across multiple namespaces in your OpenShift cluster.

Prerequisites

- You need an account with permission to create and manage **CustomResourceDefinition** and RBAC (**ClusterRole**, and **RoleBinding**) resources.

Procedure

1. Edit the AMQ Streams installation files to use the namespace the Cluster Operator is going to be installed into.
   For example, in this procedure the Cluster Operator is installed into the namespace **my-cluster-operator-namespace**.

   On Linux, use:

   ```
   sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

   On MacOS, use:

   ```
   sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

2. Edit the **install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** file to add a list of all the namespaces the Cluster Operator will watch to the **STRIMZI_NAMESPACE** environment variable.
   For example, in this procedure the Cluster Operator will watch the namespaces **watched-namespace-1**, **watched-namespace-2**, **watched-namespace-3**.

   ```
   apiVersion: apps/v1
   kind: Deployment
   spec:
     # ...
     template:
       spec:
         serviceAccountName: strimzi-cluster-operator
         containers:
         - name: strimzi-cluster-operator
           image: registry.redhat.io/amq-streams/strimzi-rhel8-operator:2.4.0
           imagePullPolicy: IfNotPresent
   ```

```
      env:
      - name: STRIMZI_NAMESPACE
        value: watched-namespace-1,watched-namespace-2,watched-namespace-3
```

3. For each namespace listed, install the **RoleBindings**.
   In this example, we replace ***watched-namespace*** in these commands with the namespaces listed in the previous step, repeating them for **watched-namespace-1**, **watched-namespace-2**, **watched-namespace-3**:

```
oc create -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -n <watched_namespace>
oc create -f install/cluster-operator/023-RoleBinding-strimzi-cluster-operator.yaml -n <watched_namespace>
oc create -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-operator-delegation.yaml -n <watched_namespace>
```

4. Deploy the Cluster Operator:

```
oc create -f install/cluster-operator -n my-cluster-operator-namespace
```

5. Check the status of the deployment:

```
oc get deployments -n my-cluster-operator-namespace
```

   Output shows the deployment name and readiness

```
NAME                      READY  UP-TO-DATE  AVAILABLE
strimzi-cluster-operator  1/1    1           1
```

   READY shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

6.2.4. Deploying the Cluster Operator to watch all namespaces

This procedure shows how to deploy the Cluster Operator to watch AMQ Streams resources across all namespaces in your OpenShift cluster.

When running in this mode, the Cluster Operator automatically manages clusters in any new namespaces that are created.

Prerequisites

- You need an account with permission to create and manage **CustomResourceDefinition** and RBAC (**ClusterRole**, and **RoleBinding**) resources.

Procedure

1. Edit the AMQ Streams installation files to use the namespace the Cluster Operator is going to be installed into.
   For example, in this procedure the Cluster Operator is installed into the namespace **my-cluster-operator-namespace**.

   On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
```

2. Edit the **install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** file to set the value of the **STRIMZI_NAMESPACE** environment variable to ***.

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      # ...
      serviceAccountName: strimzi-cluster-operator
      containers:
      - name: strimzi-cluster-operator
        image: registry.redhat.io/amq-streams/strimzi-rhel8-operator:2.4.0
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: "*"
        # ...
```

3. Create **ClusterRoleBindings** that grant cluster-wide access for all namespaces to the Cluster Operator.

```
oc create clusterrolebinding strimzi-cluster-operator-namespaced --clusterrole=strimzi-cluster-operator-namespaced --serviceaccount my-cluster-operator-namespace:strimzi-cluster-operator
oc create clusterrolebinding strimzi-cluster-operator-watched --clusterrole=strimzi-cluster-operator-watched --serviceaccount my-cluster-operator-namespace:strimzi-cluster-operator
oc create clusterrolebinding strimzi-cluster-operator-entity-operator-delegation --clusterrole=strimzi-entity-operator --serviceaccount my-cluster-operator-namespace:strimzi-cluster-operator
```

4. Deploy the Cluster Operator to your OpenShift cluster.

```
oc create -f install/cluster-operator -n my-cluster-operator-namespace
```

5. Check the status of the deployment:

```
oc get deployments -n my-cluster-operator-namespace
```

Output shows the deployment name and readiness

```
NAME                      READY  UP-TO-DATE  AVAILABLE
strimzi-cluster-operator  1/1    1           1
```

READY shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

6.3. DEPLOYING KAFKA

To be able to manage a Kafka cluster with the Cluster Operator, you must deploy it as a **Kafka** resource. AMQ Streams provides example deployment files to do this. You can use these files to deploy the Topic Operator and User Operator at the same time.

After you have deployed the Cluster Operator, use a **Kafka** resource to deploy the following components:

- Kafka cluster
- Topic Operator
- User Operator

When installing Kafka, AMQ Streams also installs a ZooKeeper cluster and adds the necessary configuration to connect Kafka with ZooKeeper.

If you haven't deployed a Kafka cluster as a **Kafka** resource, you can't use the Cluster Operator to manage it. This applies, for example, to a Kafka cluster running outside of OpenShift. However, you can use the Topic Operator and User Operator with a Kafka cluster that is **not managed** by AMQ Streams, by deploying them as standalone components. You can also deploy and use other Kafka components with a Kafka cluster not managed by AMQ Streams.

6.3.1. Deploying the Kafka cluster

This procedure shows how to deploy a Kafka cluster to your OpenShift cluster using the Cluster Operator.

The deployment uses a YAML file to provide the specification to create a **Kafka** resource.

AMQ Streams provides the following example files you can use to create a Kafka cluster:

kafka-persistent.yaml

Deploys a persistent cluster with three ZooKeeper and three Kafka nodes.

kafka-jbod.yaml

Deploys a persistent cluster with three ZooKeeper and three Kafka nodes (each using multiple persistent volumes).

kafka-persistent-single.yaml

Deploys a persistent cluster with a single ZooKeeper node and a single Kafka node.

kafka-ephemeral.yaml

Deploys an ephemeral cluster with three ZooKeeper and three Kafka nodes.

kafka-ephemeral-single.yaml

Deploys an ephemeral cluster with three ZooKeeper nodes and a single Kafka node.

In this procedure, we use the examples for an *ephemeral* and *persistent* Kafka cluster deployment.

Ephemeral cluster

In general, an ephemeral (or temporary) Kafka cluster is suitable for development and testing

purposes, not for production. This deployment uses **emptyDir** volumes for storing broker information (for ZooKeeper) and topics or partitions (for Kafka). Using an **emptyDir** volume means that its content is strictly related to the pod life cycle and is deleted when the pod goes down.

Persistent cluster

A persistent Kafka cluster uses persistent volumes to store ZooKeeper and Kafka data. A **PersistentVolume** is acquired using a **PersistentVolumeClaim** to make it independent of the actual type of the **PersistentVolume**. The **PersistentVolumeClaim** can use a **StorageClass** to trigger automatic volume provisioning. When no **StorageClass** is specified, OpenShift will try to use the default **StorageClass**.
The following examples show some common types of persistent volumes:

- If your OpenShift cluster runs on Amazon AWS, OpenShift can provision Amazon EBS volumes
- If your OpenShift cluster runs on Microsoft Azure, OpenShift can provision Azure Disk Storage volumes
- If your OpenShift cluster runs on Google Cloud, OpenShift can provision Persistent Disk volumes
- If your OpenShift cluster runs on bare metal, OpenShift can provision local persistent volumes

The example YAML files specify the latest supported Kafka version, and configuration for its supported log message format version and inter-broker protocol version. The **inter.broker.protocol.version** property for the Kafka **config** must be the version supported by the specified Kafka version (**spec.kafka.version**). The property represents the version of Kafka protocol used in a Kafka cluster.

From Kafka 3.0.0, when the **inter.broker.protocol.version** is set to **3.0** or higher, the **log.message.format.version** option is ignored and doesn't need to be set.

An update to the **inter.broker.protocol.version** is required when upgrading Kafka.

The example clusters are named **my-cluster** by default. The cluster name is defined by the name of the resource and cannot be changed after the cluster has been deployed. To change the cluster name before you deploy the cluster, edit the **Kafka.metadata.name** property of the **Kafka** resource in the relevant YAML file.

Default cluster name and specified Kafka versions

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    version: 3.4.0
    #...
    config:
      #...
      log.message.format.version: "3.4"
      inter.broker.protocol.version: "3.4"
  # ...
```

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create and deploy an ephemeral or persistent cluster.
   - To create and deploy an ephemeral cluster:

     ```
     oc apply -f examples/kafka/kafka-ephemeral.yaml
     ```

   - To create and deploy a persistent cluster:

     ```
     oc apply -f examples/kafka/kafka-persistent.yaml
     ```

2. Check the status of the deployment:

   ```
   oc get pods -n <my_cluster_operator_namespace>
   ```

   Output shows the pod names and readiness

   ```
   NAME                          READY   STATUS    RESTARTS
   my-cluster-entity-operator  3/3     Running   0
   my-cluster-kafka-0          1/1     Running   0
   my-cluster-kafka-1          1/1     Running   0
   my-cluster-kafka-2          1/1     Running   0
   my-cluster-zookeeper-0      1/1     Running   0
   my-cluster-zookeeper-1      1/1     Running   0
   my-cluster-zookeeper-2      1/1     Running   0
   ```

   my-cluster is the name of the Kafka cluster.

   A sequential index number starting with **0** identifies each Kafka and ZooKeeper pod created.

   With the default deployment, you create an Entity Operator cluster, 3 Kafka pods, and 3 ZooKeeper pods.

   READY shows the number of replicas that are ready/expected. The deployment is successful when the **STATUS** shows as **Running**.

Additional resources

Kafka cluster configuration

6.3.2. Deploying the Topic Operator using the Cluster Operator

This procedure describes how to deploy the Topic Operator using the Cluster Operator.

You configure the **entityOperator** property of the **Kafka** resource to include the **topicOperator**. By default, the Topic Operator watches for **KafkaTopic** resources in the namespace of the Kafka cluster deployed by the Cluster Operator. You can also specify a namespace using **watchedNamespace** in the Topic Operator **spec**. A single Topic Operator can watch a single namespace. One namespace should be watched by only one Topic Operator.

If you use AMQ Streams to deploy multiple Kafka clusters into the same namespace, enable the Topic Operator for only one Kafka cluster or use the **watchedNamespace** property to configure the Topic Operators to watch other namespaces.

If you want to use the Topic Operator with a Kafka cluster that is not managed by AMQ Streams, you must deploy the Topic Operator as a standalone component .

For more information about configuring the **entityOperator** and **topicOperator** properties, see Configuring the Entity Operator.

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Edit the **entityOperator** properties of the **Kafka** resource to include **topicOperator**:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the Topic Operator **spec** using the properties described in the **EntityTopicOperatorSpec** schema reference.
   Use an empty object (**{}**) if you want all properties to use their default values.

3. Create or update the resource:

```
oc apply -f <kafka_configuration_file>
```

4. Check the status of the deployment:

```
oc get pods -n <my_cluster_operator_namespace>
```

Output shows the pod name and readiness

```
NAME                          READY   STATUS    RESTARTS
my-cluster-entity-operator  3/3     Running   0
# ...
```

my-cluster is the name of the Kafka cluster.

READY shows the number of replicas that are ready/expected. The deployment is successful when the **STATUS** shows as **Running**.

6.3.3. Deploying the User Operator using the Cluster Operator

This procedure describes how to deploy the User Operator using the Cluster Operator.

You configure the **entityOperator** property of the **Kafka** resource to include the **userOperator**. By default, the User Operator watches for **KafkaUser** resources in the namespace of the Kafka cluster deployment. You can also specify a namespace using **watchedNamespace** in the User Operator **spec**. A single User Operator can watch a single namespace. One namespace should be watched by only one User Operator.

If you want to use the User Operator with a Kafka cluster that is not managed by AMQ Streams, you must deploy the User Operator as a standalone component.

For more information about configuring the **entityOperator** and **userOperator** properties, see Configuring the Entity Operator.

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Edit the **entityOperator** properties of the **Kafka** resource to include **userOperator**:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the User Operator **spec** using the properties described in **EntityUserOperatorSpec** schema reference.
   Use an empty object (**{}**) if you want all properties to use their default values.

3. Create or update the resource:

```
oc apply -f <kafka_configuration_file>
```

4. Check the status of the deployment:

```
oc get pods -n <my_cluster_operator_namespace>
```

Output shows the pod name and readiness

```
NAME                          READY   STATUS    RESTARTS
my-cluster-entity-operator  3/3     Running   0
# ...
```

my-cluster is the name of the Kafka cluster.

READY shows the number of replicas that are ready/expected. The deployment is successful when the **STATUS** shows as **Running**.

6.4. DEPLOYING KAFKA CONNECT

Kafka Connect is a tool for streaming data between Apache Kafka and other systems. For example, Kafka Connect might integrate Kafka with external databases or storage and messaging systems.

In AMQ Streams, Kafka Connect is deployed in distributed mode. Kafka Connect can also work in standalone mode, but this is not supported by AMQ Streams.

Using the concept of *connectors*, Kafka Connect provides a framework for moving large amounts of data into and out of your Kafka cluster while maintaining scalability and reliability.

The Cluster Operator manages Kafka Connect clusters deployed using the **KafkaConnect** resource and connectors created using the **KafkaConnector** resource.

In order to use Kafka Connect, you need to do the following.

- Deploy a Kafka Connect cluster
- Add connectors to integrate with other systems

NOTE

The term *connector* is used interchangeably to mean a connector instance running within a Kafka Connect cluster, or a connector class. In this guide, the term *connector* is used when the meaning is clear from the context.

6.4.1. Deploying Kafka Connect to your OpenShift cluster

This procedure shows how to deploy a Kafka Connect cluster to your OpenShift cluster using the Cluster Operator.

A Kafka Connect cluster deployment is implemented with a configurable number of nodes (also called *workers*) that distribute the workload of connectors as *tasks* so that the message flow is highly scalable and reliable.

The deployment uses a YAML file to provide the specification to create a **KafkaConnect** resource.

AMQ Streams provides example configuration files. In this procedure, we use the following example file:

- **examples/connect/kafka-connect.yaml**

Prerequisites

- The Cluster Operator must be deployed.
- Running Kafka cluster.

Procedure

1. Deploy Kafka Connect to your OpenShift cluster. Use the **examples/connect/kafka-connect.yaml** file to deploy Kafka Connect.

```
oc apply -f examples/connect/kafka-connect.yaml
```

2. Check the status of the deployment:

```
oc get pods -n <my_cluster_operator_namespace>
```

Output shows the deployment name and readiness

```
NAME                                READY  STATUS   RESTARTS
my-connect-cluster-connect-<pod_id>  1/1    Running  0
```

my-connect-cluster is the name of the Kafka Connect cluster.

A pod ID identifies each pod created.

With the default deployment, you create a single Kafka Connect pod.

READY shows the number of replicas that are ready/expected. The deployment is successful when the **STATUS** shows as **Running**.

Additional resources

Kafka Connect cluster configuration

6.4.2. Configuring Kafka Connect for multiple instances

If you are running multiple instances of Kafka Connect, you have to change the default configuration of the following **config** properties:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: connect-cluster (1)
    offset.storage.topic: connect-cluster-offsets (2)
    config.storage.topic: connect-cluster-configs (3)
    status.storage.topic: connect-cluster-status (4)
    # ...
# ...
```

1. The Kafka Connect cluster ID within Kafka.
2. Kafka topic that stores connector offsets.
3. Kafka topic that stores connector and task status configurations.
4. Kafka topic that stores connector and task status updates.

NOTE

Values for the three topics must be the same for all Kafka Connect instances with the same **group.id**.

Unless you change the default settings, each Kafka Connect instance connecting to the same Kafka cluster is deployed with the same values. What happens, in effect, is all instances are coupled to run in a cluster and use the same topics.

If multiple Kafka Connect clusters try to use the same topics, Kafka Connect will not work as expected and generate errors.

If you wish to run multiple Kafka Connect instances, change the values of these properties for each instance.

6.4.3. Adding connectors

Kafka Connect uses connectors to integrate with other systems to stream data. A connector is an instance of a Kafka **Connector** class, which can be one of the following type:

Source connector

A source connector is a runtime entity that fetches data from an external system and feeds it to Kafka as messages.

Sink connector

A sink connector is a runtime entity that fetches messages from Kafka topics and feeds them to an external system.

Kafka Connect uses a plugin architecture to provide the implementation artifacts for connectors. Plugins allow connections to other systems and provide additional configuration to manipulate data. Plugins include connectors and other components, such as data converters and transforms. A connector operates with a specific type of external system. Each connector defines a schema for its configuration. You supply the configuration to Kafka Connect to create a connector instance within Kafka Connect. Connector instances then define a set of tasks for moving data between systems.

Add connector plugins to Kafka Connect in one of the following ways:

- Configure Kafka Connect to build a new container image with plugins automatically
- Create a Docker image from the base Kafka Connect image (manually or using continuous integration)

After plugins have been added to the container image, you can start, stop, and manage connector instances in the following ways:

- Using AMQ Streams's **KafkaConnector** custom resource
- Using the Kafka Connect API

You can also create new connector instances using these options.

6.4.3.1. Building a new container image with connector plugins automatically

Configure Kafka Connect so that AMQ Streams automatically builds a new container image with additional connectors. You define the connector plugins using the **.spec.build.plugins** property of the **KafkaConnect** custom resource. AMQ Streams will automatically download and add the connector plugins into a new container image. The container is pushed into the container repository specified in **.spec.build.output** and automatically used in the Kafka Connect deployment.

Prerequisites

- The Cluster Operator must be deployed.
- A container registry.

You need to provide your own container registry where images can be pushed to, stored, and pulled from. AMQ Streams supports private container registries as well as public registries such as Quay or Docker Hub.

Procedure

1. Configure the **KafkaConnect** custom resource by specifying the container registry in **.spec.build.output**, and additional connectors in **.spec.build.plugins**:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: 1
  #...
  build:
    output: 2
      type: docker
      image: my-registry.io/my-org/my-connect-cluster:latest
      pushSecret: my-registry-credentials
    plugins: 3
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-postgres/2.1.3.Final/debezium-connector-postgres-2.1.3.Final-plugin.tar.gz
            sha512sum: c4ddc97846de561755dc0b021a62aba656098829c70eb3ade3b817ce06d852ca12ae50c0281cc791a5a131cb7fc21fb15f4b8ee76c6cae5dd07f9c11cb7c6e79
      - name: camel-telegram
        artifacts:
          - type: tgz
            url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-telegram-kafka-connector/0.11.5/camel-telegram-kafka-connector-0.11.5-package.tar.gz
            sha512sum: d6d9f45e0d1dbfcc9f6d1c7ca2046168c764389c78bc4b867dab32d24f710bb74ccf2a007d7d7a8af2dfca09d9a52ccbc2831fc715c195a3634cca055185bd91
  #...
```

1. The specification for the Kafka Connect cluster.
2. (Required) Configuration of the container registry where new images are pushed.
3. (Required) List of connector plugins and their artifacts to add to the new container image. Each plugin must be configured with at least one **artifact**.

2. Create or update the resource:

```
$ oc apply -f <kafka_connect_configuration_file>
```

3. Wait for the new container image to build, and for the Kafka Connect cluster to be deployed.
4. Use the Kafka Connect REST API or **KafkaConnector** custom resources to use the connector plugins you added.

Additional resources

- Kafka Connect **Build** schema reference

6.4.3.2. Building a new container image with connector plugins from the Kafka Connect base image

Create a custom Docker image with connector plugins from the Kafka Connect base image Add the custom image to the **/opt/kafka/plugins** directory.

You can use the Kafka container image on Red Hat Ecosystem Catalog as a base image for creating your own custom image with additional connector plugins.

At startup, the AMQ Streams version of Kafka Connect loads any third-party connector plugins contained in the **/opt/kafka/plugins** directory.

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create a new **Dockerfile** using **registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0** as the base image:

```
FROM registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Example plugins file

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-<version>.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mongodb-<version>.jar
│   ├── debezium-core-<version>.jar
│   ├── LICENSE.txt
│   ├── mongodb-driver-core-<version>.jar
│   ├── README.md
│   └── # ...
├── debezium-connector-mysql
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mysql-<version>.jar
```

```
│   ├── debezium-core-<version>.jar
│   ├── LICENSE.txt
│   ├── mysql-binlog-connector-java-<version>.jar
│   ├── mysql-connector-java-<version>.jar
│   ├── README.md
│   └── # ...
└── debezium-connector-postgres
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-postgres-<version>.jar
    ├── debezium-core-<version>.jar
    ├── LICENSE.txt
    ├── postgresql-<version>.jar
    ├── protobuf-java-<version>.jar
    ├── README.md
    └── # ...
```

The COPY command points to the plugin files to copy to the container image.

This example adds plugins for Debezium connectors (MongoDB, MySQL, and PostgreSQL), though not all files are listed for brevity. Debezium running in Kafka Connect looks the same as any other Kafka Connect task.

2. Build the container image.

3. Push your custom image to your container registry.

4. Point to the new container image.
You can point to the image in one of the following ways:

   - Edit the **KafkaConnect.spec.image** property of the **KafkaConnect** custom resource. If set, this property overrides the **STRIMZI_KAFKA_CONNECT_IMAGES** environment variable in the Cluster Operator.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: 1
  #...
  image: my-new-container-image 2
  config: 3
    #...
```

1 The specification for the Kafka Connect cluster.

2 The docker image for the pods.

3 Configuration of the Kafka Connect *workers* (not connectors).

   - Edit the **STRIMZI_KAFKA_CONNECT_IMAGES** environment variable in the **install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** file to point to the new container image, and then reinstall the Cluster Operator.

Additional resources

- Container image configuration and the **KafkaConnect.spec.image property**
- Cluster Operator configuration and the **STRIMZI_KAFKA_CONNECT_IMAGES** variable

6.4.3.3. Deploying KafkaConnector resources

Deploy **KafkaConnector** resources to manage connectors. The **KafkaConnector** custom resource offers an OpenShift-native approach to management of connectors by the Cluster Operator. You don't need to send HTTP requests to manage connectors, as with the Kafka Connect REST API. You manage a running connector instance by updating its corresponding **KafkaConnector** resource, and then applying the updates. The Cluster Operator updates the configurations of the running connector instances. You remove a connector by deleting its corresponding **KafkaConnector**.

KafkaConnector resources must be deployed to the same namespace as the Kafka Connect cluster they link to.

In the configuration shown in this procedure, the **autoRestart** property is set to **true**. This enables automatic restarts of failed connectors and tasks. Up to seven restart attempts are made, after which restarts must be made manually. You annotate the **KafkaConnector** resource to restart a connector or restart a connector task manually.

Example connectors

You can use your own connectors or try the examples provided by AMQ Streams. Up until Apache Kafka 3.1.0, example file connector plugins were included with Apache Kafka. Starting from the 3.1.1 and 3.2.0 releases of Apache Kafka, the examples need to be added to the plugin path as any other connector .

AMQ Streams provides an example **KafkaConnector** configuration file (**examples/connect/source-connector.yaml**) for the example file connector plugins, which creates the following connector instances as **KafkaConnector** resources:

- A **FileStreamSourceConnector** instance that reads each line from the Kafka license file (the source) and writes the data as messages to a single Kafka topic.
- A **FileStreamSinkConnector** instance that reads messages from the Kafka topic and writes the messages to a temporary file (the sink).

We use the example file to create connectors in this procedure.

NOTE

The example connectors are not intended for use in a production environment.

Prerequisites

- A Kafka Connect deployment
- The Cluster Operator is running

Procedure

1. Add the **FileStreamSourceConnector** and **FileStreamSinkConnector** plugins to Kafka Connect in one of the following ways:

- Configure Kafka Connect to build a new container image with plugins automatically
- Create a Docker image from the base Kafka Connect image (manually or using continuous integration)

2. Set the **strimzi.io/use-connector-resources annotation** to **true** in the Kafka Connect configuration.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
    # ...
```

With the **KafkaConnector** resources enabled, the Cluster Operator watches for them.

3. Edit the **examples/connect/source-connector.yaml** file:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector 1
  labels:
    strimzi.io/cluster: my-connect-cluster 2
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector 3
  tasksMax: 2 4
  autoRestart: 5
    enabled: true
  config: 6
    file: "/opt/kafka/LICENSE" 7
    topic: my-topic 8
    # ...
```

1. Name of the **KafkaConnector** resource, which is used as the name of the connector. Use any name that is valid for an OpenShift resource.
2. Name of the Kafka Connect cluster to create the connector instance in. Connectors must be deployed to the same namespace as the Kafka Connect cluster they link to.
3. Full name or alias of the connector class. This should be present in the image being used by the Kafka Connect cluster.
4. Maximum number of Kafka Connect tasks that the connector can create.
5. Enables automatic restarts of failed connectors and tasks.
6. Connector configuration as key-value pairs.
7. This example source connector configuration reads data from the **/opt/kafka/LICENSE** file.

8 Kafka topic to publish the source data to.

4. Create the source **KafkaConnector** in your OpenShift cluster:

```
oc apply -f examples/connect/source-connector.yaml
```

5. Create an **examples/connect/sink-connector.yaml** file:

```
touch examples/connect/sink-connector.yaml
```

6. Paste the following YAML into the **sink-connector.yaml** file:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector 1
  tasksMax: 2
  config: 2
    file: "/tmp/my-file" 3
    topics: my-topic 4
```

1 Full name or alias of the connector class. This should be present in the image being used by the Kafka Connect cluster.

2 Connector configuration as key-value pairs.

3 Temporary file to publish the source data to.

4 Kafka topic to read the source data from.

7. Create the sink **KafkaConnector** in your OpenShift cluster:

```
oc apply -f examples/connect/sink-connector.yaml
```

8. Check that the connector resources were created:

```
oc get kctr --selector strimzi.io/cluster=<my_connect_cluster> -o name

my-source-connector
my-sink-connector
```

Replace <my_connect_cluster> with the name of your Kafka Connect cluster.

9. In the container, execute **kafka-console-consumer.sh** to read the messages that were written to the topic by the source connector:

```
oc exec <my_kafka_cluster>-kafka-0 -i -t -- bin/kafka-console-consumer.sh --bootstrap-server <my_kafka_cluster>-kafka-bootstrap.NAMESPACE.svc:9092 --topic my-topic --from-beginning
```

Replace <my_kafka_cluster> with the name of your Kafka cluster.

Source and sink connector configuration options
The connector configuration is defined in the **spec.config** property of the **KafkaConnector** resource.

The **FileStreamSourceConnector** and **FileStreamSinkConnector** classes support the same configuration options as the Kafka Connect REST API. Other connectors support different configuration options.

Table 6.1. Configuration options for the FileStreamSource connector class

Name	Type	Default value	Description
file	String	Null	Source file to write messages to. If not specified, the standard input is used.
topic	List	Null	The Kafka topic to publish data to.

Table 6.2. Configuration options for FileStreamSinkConnector class

Name	Type	Default value	Description
file	String	Null	Destination file to write messages to. If not specified, the standard output is used.
topics	List	Null	One or more Kafka topics to read data from.
topics.regex	String	Null	A regular expression matching one or more Kafka topics to read data from.

6.4.3.4. Manually restarting connectors

If you are using **KafkaConnector** resources to manage connectors, use the **restart** annotation to manually trigger a restart of a connector.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Find the name of the **KafkaConnector** custom resource that controls the Kafka connector you want to restart:

```
oc get KafkaConnector
```

2. Restart the connector by annotating the **KafkaConnector** resource in OpenShift.

```
oc annotate KafkaConnector <kafka_connector_name> strimzi.io/restart=true
```

   The **restart** annotation is set to **true**.

3. Wait for the next reconciliation to occur (every two minutes by default).
   The Kafka connector is restarted, as long as the annotation was detected by the reconciliation process. When Kafka Connect accepts the restart request, the annotation is removed from the **KafkaConnector** custom resource.

6.4.3.5. Manually restarting Kafka connector tasks

If you are using **KafkaConnector** resources to manage connectors, use the **restart-task** annotation to manually trigger a restart of a connector task.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Find the name of the **KafkaConnector** custom resource that controls the Kafka connector task you want to restart:

```
oc get KafkaConnector
```

2. Find the ID of the task to be restarted from the **KafkaConnector** custom resource. Task IDs are non-negative integers, starting from 0:

```
oc describe KafkaConnector <kafka_connector_name>
```

3. Use the ID to restart the connector task by annotating the **KafkaConnector** resource in OpenShift:

```
oc annotate KafkaConnector <kafka_connector_name> strimzi.io/restart-task=0
```

   In this example, task **0** is restarted.

4. Wait for the next reconciliation to occur (every two minutes by default).
   The Kafka connector task is restarted, as long as the annotation was detected by the reconciliation process. When Kafka Connect accepts the restart request, the annotation is removed from the **KafkaConnector** custom resource.

6.4.3.6. Exposing the Kafka Connect API

Use the Kafka Connect REST API as an alternative to using **KafkaConnector** resources to manage connectors. The Kafka Connect REST API is available as a service running on ***<connect_cluster_name>*-connect-api:8083**, where *<connect_cluster_name>* is the name of your Kafka Connect cluster. The service is created when you create a Kafka Connect instance.

The operations supported by the Kafka Connect REST API are described in the Apache Kafka Connect API documentation.

NOTE

The **strimzi.io/use-connector-resources** annotation enables KafkaConnectors. If you applied the annotation to your **KafkaConnect** resource configuration, you need to remove it to use the Kafka Connect API. Otherwise, manual changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator.

You can add the connector configuration as a JSON object.

Example curl request to add connector configuration

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
    }
}'
```

The API is only accessible within the OpenShift cluster. If you want to make the Kafka Connect API accessible to applications running outside of the OpenShift cluster, you can expose it manually by creating one of the following features:

- **LoadBalancer** or **NodePort** type services
- **Ingress** resources (Kubernetes only)
- OpenShift routes (OpenShift only)

NOTE

The connection is insecure, so allow external access advisedly.

If you decide to create services, use the labels from the **selector** of the ***<connect_cluster_name>*-connect-api** service to configure the pods to which the service will route the traffic:

Selector configuration for the service

```
# ...
selector:
```

```
    strimzi.io/cluster: my-connect-cluster 1
    strimzi.io/kind: KafkaConnect
    strimzi.io/name: my-connect-cluster-connect 2
  #...
```

1 Name of the Kafka Connect custom resource in your OpenShift cluster.

2 Name of the Kafka Connect deployment created by the Cluster Operator.

You must also create a **NetworkPolicy** that allows HTTP requests from external clients.

Example NetworkPolicy to allow requests to the Kafka Connect API

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-custom-connect-network-policy
spec:
  ingress:
  - from:
    - podSelector: 1
        matchLabels:
          app: my-connector-manager
    ports:
    - port: 8083
      protocol: TCP
  podSelector:
    matchLabels:
      strimzi.io/cluster: my-connect-cluster
      strimzi.io/kind: KafkaConnect
      strimzi.io/name: my-connect-cluster-connect
  policyTypes:
  - Ingress
```

1 The label of the pod that is allowed to connect to the API.

To add the connector configuration outside the cluster, use the URL of the resource that exposes the API in the curl command.

6.4.3.7. Limiting access to the Kafka Connect API

It is crucial to restrict access to the Kafka Connect API only to trusted users to prevent unauthorized actions and potential security issues. The Kafka Connect API provides extensive capabilities for altering connector configurations, which makes it all the more important to take security precautions. Someone with access to the Kafka Connect API could potentially obtain sensitive information that an administrator may assume is secure.

The Kafka Connect REST API can be accessed by anyone who has authenticated access to the OpenShift cluster and knows the endpoint URL, which includes the hostname/IP address and port number.

For example, suppose an organization uses a Kafka Connect cluster and connectors to stream sensitive data from a customer database to a central database. The administrator uses a configuration provider

plugin to store sensitive information related to connecting to the customer database and the central database, such as database connection details and authentication credentials. The configuration provider protects this sensitive information from being exposed to unauthorized users. However, someone who has access to the Kafka Connect API can still obtain access to the customer database without the consent of the administrator. They can do this by setting up a fake database and configuring a connector to connect to it. They then modify the connector configuration to point to the customer database, but instead of sending the data to the central database, they send it to the fake database. By configuring the connector to connect to the fake database, the login details and credentials for connecting to the customer database are intercepted, even though they are stored securely in the configuration provider.

If you are using the **KafkaConnector** custom resources, then by default the OpenShift RBAC rules permit only OpenShift cluster administrators to make changes to connectors. You can also designate non-cluster administrators to manage AMQ Streams resources. With **KafkaConnector** resources enabled in your Kafka Connect configuration, changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator. If you are not using the **KafkaConnector** resource, the default RBAC rules do not limit access to the Kafka Connect API. If you want to limit direct access to the Kafka Connect REST API using OpenShift RBAC, you need to enable and use the **KafkaConnector** resources.

For improved security, we recommend configuring the following properties for the Kafka Connect API:

org.apache.kafka.disallowed.login.modules

(Kafka 3.4 or later) Set the **org.apache.kafka.disallowed.login.modules** Java system property to prevent the use of insecure login modules. For example, specifying **com.sun.security.auth.module.JndiLoginModule** prevents the use of the Kafka **JndiLoginModule**.

Example configuration for disallowing login modules

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  jvmOptions:
    javaSystemProperties:
      - name: org.apache.kafka.disallowed.login.modules
        value: com.sun.security.auth.module.JndiLoginModule,
org.apache.kafka.common.security.kerberos.KerberosLoginModule
  # ...
```

Only allow trusted login modules and follow the latest advice from Kafka for the version you are using. As a best practice, you should explicitly disallow insecure login modules in your Kafka Connect configuration by using the **org.apache.kafka.disallowed.login.modules** system property.

connector.client.config.override.policy

Set the **connector.client.config.override.policy** property to **None** to prevent connector configurations from overriding the Kafka Connect configuration and the consumers and producers it uses.

Example configuration to specify connector override policy

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    connector.client.config.override.policy: None
# ...
```

6.4.3.8. Switching from using the Kafka Connect API to using KafkaConnector custom resources

You can switch from using the Kafka Connect API to using **KafkaConnector** custom resources to manage your connectors. To make the switch, do the following in the order shown:

1. Deploy **KafkaConnector** resources with the configuration to create your connector instances.
2. Enable **KafkaConnector** resources in your Kafka Connect configuration by setting the **strimzi.io/use-connector-resources** annotation to **true**.

WARNING

If you enable **KafkaConnector** resources before creating them, you delete all connectors.

To switch from using **KafkaConnector** resources to using the Kafka Connect API, first remove the annotation that enables the **KafkaConnector** resources from your Kafka Connect configuration. Otherwise, manual changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator.

When making the switch, check the status of the **KafkaConnect** resource. The value of **metadata.generation** (the current version of the deployment) must match **status.observedGeneration** (the latest reconciliation of the resource). When the Kafka Connect cluster is **Ready**, you can delete the **KafkaConnector** resources.

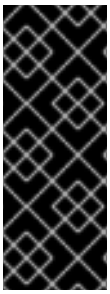
6.5. DEPLOYING KAFKA MIRRORMAKER

The Cluster Operator deploys one or more Kafka MirrorMaker replicas to replicate data between Kafka clusters. This process is called mirroring to avoid confusion with the Kafka partitions replication concept. MirrorMaker consumes messages from the source cluster and republishes those messages to the target cluster.

6.5.1. Deploying Kafka MirrorMaker to your OpenShift cluster

This procedure shows how to deploy a Kafka MirrorMaker cluster to your OpenShift cluster using the Cluster Operator.

The deployment uses a YAML file to provide the specification to create a **KafkaMirrorMaker** or **KafkaMirrorMaker2** resource depending on the version of MirrorMaker deployed.



IMPORTANT

Kafka MirrorMaker 1 (referred to as just *MirrorMaker* in the documentation) has been deprecated in Apache Kafka 3.0.0 and will be removed in Apache Kafka 4.0.0. As a result, the **KafkaMirrorMaker** custom resource which is used to deploy Kafka MirrorMaker 1 has been deprecated in AMQ Streams as well. The **KafkaMirrorMaker** resource will be removed from AMQ Streams when we adopt Apache Kafka 4.0.0. As a replacement, use the **KafkaMirrorMaker2** custom resource with the **IdentityReplicationPolicy**.

AMQ Streams provides example configuration files. In this procedure, we use the following example files:

- **examples/mirror-maker/kafka-mirror-maker.yaml**
- **examples/mirror-maker/kafka-mirror-maker-2.yaml**

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Deploy Kafka MirrorMaker to your OpenShift cluster:
   For MirrorMaker:

   ```
   oc apply -f examples/mirror-maker/kafka-mirror-maker.yaml
   ```

   For MirrorMaker 2:

   ```
   oc apply -f examples/mirror-maker/kafka-mirror-maker-2.yaml
   ```

2. Check the status of the deployment:

   ```
   oc get pods -n <my_cluster_operator_namespace>
   ```

   Output shows the deployment name and readiness

   ```
   NAME                                  READY  STATUS   RESTARTS
   my-mirror-maker-mirror-maker-<pod_id>   1/1    Running  1
   my-mm2-cluster-mirrormaker2-<pod_id>    1/1    Running  1
   ```

   my-mirror-maker is the name of the Kafka MirrorMaker cluster. **my-mm2-cluster** is the name of the Kafka MirrorMaker 2 cluster.

   A pod ID identifies each pod created.

   With the default deployment, you install a single MirrorMaker or MirrorMaker 2 pod.

   READY shows the number of replicas that are ready/expected. The deployment is successful when the **STATUS** shows as **Running**.

Additional resources

- Kafka MirrorMaker cluster configuration

6.6. DEPLOYING KAFKA BRIDGE

The Cluster Operator deploys one or more Kafka bridge replicas to send data between Kafka clusters and clients via HTTP API.

6.6.1. Deploying Kafka Bridge to your OpenShift cluster

This procedure shows how to deploy a Kafka Bridge cluster to your OpenShift cluster using the Cluster Operator.

The deployment uses a YAML file to provide the specification to create a **KafkaBridge** resource.

AMQ Streams provides example configuration files. In this procedure, we use the following example file:

- **examples/bridge/kafka-bridge.yaml**

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Deploy Kafka Bridge to your OpenShift cluster:

```
oc apply -f examples/bridge/kafka-bridge.yaml
```

2. Check the status of the deployment:

```
oc get pods -n <my_cluster_operator_namespace>
```

Output shows the deployment name and readiness

```
NAME                    READY  STATUS   RESTARTS
my-bridge-bridge-<pod_id>  1/1    Running  0
```

my-bridge is the name of the Kafka Bridge cluster.

A pod ID identifies each pod created.

With the default deployment, you install a single Kafka Bridge pod.

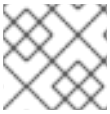
READY shows the number of replicas that are ready/expected. The deployment is successful when the **STATUS** shows as **Running**.

Additional resources

- Kafka Bridge cluster configuration
- Using the AMQ Streams Kafka Bridge

6.6.2. Exposing the Kafka Bridge service to your local machine

Use port forwarding to expose the AMQ Streams Kafka Bridge service to your local machine on http://localhost:8080.



NOTE

Port forwarding is only suitable for development and testing purposes.

Procedure

1. List the names of the pods in your OpenShift cluster:

```
oc get pods -o name

pod/kafka-consumer
# ...
pod/my-bridge-bridge-<pod_id>
```

2. Connect to the Kafka Bridge pod on port **8080**:

```
oc port-forward pod/my-bridge-bridge-<pod_id> 8080:8080 &
```

NOTE

If port 8080 on your local machine is already in use, use an alternative HTTP port, such as **8008**.

API requests are now forwarded from port 8080 on your local machine to port 8080 in the Kafka Bridge pod.

6.6.3. Accessing the Kafka Bridge outside of OpenShift

After deployment, the AMQ Streams Kafka Bridge can only be accessed by applications running in the same OpenShift cluster. These applications use the ***<kafka_bridge_name>*-bridge-service** service to access the API.

If you want to make the Kafka Bridge accessible to applications running outside of the OpenShift cluster, you can expose it manually by creating one of the following features:

- **LoadBalancer** or **NodePort** type services
- **Ingress** resources (Kubernetes only)
- OpenShift routes (OpenShift only)

If you decide to create Services, use the labels from the **selector** of the ***<kafka_bridge_name>*-bridge-service** service to configure the pods to which the service will route the traffic:

```
# ...
selector:
  strimzi.io/cluster: kafka-bridge-name 1
```

```
  strimzi.io/kind: KafkaBridge
#...
```

1 Name of the Kafka Bridge custom resource in your OpenShift cluster.

6.7. ALTERNATIVE STANDALONE DEPLOYMENT OPTIONS FOR AMQ STREAMS OPERATORS

You can perform a standalone deployment of the Topic Operator and User Operator. Consider a standalone deployment of these operators if you are using a Kafka cluster that is not managed by the Cluster Operator.

You deploy the operators to OpenShift. Kafka can be running outside of OpenShift. For example, you might be using a Kafka as a managed service. You adjust the deployment configuration for the standalone operator to match the address of your Kafka cluster.

6.7.1. Deploying the standalone Topic Operator

This procedure shows how to deploy the Topic Operator as a standalone component for topic management. You can use a standalone Topic Operator with a Kafka cluster that is not managed by the Cluster Operator.

A standalone deployment can operate with any Kafka cluster.

Standalone deployment files are provided with AMQ Streams. Use the **05-Deployment-strimzi-topic-operator.yaml** deployment file to deploy the Topic Operator. Add or set the environment variables needed to make a connection to a Kafka cluster.

The Topic Operator watches for **KafkaTopic** resources in a single namespace. You specify the namespace to watch, and the connection to the Kafka cluster, in the Topic Operator configuration. A single Topic Operator can watch a single namespace. One namespace should be watched by only one Topic Operator. If you want to use more than one Topic Operator, configure each of them to watch different namespaces. In this way, you can use Topic Operators with multiple Kafka clusters.

Prerequisites

- You are running a Kafka cluster for the Topic Operator to connect to.
  As long as the standalone Topic Operator is correctly configured for connection, the Kafka cluster can be running on a bare-metal environment, a virtual machine, or as a managed cloud application service.

Procedure

1. Edit the **env** properties in the **install/topic-operator/05-Deployment-strimzi-topic-operator.yaml** standalone deployment file.

   Example standalone Topic Operator deployment configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-topic-operator
  labels:
```

```
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-topic-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE 1
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS 2
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_RESOURCE_LABELS 3
              value: "strimzi.io/cluster=my-cluster"
            - name: STRIMZI_ZOOKEEPER_CONNECT 4
              value: my-cluster-zookeeper-client:2181
            - name: STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS 5
              value: "18000"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS 6
              value: "120000"
            - name: STRIMZI_TOPIC_METADATA_MAX_ATTEMPTS 7
              value: "6"
            - name: STRIMZI_LOG_LEVEL 8
              value: INFO
            - name: STRIMZI_TLS_ENABLED 9
              value: "false"
            - name: STRIMZI_JAVA_OPTS 10
              value: "-Xmx=512M -Xms=256M"
            - name: STRIMZI_JAVA_SYSTEM_PROPERTIES 11
              value: "-Djavax.net.debug=verbose -DpropertyName=value"
            - name: STRIMZI_PUBLIC_CA 12
              value: "false"
            - name: STRIMZI_TLS_AUTH_ENABLED 13
              value: "false"
            - name: STRIMZI_SASL_ENABLED 14
              value: "false"
            - name: STRIMZI_SASL_USERNAME 15
              value: "admin"
            - name: STRIMZI_SASL_PASSWORD 16
              value: "password"
            - name: STRIMZI_SASL_MECHANISM 17
              value: "scram-sha-512"
            - name: STRIMZI_SECURITY_PROTOCOL 18
              value: "SSL"
```

1 The OpenShift namespace for the Topic Operator to watch for **KafkaTopic** resources. Specify the namespace of the Kafka cluster.

2 The host and port pair of the bootstrap broker address to discover and connect to all brokers in the Kafka cluster. Use a comma-separated list to specify two or three broker addresses in case a server is down.

3 The label to identify the **KafkaTopic** resources managed by the Topic Operator. This does not have to be the name of the Kafka cluster. It can be the label assigned to the **KafkaTopic** resource. If you deploy more than one Topic Operator, the labels must be unique for each. That is, the operators cannot manage the same resources.

4 The host and port pair of the address to connect to the ZooKeeper cluster. This must be the same ZooKeeper cluster that your Kafka cluster is using.

5 The ZooKeeper session timeout, in milliseconds. The default is **18000** (18 seconds).

6 The interval between periodic reconciliations, in milliseconds. The default is **120000** (2 minutes).

7 The number of attempts at getting topic metadata from Kafka. The time between each attempt is defined as an exponential backoff. Consider increasing this value when topic creation takes more time due to the number of partitions or replicas. The default is **6** attempts.

8 The level for printing logging messages. You can set the level to **ERROR**, **WARNING**, **INFO**, **DEBUG**, or **TRACE**.

9 Enables TLS support for encrypted communication with the Kafka brokers.

10 (Optional) The Java options used by the JVM running the Topic Operator.

11 (Optional) The debugging (**-D**) options set for the Topic Operator.

12 (Optional) Skips the generation of trust store certificates if TLS is enabled through **STRIMZI_TLS_ENABLED**. If this environment variable is enabled, the brokers must use a public trusted certificate authority for their TLS certificates. The default is **false**.

13 (Optional) Generates key store certificates for mTLS authentication. Setting this to **false** disables client authentication with mTLS to the Kafka brokers. The default is **true**.

14 (Optional) Enables SASL support for client authentication when connecting to Kafka brokers. The default is **false**.

15 (Optional) The SASL username for client authentication. Mandatory only if SASL is enabled through **STRIMZI_SASL_ENABLED**.

16 (Optional) The SASL password for client authentication. Mandatory only if SASL is enabled through **STRIMZI_SASL_ENABLED**.

17 (Optional) The SASL mechanism for client authentication. Mandatory only if SASL is enabled through **STRIMZI_SASL_ENABLED**. You can set the value to **plain**, **scram-sha-256**, or **scram-sha-512**.

18 (Optional) The security protocol used for communication with Kafka brokers. The default value is "PLAINTEXT". You can set the value to **PLAINTEXT**, **SSL**, **SASL_PLAINTEXT**, or **SASL_SSL**.

2. If you want to connect to Kafka brokers that are using certificates from a public certificate authority, set **STRIMZI_PUBLIC_CA** to **true**. Set this property to **true**, for example, if you are using Amazon AWS MSK service.

3. If you enabled mTLS with the **STRIMZI_TLS_ENABLED** environment variable, specify the keystore and truststore used to authenticate connection to the Kafka cluster.

   Example mTLS configuration

   ```
   # ....
   env:
     - name: STRIMZI_TRUSTSTORE_LOCATION (1)
       value: "/path/to/truststore.p12"
     - name: STRIMZI_TRUSTSTORE_PASSWORD (2)
       value: "TRUSTSTORE-PASSWORD"
     - name: STRIMZI_KEYSTORE_LOCATION (3)
       value: "/path/to/keystore.p12"
     - name: STRIMZI_KEYSTORE_PASSWORD (4)
       value: "KEYSTORE-PASSWORD"
   # ...
   ```

   (1) The truststore contains the public keys of the Certificate Authorities used to sign the Kafka and ZooKeeper server certificates.

   (2) The password for accessing the truststore.

   (3) The keystore contains the private key for mTLS authentication.

   (4) The password for accessing the keystore.

4. Deploy the Topic Operator.

   ```
   oc create -f install/topic-operator
   ```

5. Check the status of the deployment:

   ```
   oc get deployments
   ```

   Output shows the deployment name and readiness

   ```
   NAME                      READY  UP-TO-DATE  AVAILABLE
   strimzi-topic-operator  1/1    1           1
   ```

   READY shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

6.7.2. Deploying the standalone User Operator

This procedure shows how to deploy the User Operator as a standalone component for user management. You can use a standalone User Operator with a Kafka cluster that is not managed by the Cluster Operator.

A standalone deployment can operate with any Kafka cluster.

Standalone deployment files are provided with AMQ Streams. Use the **05-Deployment-strimzi-user-operator.yaml** deployment file to deploy the User Operator. Add or set the environment variables needed to make a connection to a Kafka cluster.

The User Operator watches for **KafkaUser** resources in a single namespace. You specify the namespace to watch, and the connection to the Kafka cluster, in the User Operator configuration. A single User Operator can watch a single namespace. One namespace should be watched by only one User Operator. If you want to use more than one User Operator, configure each of them to watch different namespaces. In this way, you can use the User Operator with multiple Kafka clusters.

Prerequisites

- You are running a Kafka cluster for the User Operator to connect to.
  As long as the standalone User Operator is correctly configured for connection, the Kafka cluster can be running on a bare-metal environment, a virtual machine, or as a managed cloud application service.

Procedure

1. Edit the following **env** properties in the **install/user-operator/05-Deployment-strimzi-user-operator.yaml** standalone deployment file.

   #### Example standalone User Operator deployment configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-user-operator
  labels:
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-user-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE 1
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS 2
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_CA_CERT_NAME 3
              value: my-cluster-clients-ca-cert
            - name: STRIMZI_CA_KEY_NAME 4
              value: my-cluster-clients-ca
            - name: STRIMZI_LABELS 5
              value: "strimzi.io/cluster=my-cluster"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS 6
              value: "120000"
```

```
        - name: STRIMZI_WORK_QUEUE_SIZE 7
          value: 10000
        - name: STRIMZI_CONTROLLER_THREAD_POOL_SIZE 8
          value: 10
        - name: STRIMZI_USER_OPERATIONS_THREAD_POOL_SIZE 9
          value: 4
        - name: STRIMZI_LOG_LEVEL 10
          value: INFO
        - name: STRIMZI_GC_LOG_ENABLED 11
          value: "true"
        - name: STRIMZI_CA_VALIDITY 12
          value: "365"
        - name: STRIMZI_CA_RENEWAL 13
          value: "30"
        - name: STRIMZI_JAVA_OPTS 14
          value: "-Xmx=512M -Xms=256M"
        - name: STRIMZI_JAVA_SYSTEM_PROPERTIES 15
          value: "-Djavax.net.debug=verbose -DpropertyName=value"
        - name: STRIMZI_SECRET_PREFIX 16
          value: "kafka-"
        - name: STRIMZI_ACLS_ADMIN_API_SUPPORTED 17
          value: "true"
        - name: STRIMZI_MAINTENANCE_TIME_WINDOWS 18
          value: '* * 8-10 * * ?;* * 14-15 * * ?'
        - name: STRIMZI_KAFKA_ADMIN_CLIENT_CONFIGURATION 19
          value: |
            default.api.timeout.ms=120000
            request.timeout.ms=60000
        - name: STRIMZI_KRAFT_ENABLED 20
          value: "false"
```

1. The OpenShift namespace for the User Operator to watch for **KafkaUser** resources. Only one namespace can be specified.

2. The host and port pair of the bootstrap broker address to discover and connect to all brokers in the Kafka cluster. Use a comma-separated list to specify two or three broker addresses in case a server is down.

3. The OpenShift **Secret** that contains the public key (**ca.crt**) value of the CA (certificate authority) that signs new user certificates for mTLS authentication.

4. The OpenShift **Secret** that contains the private key (**ca.key**) value of the CA that signs new user certificates for mTLS authentication.

5. The label to identify the **KafkaUser** resources managed by the User Operator. This does not have to be the name of the Kafka cluster. It can be the label assigned to the **KafkaUser** resource. If you deploy more than one User Operator, the labels must be unique for each. That is, the operators cannot manage the same resources.

6. The interval between periodic reconciliations, in milliseconds. The default is **120000** (2 minutes).

7.

The size of the controller event queue. The size of the queue should be at least as big as the maximal amount of users you expect the User Operator to operate. The default is

8 The size of the worker pool for reconciling the users. Bigger pool might require more resources, but it will also handle more **KafkaUser** resources The default is **50**.

9 The size of the worker pool for Kafka Admin API and OpenShift operations. Bigger pool might require more resources, but it will also handle more **KafkaUser** resources The default is **4**.

10 The level for printing logging messages. You can set the level to **ERROR**, **WARNING**, **INFO**, **DEBUG**, or **TRACE**.

11 Enables garbage collection (GC) logging. The default is **true**.

12 The validity period for the CA. The default is **365** days.

13 The renewal period for the CA. The renewal period is measured backwards from the expiry date of the current certificate. The default is **30** days to initiate certificate renewal before the old certificates expire.

14 (Optional) The Java options used by the JVM running the User Operator

15 (Optional) The debugging (**-D**) options set for the User Operator

16 (Optional) Prefix for the names of OpenShift secrets created by the User Operator.

17 (Optional) Indicates whether the Kafka cluster supports management of authorization ACL rules using the Kafka Admin API. When set to **false**, the User Operator will reject all resources with **simple** authorization ACL rules. This helps to avoid unnecessary exceptions in the Kafka cluster logs. The default is **true**.

18 (Optional) Semi-colon separated list of Cron Expressions defining the maintenance time windows during which the expiring user certificates will be renewed.

19 (Optional) Configuration options for configuring the Kafka Admin client used by the User Operator in the properties format.

20 (Optional) Indicates whether the Kafka cluster the User Operator is connecting to is using KRaft instead of ZooKeeper. Set this variable to **true** if the Kafka cluster uses KRaft. The default is **false**. Note that some features are not available when running against KRaft clusters. For example, management of SCRAM-SHA-512 users is disabled because Apache Kafka currently does not support it.

2. If you are using mTLS to connect to the Kafka cluster, specify the secrets used to authenticate connection. Otherwise, go to the next step.

Example mTLS configuration

```
# ....
env:
  - name: STRIMZI_CLUSTER_CA_CERT_SECRET_NAME 1
    value: my-cluster-cluster-ca-cert
  - name: STRIMZI_EO_KEY_SECRET_NAME 2
    value: my-cluster-entity-operator-certs
# ..."
```

```
-
```

1. The OpenShift **Secret** that contains the public key (**ca.crt**) value of the CA that signs Kafka broker certificates.
2. The OpenShift **Secret** that contains the certificate public key (**entity-operator.crt**) and private key (**entity-operator.key**) that is used for mTLS authentication against the Kafka cluster.

3. Deploy the User Operator.

```
oc create -f install/user-operator
```

4. Check the status of the deployment:

```
oc get deployments
```

Output shows the deployment name and readiness

```
NAME                      READY  UP-TO-DATE  AVAILABLE
strimzi-user-operator  1/1    1           1
```

READY shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

CHAPTER 7. SETTING UP CLIENT ACCESS TO A KAFKA CLUSTER

After you have deployed AMQ Streams, you can set up client access to your Kafka cluster. To verify the deployment, you can deploy example producer and consumer clients. Otherwise, create listeners that provide client access within or outside the OpenShift cluster.

7.1. DEPLOYING EXAMPLE CLIENTS

Deploy example producer and consumer clients to send and receive messages. You can use these clients to verify a deployment of AMQ Streams.

Prerequisites

- The Kafka cluster is available for the clients.

Procedure

1. Deploy a Kafka producer.

```
oc run kafka-producer -ti --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 --rm=true --restart=Never -- bin/kafka-console-producer.sh --bootstrap-server cluster-name-kafka-bootstrap:9092 --topic my-topic
```

2. Type a message into the console where the producer is running.
3. Press *Enter* to send the message.
4. Deploy a Kafka consumer.

```
oc run kafka-consumer -ti --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 --rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server cluster-name-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

5. Confirm that you see the incoming messages in the consumer console.

7.2. CONFIGURING LISTENERS TO CONNECT TO KAFKA BROKERS

Use listeners for client connection to Kafka brokers. AMQ Streams provides a generic **GenericKafkaListener** schema with properties to configure listeners through the **Kafka** resource. The **GenericKafkaListener** provides a flexible approach to listener configuration. You can specify properties to configure *internal* listeners for connecting within the OpenShift cluster or *external* listeners for connecting outside the OpenShift cluster.

Specify a connection **type** to expose Kafka in the listener configuration. The type chosen depends on your requirements, and your environment and infrastructure. The following listener types are supported:

Internal listeners

- **internal** to connect within the same OpenShift cluster
- **cluster-ip** to expose Kafka using per-broker **ClusterIP** services

External listeners

- **nodeport** to use ports on OpenShift nodes
- **loadbalancer** to use loadbalancer services
- **ingress** to use Kubernetes **Ingress** and the Ingress NGINX Controller for Kubernetes (Kubernetes only)
- **route** to use OpenShift **Route** and the default HAProxy router (OpenShift only)

IMPORTANT

Do not use **ingress** on OpenShift, use the **route** type instead. The Ingress NGINX Controller is only intended for use on Kubernetes. The **route** type is only supported on OpenShift.

An **internal** type listener configuration uses a headless service and the DNS names given to the broker pods. You might want to join your OpenShift network to an outside network. In which case, you can configure an **internal** type listener (using the **useServiceDnsDomain** property) so that the OpenShift service DNS domain (typically **.cluster.local**) is not used. You can also configure a **cluster-ip** type of listener that exposes a Kafka cluster based on per-broker **ClusterIP** services. This is a useful option when you can't route through the headless service or you wish to incorporate a custom access mechanism. For example, you might use this listener when building your own type of external listener for a specific Ingress controller or the OpenShift Gateway API.

External listeners handle access to a Kafka cluster from networks that require different authentication mechanisms. You can configure external listeners for client access outside an OpenShift environment using a specified connection mechanism, such as a loadbalancer or route. For example, loadbalancers might not be suitable for certain infrastructure, such as bare metal, where node ports provide a better option.

Each listener is defined as an array in the **Kafka** resource.

Example listener configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
        configuration:
          useServiceDnsDomain: true
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
```

```
      type: tls
  - name: external
    port: 9094
    type: route
    tls: true
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-certificate.crt
        key: my-key.key
# ...
```

You can configure as many listeners as required, as long as their names and ports are unique. You can also configure listeners for secure connection using authentication.

If you want to know more about the pros and cons of each connection type, refer to Accessing Apache Kafka in Strimzi.

NOTE

If you scale your Kafka cluster while using external listeners, it might trigger a rolling update of all Kafka brokers. This depends on the configuration.

Additional resources

- **GenericKafkaListener** schema reference

7.3. SETTING UP CLIENT ACCESS TO A KAFKA CLUSTER USING LISTENERS

Using the address of the Kafka cluster, you can provide access to a client in the same OpenShift cluster; or provide external access to a client on a different OpenShift namespace or outside OpenShift entirely. This procedure shows how to configure client access to a Kafka cluster from outside OpenShift or from another OpenShift cluster.

A Kafka listener provides access to the Kafka cluster. Client access is secured using the following configuration:

1. An external listener is configured for the Kafka cluster, with TLS encryption and mTLS authentication, and Kafka **simple** authorization enabled.

2. A **KafkaUser** is created for the client, with mTLS authentication, and Access Control Lists (ACLs) defined for **simple** authorization.

You can configure your listener to use mutual **tls**, **scram-sha-512**, or **oauth** authentication. mTLS always uses encryption, but encryption is also recommended when using SCRAM-SHA-512 and OAuth 2.0 authentication.

You can configure **simple**, **oauth**, **opa**, or **custom** authorization for Kafka brokers. When enabled, authorization is applied to all enabled listeners.

When you configure the **KafkaUser** authentication and authorization mechanisms, ensure they match the equivalent Kafka configuration:

- **KafkaUser.spec.authentication** matches **Kafka.spec.kafka.listeners[*].authentication**

- **KafkaUser.spec.authorization** matches **Kafka.spec.kafka.authorization**

You should have at least one listener supporting the authentication you want to use for the **KafkaUser**.

NOTE

Authentication between Kafka users and Kafka brokers depends on the authentication settings for each. For example, it is not possible to authenticate a user with mTLS if it is not also enabled in the Kafka configuration.

AMQ Streams operators automate the configuration process and create the certificates required for authentication:

- The Cluster Operator creates the listeners and sets up the cluster and client certificate authority (CA) certificates to enable authentication with the Kafka cluster.
- The User Operator creates the user representing the client and the security credentials used for client authentication, based on the chosen authentication type.

You add the certificates to your client configuration.

In this procedure, the CA certificates generated by the Cluster Operator are used, but you can replace them by installing your own certificates. You can also configure your listener to use a Kafka listener certificate managed by an external CA (certificate authority).

Certificates are available in PEM (.crt) and PKCS #12 (.p12) formats. This procedure uses PEM certificates. Use PEM certificates with clients that use certificates in X.509 format.

NOTE

For internal clients in the same OpenShift cluster and namespace, you can mount the cluster CA certificate in the pod specification. For more information, see Configuring internal clients to trust the cluster CA.

Prerequisites

- The Kafka cluster is available for connection by a client running outside the OpenShift cluster
- The Cluster Operator and User Operator are running in the cluster

Procedure

1. Configure the Kafka cluster with a Kafka listener.
   - Define the authentication required to access the Kafka broker through the listener.
   - Enable authorization on the Kafka broker.

   Example listener configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
```

```
spec:
  kafka:
    # ...
    listeners: 1
    - name: external 2
      port: 9094 3
      type: <listener_type> 4
      tls: true 5
      authentication:
        type: tls 6
      configuration: 7
        #...
    authorization: 8
      type: simple
      superUsers:
        - super-user-name 9
  # ...
```

1. Configuration options for enabling external listeners are described in the Generic Kafka listener schema reference.
2. Name to identify the listener. Must be unique within the Kafka cluster.
3. Port number used by the listener inside Kafka. The port number has to be unique within a given Kafka cluster. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX. Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.
4. External listener type specified as **route** (OpenShift only), **loadbalancer**, **nodeport** or **ingress** (Kubernetes only). An internal listener is specified as **internal** or **cluster-ip**.
5. Required. TLS encryption on the listener. For **route** and **ingress** type listeners it must be set to **true**. For mTLS authentication, also use the **authentication** property.
6. Client authentication mechanism on the listener. For server and client authentication using mTLS, you specify **tls: true** and **authentication.type: tls**.
7. (Optional) Depending on the requirements of the listener type, you can specify additional listener configuration.
8. Authorization specified as **simple**, which uses the **AclAuthorizer** Kafka plugin.
9. (Optional) Super users can access all brokers regardless of any access restrictions defined in ACLs.

WARNING

An OpenShift Route address comprises the name of the Kafka cluster, the name of the listener, and the name of the namespace it is created in. For example, **my-cluster-kafka-listener1-bootstrap-myproject** (*CLUSTER-NAME*-kafka-*LISTENER-NAME*-bootstrap-*NAMESPACE*). If you are using a **route** listener type, be careful that the whole length of the address does not exceed a maximum limit of 63 characters.

2. Create or update the **Kafka** resource.

```
oc apply -f <kafka_configuration_file>
```

The Kafka cluster is configured with a Kafka broker listener using mTLS authentication.

A service is created for each Kafka broker pod.

A service is created to serve as the *bootstrap address* for connection to the Kafka cluster.

A service is also created as the *external bootstrap address* for external connection to the Kafka cluster using **nodeport** listeners.

The cluster CA certificate to verify the identity of the kafka brokers is also created in the secret ***<cluster_name>*-cluster-ca-cert**.

NOTE

If you scale your Kafka cluster while using external listeners, it might trigger a rolling update of all Kafka brokers. This depends on the configuration.

3. Retrieve the bootstrap address you can use to access the Kafka cluster from the status of the **Kafka** resource.

```
oc get kafka <kafka_cluster_name> -o=jsonpath='{.status.listeners[?
(@.name=="<listener_name>")].bootstrapServers}{"\n"}'
```

For example:

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="external")].bootstrapServers}{"\n"}'
```

Use the bootstrap address in your Kafka client to connect to the Kafka cluster.

4. Create or modify a user representing the client that requires access to the Kafka cluster.
   - Specify the same authentication type as the **Kafka** listener.
   - Specify the authorization ACLs for **simple** authorization.

Example user configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster (1)
spec:
  authentication:
    type: tls (2)
  authorization:
    type: simple
    acls: (3)
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Describe
          - Read
      - resource:
          type: group
          name: my-group
          patternType: literal
        operations:
          - Read
```

(1) The label must match the label of the Kafka cluster.

(2) Authentication specified as mutual **tls**.

(3) Simple authorization requires an accompanying list of ACL rules to apply to the user. The rules define the operations allowed on Kafka resources based on the *username* (**my-user**).

5. Create or modify the **KafkaUser** resource.

```
oc apply -f USER-CONFIG-FILE
```

The user is created, as well as a secret with the same name as the **KafkaUser** resource. The secret contains a public and private key for mTLS authentication.

Example secret

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: <public_key> # Public key of the clients CA
  user.crt: <user_certificate> # Public key of the user
```

```
user.key: <user_private_key> # Private key of the user
user.p12: <store> # PKCS #12 store for user certificates and keys
user.password: <password_for_store> # Protects the PKCS #12 store
```

6. Extract the cluster CA certificate from the ***<cluster_name>*-cluster-ca-cert** secret of the Kafka cluster.

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

7. Extract the user CA certificate from the ***<user_name>*** secret.

```
oc get secret <user_name> -o jsonpath='{.data.user\.crt}' | base64 -d > user.crt
```

8. Extract the private key of the user from the ***<user_name>*** secret.

```
oc get secret <user_name> -o jsonpath='{.data.user\.key}' | base64 -d > user.key
```

9. Configure your client with the bootstrap address hostname and port for connecting to the Kafka cluster:

```
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "<hostname>:<port>");
```

10. Configure your client with the truststore credentials to verify the identity of the Kafka cluster.
Specify the public cluster CA certificate.

Example truststore configuration

```
props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
props.put(SslConfigs.SSL_TRUSTSTORE_TYPE_CONFIG, "PEM");
props.put(SslConfigs.SSL_TRUSTSTORE_CERTIFICATES_CONFIG,
"<ca.crt_file_content>");
```

SSL is the specified security protocol for mTLS authentication. Specify **SASL_SSL** for SCRAM-SHA-512 authentication over TLS. PEM is the file format of the truststore.

11. Configure your client with the keystore credentials to verify the user when connecting to the Kafka cluster.
Specify the public certificate and private key.

Example keystore configuration

```
props.put(CommonClientConfigs.SECURITY_PROTOCOL_CONFIG, "SSL");
props.put(SslConfigs.SSL_KEYSTORE_TYPE_CONFIG, "PEM");
props.put(SslConfigs.SSL_KEYSTORE_CERTIFICATE_CHAIN_CONFIG,
"<user.crt_file_content>");
props.put(SslConfigs.SSL_KEYSTORE_KEY_CONFIG, "<user.key_file_content>");
```

Add the keystore certificate and the private key directly to the configuration. Add as a single-line format. Between the **BEGIN CERTIFICATE** and **END CERTIFICATE** delimiters, start with a newline character (**\n**). End each line from the original certificate with **\n** too.

Example keystore configuration

```
props.put(SslConfigs.SSL_KEYSTORE_CERTIFICATE_CHAIN_CONFIG, "-----BEGIN CERTIFICATE-----
\n<user_certificate_content_line_1>\n<user_certificate_content_line_n>\n-----END CERTIFICATE---");
props.put(SslConfigs.SSL_KEYSTORE_KEY_CONFIG, "----BEGIN PRIVATE KEY-----
\n<user_key_content_line_1>\n<user_key_content_line_n>\n-----END PRIVATE KEY-----");
```

Additional resources

- Section 8.1.1, "Listener authentication"
- Section 8.1.2, "Kafka authorization"
- If you are using an authorization server, you can use token-based authentication and authorization:
  - Section 8.4, "Using OAuth 2.0 token-based authentication"
  - Section 8.5, "Using OAuth 2.0 token-based authorization"

7.4. ACCESSING KAFKA USING NODE PORTS

Use node ports to access an AMQ Streams Kafka cluster from an external client outside the OpenShift cluster.

To connect to a broker, you specify a hostname and port number for the Kafka bootstrap address, as well as the certificate used for TLS encryption.

The procedure shows basic **nodeport** listener configuration. You can use listener properties to enable TLS encryption (**tls**) and specify a client authentication mechanism (**authentication**). Add additional configuration using **configuration** properties. For example, you can use the following configuration properties with **nodeport** listeners:

preferredNodePortAddressType
: Specifies the first address type that's checked as the node address.

externalTrafficPolicy
: Specifies whether the service routes external traffic to node-local or cluster-wide endpoints.

nodePort
: Overrides the assigned node port numbers for the bootstrap and broker services.

For more information on listener configuration, see the **GenericKafkaListener** schema reference.

Prerequisites

- A running Cluster Operator

In this procedure, the Kafka cluster name is **my-cluster**. The name of the listener is **external**.

Procedure

1. Configure a **Kafka** resource with an external listener set to the **nodeport** type.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: nodeport
        tls: true
        authentication:
          type: tls
        # ...
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

```
oc apply -f <kafka_configuration_file>
```

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret **my-cluster-cluster-ca-cert**.

NodePort type services are created for each Kafka broker, as well as an external bootstrap service.

Node port services created for the bootstrap and brokers

```
NAME                                TYPE      CLUSTER-IP      PORT(S)
my-cluster-kafka-external-0         NodePort  172.30.55.13    9094:31789/TCP
my-cluster-kafka-external-1         NodePort  172.30.250.248  9094:30028/TCP
my-cluster-kafka-external-2         NodePort  172.30.115.81   9094:32650/TCP
my-cluster-kafka-external-bootstrap  NodePort  172.30.30.23    9094:32650/TCP
```

The bootstrap address used for client connection is propagated to the **status** of the **Kafka** resource.

Example status for the bootstrap address

```
status:
  clusterId: Y_RJQDGKRXmNF7fEcWldJQ
  conditions:
    - lastTransitionTime: '2023-01-31T14:59:37.113630Z'
      status: 'True'
      type: Ready
  listeners:
    # ...
    - addresses:
```

```
      - host: ip-10-0-224-199.us-west-2.compute.internal
        port: 32650
    bootstrapServers: 'ip-10-0-224-199.us-west-2.compute.internal:32650'
    certificates:
      - |
        -----BEGIN CERTIFICATE-----

        -----END CERTIFICATE-----
    name: external
    type: external
  observedGeneration: 2
 # ...
```

3. Retrieve the bootstrap address you can use to access the Kafka cluster from the status of the **Kafka** resource.

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="external")].bootstrapServers}{"\n"}'

ip-10-0-224-199.us-west-2.compute.internal:32650
```

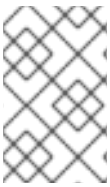
4. Extract the cluster CA certificate.

```
oc get secret my-cluster-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

5. Configure your client to connect to the brokers.

   a. Specify the bootstrap host and port in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, **ip-10-0-224-199.us-west-2.compute.internal:32650**.

   b. Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.
   If you enabled a client authentication mechanism, you will also need to configure it in your client.



NOTE

If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

7.5. ACCESSING KAFKA USING LOADBALANCERS

Use loadbalancers to access an AMQ Streams Kafka cluster from an external client outside the OpenShift cluster.

To connect to a broker, you specify a hostname and port number for the Kafka bootstrap address, as well as the certificate used for TLS encryption.

The procedure shows basic **loadbalancer** listener configuration. You can use listener properties to enable TLS encryption (**tls**) and specify a client authentication mechanism (**authentication**). Add additional configuration using **configuration** properties. For example, you can use the following configuration properties with **loadbalancer** listeners:

loadBalancerSourceRanges

Restricts traffic to a specified list of CIDR (Classless Inter-Domain Routing) ranges.

externalTrafficPolicy

Specifies whether the service routes external traffic to node-local or cluster-wide endpoints.

loadBalancerIP

Requests a specific IP address when creating a loadbalancer.

For more information on listener configuration, see the **GenericKafkaListener** schema reference.

Prerequisites

- A running Cluster Operator

In this procedure, the Kafka cluster name is **my-cluster**. The name of the listener is **external**.

Procedure

1. Configure a **Kafka** resource with an external listener set to the **loadbalancer** type. For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9095
        type: loadbalancer
        tls: true
        authentication:
          type: tls
        # ...
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

```
oc apply -f <kafka_configuration_file>
```

A cluster CA certificate to verify the identity of the kafka brokers is also created in the secret **my-cluster-cluster-ca-cert**.

loadbalancer type services and loadbalancers are created for each Kafka broker, as well as an external bootstrap service.

Loadbalancer services and loadbalancers created for the bootstraps and brokers

```
NAME                                TYPE           CLUSTER-IP      PORT(S)
my-cluster-kafka-external-0         LoadBalancer   172.30.204.234  9095:30011/TCP
my-cluster-kafka-external-1         LoadBalancer   172.30.164.89   9095:32544/TCP
my-cluster-kafka-external-2         LoadBalancer   172.30.73.151   9095:32504/TCP
my-cluster-kafka-external-bootstrap  LoadBalancer   172.30.30.228   9095:30371/TCP

NAME                                EXTERNAL-IP (loadbalancer)
my-cluster-kafka-external-0         a8a519e464b924000b6c0f0a05e19f0d-1132975133.us-
west-2.elb.amazonaws.com
my-cluster-kafka-external-1         ab6adc22b556343afb0db5ea05d07347-611832211.us-
west-2.elb.amazonaws.com
my-cluster-kafka-external-2         a9173e8ccb1914778aeb17eca98713c0-777597560.us-
west-2.elb.amazonaws.com
my-cluster-kafka-external-bootstrap  a8d4a6fb363bf447fb6e475fc3040176-36312313.us-
west-2.elb.amazonaws.com
```

The bootstrap address used for client connection is propagated to the **status** of the **Kafka** resource.

Example status for the bootstrap address

```
status:
  clusterId: Y_RJQDGKRXmNF7fEcWldJQ
  conditions:
    - lastTransitionTime: '2023-01-31T14:59:37.113630Z'
      status: 'True'
      type: Ready
  listeners:
    # ...
    - addresses:
        - host: >-
            a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com
          port: 9095
      bootstrapServers: >-
        a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9095
      certificates:
        - |
          -----BEGIN CERTIFICATE-----

          -----END CERTIFICATE-----
      name: external
      type: external
  observedGeneration: 2
 # ...
```

The DNS addresses used for client connection are propagated to the **status** of each loadbalancer service.

Example status for the bootstrap loadbalancer

```
status:
  loadBalancer:
    ingress:
```

```
    - hostname: >-
        a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com
# ...
```

3. Retrieve the bootstrap address you can use to access the Kafka cluster from the status of the **Kafka** resource.

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="external")].bootstrapServers}{"\n"}'

a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9095
```

4. Extract the cluster CA certificate.

```
oc get secret my-cluster-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

5. Configure your client to connect to the brokers.

   a. Specify the bootstrap host and port in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, **a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9095**.

   b. Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.
   If you enabled a client authentication mechanism, you will also need to configure it in your client.

NOTE

If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

7.6. ACCESSING KAFKA USING OPENSHIFT ROUTES

Use OpenShift routes to access an AMQ Streams Kafka cluster from clients outside the OpenShift cluster.

To be able to use routes, add configuration for a **route** type listener in the **Kafka** custom resource. When applied, the configuration creates a dedicated route and service for an external bootstrap and each broker in the cluster. Clients connect to the bootstrap route, which routes them through the bootstrap service to connect to a broker. Per-broker connections are then established using DNS names, which route traffic from the client to the broker through the broker-specific routes and services.

To connect to a broker, you specify a hostname for the route bootstrap address, as well as the certificate used for TLS encryption. For access using routes, the port is always 443.

WARNING

An OpenShift route address comprises the name of the Kafka cluster, the name of the listener, and the name of the project it is created in. For example, **my-cluster-kafka-external-bootstrap-myproject** (<cluster_name>-kafka-<listener_name>-bootstrap-<namespace>). Be careful that the whole length of the address does not exceed a maximum limit of 63 characters.

The procedure shows basic listener configuration. TLS encryption (**tls**) must be enabled. You can also specify a client authentication mechanism (**authentication**). Add additional configuration using **configuration** properties. For example, you can use the **host** configuration property with **route** listeners to specify the hostnames used by the bootstrap and per-broker services.

For more information on listener configuration, see the **GenericKafkaListener** schema reference.

TLS passthrough

TLS passthrough is enabled for routes created by AMQ Streams. Kafka uses a binary protocol over TCP, but routes are designed to work with a HTTP protocol. To be able to route TCP traffic through routes, AMQ Streams uses TLS passthrough with Server Name Indication (SNI).

SNI helps with identifying and passing connection to Kafka brokers. In passthrough mode, TLS encryption is always used. Because the connection passes to the brokers, the listeners use TLS certificates signed by the internal cluster CA and not the ingress certificates. To configure listeners to use your own listener certificates, use the **brokerCertChainAndKey** property.

Prerequisites

- A running Cluster Operator

In this procedure, the Kafka cluster name is **my-cluster**. The name of the listener is **external**.

Procedure

1. Configure a **Kafka** resource with an external listener set to the **route** type.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: route
```

```
      tls: true 1
      authentication:
        type: tls
      # ...
  # ...
zookeeper:
  # ...
```

1 For **route** type listeners, TLS encryption must be enabled (**true**).

2. Create or update the resource.

```
oc apply -f <kafka_configuration_file>
```

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret **my-cluster-cluster-ca-cert**.

ClusterIP type services are created for each Kafka broker, as well as an external bootstrap service.

A **route** is also created for each service, with a DNS address (host/port) to expose them using the default OpenShift HAProxy router.

The routes are preconfigured with TLS passthrough.

Routes created for the bootstraps and brokers

```
NAME                            HOST/PORT                                                  SERVICES
PORT  TERMINATION
my-cluster-kafka-external-0     my-cluster-kafka-external-0-my-project.router.com
my-cluster-kafka-external-0     9094  passthrough
my-cluster-kafka-external-1     my-cluster-kafka-external-1-my-project.router.com
my-cluster-kafka-external-1     9094  passthrough
my-cluster-kafka-external-2     my-cluster-kafka-external-2-my-project.router.com
my-cluster-kafka-external-2     9094  passthrough
my-cluster-kafka-external-bootstrap  my-cluster-kafka-external-bootstrap-my-
project.router.com  my-cluster-kafka-external-bootstrap  9094  passthrough
```

The DNS addresses used for client connection are propagated to the **status** of each route.

Example status for the bootstrap route

```
status:
  ingress:
    - host: >-
        my-cluster-kafka-external-bootstrap-my-project.router.com
  # ...
```

3. Use a target broker to check the client-server TLS connection on port 443 using the OpenSSL **s_client**.

```
openssl s_client -connect my-cluster-kafka-external-0-my-project.router.com:443 -
servername my-cluster-kafka-external-0-my-project.router.com -showcerts
```

The server name is the SNI for passing the connection to the broker.

If the connection is successful, the certificates for the broker are returned.

Certificates for the broker

```
Certificate chain
 0 s:O = io.strimzi, CN = my-cluster-kafka
   i:O = io.strimzi, CN = cluster-ca v0
```

4. Retrieve the address of the bootstrap service from the status of the **Kafka** resource.

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="external")].bootstrapServers}{"\n"}'

my-cluster-kafka-external-bootstrap-my-project.router.com:443
```

The address comprises the cluster name, the listener name, the project name and the domain of the router (**router.com** in this example).

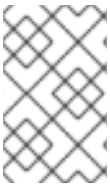
5. Extract the cluster CA certificate.

```
oc get secret my-cluster-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

6. Configure your client to connect to the brokers.

   a. Specify the address for the bootstrap service and port 443 in your Kafka client as the bootstrap address to connect to the Kafka cluster.

   b. Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.
   If you enabled a client authentication mechanism, you will also need to configure it in your client.



NOTE

If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

CHAPTER 8. MANAGING SECURE ACCESS TO KAFKA

You can secure your Kafka cluster by managing the access each client has to the Kafka brokers.

A secure connection between Kafka brokers and clients can encompass:

- Encryption for data exchange
- Authentication to prove identity
- Authorization to allow or decline actions executed by users

This chapter explains how to set up secure connections between Kafka brokers and clients, with sections describing:

- Security options for Kafka clusters and clients
- How to secure Kafka brokers
- How to use an authorization server for OAuth 2.0 token-based authentication and authorization

8.1. SECURITY OPTIONS FOR KAFKA

Use the **Kafka** resource to configure the mechanisms used for Kafka authentication and authorization.

8.1.1. Listener authentication

Configure client authentication for Kafka brokers when creating listeners. Specify the listener authentication type using the **Kafka.spec.kafka.listeners.authentication** property in the **Kafka** resource.

For clients inside the OpenShift cluster, you can create **plain** (without encryption) or **tls** *internal* listeners. The **internal** listener type use a headless service and the DNS names given to the broker pods. As an alternative to the headless service, you can also create a **cluster-ip** type of internal listener to expose Kafka using per-broker **ClusterIP** services. For clients outside the OpenShift cluster, you create *external* listeners and specify a connection mechanism, which can be **nodeport**, **loadbalancer**, **ingress** (Kubernetes only), or **route** (OpenShift only).

For more information on the configuration options for connecting an external client, see Chapter 7, *Setting up client access to a Kafka cluster* .

Supported authentication options:

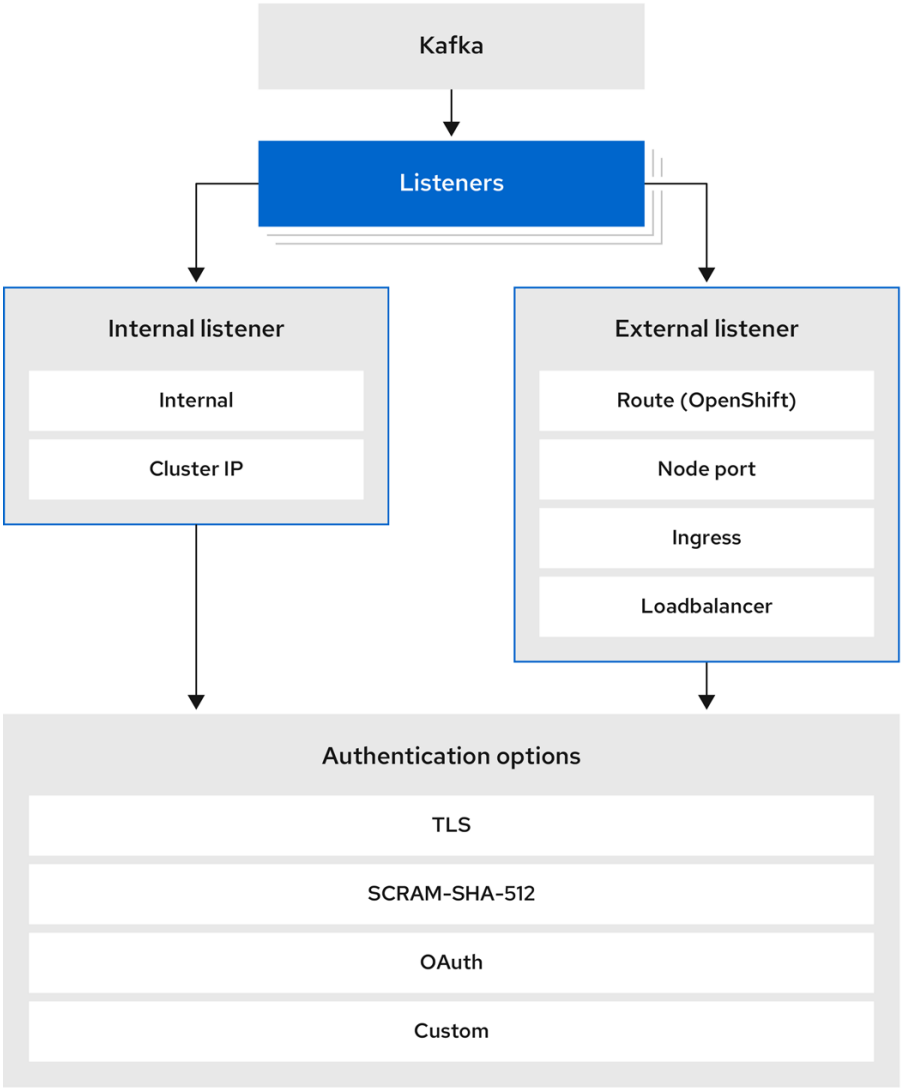
1. mTLS authentication (only on the listeners with TLS enabled encryption)
2. SCRAM-SHA-512 authentication
3. OAuth 2.0 token-based authentication
4. Custom authentication

The authentication option you choose depends on how you wish to authenticate client access to Kafka brokers.

NOTE

Try exploring the standard authentication options before using custom authentication. Custom authentication allows for any type of kafka-supported authentication. It can provide more flexibility, but also adds complexity.

Figure 8.1. Kafka listener authentication options



The listener **authentication** property is used to specify an authentication mechanism specific to that listener.

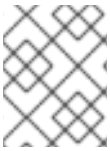
If no **authentication** property is specified then the listener does not authenticate clients which connect through that listener. The listener will accept all connections without authentication.

Authentication must be configured when using the User Operator to manage **KafkaUsers**.

The following example shows:

- A **plain** listener configured for SCRAM-SHA-512 authentication
- A **tls** listener with mTLS authentication
- An **external** listener with mTLS authentication

Each listener is configured with a unique name and port within a Kafka cluster.



NOTE

Listeners cannot be configured to use the ports reserved for inter-broker communication (9091 or 9090) and metrics (9404).

Example listener authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: true
        authentication:
          type: scram-sha-512
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external
        port: 9094
        type: loadbalancer
        tls: true
        authentication:
          type: tls
# ...
```

8.1.1.1. mTLS authentication

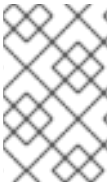
mTLS authentication is always used for the communication between Kafka brokers and ZooKeeper pods.

AMQ Streams can configure Kafka to use TLS (Transport Layer Security) to provide encrypted communication between Kafka brokers and clients either with or without mutual authentication. For mutual, or two-way, authentication, both the server and the client present certificates. When you configure mTLS authentication, the broker authenticates the client (client authentication) and the client authenticates the broker (server authentication).

mTLS listener configuration in the **Kafka** resource requires the following:

- **tls: true** to specify TLS encryption and server authentication
- **authentication.type: tls** to specify the client authentication

When a Kafka cluster is created by the Cluster Operator, it creates a new secret with the name **<cluster_name>-cluster-ca-cert**. The secret contains a CA certificate. The CA certificate is in PEM and PKCS #12 format. To verify a Kafka cluster, add the CA certificate to the truststore in your client configuration. To verify a client, add a user certificate and key to the keystore in your client configuration. For more information on configuring a client for mTLS, see Section 8.2.2, "User authentication".



NOTE

TLS authentication is more commonly one-way, with one party authenticating the identity of another. For example, when HTTPS is used between a web browser and a web server, the browser obtains proof of the identity of the web server.

8.1.1.2. SCRAM-SHA-512 authentication

SCRAM (Salted Challenge Response Authentication Mechanism) is an authentication protocol that can establish mutual authentication using passwords. AMQ Streams can configure Kafka to use SASL (Simple Authentication and Security Layer) SCRAM-SHA-512 to provide authentication on both unencrypted and encrypted client connections.

When SCRAM-SHA-512 authentication is used with a TLS connection, the TLS protocol provides the encryption, but is not used for authentication.

The following properties of SCRAM make it safe to use SCRAM-SHA-512 even on unencrypted connections:

- The passwords are not sent in the clear over the communication channel. Instead the client and the server are each challenged by the other to offer proof that they know the password of the authenticating user.
- The server and client each generate a new challenge for each authentication exchange. This means that the exchange is resilient against replay attacks.

When **KafkaUser.spec.authentication.type** is configured with **scram-sha-512** the User Operator will generate a random 12-character password consisting of upper and lowercase ASCII letters and numbers.

8.1.1.3. Network policies

By default, AMQ Streams automatically creates a **NetworkPolicy** resource for every listener that is enabled on a Kafka broker. This **NetworkPolicy** allows applications to connect to listeners in all namespaces. Use network policies as part of the listener configuration.

If you want to restrict access to a listener at the network level to only selected applications or namespaces, use the **networkPolicyPeers** property. Each listener can have a different **networkPolicyPeers** configuration. For more information on network policy peers, refer to the NetworkPolicyPeer API reference.

If you want to use custom network policies, you can set the **STRIMZI_NETWORK_POLICY_GENERATION** environment variable to **false** in the Cluster Operator configuration. For more information, see Section 13.2.3, "Configuring the Cluster Operator with environment variables".

NOTE

Your configuration of OpenShift must support ingress **NetworkPolicies** in order to use network policies in AMQ Streams.

8.1.1.4. Providing listener certificates

You can provide your own server certificates, called *Kafka listener certificates*, for TLS listeners or external listeners which have TLS encryption enabled. For more information, see Section 8.3.4, "Providing your own Kafka listener certificates for TLS encryption".

Additional resources

- **GenericKafkaListener** schema reference

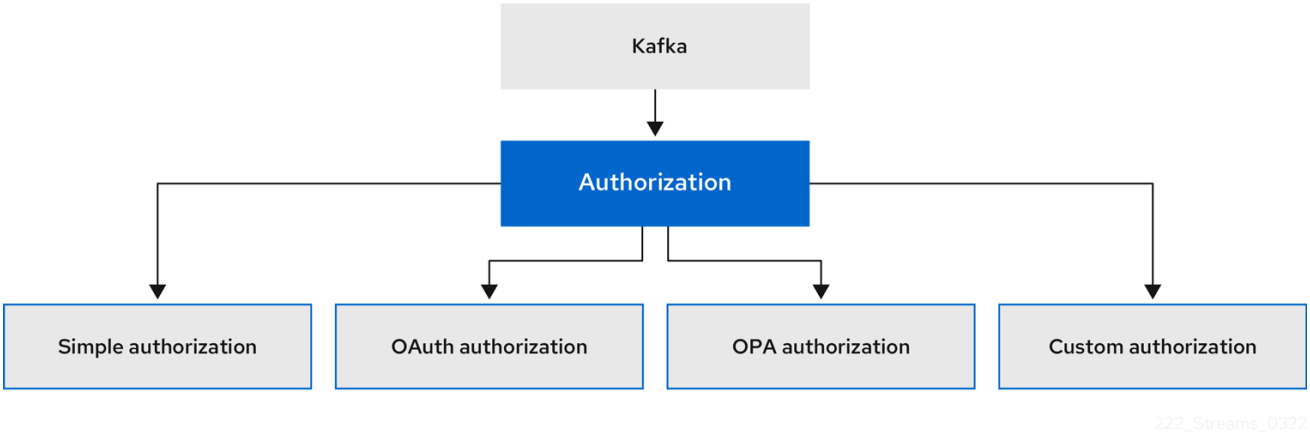
8.1.2. Kafka authorization

Configure authorization for Kafka brokers using the **Kafka.spec.kafka.authorization** property in the **Kafka** resource. If the **authorization** property is missing, no authorization is enabled and clients have no restrictions. When enabled, authorization is applied to all enabled listeners. The authorization method is defined in the **type** field.

Supported authorization options:

- Simple authorization
- OAuth 2.0 authorization (if you are using OAuth 2.0 token based authentication)
- Open Policy Agent (OPA) authorization
- Custom authorization

Figure 8.2. Kafka cluster authorization options



8.1.2.1. Super users

Super users can access all resources in your Kafka cluster regardless of any access restrictions, and are supported by all authorization mechanisms.

To designate super users for a Kafka cluster, add a list of user principals to the **superUsers** property. If a user uses mTLS authentication, the username is the common name from the TLS certificate subject

prefixed with **CN=**. If you are not using the User Operator and using your own certificates for mTLS, the username is the full certificate subject. A full certificate subject can have the following fields: **CN=user,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=my_country_code**. Omit any fields that are not present.

An example configuration with super users

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: simple
      superUsers:
        - CN=client_1
        - user_2
        - CN=client_3
        - CN=client_4,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=US
        - CN=client_5,OU=my_ou,O=my_org,C=GB
        - CN=client_6,O=my_org
    # ...
```

8.2. SECURITY OPTIONS FOR KAFKA CLIENTS

Use the **KafkaUser** resource to configure the authentication mechanism, authorization mechanism, and access rights for Kafka clients. In terms of configuring security, clients are represented as users.

You can authenticate and authorize user access to Kafka brokers. Authentication permits access, and authorization constrains the access to permissible actions.

You can also create *super users* that have unconstrained access to Kafka brokers.

The authentication and authorization mechanisms must match the specification for the listener used to access the Kafka brokers.

For more information on configuring a **KafkaUser** resource to access Kafka brokers securely, see Section 7.3, “Setting up client access to a Kafka cluster using listeners” .

8.2.1. Identifying a Kafka cluster for user handling

A **KafkaUser** resource includes a label that defines the appropriate name of the Kafka cluster (derived from the name of the **Kafka** resource) to which it belongs.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
```

The label is used by the User Operator to identify the **KafkaUser** resource and create a new user, and also in subsequent handling of the user.

If the label does not match the Kafka cluster, the User Operator cannot identify the **KafkaUser** and the user is not created.

If the status of the **KafkaUser** resource remains empty, check your label.

8.2.2. User authentication

Use the **KafkaUser** custom resource to configure authentication credentials for users (clients) that require access to a Kafka cluster. Configure the credentials using the **authentication** property in **KafkaUser.spec**. By specifying a **type**, you control what credentials are generated.

Supported authentication types:

- **tls** for mTLS authentication
- **tls-external** for mTLS authentication using external certificates
- **scram-sha-512** for SCRAM-SHA-512 authentication

If **tls** or **scram-sha-512** is specified, the User Operator creates authentication credentials when it creates the user. If **tls-external** is specified, the user still uses mTLS, but no authentication credentials are created. Use this option when you're providing your own certificates. When no authentication type is specified, the User Operator does not create the user or its credentials.

You can use **tls-external** to authenticate with mTLS using a certificate issued outside the User Operator. The User Operator does not generate a TLS certificate or a secret. You can still manage ACL rules and quotas through the User Operator in the same way as when you're using the **tls** mechanism. This means that you use the **CN=*USER-NAME*** format when specifying ACL rules and quotas. *USER-NAME* is the common name given in a TLS certificate.

8.2.2.1. mTLS authentication

To use mTLS authentication, you set the **type** field in the **KafkaUser** resource to **tls**.

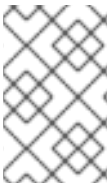
Example user with mTLS authentication enabled

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  # ...
```

The authentication type must match the equivalent configuration for the **Kafka** listener used to access the Kafka cluster.

When the user is created by the User Operator, it creates a new secret with the same name as the **KafkaUser** resource. The secret contains a private and public key for mTLS. The public key is contained in a user certificate, which is signed by a clients CA (certificate authority) when it is created. All keys are

in X.509 format.



NOTE

If you are using the clients CA generated by the Cluster Operator, the user certificates generated by the User Operator are also renewed when the clients CA is renewed by the Cluster Operator.

The user secret provides keys and certificates in PEM and PKCS #12 formats .

Example secret with user credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: <public_key> # Public key of the clients CA
  user.crt: <user_certificate> # Public key of the user
  user.key: <user_private_key> # Private key of the user
  user.p12: <store> # PKCS #12 store for user certificates and keys
  user.password: <password_for_store> # Protects the PKCS #12 store
```

When you configure a client, you specify the following:

- **Truststore** properties for the public cluster CA certificate to verify the identity of the Kafka cluster
- **Keystore** properties for the user authentication credentials to verify the client

The configuration depends on the file format (PEM or PKCS #12). This example uses PKCS #12 stores, and the passwords required to access the credentials in the stores.

Example client configuration using mTLS in PKCS #12 format

```
bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093 1
security.protocol=SSL 2
ssl.truststore.location=/tmp/ca.p12 3
ssl.truststore.password=<truststore_password> 4
ssl.keystore.location=/tmp/user.p12 5
ssl.keystore.password=<keystore_password> 6
```

1. The bootstrap server address to connect to the Kafka cluster.
2. The security protocol option when using TLS for encryption.
3. The truststore location contains the public key certificate (**ca.p12**) for the Kafka cluster. A cluster CA certificate and password is generated by the Cluster Operator in the **<cluster_name>-cluster-ca-cert** secret when the Kafka cluster is created.

4 The password (**ca.password**) for accessing the truststore.

5 The keystore location contains the public key certificate (**user.p12**) for the Kafka user.

6 The password (**user.password**) for accessing the keystore.

8.2.2.2. mTLS authentication using a certificate issued outside the User Operator

To use mTLS authentication using a certificate issued outside the User Operator, you set the **type** field in the **KafkaUser** resource to **tls-external**. A secret and credentials are not created for the user.

Example user with mTLS authentication that uses a certificate issued outside the User Operator

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls-external
  # ...
```

8.2.2.3. SCRAM-SHA-512 authentication

To use the SCRAM-SHA-512 authentication mechanism, you set the **type** field in the **KafkaUser** resource to **scram-sha-512**.

Example user with SCRAM-SHA-512 authentication enabled

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  # ...
```

When the user is created by the User Operator, it creates a new secret with the same name as the **KafkaUser** resource. The secret contains the generated password in the **password** key, which is encoded with base64. In order to use the password, it must be decoded.

Example secret with user credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
```

```
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  password: Z2VuZXJhdGVkcGFzc3dvcmQ= (1)
  sasl.jaas.config:
b3JnLmFwYWNoZS5rYWZrYS5jb21tb24uc2VjdXJpdHkuc2NyYW0uU2NyYW1Mb2dpbk1vZHVsZSByZ
XF1aXJlZCB1c2VybmFtZT0ibXktdXNlciIgcGFzc3dvcmQ9ImdlbmVyYXRlZHBhc3N3b3JkIjsK (2)
```

(1) The generated password, base64 encoded.

(2) The JAAS configuration string for SASL SCRAM-SHA-512 authentication, base64 encoded.

Decoding the generated password:

```
echo "Z2VuZXJhdGVkcGFzc3dvcmQ=" | base64 --decode
```

8.2.2.3.1. Custom password configuration

When a user is created, AMQ Streams generates a random password. You can use your own password instead of the one generated by AMQ Streams. To do so, create a secret with the password and reference it in the **KafkaUser** resource.

Example user with a password set for SCRAM-SHA-512 authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
    password:
      valueFrom:
        secretKeyRef:
          name: my-secret (1)
          key: my-password (2)
  # ...
```

(1) The name of the secret containing the predefined password.

(2) The key for the password stored inside the secret.

8.2.3. User authorization

Use the **KafkaUser** custom resource to configure authorization rules for users (clients) that require access to a Kafka cluster. Configure the rules using the **authorization** property in **KafkaUser.spec**. By specifying a **type**, you control what rules are used.

To use simple authorization, you set the **type** property to **simple** in **KafkaUser.spec.authorization**. The simple authorization uses the Kafka Admin API to manage the ACL rules inside your Kafka cluster. Whether ACL management in the User Operator is enabled or not depends on your authorization configuration in the Kafka cluster.

- For simple authorization, ACL management is always enabled.
- For OPA authorization, ACL management is always disabled. Authorization rules are configured in the OPA server.
- For Red Hat Single Sign-On authorization, you can manage the ACL rules directly in Red Hat Single Sign-On. You can also delegate authorization to the simple authorizer as a fallback option in the configuration. When delegation to the simple authorizer is enabled, the User Operator will enable management of ACL rules as well.
- For custom authorization using a custom authorization plugin, use the **supportsAdminApi** property in the **.spec.kafka.authorization** configuration of the **Kafka** custom resource to enable or disable the support.

Authorization is cluster-wide. The authorization type must match the equivalent configuration in the **Kafka** custom resource.

If ACL management is not enabled, AMQ Streams rejects a resource if it contains any ACL rules.

If you're using a standalone deployment of the User Operator, ACL management is enabled by default. You can disable it using the **STRIMZI_ACLS_ADMIN_API_SUPPORTED** environment variable.

If no authorization is specified, the User Operator does not provision any access rights for the user. Whether such a **KafkaUser** can still access resources depends on the authorizer being used. For example, for the **AclAuthorizer** this is determined by its **allow.everyone.if.no.acl.found** configuration.

8.2.3.1. ACL rules

AclAuthorizer uses ACL rules to manage access to Kafka brokers.

ACL rules grant access rights to the user, which you specify in the **acls** property.

For more information about the **AclRule** object, see the **AclRule** schema reference.

8.2.3.2. Super user access to Kafka brokers

If a user is added to a list of super users in a Kafka broker configuration, the user is allowed unlimited access to the cluster regardless of any authorization constraints defined in ACLs in **KafkaUser**.

For more information on configuring super user access to brokers, see Kafka authorization.

8.2.3.3. User quotas

You can configure the **spec** for the **KafkaUser** resource to enforce quotas so that a user does not exceed a configured level of access to Kafka brokers. You can set size-based network usage and time-based CPU utilization thresholds. You can also add a partition mutation quota to control the rate at which requests to change partitions are accepted for user requests.

An example KafkaUser with user quotas

```
apiVersion: kafka.strimzi.io/v1beta2
```

```
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  quotas:
    producerByteRate: 1048576 (1)
    consumerByteRate: 2097152 (2)
    requestPercentage: 55 (3)
    controllerMutationRate: 10 (4)
```

1. Byte-per-second quota on the amount of data the user can push to a Kafka broker
2. Byte-per-second quota on the amount of data the user can fetch from a Kafka broker
3. CPU utilization limit as a percentage of time for a client group
4. Number of concurrent partition creation and deletion operations (mutations) allowed per second

For more information on these properties, see the **KafkaUserQuotas** schema reference.

8.3. SECURING ACCESS TO KAFKA BROKERS

To establish secure access to Kafka brokers, you configure and apply:

- A **Kafka** resource to:
  - Create listeners with a specified authentication type
  - Configure authorization for the whole Kafka cluster
- A **KafkaUser** resource to access the Kafka brokers securely through the listeners

Configure the **Kafka** resource to set up:

- Listener authentication
- Network policies that restrict access to Kafka listeners
- Kafka authorization
- Super users for unconstrained access to brokers

Authentication is configured independently for each listener. Authorization is always configured for the whole Kafka cluster.

The Cluster Operator creates the listeners and sets up the cluster and client certificate authority (CA) certificates to enable authentication within the Kafka cluster.

You can replace the certificates generated by the Cluster Operator by installing your own certificates.

You can also provide your own server certificates and private keys for any listener with TLS encryption enabled. These user-provided certificates are called *Kafka listener certificates*. Providing Kafka listener

certificates allows you to leverage existing security infrastructure, such as your organization's private CA or a public CA. Kafka clients will need to trust the CA which was used to sign the listener certificate. You must manually renew Kafka listener certificates when needed. Certificates are available in PKCS #12 format (.p12) and PEM (.crt) formats.

Use **KafkaUser** to enable the authentication and authorization mechanisms that a specific client uses to access Kafka.

Configure the **KafkaUser** resource to set up:

- Authentication to match the enabled listener authentication
- Authorization to match the enabled Kafka authorization
- Quotas to control the use of resources by clients

The User Operator creates the user representing the client and the security credentials used for client authentication, based on the chosen authentication type.

Refer to the schema reference for more information on access configuration properties:

- **Kafka** schema reference
- **KafkaUser** schema reference
- **GenericKafkaListener** schema reference

8.3.1. Securing Kafka brokers

This procedure shows the steps involved in securing Kafka brokers when running AMQ Streams.

The security implemented for Kafka brokers must be compatible with the security implemented for the clients requiring access.

- **Kafka.spec.kafka.listeners[*].authentication** matches **KafkaUser.spec.authentication**
- **Kafka.spec.kafka.authorization** matches **KafkaUser.spec.authorization**

The steps show the configuration for simple authorization and a listener using mTLS authentication. For more information on listener configuration, see the **GenericKafkaListener** schema reference.

Alternatively, you can use SCRAM-SHA or OAuth 2.0 for listener authentication, and OAuth 2.0 or OPA for Kafka authorization.

Procedure

1. Configure the **Kafka** resource.
   a. Configure the **authorization** property for authorization.
   b. Configure the **listeners** property to create a listener with authentication.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
```

```
# ...
authorization: 1
  type: simple
  superUsers: 2
    - CN=client_1
    - user_2
    - CN=client_3
listeners:
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication:
      type: tls 3
# ...
zookeeper:
  # ...
```

1. Authorization enables **simple** authorization on the Kafka broker using the **AclAuthorizer** Kafka plugin.
2. List of user principals with unlimited access to Kafka. *CN* is the common name from the client certificate when mTLS authentication is used.
3. Listener authentication mechanisms may be configured for each listener, and specified as mTLS, SCRAM-SHA-512, or token-based OAuth 2.0.

If you are configuring an external listener, the configuration is dependent on the chosen connection mechanism.

2. Create or update the **Kafka** resource.

```
oc apply -f <kafka_configuration_file>
```

The Kafka cluster is configured with a Kafka broker listener using mTLS authentication.

A service is created for each Kafka broker pod.

A service is created to serve as the *bootstrap address* for connection to the Kafka cluster.

The cluster CA certificate to verify the identity of the kafka brokers is also created in the secret ***<cluster_name>*-cluster-ca-cert**.

8.3.2. Securing user access to Kafka

Create or modify a **KafkaUser** to represent a client that requires secure access to the Kafka cluster.

When you configure the **KafkaUser** authentication and authorization mechanisms, ensure they match the equivalent **Kafka** configuration:

- **KafkaUser.spec.authentication** matches **Kafka.spec.kafka.listeners[*].authentication**
- **KafkaUser.spec.authorization** matches **Kafka.spec.kafka.authorization**

This procedure shows how a user is created with mTLS authentication. You can also create a user with SCRAM-SHA authentication.

The authentication required depends on the type of authentication configured for the Kafka broker listener.

NOTE

Authentication between Kafka users and Kafka brokers depends on the authentication settings for each. For example, it is not possible to authenticate a user with mTLS if it is not also enabled in the Kafka configuration.

Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.
- A running User Operator (typically deployed with the Entity Operator).

The authentication type in **KafkaUser** should match the authentication configured in **Kafka** brokers.

Procedure

1. Configure the **KafkaUser** resource.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication: 1
    type: tls
  authorization:
    type: simple 2
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Describe
          - Read
      - resource:
          type: group
          name: my-group
          patternType: literal
        operations:
          - Read
```

1. User authentication mechanism, defined as mutual **tls** or **scram-sha-512**.
2. Simple authorization, which requires an accompanying list of ACL rules.

2. Create or update the **KafkaUser** resource.

```
oc apply -f <user_config_file>
```

The user is created, as well as a Secret with the same name as the **KafkaUser** resource. The Secret contains a private and public key for mTLS authentication.

For information on configuring a Kafka client with properties for secure connection to Kafka brokers, see Section 7.3, “Setting up client access to a Kafka cluster using listeners” .

8.3.3. Restricting access to Kafka listeners using network policies

You can restrict access to a listener to only selected applications by using the **networkPolicyPeers** property.

Prerequisites

- An OpenShift cluster with support for Ingress NetworkPolicies.
- The Cluster Operator is running.

Procedure

1. Open the **Kafka** resource.
2. In the **networkPolicyPeers** property, define the application pods or namespaces that will be allowed to access the Kafka cluster.
   For example, to configure a **tls** listener to allow connections only from application pods with the label **app** set to **kafka-client**:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
        networkPolicyPeers:
          - podSelector:
              matchLabels:
                app: kafka-client
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.
   Use **oc apply**:

```
oc apply -f your-file
```

Additional resources

- **networkPolicyPeers** configuration
- NetworkPolicyPeer API reference

8.3.4. Providing your own Kafka listener certificates for TLS encryption

Listeners provide client access to Kafka brokers. Configure listeners in the **Kafka** resource, including the configuration required for client access using TLS.

By default, the listeners use certificates signed by the internal CA (certificate authority) certificates generated by AMQ Streams. A CA certificate is generated by the Cluster Operator when it creates a Kafka cluster. When you configure a client for TLS, you add the CA certificate to its truststore configuration to verify the Kafka cluster. You can also install and use your own CA certificates. Or you can configure a listener using **brokerCertChainAndKey** properties and use a custom server certificate.

The **brokerCertChainAndKey** properties allow you to access Kafka brokers using your own custom certificates at the listener-level. You create a secret with your own private key and server certificate, then specify the key and certificate in the listener's **brokerCertChainAndKey** configuration. You can use a certificate signed by a public (external) CA or a private CA. If signed by a public CA, you usually won't need to add it to a client's truststore configuration. Custom certificates are not managed by AMQ Streams, so you need to renew them manually.

NOTE

Listener certificates are used for TLS encryption and server authentication only. They are not used for TLS client authentication. If you want to use your own certificate for TLS client authentication as well, you must install and use your own clients CA.

Prerequisites

- The Cluster Operator is running.
- Each listener requires the following:
  - A compatible server certificate signed by an external CA. (Provide an X.509 certificate in PEM format.)
    You can use one listener certificate for multiple listeners.
  - Subject Alternative Names (SANs) are specified in the certificate for each listener. For more information, see Section 8.3.5, "Alternative subjects in server certificates for Kafka listeners".

If you are not using a self-signed certificate, you can provide a certificate that includes the whole CA chain in the certificate.

You can only use the **brokerCertChainAndKey** properties if TLS encryption (**tls: true**) is configured for the listener.

NOTE

AMQ Streams does not support the use of encrypted private keys for TLS. The private key stored in the secret must be unencrypted for this to work.

Procedure

1. Create a **Secret** containing your private key and server certificate:

```
oc create secret generic my-secret --from-file=my-listener-key.key --from-file=my-listener-certificate.crt
```

2. Edit the **Kafka** resource for your cluster.
Configure the listener to use your **Secret**, certificate file, and private key file in the **configuration.brokerCertChainAndKey** property.

Example configuration for a loadbalancer external listener with TLS encryption enabled

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
# ...
```

Example configuration for a TLS listener

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: tls
    port: 9093
    type: internal
    tls: true
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
# ...
```

3. Apply the new configuration to create or update the resource:

```
oc apply -f kafka.yaml
```

The Cluster Operator starts a rolling update of the Kafka cluster, which updates the configuration of the listeners.

NOTE

A rolling update is also started if you update a Kafka listener certificate in a **Secret** that is already used by a listener.

8.3.5. Alternative subjects in server certificates for Kafka listeners

In order to use TLS hostname verification with your own Kafka listener certificates, you must use the correct Subject Alternative Names (SANs) for each listener. The certificate SANs must specify hostnames for the following:

- All of the Kafka brokers in your cluster
- The Kafka cluster bootstrap service

You can use wildcard certificates if they are supported by your CA.

8.3.5.1. Examples of SANs for internal listeners

Use the following examples to help you specify hostnames of the SANs in your certificates for your internal listeners.

Replace **<cluster-name>** with the name of the Kafka cluster and **<namespace>** with the OpenShift namespace where the cluster is running.

Wildcards example for a type: internal listener

```
//Kafka brokers
*.<cluster-name>-kafka-brokers
*.<cluster-name>-kafka-brokers.<namespace>.svc

// Bootstrap service
<cluster-name>-kafka-bootstrap
<cluster-name>-kafka-bootstrap.<namespace>.svc
```

Non-wildcards example for a type: internal listener

```
// Kafka brokers
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers.<namespace>.svc
<cluster-name>-kafka-1.<cluster-name>-kafka-brokers
<cluster-name>-kafka-1.<cluster-name>-kafka-brokers.<namespace>.svc
# ...

// Bootstrap service
<cluster-name>-kafka-bootstrap
<cluster-name>-kafka-bootstrap.<namespace>.svc
```

Non-wildcards example for a type: cluster-ip listener

```
// Kafka brokers
```

```
<cluster-name>-kafka-<listener-name>-0
<cluster-name>-kafka-<listener-name>-0.<namespace>.svc
<cluster-name>-kafka-<listener-name>-1
<cluster-name>-kafka-<listener-name>-1.<namespace>.svc
# ...

// Bootstrap service
<cluster-name>-kafka-<listener-name>-bootstrap
<cluster-name>-kafka-<listener-name>-bootstrap.<namespace>.svc
```

8.3.5.2. Examples of SANs for external listeners

For external listeners which have TLS encryption enabled, the hostnames you need to specify in certificates depends on the external listener **type**.

Table 8.1. SANs for each type of external listener

External listener type	In the SANs, specify...
ingress	Addresses of all Kafka broker **Ingress** resources and the address of the bootstrap **Ingress**. You can use a matching wildcard name.
route	Addresses of all Kafka broker **Routes** and the address of the bootstrap **Route**. You can use a matching wildcard name.
loadbalancer	Addresses of all Kafka broker **loadbalancers** and the bootstrap **loadbalancer** address. You can use a matching wildcard name.
nodeport	Addresses of all OpenShift worker nodes that the Kafka broker pods might be scheduled to. You can use a matching wildcard name.

Additional resources

- Section 8.3.4, "Providing your own Kafka listener certificates for TLS encryption"

8.4. USING OAUTH 2.0 TOKEN-BASED AUTHENTICATION

AMQ Streams supports the use of OAuth 2.0 authentication using the *OAUTHBEARER* and *PLAIN* mechanisms.

OAuth 2.0 enables standardized token-based authentication and authorization between applications, using a central authorization server to issue tokens that grant limited access to resources.

You can configure OAuth 2.0 authentication, then OAuth 2.0 authorization.

Kafka brokers and clients both need to be configured to use OAuth 2.0. OAuth 2.0 authentication can also be used in conjunction with **simple** or OPA-based Kafka authorization.

Using OAuth 2.0 token-based authentication, application clients can access resources on application servers (called *resource servers*) without exposing account credentials.

The application client passes an access token as a means of authenticating, which application servers can also use to determine the level of access to grant. The authorization server handles the granting of access and inquiries about access.

In the context of AMQ Streams:

- Kafka brokers act as OAuth 2.0 resource servers
- Kafka clients act as OAuth 2.0 application clients

Kafka clients authenticate to Kafka brokers. The brokers and clients communicate with the OAuth 2.0 authorization server, as necessary, to obtain or validate access tokens.

For a deployment of AMQ Streams, OAuth 2.0 integration provides:

- Server-side OAuth 2.0 support for Kafka brokers
- Client-side OAuth 2.0 support for Kafka MirrorMaker, Kafka Connect and the Kafka Bridge

8.4.1. OAuth 2.0 authentication mechanisms

AMQ Streams supports the OAUTHBEARER and PLAIN mechanisms for OAuth 2.0 authentication. Both mechanisms allow Kafka clients to establish authenticated sessions with Kafka brokers. The authentication flow between clients, the authorization server, and Kafka brokers is different for each mechanism.

We recommend that you configure clients to use OAUTHBEARER whenever possible. OAUTHBEARER provides a higher level of security than PLAIN because client credentials are *never* shared with Kafka brokers. Consider using PLAIN only with Kafka clients that do not support OAUTHBEARER.

You configure Kafka broker listeners to use OAuth 2.0 authentication for connecting clients. If necessary, you can use the OAUTHBEARER and PLAIN mechanisms on the same **oauth** listener. The properties to support each mechanism must be explicitly specified in the **oauth** listener configuration.

OAUTHBEARER overview

OAUTHBEARER is automatically enabled in the **oauth** listener configuration for the Kafka broker. You can set the **enableOauthBearer** property to **true**, though this is not required.

```
# ...
authentication:
  type: oauth
  # ...
  enableOauthBearer: true
```

Many Kafka client tools use libraries that provide basic support for OAUTHBEARER at the protocol level. To support application development, AMQ Streams provides an *OAuth callback handler* for the upstream Kafka Client Java libraries (but not for other libraries). Therefore, you do not need to write your own callback handlers. An application client can use the callback handler to provide the access token. Clients written in other languages, such as Go, must use custom code to connect to the authorization server and obtain the access token.

With OAUTHBEARER, the client initiates a session with the Kafka broker for credentials exchange, where credentials take the form of a bearer token provided by the callback handler. Using the callbacks, you can configure token provision in one of three ways:

- Client ID and Secret (by using the *OAuth 2.0 client credentials* mechanism)
- A long-lived access token, obtained manually at configuration time
- A long-lived refresh token, obtained manually at configuration time

NOTE

OAUTHBEARER authentication can only be used by Kafka clients that support the OAUTHBEARER mechanism at the protocol level.

PLAIN overview

To use PLAIN, you must enable it in the **oauth** listener configuration for the Kafka broker.

In the following example, PLAIN is enabled in addition to OAUTHBEARER, which is enabled by default. If you want to use PLAIN only, you can disable OAUTHBEARER by setting **enableOauthBearer** to **false**.

```
# ...
authentication:
  type: oauth
  # ...
  enablePlain: true
  tokenEndpointUri: https://OAUTH-SERVER-ADDRESS/auth/realms/external/protocol/openid-connect/token
```

PLAIN is a simple authentication mechanism used by all Kafka client tools. To enable PLAIN to be used with OAuth 2.0 authentication, AMQ Streams provides *OAuth 2.0 over PLAIN* server-side callbacks.

With the AMQ Streams implementation of PLAIN, the client credentials are not stored in ZooKeeper. Instead, client credentials are handled centrally behind a compliant authorization server, similar to when OAUTHBEARER authentication is used.

When used with the OAuth 2.0 over PLAIN callbacks, Kafka clients authenticate with Kafka brokers using either of the following methods:

- Client ID and secret (by using the OAuth 2.0 client credentials mechanism)
- A long-lived access token, obtained manually at configuration time

For both methods, the client must provide the PLAIN **username** and **password** properties to pass credentials to the Kafka broker. The client uses these properties to pass a client ID and secret or username and access token.

Client IDs and secrets are used to obtain access tokens.

Access tokens are passed as **password** property values. You pass the access token with or without an **$accessToken:** prefix.

- If you configure a token endpoint (**tokenEndpointUri**) in the listener configuration, you need the prefix.

- If you don't configure a token endpoint (**tokenEndpointUri**) in the listener configuration, you don't need the prefix. The Kafka broker interprets the password as a raw access token.

If the **password** is set as the access token, the **username** must be set to the same principal name that the Kafka broker obtains from the access token. You can specify username extraction options in your listener using the **userNameClaim**, **fallbackUserNameClaim**, **fallbackUsernamePrefix**, and **userInfoEndpointUri** properties. The username extraction process also depends on your authorization server; in particular, how it maps client IDs to account names.

NOTE

OAuth over PLAIN does not support **password grant** mechanism. You can only 'proxy' through SASL PLAIN mechanism the **client credentials** (clientId + secret) or the access token as described above.

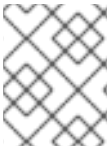
Additional resources

- Section 8.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers"

8.4.2. OAuth 2.0 Kafka broker configuration

Kafka broker configuration for OAuth 2.0 involves:

- Creating the OAuth 2.0 client in the authorization server
- Configuring OAuth 2.0 authentication in the Kafka custom resource



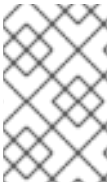
NOTE

In relation to the authorization server, Kafka brokers and Kafka clients are both regarded as OAuth 2.0 clients.

8.4.2.1. OAuth 2.0 client configuration on an authorization server

To configure a Kafka broker to validate the token received during session initiation, the recommended approach is to create an OAuth 2.0 *client* definition in an authorization server, configured as *confidential*, with the following client credentials enabled:

- Client ID of **kafka** (for example)
- Client ID and Secret as the authentication mechanism



NOTE

You only need to use a client ID and secret when using a non-public introspection endpoint of the authorization server. The credentials are not typically required when using public authorization server endpoints, as with fast local JWT token validation.

8.4.2.2. OAuth 2.0 authentication configuration in the Kafka cluster

To use OAuth 2.0 authentication in the Kafka cluster, you specify, for example, a **tls** listener configuration for your Kafka cluster custom resource with the authentication method **oauth**:

Assigining the authentication method type for OAuth 2.0

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
      #...
```

You can configure OAuth 2.0 authentication in your listeners. We recommend using OAuth 2.0 authentication together with TLS encryption (**tls: true**). Without encryption, the connection is vulnerable to network eavesdropping and unauthorized access through token theft.

You configure an **external** listener with **type: oauth** for a secure transport layer to communicate with the client.

Using OAuth 2.0 with an external listener

```
# ...
listeners:
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: oauth
    #...
```

The **tls** property is *false* by default, so it must be enabled.

When you have defined the type of authentication as OAuth 2.0, you add configuration based on the type of validation, either as fast local JWT validation or token validation using an introspection endpoint.

The procedure to configure OAuth 2.0 for listeners, with descriptions and examples, is described in Configuring OAuth 2.0 support for Kafka brokers.

8.4.2.3. Fast local JWT token validation configuration

Fast local JWT token validation checks a JWT token signature locally.

The local check ensures that a token:

- Conforms to type by containing a (*typ*) claim value of **Bearer** for an access token
- Is valid (not expired)
- Has an issuer that matches a **validIssuerURI**

You specify a **validIssuerURI** attribute when you configure the listener, so that any tokens not issued by the authorization server are rejected.

The authorization server does not need to be contacted during fast local JWT token validation. You activate fast local JWT token validation by specifying a **jwksEndpointUri** attribute, the endpoint exposed by the OAuth 2.0 authorization server. The endpoint contains the public keys used to validate signed JWT tokens, which are sent as credentials by Kafka clients.

NOTE

All communication with the authorization server should be performed using TLS encryption.

You can configure a certificate truststore as an OpenShift Secret in your AMQ Streams project namespace, and use a **tlsTrustedCertificates** attribute to point to the OpenShift Secret containing the truststore file.

You might want to configure a **userNameClaim** to properly extract a username from the JWT token. If you want to use Kafka ACL authorization, you need to identify the user by their username during authentication. (The **sub** claim in JWT tokens is typically a unique ID, not a username.)

Example configuration for fast local JWT token validation

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    #...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          jwksEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-connect/certs>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
    #...
```

8.4.2.4. OAuth 2.0 introspection endpoint configuration

Token validation using an OAuth 2.0 introspection endpoint treats a received access token as opaque. The Kafka broker sends an access token to the introspection endpoint, which responds with the token information necessary for validation. Importantly, it returns up-to-date information if the specific access token is valid, and also information about when the token expires.

To configure OAuth 2.0 introspection-based validation, you specify an **introspectionEndpointUri** attribute rather than the **jwksEndpointUri** attribute specified for fast local JWT token validation. Depending on the authorization server, you typically have to specify a **clientId** and **clientSecret**, because the introspection endpoint is usually protected.

Example configuration for an introspection endpoint

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          clientId: kafka-broker
          clientSecret:
            secretName: my-cluster-oauth
            key: clientSecret
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          introspectionEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-
connect/token/introspect>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
```

8.4.3. Session re-authentication for Kafka brokers

You can configure **oauth** listeners to use Kafka *session re-authentication* for OAuth 2.0 sessions between Kafka clients and Kafka brokers. This mechanism enforces the expiry of an authenticated session between the client and the broker after a defined period of time. When a session expires, the client immediately starts a new session by reusing the existing connection rather than dropping it.

Session re-authentication is disabled by default. To enable it, you set a time value for **maxSecondsWithoutReauthentication** in the **oauth** listener configuration. The same property is used to configure session re-authentication for OAUTHBEARER and PLAIN authentication. For an example configuration, see Section 8.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers".

Session re-authentication must be supported by the Kafka client libraries used by the client.

Session re-authentication can be used with *fast local JWT* or *introspection endpoint* token validation.

Client re-authentication

When the broker's authenticated session expires, the client must re-authenticate to the existing session by sending a new, valid access token to the broker, without dropping the connection.

If token validation is successful, a new client session is started using the existing connection. If the client fails to re-authenticate, the broker will close the connection if further attempts are made to send or receive messages. Java clients that use Kafka client library 2.2 or later automatically re-authenticate if the re-authentication mechanism is enabled on the broker.

Session re-authentication also applies to refresh tokens, if used. When the session expires, the client refreshes the access token by using its refresh token. The client then uses the new access token to re-authenticate to the existing session.

Session expiry for OAUTHBEARER and PLAIN

When session re-authentication is configured, session expiry works differently for OAUTHBEARER and PLAIN authentication.

For OAUTHBEARER and PLAIN, using the client ID and secret method:

- The broker's authenticated session will expire at the configured **maxSecondsWithoutReauthentication**.
- The session will expire earlier if the access token expires before the configured time.

For PLAIN using the long-lived access token method:

- The broker's authenticated session will expire at the configured **maxSecondsWithoutReauthentication**.
- Re-authentication will fail if the access token expires before the configured time. Although session re-authentication is attempted, PLAIN has no mechanism for refreshing tokens.

If **maxSecondsWithoutReauthentication** is *not* configured, OAUTHBEARER and PLAIN clients can remain connected to brokers indefinitely, without needing to re-authenticate. Authenticated sessions do not end with access token expiry. However, this can be considered when configuring authorization, for example, by using **keycloak** authorization or installing a custom authorizer.

Additional resources

- Section 8.4.2, "OAuth 2.0 Kafka broker configuration"
- Section 8.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers"
- **KafkaListenerAuthenticationOAuth** schema reference
- KIP-368

8.4.4. OAuth 2.0 Kafka client configuration

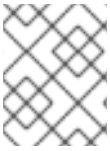
A Kafka client is configured with either:

- The credentials required to obtain a valid access token from an authorization server (client ID and Secret)
- A valid long-lived access token or refresh token, obtained using tools provided by an authorization server

The only information ever sent to the Kafka broker is an access token. The credentials used to authenticate with the authorization server to obtain the access token are never sent to the broker.

When a client obtains an access token, no further communication with the authorization server is needed.

The simplest mechanism is authentication with a client ID and Secret. Using a long-lived access token, or a long-lived refresh token, adds more complexity because there is an additional dependency on authorization server tools.



NOTE

If you are using long-lived access tokens, you may need to configure the client in the authorization server to increase the maximum lifetime of the token.

If the Kafka client is not configured with an access token directly, the client exchanges credentials for an access token during Kafka session initiation by contacting the authorization server. The Kafka client exchanges either:

- Client ID and Secret
- Client ID, refresh token, and (optionally) a secret
- Username and password, with client ID and (optionally) a secret

8.4.5. OAuth 2.0 client authentication flows

OAuth 2.0 authentication flows depend on the underlying Kafka client and Kafka broker configuration. The flows must also be supported by the authorization server used.

The Kafka broker listener configuration determines how clients authenticate using an access token. The client can pass a client ID and secret to request an access token.

If a listener is configured to use PLAIN authentication, the client can authenticate with a client ID and secret or username and access token. These values are passed as the **username** and **password** properties of the PLAIN mechanism.

Listener configuration supports the following token validation options:

- You can use fast local token validation based on JWT signature checking and local token introspection, without contacting an authorization server. The authorization server provides a JWKS endpoint with public certificates that are used to validate signatures on the tokens.
- You can use a call to a token introspection endpoint provided by an authorization server. Each time a new Kafka broker connection is established, the broker passes the access token received from the client to the authorization server. The Kafka broker checks the response to confirm whether or not the token is valid.

NOTE

An authorization server might only allow the use of opaque access tokens, which means that local token validation is not possible.

Kafka client credentials can also be configured for the following types of authentication:

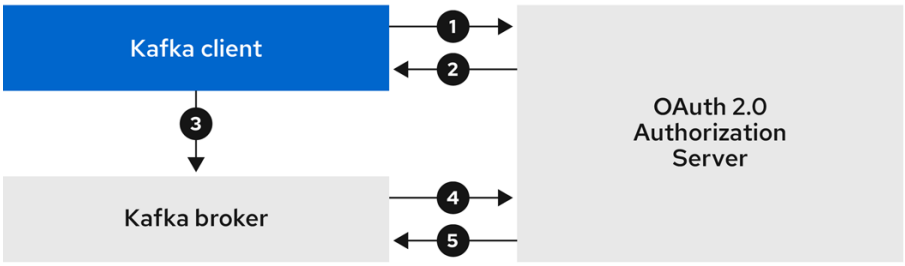
- Direct local access using a previously generated long-lived access token
- Contact with the authorization server for a new access token to be issued (using a client ID and a secret, or a refresh token, or a username and a password)

8.4.5.1. Example client authentication flows using the SASL OAUTHBEARER mechanism

You can use the following communication flows for Kafka authentication using the SASL OAUTHBEARER mechanism.

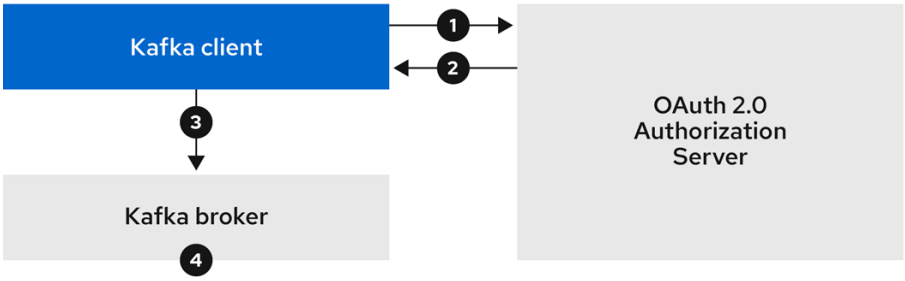
- Client using client ID and secret, with broker delegating validation to authorization server
- Client using client ID and secret, with broker performing fast local token validation
- Client using long-lived access token, with broker delegating validation to authorization server
- Client using long-lived access token, with broker performing fast local validation

Client using client ID and secret, with broker delegating validation to authorization server



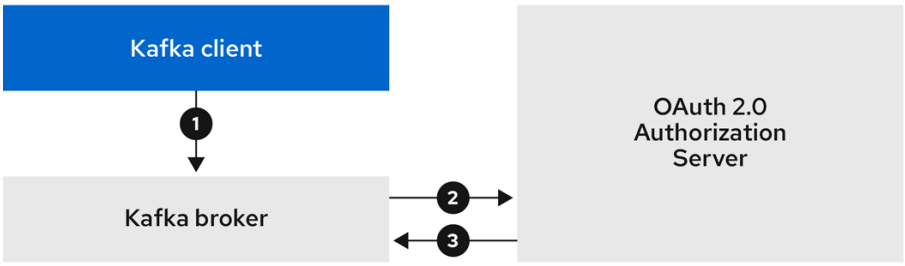
1. The Kafka client requests an access token from the authorization server using a client ID and secret, and optionally a refresh token. Alternatively, the client may authenticate using a username and a password.
2. The authorization server generates a new access token.
3. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the access token.
4. The Kafka broker validates the access token by calling a token introspection endpoint on the authorization server using its own client ID and secret.
5. A Kafka client session is established if the token is valid.

Client using client ID and secret, with broker performing fast local token validation



1. The Kafka client authenticates with the authorization server from the token endpoint, using a client ID and secret, and optionally a refresh token. Alternatively, the client may authenticate using a username and a password.
2. The authorization server generates a new access token.
3. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the access token.
4. The Kafka broker validates the access token locally using a JWT token signature check, and local token introspection.

Client using long-lived access token, with broker delegating validation to authorization server



1. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the long-lived access token.
2. The Kafka broker validates the access token by calling a token introspection endpoint on the authorization server, using its own client ID and secret.
3. A Kafka client session is established if the token is valid.

Client using long-lived access token, with broker performing fast local validation

1. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the long-lived access token.
2. The Kafka broker validates the access token locally using a JWT token signature check and local token introspection.

WARNING

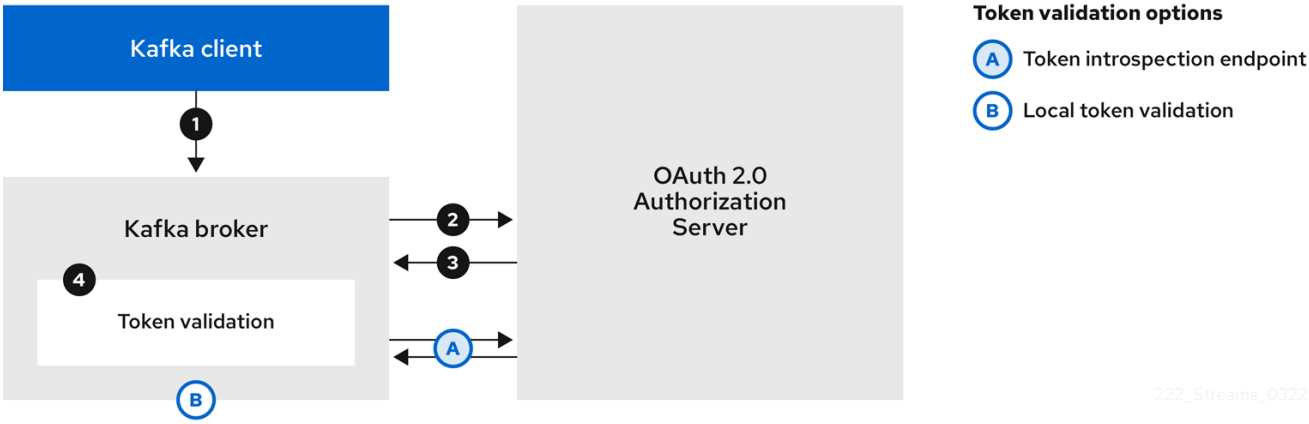
Fast local JWT token signature validation is suitable only for short-lived tokens as there is no check with the authorization server if a token has been revoked. Token expiration is written into the token, but revocation can happen at any time, so cannot be accounted for without contacting the authorization server. Any issued token would be considered valid until it expires.

8.4.5.2. Example client authentication flows using the SASL PLAIN mechanism

You can use the following communication flows for Kafka authentication using the OAuth PLAIN mechanism.

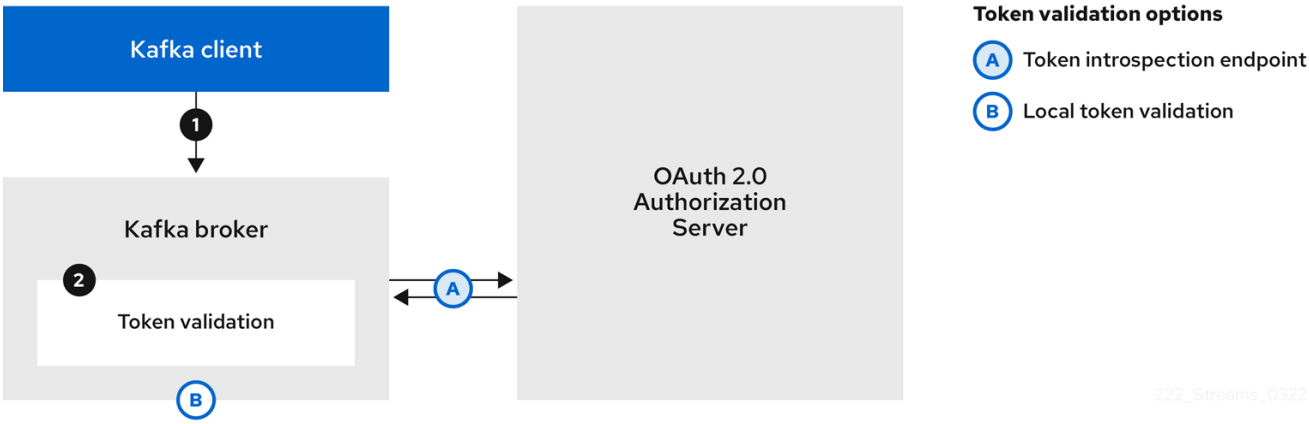
- Client using a client ID and secret, with the broker obtaining the access token for the client
- Client using a long-lived access token without a client ID and secret

Client using a client ID and secret, with the broker obtaining the access token for the client



1. The Kafka client passes a **clientId** as a username and a **secret** as a password.
2. The Kafka broker uses a token endpoint to pass the **clientId** and **secret** to the authorization server.
3. The authorization server returns a fresh access token or an error if the client credentials are not valid.
4. The Kafka broker validates the token in one of the following ways:
    a. If a token introspection endpoint is specified, the Kafka broker validates the access token by calling the endpoint on the authorization server. A session is established if the token validation is successful.
    b. If local token introspection is used, a request is not made to the authorization server. The Kafka broker validates the access token locally using a JWT token signature check.

Client using a long-lived access token without a client ID and secret



1. The Kafka client passes a username and password. The password provides the value of an access token that was obtained manually and configured before running the client.
2. The password is passed with or without an **$accessToken:** string prefix depending on whether or not the Kafka broker listener is configured with a token endpoint for authentication.

a. If the token endpoint is configured, the password should be prefixed by **$accessToken:** to let the broker know that the password parameter contains an access token rather than a client secret. The Kafka broker interprets the username as the account username.

b. If the token endpoint is not configured on the Kafka broker listener (enforcing a **no-client-credentials mode**), the password should provide the access token without the prefix. The Kafka broker interprets the username as the account username. In this mode, the client doesn't use a client ID and secret, and the **password** parameter is always interpreted as a raw access token.

3. The Kafka broker validates the token in one of the following ways:

a. If a token introspection endpoint is specified, the Kafka broker validates the access token by calling the endpoint on the authorization server. A session is established if token validation is successful.

b. If local token introspection is used, there is no request made to the authorization server. Kafka broker validates the access token locally using a JWT token signature check.

8.4.6. Configuring OAuth 2.0 authentication

OAuth 2.0 is used for interaction between Kafka clients and AMQ Streams components.

In order to use OAuth 2.0 for AMQ Streams, you must:

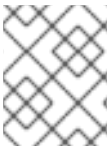
1. Deploy an authorization server and configure the deployment to integrate with AMQ Streams
2. Deploy or update the Kafka cluster with Kafka broker listeners configured to use OAuth 2.0
3. Update your Java-based Kafka clients to use OAuth 2.0
4. Update Kafka component clients to use OAuth 2.0

8.4.6.1. Configuring an OAuth 2.0 authorization server

This procedure describes in general what you need to do to configure an authorization server for integration with AMQ Streams.

These instructions are not product specific.

The steps are dependent on the chosen authorization server. Consult the product documentation for the authorization server for information on how to set up OAuth 2.0 access.



NOTE

If you already have an authorization server deployed, you can skip the deployment step and use your current deployment.

Procedure

1. Deploy the authorization server to your cluster.
2. Access the CLI or admin console for the authorization server to configure OAuth 2.0 for AMQ Streams.
   Now prepare the authorization server to work with AMQ Streams.

3. Configure a **kafka-broker** client.
4. Configure clients for each Kafka client component of your application.

What to do next

After deploying and configuring the authorization server, configure the Kafka brokers to use OAuth 2.0 .

8.4.6.2. Configuring OAuth 2.0 support for Kafka brokers

This procedure describes how to configure Kafka brokers so that the broker listeners are enabled to use OAuth 2.0 authentication using an authorization server.

We advise use of OAuth 2.0 over an encrypted interface through through a listener with **tls: true**. Plain listeners are not recommended.

If the authorization server is using certificates signed by the trusted CA and matching the OAuth 2.0 server hostname, TLS connection works using the default settings. Otherwise, you may need to configure the truststore with proper certificates or disable the certificate hostname validation.

When configuring the Kafka broker you have two options for the mechanism used to validate the access token during OAuth 2.0 authentication of the newly connected Kafka client:

- Configuring fast local JWT token validation
- Configuring token validation using an introspection endpoint

Before you start

For more information on the configuration of OAuth 2.0 authentication for Kafka broker listeners, see:

- **KafkaListenerAuthenticationOAuth** schema reference
- OAuth 2.0 authentication mechanisms

Prerequisites

- AMQ Streams and Kafka are running
- An OAuth 2.0 authorization server is deployed

Procedure

1. Update the Kafka broker configuration (**Kafka.spec.kafka**) of your **Kafka** resource in an editor.

```
oc edit kafka my-cluster
```

2. Configure the Kafka broker **listeners** configuration.
The configuration for each type of listener does not have to be the same, as they are independent.

   The examples here show the configuration options as configured for external listeners.

   Example 1: Configuring fast local JWT token validation

```
#...
```

```
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth (1)
    validIssuerUri: <https://<auth-server-address>/auth/realms/external> (2)
    jwksEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-
connect/certs> (3)
    userNameClaim: preferred_username (4)
    maxSecondsWithoutReauthentication: 3600 (5)
    tlsTrustedCertificates: (6)
    - secretName: oauth-server-cert
      certificate: ca.crt
    disableTlsHostnameVerification: true (7)
    jwksExpirySeconds: 360 (8)
    jwksRefreshSeconds: 300 (9)
    jwksMinRefreshPauseSeconds: 1 (10)
```

(1) Listener type set to **oauth**.

(2) URI of the token issuer used for authentication.

(3) URI of the JWKS certificate endpoint used for local JWT validation.

(4) The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The **userNameClaim** value will depend on the authentication flow and the authorization server used.

(5) (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.

(6) (Optional) Trusted certificates for TLS connection to the authorization server.

(7) (Optional) Disable TLS hostname verification. Default is **false**.

(8) The duration the JWKS certificates are considered valid before they expire. Default is **360** seconds. If you specify a longer time, consider the risk of allowing access to revoked certificates.

(9) The period between refreshes of JWKS certificates. The interval must be at least 60 seconds shorter than the expiry interval. Default is **300** seconds.

(10) The minimum pause in seconds between consecutive attempts to refresh JWKS public keys. When an unknown signing key is encountered, the JWKS keys refresh is scheduled outside the regular periodic schedule with at least the specified pause since the last refresh attempt. The refreshing of keys follows the rule of exponential backoff, retrying on unsuccessful refreshes with ever increasing pause, until it reaches **jwksRefreshSeconds**. The default value is 1.

Example 2: Configuring token validation using an introspection endpoint

```
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth
    validIssuerUri: <https://<auth-server-address>/auth/realms/external>
    introspectionEndpointUri: <https://<auth-server-
address>/auth/realms/external/protocol/openid-connect/token/introspect> (1)
    clientId: kafka-broker (2)
    clientSecret: (3)
      secretName: my-cluster-oauth
      key: clientSecret
    userNameClaim: preferred_username (4)
    maxSecondsWithoutReauthentication: 3600 (5)
```

1. URI of the token introspection endpoint.
2. Client ID to identify the client.
3. Client Secret and client ID is used for authentication.
4. The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The **userNameClaim** value will depend on the authorization server used.
5. (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.

Depending on how you apply OAuth 2.0 authentication, and the type of authorization server, there are additional (optional) configuration settings you can use:

```
  # ...
  authentication:
    type: oauth
    # ...
    checkIssuer: false (1)
    checkAudience: true (2)
    fallbackUserNameClaim: client_id (3)
    fallbackUserNamePrefix: client-account- (4)
    validTokenType: bearer (5)
    userInfoEndpointUri: https://OAUTH-SERVER-
ADDRESS/auth/realms/external/protocol/openid-connect/userinfo (6)
    enableOauthBearer: false (7)
    enablePlain: true (8)
    tokenEndpointUri: https://OAUTH-SERVER-
ADDRESS/auth/realms/external/protocol/openid-connect/token (9)
    customClaimCheck: "@.custom == 'custom-value'" (10)
    clientAudience: AUDIENCE (11)
```

```
clientScope: SCOPE 12
connectTimeoutSeconds: 60 13
readTimeoutSeconds: 60 14
httpRetries: 2 15
httpRetryPauseMs: 300 16
groupsClaim: "$.groups" 17
groupsClaimDelimiter: "," 18
```

1. If your authorization server does not provide an **iss** claim, it is not possible to perform an issuer check. In this situation, set **checkIssuer** to **false** and do not specify a **validIssuerUri**. Default is **true**.

2. If your authorization server provides an **aud** (audience) claim, and you want to enforce an audience check, set **checkAudience** to **true**. Audience checks identify the intended recipients of tokens. As a result, the Kafka broker will reject tokens that do not have its **clientId** in their **aud** claim. Default is **false**.

3. An authorization server may not provide a single attribute to identify both regular users and clients. When a client authenticates in its own name, the server might provide a *client ID*. When a user authenticates using a username and password, to obtain a refresh token or an access token, the server might provide a *username* attribute in addition to a client ID. Use this fallback option to specify the username claim (attribute) to use if a primary user ID attribute is not available.

4. In situations where **fallbackUserNameClaim** is applicable, it may also be necessary to prevent name collisions between the values of the username claim, and those of the fallback username claim. Consider a situation where a client called **producer** exists, but also a regular user called **producer** exists. In order to differentiate between the two, you can use this property to add a prefix to the user ID of the client.

5. (Only applicable when using **introspectionEndpointUri**) Depending on the authorization server you are using, the introspection endpoint may or may not return the *token type* attribute, or it may contain different values. You can specify a valid token type value that the response from the introspection endpoint has to contain.

6. (Only applicable when using **introspectionEndpointUri**) The authorization server may be configured or implemented in such a way to not provide any identifiable information in an Introspection Endpoint response. In order to obtain the user ID, you can configure the URI of the **userinfo** endpoint as a fallback. The **userNameClaim**, **fallbackUserNameClaim**, and **fallbackUserNamePrefix** settings are applied to the response of **userinfo** endpoint.

7. Set this to **false** to disable the OAUTHBEARER mechanism on the listener. At least one of PLAIN or OAUTHBEARER has to be enabled. Default is **true**.

8. Set to **true** to enable PLAIN authentication on the listener, which is supported for clients on all platforms.

9. Additional configuration for the PLAIN mechanism. If specified, clients can authenticate over PLAIN by passing an access token as the **password** using an **$accessToken:** prefix. For production, always use **https://** urls.

10. Additional custom rules can be imposed on the JWT access token during validation by setting this to a JsonPath filter query. If the access token does not contain the necessary data, it is rejected. When using the **introspectionEndpointUri**, the custom check is applied to the introspection endpoint response JSON.

11 An **audience** parameter passed to the token endpoint. An *audience* is used when obtaining an access token for inter-broker authentication. It is also used in the name of a client for

12 A **scope** parameter passed to the token endpoint. A *scope* is used when obtaining an access token for inter-broker authentication. It is also used in the name of a client for OAuth 2.0 over PLAIN client authentication using a **clientId** and **secret**. This only affects the ability to obtain the token, and the content of the token, depending on the authorization server. It does not affect token validation rules by the listener.

13 The connect timeout in seconds when connecting to the authorization server. The default value is 60.

14 The read timeout in seconds when connecting to the authorization server. The default value is 60.

15 The maximum number of times to retry a failed HTTP request to the authorization server. The default value is **0**, meaning that no retries are performed. To use this option effectively, consider reducing the timeout times for the **connectTimeoutSeconds** and **readTimeoutSeconds** options. However, note that retries may prevent the current worker thread from being available to other requests, and if too many requests stall, it could make the Kafka broker unresponsive.

16 The time to wait before attempting another retry of a failed HTTP request to the authorization server. By default, this time is set to zero, meaning that no pause is applied. This is because many issues that cause failed requests are per-request network glitches or proxy issues that can be resolved quickly. However, if your authorization server is under stress or experiencing high traffic, you may want to set this option to a value of 100 ms or more to reduce the load on the server and increase the likelihood of successful retries.

17 A JsonPath query that is used to extract groups information from either the JWT token or the introspection endpoint response. This option is not set by default. By configuring this option, a custom authorizer can make authorization decisions based on user groups.

18 A delimiter used to parse groups information when it is returned as a single delimited string. The default value is ',' (comma).

3. Save and exit the editor, then wait for rolling updates to complete.
4. Check the update in the logs or by watching the pod state transitions:

```
oc logs -f ${POD_NAME} -c ${CONTAINER_NAME}
oc get pod -w
```

The rolling update configures the brokers to use OAuth 2.0 authentication.

What to do next

- Configure your Kafka clients to use OAuth 2.0

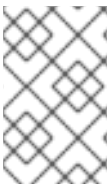
8.4.6.3. Configuring Kafka Java clients to use OAuth 2.0

Configure Kafka producer and consumer APIs to use OAuth 2.0 for interaction with Kafka brokers. Add a callback plugin to your client **pom.xml** file, then configure your client for OAuth 2.0.

Specify the following in your client configuration:

- A SASL (Simple Authentication and Security Layer) security protocol:
  - **SASL_SSL** for authentication over TLS encrypted connections
  - **SASL_PLAINTEXT** for authentication over unencrypted connections
    Use **SASL_SSL** for production and **SASL_PLAINTEXT** for local development only. When using **SASL_SSL**, additional **ssl.truststore** configuration is needed. The truststore configuration is required for secure connection (**https://**) to the OAuth 2.0 authorization server. To verify the OAuth 2.0 authorization server, add the CA certificate for the authorization server to the truststore in your client configuration. You can configure a truststore in PEM or PKCS #12 format.
- A Kafka SASL mechanism:
  - **OAUTHBEARER** for credentials exchange using a bearer token
  - **PLAIN** to pass client credentials (clientId + secret) or an access token
- A JAAS (Java Authentication and Authorization Service) module that implements the SASL mechanism:
  - **org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule** implements the OAUTHBEARER mechanism
  - **org.apache.kafka.common.security.plain.PlainLoginModule** implements the PLAIN mechanism
- SASL authentication properties, which support the following authentication methods:
  - OAuth 2.0 client credentials
  - OAuth 2.0 password grant (deprecated)
  - Access token
  - Refresh token

Add the SASL authentication properties as JAAS configuration (**sasl.jaas.config**). How you configure the authentication properties depends on the authentication method you are using to access the OAuth 2.0 authorization server. In this procedure, the properties are specified in a properties file, then loaded into the client configuration.



NOTE

You can also specify authentication properties as environment variables, or as Java system properties. For Java system properties, you can set them using **setProperty** and pass them on the command line using the **-D** option.

Prerequisites

- AMQ Streams and Kafka are running
- An OAuth 2.0 authorization server is deployed and configured for OAuth access to Kafka brokers
- Kafka brokers are configured for OAuth 2.0

Procedure

1. Add the client library with OAuth 2.0 support to the **pom.xml** file for the Kafka client:

```
<dependency>
 <groupId>io.strimzi</groupId>
 <artifactId>kafka-oauth-client</artifactId>
 <version>0.12.0.redhat-00006</version>
</dependency>
```

2. Configure the client properties by specifying the following configuration in a properties file:
   - The security protocol
   - The SASL mechanism
   - The JAAS module and authentication properties according to the method being used
     For example, we can add the following to a **client.properties** file:

     Client credentials mechanism properties

```
security.protocol=SASL_SSL (1)
sasl.mechanism=OAUTHBEARER (2)
ssl.truststore.location=/tmp/truststore.p12 (3)
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \ (4)
  oauth.client.id="<client_id>" \ (5)
  oauth.client.secret="<client_secret>" \ (6)
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \ (7)
  oauth.ssl.truststore.password="$STOREPASS" \ (8)
  oauth.ssl.truststore.type="PKCS12" \ (9)
  oauth.scope="<scope>" \ (10)
  oauth.audience="<audience>" ; (11)
```

(1) **SASL_SSL** security protocol for TLS-encrypted connections. Use **SASL_PLAINTEXT** over unencrypted connections for local development only.

(2) The SASL mechanism specified as **OAUTHBEARER** or **PLAIN**.

(3) The truststore configuration for secure access to the Kafka cluster.

(4) URI of the authorization server token endpoint.

(5) Client ID, which is the name used when creating the *client* in the authorization server.

(6) Client secret created when creating the *client* in the authorization server.

(7) The location contains the public key certificate (**truststore.p12**) for the authorization server.

(8) The password for accessing the truststore.

9 The truststore type.

10 (Optional) The **scope** for requesting the token from the token endpoint. An authorization server may require a client to specify the scope.

11 (Optional) The **audience** for requesting the token from the token endpoint. An authorization server may require a client to specify the audience.

Password grants mechanism properties

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.password.grant.username="<username>" \ 3
  oauth.password.grant.password="<password>" \ 4
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.scope="<scope>" \
  oauth.audience="<audience>" ;
```

1 Client ID, which is the name used when creating the *client* in the authorization server.

2 (Optional) Client secret created when creating the *client* in the authorization server.

3 Username for password grant authentication. OAuth password grant configuration (username and password) uses the OAuth 2.0 password grant method. To use password grants, create a user account for a client on your authorization server with limited permissions. The account should act like a service account. Use in environments where user accounts are required for authentication, but consider using a refresh token first.

4 Password for password grant authentication.

NOTE

SASL PLAIN does not support passing a username and password (password grants) using the OAuth 2.0 password grant method.

Access token properties

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
```

```
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.access.token="<access_token>" ; 1
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
```

1 Long-lived access token for Kafka clients.

Refresh token properties

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.refresh.token="<refresh_token>" ; 3
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
```

1 Client ID, which is the name used when creating the *client* in the authorization server.

2 (Optional) Client secret created when creating the *client* in the authorization server.

3 Long-lived refresh token for Kafka clients.

3. Input the client properties for OAUTH 2.0 authentication into the Java client code.

Example showing input of client properties

```
Properties props = new Properties();
try (FileReader reader = new FileReader("client.properties", StandardCharsets.UTF_8)) {
  props.load(reader);
}
```

4. Verify that the Kafka client can access the Kafka brokers.

8.4.6.4. Configuring OAuth 2.0 for Kafka components

This procedure describes how to configure Kafka components to use OAuth 2.0 authentication using an authorization server.

You can configure authentication for:

- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

In this scenario, the Kafka component and the authorization server are running in the same cluster.

Before you start

For more information on the configuration of OAuth 2.0 authentication for Kafka components, see the **KafkaClientAuthenticationOAuth** schema reference. The schema reference includes examples of configuration options.

Prerequisites

- AMQ Streams and Kafka are running
- An OAuth 2.0 authorization server is deployed and configured for OAuth access to Kafka brokers
- Kafka brokers are configured for OAuth 2.0

Procedure

1. Create a client secret and mount it to the component as an environment variable.
   For example, here we are creating a client **Secret** for the Kafka Bridge:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Secret
metadata:
 name: my-bridge-oauth
type: Opaque
data:
 clientSecret: MGQ1OTRmMzYtZTllZS00MDY2LWI5OGEtMTM5MzM2NjdlZjQw (1)
```

   (1) The **clientSecret** key must be in base64 format.

2. Create or edit the resource for the Kafka component so that OAuth 2.0 authentication is configured for the authentication property.
   For OAuth 2.0 authentication, you can use the following options:

   - Client ID and secret
   - Client ID and refresh token
   - Access token
   - Username and password
   - TLS

   For example, here OAuth 2.0 is assigned to the Kafka Bridge client using a client ID and secret, and TLS:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: oauth (1)
    tokenEndpointUri: https://<auth-server-address>/auth/realms/master/protocol/openid-
connect/token (2)
    clientId: kafka-bridge
    clientSecret:
      secretName: my-bridge-oauth
      key: clientSecret
    tlsTrustedCertificates: (3)
    - secretName: oauth-server-cert
      certificate: tls.crt
```

1. Authentication type set to **oauth**.
2. URI of the token endpoint for authentication.
3. Trusted certificates for TLS connection to the authorization server.

Depending on how you apply OAuth 2.0 authentication, and the type of authorization server, there are additional configuration options you can use:

```
# ...
spec:
  # ...
  authentication:
    # ...
    disableTlsHostnameVerification: true (1)
    checkAccessTokenType: false (2)
    accessTokenIsJwt: false (3)
    scope: any (4)
    audience: kafka (5)
    connectTimeoutSeconds: 60 (6)
    readTimeoutSeconds: 60 (7)
    httpRetries: 2 (8)
    httpRetryPauseMs: 300 (9)
```

1. (Optional) Disable TLS hostname verification. Default is **false**.
2. If the authorization server does not return a **typ** (type) claim inside the JWT token, you can apply **checkAccessTokenType: false** to skip the token type check. Default is **true**.
3. If you are using opaque tokens, you can apply **accessTokenIsJwt: false** so that access tokens are not treated as JWT tokens.
4. (Optional) The **scope** for requesting the token from the token endpoint. An authorization server may require a client to specify the scope. In this case it is **any**.

5 (Optional) The **audience** for requesting the token from the token endpoint. An authorization server may require a client to specify the audience. In this case it is **kafka**.

6 (Optional) The connect timeout in seconds when connecting to the authorization server. The default value is 60.

7 (Optional) The read timeout in seconds when connecting to the authorization server. The default value is 60.

8 (Optional) The maximum number of times to retry a failed HTTP request to the authorization server. The default value is **0**, meaning that no retries are performed. To use this option effectively, consider reducing the timeout times for the **connectTimeoutSeconds** and **readTimeoutSeconds** options. However, note that retries may prevent the current worker thread from being available to other requests, and if too many requests stall, it could make the Kafka broker unresponsive.

9 (Optional) The time to wait before attempting another retry of a failed HTTP request to the authorization server. By default, this time is set to zero, meaning that no pause is applied. This is because many issues that cause failed requests are per-request network glitches or proxy issues that can be resolved quickly. However, if your authorization server is under stress or experiencing high traffic, you may want to set this option to a value of 100 ms or more to reduce the load on the server and increase the likelihood of successful retries.

3. Apply the changes to the deployment of your Kafka resource.

```
oc apply -f your-file
```

4. Check the update in the logs or by watching the pod state transitions:

```
oc logs -f ${POD_NAME} -c ${CONTAINER_NAME}
oc get pod -w
```

The rolling updates configure the component for interaction with Kafka brokers using OAuth 2.0 authentication.

8.5. USING OAUTH 2.0 TOKEN-BASED AUTHORIZATION

If you are using OAuth 2.0 with Red Hat Single Sign-On for token-based authentication, you can also use Red Hat Single Sign-On to configure authorization rules to constrain client access to Kafka brokers. Authentication establishes the identity of a user. Authorization decides the level of access for that user.

AMQ Streams supports the use of OAuth 2.0 token-based authorization through Red Hat Single Sign-On Authorization Services, which allows you to manage security policies and permissions centrally.

Security policies and permissions defined in Red Hat Single Sign-On are used to grant access to resources on Kafka brokers. Users and clients are matched against policies that permit access to perform specific actions on Kafka brokers.

Kafka allows all users full access to brokers by default, and also provides the **AclAuthorizer** plugin to configure authorization based on Access Control Lists (ACLs).

ZooKeeper stores ACL rules that grant or deny access to resources based on *username*. However, OAuth 2.0 token-based authorization with Red Hat Single Sign-On offers far greater flexibility on how you wish to implement access control to Kafka brokers. In addition, you can configure your Kafka brokers

to use OAuth 2.0 authorization and ACLs.

Additional resources

- Using OAuth 2.0 token-based authentication
- Kafka Authorization
- Red Hat Single Sign-On documentation

8.5.1. OAuth 2.0 authorization mechanism

OAuth 2.0 authorization in AMQ Streams uses Red Hat Single Sign-On server Authorization Services REST endpoints to extend token-based authentication with Red Hat Single Sign-On by applying defined security policies on a particular user, and providing a list of permissions granted on different resources for that user. Policies use roles and groups to match permissions to users. OAuth 2.0 authorization enforces permissions locally based on the received list of grants for the user from Red Hat Single Sign-On Authorization Services.

8.5.1.1. Kafka broker custom authorizer

A Red Hat Single Sign-On *authorizer* (**KeycloakRBACAuthorizer**) is provided with AMQ Streams. To be able to use the Red Hat Single Sign-On REST endpoints for Authorization Services provided by Red Hat Single Sign-On, you configure a custom authorizer on the Kafka broker.

The authorizer fetches a list of granted permissions from the authorization server as needed, and enforces authorization locally on the Kafka Broker, making rapid authorization decisions for each client request.

8.5.2. Configuring OAuth 2.0 authorization support

This procedure describes how to configure Kafka brokers to use OAuth 2.0 authorization using Red Hat Single Sign-On Authorization Services.

Before you begin

Consider the access you require or want to limit for certain users. You can use a combination of Red Hat Single Sign-On *groups*, *roles*, *clients*, and *users* to configure access in Red Hat Single Sign-On.

Typically, groups are used to match users based on organizational departments or geographical locations. And roles are used to match users based on their function.

With Red Hat Single Sign-On, you can store users and groups in LDAP, whereas clients and roles cannot be stored this way. Storage and access to user data may be a factor in how you choose to configure authorization policies.

NOTE

Super users always have unconstrained access to a Kafka broker regardless of the authorization implemented on the Kafka broker.

Prerequisites

- AMQ Streams must be configured to use OAuth 2.0 with Red Hat Single Sign-On for token-based authentication. You use the same Red Hat Single Sign-On server endpoint when you set up authorization.
- OAuth 2.0 authentication must be configured with the **maxSecondsWithoutReauthentication** option to enable re-authentication.

Procedure

1. Access the Red Hat Single Sign-On Admin Console or use the Red Hat Single Sign-On Admin CLI to enable Authorization Services for the Kafka broker client you created when setting up OAuth 2.0 authentication.
2. Use Authorization Services to define resources, authorization scopes, policies, and permissions for the client.
3. Bind the permissions to users and clients by assigning them roles and groups.
4. Configure the Kafka brokers to use Red Hat Single Sign-On authorization by updating the Kafka broker configuration (**Kafka.spec.kafka**) of your **Kafka** resource in an editor.

```
oc edit kafka my-cluster
```

5. Configure the Kafka broker **kafka** configuration to use **keycloak** authorization, and to be able to access the authorization server and Authorization Services.
For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    authorization:
      type: keycloak (1)
      tokenEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-connect/token> (2)
      clientId: kafka (3)
      delegateToKafkaAcls: false (4)
      disableTlsHostnameVerification: false (5)
      superUsers: (6)
      - CN=fred
      - sam
      - CN=edward
      tlsTrustedCertificates: (7)
      - secretName: oauth-server-cert
        certificate: ca.crt
      grantsRefreshPeriodSeconds: 60 (8)
      grantsRefreshPoolSize: 5 (9)
      connectTimeoutSeconds: 60 (10)
      readTimeoutSeconds: 60 (11)
      httpRetries: 2 (12)
    #...
```

1. Type **keycloak** enables Red Hat Single Sign-On authorization.
2. URI of the Red Hat Single Sign-On token endpoint. For production, always use **https://** urls. When you configure token-based **oauth** authentication, you specify a **jwksEndpointUri** as the URI for local JWT validation. The hostname for the **tokenEndpointUri** URI must be the same.
3. The client ID of the OAuth 2.0 client definition in Red Hat Single Sign-On that has Authorization Services enabled. Typically, **kafka** is used as the ID.
4. (Optional) Delegate authorization to Kafka **AclAuthorizer** if access is denied by Red Hat Single Sign-On Authorization Services policies. Default is **false**.
5. (Optional) Disable TLS hostname verification. Default is **false**.
6. (Optional) Designated super users.
7. (Optional) Trusted certificates for TLS connection to the authorization server.
8. (Optional) The time between two consecutive grants refresh runs. That is the maximum time for active sessions to detect any permissions changes for the user on Red Hat Single Sign-On. The default value is 60.
9. (Optional) The number of threads to use to refresh (in parallel) the grants for the active sessions. The default value is 5.
10. (Optional) The connect timeout in seconds when connecting to the Red Hat Single Sign-On token endpoint. The default value is 60.
11. (Optional) The read timeout in seconds when connecting to the Red Hat Single Sign-On token endpoint. The default value is 60.
12. (Optional) The maximum number of times to retry (without pausing) a failed HTTP request to the authorization server. The default value is **0**, meaning that no retries are performed. To use this option effectively, consider reducing the timeout times for the **connectTimeoutSeconds** and **readTimeoutSeconds** options. However, note that retries may prevent the current worker thread from being available to other requests, and if too many requests stall, it could make the Kafka broker unresponsive.

6. Save and exit the editor, then wait for rolling updates to complete.
7. Check the update in the logs or by watching the pod state transitions:

```
oc logs -f ${POD_NAME} -c kafka
oc get pod -w
```

   The rolling update configures the brokers to use OAuth 2.0 authorization.

8. Verify the configured permissions by accessing Kafka brokers as clients or users with specific roles, making sure they have the necessary access, or do not have the access they are not supposed to have.

8.5.3. Managing policies and permissions in Red Hat Single Sign-On Authorization Services

This section describes the authorization models used by Red Hat Single Sign-On Authorization Services and Kafka, and defines the important concepts in each model.

To grant permissions to access Kafka, you can map Red Hat Single Sign-On Authorization Services objects to Kafka resources by creating an *OAuth client specification* in Red Hat Single Sign-On. Kafka permissions are granted to user accounts or service accounts using Red Hat Single Sign-On Authorization Services rules.

Examples are shown of the different user permissions required for common Kafka operations, such as creating and listing topics.

8.5.3.1. Kafka and Red Hat Single Sign-On authorization models overview

Kafka and Red Hat Single Sign-On Authorization Services use different authorization models.

Kafka authorization model
Kafka's authorization model uses *resource types*. When a Kafka client performs an action on a broker, the broker uses the configured **KeycloakRBACAuthorizer** to check the client's permissions, based on the action and resource type.

Kafka uses five resource types to control access: **Topic**, **Group**, **Cluster**, **TransactionalId**, and **DelegationToken**. Each resource type has a set of available permissions.

Topic

- **Create**
- **Write**
- **Read**
- **Delete**
- **Describe**
- **DescribeConfigs**
- **Alter**
- **AlterConfigs**

Group

- **Read**
- **Describe**
- **Delete**

Cluster

- **Create**
- **Describe**
- **Alter**

- **DescribeConfigs**
- **AlterConfigs**
- **IdempotentWrite**
- **ClusterAction**

TransactionalId

- **Describe**
- **Write**

DelegationToken

- **Describe**

Red Hat Single Sign-On Authorization Services model
The Red Hat Single Sign-On Authorization Services model has four concepts for defining and granting permissions: *resources*, *authorization scopes*, *policies*, and *permissions*.

Resources

A resource is a set of resource definitions that are used to match resources with permitted actions. A resource might be an individual topic, for example, or all topics with names starting with the same prefix. A resource definition is associated with a set of available authorization scopes, which represent a set of all actions available on the resource. Often, only a subset of these actions is actually permitted.

Authorization scopes

An authorization scope is a set of all the available actions on a specific resource definition. When you define a new resource, you add scopes from the set of all scopes.

Policies

A policy is an authorization rule that uses criteria to match against a list of accounts. Policies can match:

- *Service accounts* based on client ID or roles
- *User accounts* based on username, groups, or roles.

Permissions

A permission grants a subset of authorization scopes on a specific resource definition to a set of users.

Additional resources

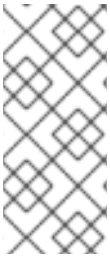
- Kafka authorization model

8.5.3.2. Map Red Hat Single Sign-On Authorization Services to the Kafka authorization model

The Kafka authorization model is used as a basis for defining the Red Hat Single Sign-On roles and resources that will control access to Kafka.

To grant Kafka permissions to user accounts or service accounts, you first create an *OAuth client*

specification in Red Hat Single Sign-On for the Kafka broker. You then specify Red Hat Single Sign-On Authorization Services rules on the client. Typically, the client id of the OAuth client that represents the broker is **kafka**. The example configuration files provided with AMQ Streams use **kafka** as the OAuth client id.



NOTE

If you have multiple Kafka clusters, you can use a single OAuth client (**kafka**) for all of them. This gives you a single, unified space in which to define and manage authorization rules. However, you can also use different OAuth client ids (for example, **my-cluster-kafka** or **cluster-dev-kafka**) and define authorization rules for each cluster within each client configuration.

The **kafka** client definition must have the **Authorization Enabled** option enabled in the Red Hat Single Sign-On Admin Console.

All permissions exist within the scope of the **kafka** client. If you have different Kafka clusters configured with different OAuth client IDs, they each need a separate set of permissions even though they're part of the same Red Hat Single Sign-On realm.

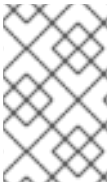
When the Kafka client uses OAUTHBEARER authentication, the Red Hat Single Sign-On authorizer (**KeycloakRBACAuthorizer**) uses the access token of the current session to retrieve a list of grants from the Red Hat Single Sign-On server. To retrieve the grants, the authorizer evaluates the Red Hat Single Sign-On Authorization Services policies and permissions.

Authorization scopes for Kafka permissions

An initial Red Hat Single Sign-On configuration usually involves uploading authorization scopes to create a list of all possible actions that can be performed on each Kafka resource type. This step is performed once only, before defining any permissions. You can add authorization scopes manually instead of uploading them.

Authorization scopes must contain all the possible Kafka permissions regardless of the resource type:

- **Create**
- **Write**
- **Read**
- **Delete**
- **Describe**
- **Alter**
- **DescribeConfig**
- **AlterConfig**
- **ClusterAction**
- **IdempotentWrite**



NOTE

If you're certain you won't need a permission (for example, **IdempotentWrite**), you can omit it from the list of authorization scopes. However, that permission won't be available to target on Kafka resources.

Resource patterns for permissions checks

Resource patterns are used for pattern matching against the targeted resources when performing permission checks. The general pattern format is ***RESOURCE-TYPE:PATTERN-NAME***.

The resource types mirror the Kafka authorization model. The pattern allows for two matching options:

- Exact matching (when the pattern does not end with *)
- Prefix matching (when the pattern ends with *)

Example patterns for resources

```
Topic:my-topic
Topic:orders-*
Group:orders-*
Cluster:*
```

Additionally, the general pattern format can be prefixed by **kafka-cluster:*CLUSTER-NAME*** followed by a comma, where *CLUSTER-NAME* refers to the **metadata.name** in the Kafka custom resource.

Example patterns for resources with cluster prefix

```
kafka-cluster:my-cluster,Topic:*
kafka-cluster:*,Group:b_*
```

When the **kafka-cluster** prefix is missing, it is assumed to be **kafka-cluster:***.

When defining a resource, you can associate it with a list of possible authorization scopes which are relevant to the resource. Set whatever actions make sense for the targeted resource type.

Though you may add any authorization scope to any resource, only the scopes supported by the resource type are considered for access control.

Policies for applying access permission

Policies are used to target permissions to one or more user accounts or service accounts. Targeting can refer to:

- Specific user or service accounts
- Realm roles or client roles
- User groups
- JavaScript rules to match a client IP address

A policy is given a unique name and can be reused to target multiple permissions to multiple resources.

Permissions to grant access

Use fine-grained permissions to pull together the policies, resources, and authorization scopes that grant access to users.

The name of each permission should clearly define which permissions it grants to which users. For example, **Dev Team B can read from topics starting with x**.

Additional resources

- For more information about how to configure permissions through Red Hat Single Sign-On Authorization Services, see Section 8.5.4, "Trying Red Hat Single Sign-On Authorization Services".

8.5.3.3. Example permissions required for Kafka operations

The following examples demonstrate the user permissions required for performing common operations on Kafka.

Create a topic

To create a topic, the **Create** permission is required for the specific topic, or for **Cluster:kafka-cluster**.

```
bin/kafka-topics.sh --create --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

List topics

If a user has the **Describe** permission on a specified topic, the topic is listed.

```
bin/kafka-topics.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Display topic details

To display a topic's details, **Describe** and **DescribeConfigs** permissions are required on the topic.

```
bin/kafka-topics.sh --describe --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Produce messages to a topic

To produce messages to a topic, **Describe** and **Write** permissions are required on the topic.

If the topic hasn't been created yet, and topic auto-creation is enabled, the permissions to create a topic are required.

```
bin/kafka-console-producer.sh  --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --producer.config=/tmp/config.properties
```

Consume messages from a topic

To consume messages from a topic, **Describe** and **Read** permissions are required on the topic. Consuming from the topic normally relies on storing the consumer offsets in a consumer group, which requires additional **Describe** and **Read** permissions on the consumer group.

Two **resources** are needed for matching. For example:

```
Topic:my-topic
Group:my-group-*
```

```
bin/kafka-console-consumer.sh --topic my-topic --group my-group-1 --from-beginning \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --consumer.config /tmp/config.properties
```

Produce messages to a topic using an idempotent producer

As well as the permissions for producing to a topic, an additional **IdempotentWrite** permission is required on the **Cluster:kafka-cluster** resource.

Two **resources** are needed for matching. For example:

```
Topic:my-topic
Cluster:kafka-cluster
```

```
bin/kafka-console-producer.sh  --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --producer.config=/tmp/config.properties --
producer-property enable.idempotence=true --request-required-acks -1
```

List consumer groups

When listing consumer groups, only the groups on which the user has the **Describe** permissions are returned. Alternatively, if the user has the **Describe** permission on the **Cluster:kafka-cluster**, all the consumer groups are returned.

```
bin/kafka-consumer-groups.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Display consumer group details

To display a consumer group's details, the **Describe** permission is required on the group and the topics associated with the group.

```
bin/kafka-consumer-groups.sh --describe --group my-group-1 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Change topic configuration

To change a topic's configuration, the **Describe** and **Alter** permissions are required on the topic.

```
bin/kafka-topics.sh --alter --topic my-topic --partitions 2 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Display Kafka broker configuration

In order to use **kafka-configs.sh** to get a broker's configuration, the **DescribeConfigs** permission is required on the **Cluster:kafka-cluster**.

```
bin/kafka-configs.sh --entity-type brokers --entity-name 0 --describe --all \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Change Kafka broker configuration

To change a Kafka broker's configuration, **DescribeConfigs** and **AlterConfigs** permissions are required on **Cluster:kafka-cluster**.

```
bin/kafka-configs --entity-type brokers --entity-name 0 --alter --add-config log.cleaner.threads=2 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Delete a topic

To delete a topic, the **Describe** and **Delete** permissions are required on the topic.

```
bin/kafka-topics.sh --delete --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Select a lead partition

To run leader selection for topic partitions, the **Alter** permission is required on the **Cluster:kafka-cluster**.

```
bin/kafka-leader-election.sh --topic my-topic --partition 0 --election-type PREFERRED  /
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --admin.config /tmp/config.properties
```

Reassign partitions

To generate a partition reassignment file, **Describe** permissions are required on the topics involved.

```
bin/kafka-reassign-partitions.sh --topics-to-move-json-file /tmp/topics-to-move.json --broker-list "0,1" --generate \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties > /tmp/partition-reassignment.json
```

To execute the partition reassignment, **Describe** and **Alter** permissions are required on **Cluster:kafka-cluster**. Also, **Describe** permissions are required on the topics involved.

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --execute \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties
```

To verify partition reassignment, **Describe**, and **AlterConfigs** permissions are required on **Cluster:kafka-cluster**, and on each of the topics involved.

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --verify \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties
```

8.5.4. Trying Red Hat Single Sign-On Authorization Services

This example explains how to use Red Hat Single Sign-On Authorization Services with **keycloak** authorization. Use Red Hat Single Sign-On Authorization Services to enforce access restrictions on Kafka clients. Red Hat Single Sign-On Authorization Services use authorization scopes, policies and permissions to define and apply access control to resources.

Red Hat Single Sign-On Authorization Services REST endpoints provide a list of granted permissions on resources for authenticated users. The list of grants (permissions) is fetched from the Red Hat Single Sign-On server as the first action after an authenticated session is established by the Kafka client. The list is refreshed in the background so that changes to the grants are detected. Grants are cached and enforced locally on the Kafka broker for each user session to provide fast authorization decisions.

AMQ Streams provides example configuration files. These include the following example files for setting up Red Hat Single Sign-On:

kafka-ephemeral-oauth-single-keycloak-authz.yaml

An example **Kafka** custom resource configured for OAuth 2.0 token-based authorization using Red Hat Single Sign-On. You can use the custom resource to deploy a Kafka cluster that uses **keycloak** authorization and token-based **oauth** authentication.

kafka-authz-realm.json

An example Red Hat Single Sign-On realm configured with sample groups, users, roles and clients. You can import the realm into a Red Hat Single Sign-On instance to set up fine-grained permissions to access Kafka.

If you want to try the example with Red Hat Single Sign-On, use these files to perform the tasks outlined in this section in the order shown.

1. Accessing the Red Hat Single Sign-On Admin Console
2. Deploying a Kafka cluster with Red Hat Single Sign-On authorization
3. Preparing TLS connectivity for a CLI Kafka client session
4. Checking authorized access to Kafka using a CLI Kafka client session

Authentication

When you configure token-based **oauth** authentication, you specify a **jwksEndpointUri** as the URI for local JWT validation. When you configure **keycloak** authorization, you specify a **tokenEndpointUri** as the URI of the Red Hat Single Sign-On token endpoint. The hostname for both URIs must be the same.

Targeted permissions with group or role policies

In Red Hat Single Sign-On, confidential clients with service accounts enabled can authenticate to the server in their own name using a client ID and a secret. This is convenient for microservices that typically act in their own name, and not as agents of a particular user (like a web site). Service accounts can have roles assigned like regular users. They cannot, however, have groups assigned. As a consequence, if you want to target permissions to microservices using service accounts, you cannot use group policies, and should instead use role policies. Conversely, if you want to limit certain permissions only to regular user accounts where authentication with a username and password is required, you can achieve that as a side effect of using the group policies rather than the role policies. This is what is used in this example for permissions that start with **ClusterManager**. Performing cluster management is usually done interactively using CLI tools. It makes sense to require the user to log in before using the resulting access token to authenticate to the Kafka broker. In this case, the access token represents the specific user, rather than the client application.

8.5.4.1. Accessing the Red Hat Single Sign-On Admin Console

Set up Red Hat Single Sign-On, then connect to its Admin Console and add the preconfigured realm. Use the example **kafka-authz-realm.json** file to import the realm. You can check the authorization rules defined for the realm in the Admin Console. The rules grant access to the resources on the Kafka cluster configured to use the example Red Hat Single Sign-On realm.

Prerequisites

- A running OpenShift cluster.

- The AMQ Streams **examples/security/keycloak-authorization/kafka-authz-realm.json** file that contains the preconfigured realm.

Procedure

1. Install the Red Hat Single Sign-On server using the Red Hat Single Sign-On Operator as described in Server Installation and Configuration in the Red Hat Single Sign-On documentation.
2. Wait until the Red Hat Single Sign-On instance is running.
3. Get the external hostname to be able to access the Admin Console.

```
NS=sso
oc get ingress keycloak -n $NS
```

   In this example, we assume the Red Hat Single Sign-On server is running in the **sso** namespace.

4. Get the password for the **admin** user.

```
oc get -n $NS pod keycloak-0 -o yaml | less
```

   The password is stored as a secret, so get the configuration YAML file for the Red Hat Single Sign-On instance to identify the name of the secret (**secretKeyRef.name**).

5. Use the name of the secret to obtain the clear text password.

```
SECRET_NAME=credential-keycloak
oc get -n $NS secret $SECRET_NAME -o yaml | grep PASSWORD | awk '{print $2}' |
base64 -D
```

   In this example, we assume the name of the secret is **credential-keycloak**.

6. Log in to the Admin Console with the username **admin** and the password you obtained.
   Use **https://*HOSTNAME*** to access the Kubernetes **Ingress**.

   You can now upload the example realm to Red Hat Single Sign-On using the Admin Console.

7. Click **Add Realm** to import the example realm.
8. Add the **examples/security/keycloak-authorization/kafka-authz-realm.json** file, and then click **Create**.
   You now have **kafka-authz** as your current realm in the Admin Console.

   The default view displays the **Master** realm.

9. In the Red Hat Single Sign-On Admin Console, go to **Clients** > **kafka** > **Authorization** > **Settings** and check that **Decision Strategy** is set to **Affirmative**.
   An affirmative policy means that at least one policy must be satisfied for a client to access the Kafka cluster.
10. In the Red Hat Single Sign-On Admin Console, go to **Groups**, **Users**, **Roles** and **Clients** to view the realm configuration.

    Groups

    Groups are used to create user groups and set user permissions. Groups are sets of users

with a name assigned. They are used to compartmentalize users into geographical, organizational or departmental units. Groups can be linked to an LDAP identity provider. You can make a user a member of a group through a custom LDAP server admin user interface, for example, to grant permissions on Kafka resources.

Users

Users are used to create users. For this example, **alice** and **bob** are defined. **alice** is a member of the **ClusterManager** group and **bob** is a member of **ClusterManager-my-cluster** group. Users can be stored in an LDAP identity provider.

Roles

Roles mark users or clients as having certain permissions. Roles are a concept analogous to groups. They are usually used to *tag* users with organizational roles and have the requisite permissions. Roles cannot be stored in an LDAP identity provider. If LDAP is a requirement, you can use groups instead, and add Red Hat Single Sign-On roles to the groups so that when users are assigned a group they also get a corresponding role.

Clients

Clients can have specific configurations. For this example, **kafka**, **kafka-cli**, **team-a-client**, and **team-b-client** clients are configured.

- The **kafka** client is used by Kafka brokers to perform the necessary OAuth 2.0 communication for access token validation. This client also contains the authorization services resource definitions, policies, and authorization scopes used to perform authorization on the Kafka brokers. The authorization configuration is defined in the **kafka** client from the **Authorization** tab, which becomes visible when **Authorization Enabled** is switched on from the **Settings** tab.

- The **kafka-cli** client is a public client that is used by the Kafka command line tools when authenticating with username and password to obtain an access token or a refresh token.

- The **team-a-client** and **team-b-client** clients are confidential clients representing services with partial access to certain Kafka topics.

11. In the Red Hat Single Sign-On Admin Console, go to **Authorization** > **Permissions** to see the granted permissions that use the resources and policies defined for the realm.
For example, the **kafka** client has the following permissions:

> Dev Team A can write to topics that start with x_ on any cluster
> Dev Team B can read from topics that start with x_ on any cluster
> Dev Team B can update consumer group offsets that start with x_ on any cluster
> ClusterManager of my-cluster Group has full access to cluster config on my-cluster
> ClusterManager of my-cluster Group has full access to consumer groups on my-cluster
> ClusterManager of my-cluster Group has full access to topics on my-cluster

Dev Team A

The Dev Team A realm role can write to topics that start with **x_** on any cluster. This combines a resource called **Topic:x_***, **Describe** and **Write** scopes, and the **Dev Team A** policy. The **Dev Team A** policy matches all users that have a realm role called **Dev Team A**.

Dev Team B

The Dev Team B realm role can read from topics that start with **x_** on any cluster. This combines **Topic:x_***, **Group:x_*** resources, **Describe** and **Read** scopes, and the **Dev Team B** policy. The **Dev Team B** policy matches all users that have a realm role called **Dev Team**

B. Matching users and clients have the ability to read from topics, and update the consumed offsets for topics and consumer groups that have names starting with **x_**.

8.5.4.2. Deploying a Kafka cluster with Red Hat Single Sign-On authorization

Deploy a Kafka cluster configured to connect to the Red Hat Single Sign-On server. Use the example **kafka-ephemeral-oauth-single-keycloak-authz.yaml** file to deploy the Kafka cluster as a **Kafka** custom resource. The example deploys a single-node Kafka cluster with **keycloak** authorization and **oauth** authentication.

Prerequisites

- The Red Hat Single Sign-On authorization server is deployed to your OpenShift cluster and loaded with the example realm.
- The Cluster Operator is deployed to your OpenShift cluster.
- The AMQ Streams **examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml** custom resource.

Procedure

1. Use the hostname of the Red Hat Single Sign-On instance you deployed to prepare a truststore certificate for Kafka brokers to communicate with the Red Hat Single Sign-On server.

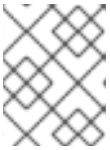
```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.pem
```

The certificate is required as Kubernetes **Ingress** is used to make a secure (HTTPS) connection.

Usually there is not one single certificate, but a certificate chain. You only have to provide the top-most issuer CA, which is listed last in the **/tmp/sso.pem** file. You can extract it manually or using the following commands:

Example command to extract the top CA certificate in a certificate chain

```
split -p "-----BEGIN CERTIFICATE-----" sso.pem sso-
for f in $(ls sso-*); do mv $f $f.pem; done
cp $(ls sso-* | sort -r | head -n 1) sso-ca.crt
```



NOTE

A trusted CA certificate is normally obtained from a trusted source, and not by using the **openssl** command.

2. Deploy the certificate to OpenShift as a secret.

```
oc create secret generic oauth-server-cert --from-file=/tmp/sso-ca.crt -n $NS
```

3. Set the hostname as an environment variable

```
SSO_HOST=SSO-HOSTNAME
```

4. Create and deploy the example Kafka cluster.

```
cat examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml | sed -E 's#\${SSO_HOST}'"#$SSO_HOST#" | oc create -n $NS -f -
```

8.5.4.3. Preparing TLS connectivity for a CLI Kafka client session

Create a new pod for an interactive CLI session. Set up a truststore with a Red Hat Single Sign-On certificate for TLS connectivity. The truststore is to connect to Red Hat Single Sign-On and the Kafka broker.

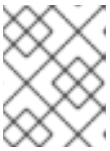
Prerequisites

- The Red Hat Single Sign-On authorization server is deployed to your OpenShift cluster and loaded with the example realm.
  In the Red Hat Single Sign-On Admin Console, check the roles assigned to the clients are displayed in **Clients** > **Service Account Roles**.
- The Kafka cluster configured to connect with Red Hat Single Sign-On is deployed to your OpenShift cluster.

Procedure

1. Run a new interactive pod container using the AMQ Streams Kafka image to connect to a running Kafka broker.

```
NS=sso
oc run -ti --restart=Never --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 kafka-cli -n $NS -- /bin/sh
```



NOTE

If **oc** times out waiting on the image download, subsequent attempts may result in an *AlreadyExists* error.

2. Attach to the pod container.

```
oc attach -ti kafka-cli -n $NS
```

3. Use the hostname of the Red Hat Single Sign-On instance to prepare a certificate for client connection using TLS.

```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.pem
```

Usually there is not one single certificate, but a certificate chain. You only have to provide the top-most issuer CA, which is listed last in the **/tmp/sso.pem** file. You can extract it manually or using the following command:

Example command to extract the top CA certificate in a certificate chain

```
split -p "-----BEGIN CERTIFICATE-----" sso.pem sso-
for f in $(ls sso-*); do mv $f $f.pem; done
cp $(ls sso-* | sort -r | head -n 1) sso-ca.crt
```

NOTE

A trusted CA certificate is normally obtained from a trusted source, and not by using the **openssl** command.

4. Create a truststore for TLS connection to the Kafka brokers.

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias sso -storepass $STOREPASS
-import -file /tmp/sso-ca.crt -noprompt
```

5. Use the Kafka bootstrap address as the hostname of the Kafka broker and the **tls** listener port (9093) to prepare a certificate for the Kafka broker.

```
KAFKA_HOST_PORT=my-cluster-kafka-bootstrap:9093
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $KAFKA_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/my-cluster-kafka.pem
```

The obtained **.pem** file is usually not one single certificate, but a certificate chain. You only have to provide the top-most issuer CA, which is listed last in the **/tmp/my-cluster-kafka.pem** file. You can extract it manually or using the following command:

Example command to extract the top CA certificate in a certificate chain

```
split -p "-----BEGIN CERTIFICATE-----" /tmp/my-cluster-kafka.pem kafka-
for f in $(ls kafka-*); do mv $f $f.pem; done
cp $(ls kafka-* | sort -r | head -n 1) my-cluster-kafka-ca.crt
```

NOTE

A trusted CA certificate is normally obtained from a trusted source, and not by using the **openssl** command. For this example we assume the client is running in a pod in the same namespace where the Kafka cluster was deployed. If the client is accessing the Kafka cluster from outside the OpenShift cluster, you would have to first determine the bootstrap address. In that case you can also get the cluster certificate directly from the OpenShift secret, and there is no need for **openssl**. For more information, see Chapter 7, *Setting up client access to a Kafka cluster* .

6. Add the certificate for the Kafka broker to the truststore.

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias my-cluster-kafka -storepass
$STOREPASS -import -file /tmp/my-cluster-kafka-ca.crt -noprompt
```

Keep the session open to check authorized access.

8.5.4.4. Checking authorized access to Kafka using a CLI Kafka client session

Check the authorization rules applied through the Red Hat Single Sign-On realm using an interactive CLI session. Apply the checks using Kafka's example producer and consumer clients to create topics with user and service accounts that have different levels of access.

Use the **team-a-client** and **team-b-client** clients to check the authorization rules. Use the **alice** admin user to perform additional administrative tasks on Kafka.

The AMQ Streams Kafka image used in this example contains Kafka producer and consumer binaries.

Prerequisites

- ZooKeeper and Kafka are running in the OpenShift cluster to be able to send and receive messages.
- The interactive CLI Kafka client session is started.
  Apache Kafka download.

Setting up client and admin user configuration

1. Prepare a Kafka configuration file with authentication properties for the **team-a-client** client.

```
SSO_HOST=SSO-HOSTNAME

cat > /tmp/team-a-client.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.client.id="team-a-client" \
  oauth.client.secret="team-a-client-secret" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-
connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHar
dler
EOF
```

The SASL OAUTHBEARER mechanism is used. This mechanism requires a client ID and client secret, which means the client first connects to the Red Hat Single Sign-On server to obtain an access token. The client then connects to the Kafka broker and uses the access token to authenticate.

2. Prepare a Kafka configuration file with authentication properties for the **team-b-client** client.

```
cat > /tmp/team-b-client.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.client.id="team-b-client" \
  oauth.client.secret="team-b-client-secret" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-
connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHar
dler
EOF
```

3. Authenticate admin user **alice** by using **curl** and performing a password grant authentication to obtain a refresh token.

```
USERNAME=alice
PASSWORD=alice-password

GRANT_RESPONSE=$(curl -X POST "https://$SSO_HOST/auth/realms/kafka-
authz/protocol/openid-connect/token" -H 'Content-Type: application/x-www-form-urlencoded'
-d
"grant_type=password&username=$USERNAME&password=$PASSWORD&client_id=kafka-
cli&scope=offline_access" -s -k)

REFRESH_TOKEN=$(echo $GRANT_RESPONSE | awk -F "refresh_token\":\"" '{printf $2}' |
awk -F "\"" '{printf $1}')
```

The refresh token is an offline token that is long-lived and does not expire.

4. Prepare a Kafka configuration file with authentication properties for the admin user **alice**.

```
cat > /tmp/alice.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.refresh.token="$REFRESH_TOKEN" \
  oauth.client.id="kafka-cli" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-
connect/token" ;
```

```
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHar
dler
EOF
```

The **kafka-cli** public client is used for the **oauth.client.id** in the **sasl.jaas.config**. Since it's a public client it does not require a secret. The client authenticates with the refresh token that was authenticated in the previous step. The refresh token requests an access token behind the scenes, which is then sent to the Kafka broker for authentication.

Producing messages with authorized access

Use the **team-a-client** configuration to check that you can produce messages to topics that start with **a_** or **x_**.

1. Write to topic **my-topic**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
my-topic \
  --producer.config=/tmp/team-a-client.properties
First message
```

This request returns a **Not authorized to access topics: [my-topic]** error.

team-a-client has a **Dev Team A** role that gives it permission to perform any supported actions on topics that start with **a_**, but can only write to topics that start with **x_**. The topic named **my-topic** matches neither of those rules.

2. Write to topic **a_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
a_messages \
  --producer.config /tmp/team-a-client.properties
First message
Second message
```

Messages are produced to Kafka successfully.

3. Press CTRL+C to exit the CLI application.

4. Check the Kafka container log for a debug log of **Authorization GRANTED** for the request.

```
oc logs my-cluster-kafka-0 -f -n $NS
```

Consuming messages with authorized access

Use the **team-a-client** configuration to consume messages from topic **a_messages**.

1. Fetch messages from topic **a_messages**.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties
```

The request returns an error because the **Dev Team A** role for **team-a-client** only has access to consumer groups that have names starting with **a_**.

2. Update the **team-a-client** properties to specify the custom consumer group it is permitted to use.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
a_consumer_group_1
```

The consumer receives all the messages from the **a_messages** topic.

Administering Kafka with authorized access

The **team-a-client** is an account without any cluster-level access, but it can be used with some administrative operations.

1. List topics.

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-a-client.properties --list
```

The **a_messages** topic is returned.

2. List consumer groups.

```
bin/kafka-consumer-groups.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --
command-config /tmp/team-a-client.properties --list
```

The **a_consumer_group_1** consumer group is returned.

Fetch details on the cluster configuration.

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-a-client.properties \
  --entity-type brokers --describe --entity-default
```

The request returns an error because the operation requires cluster level permissions that **team-a-client** does not have.

Using clients with different permissions

Use the **team-b-client** configuration to produce messages to topics that start with **b_**.

1. Write to topic **a_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
a_messages \
  --producer.config /tmp/team-b-client.properties
Message 1
```

This request returns a **Not authorized to access topics: [a_messages]** error.

2. Write to topic **b_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
b_messages \
  --producer.config /tmp/team-b-client.properties
```

```
Message 1
Message 2
Message 3
```

Messages are produced to Kafka successfully.

3. Write to topic **x_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --producer.config /tmp/team-b-client.properties
Message 1
```

A **Not authorized to access topics: [x_messages]** error is returned, The **team-b-client** can only read from topic **x_messages**.

4. Write to topic **x_messages** using **team-a-client**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --producer.config /tmp/team-a-client.properties
Message 1
```

This request returns a **Not authorized to access topics: [x_messages]** error. The **team-a-client** can write to the **x_messages** topic, but it does not have a permission to create a topic if it does not yet exist. Before **team-a-client** can write to the **x_messages** topic, an admin *power user* must create it with the correct configuration, such as the number of partitions and replicas.

Managing Kafka with an authorized admin user

Use admin user **alice** to manage Kafka. **alice** has full access to manage everything on any Kafka cluster.

1. Create the **x_messages** topic as **alice**.

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/alice.properties \
  --topic x_messages --create --replication-factor 1 --partitions 1
```

The topic is created successfully.

2. List all topics as **alice**.

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/alice.properties --list
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-a-client.properties --list
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-b-client.properties --list
```

Admin user **alice** can list all the topics, whereas **team-a-client** and **team-b-client** can only list the topics they have access to.

The **Dev Team A** and **Dev Team B** roles both have **Describe** permission on topics that start with **x_**, but they cannot see the other team's topics because they do not have **Describe** permissions on them.

3. Use the **team-a-client** to produce messages to the **x_messages** topic:

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --producer.config /tmp/team-a-client.properties
Message 1
Message 2
Message 3
```

   As **alice** created the **x_messages** topic, messages are produced to Kafka successfully.

4. Use the **team-b-client** to produce messages to the **x_messages** topic.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --producer.config /tmp/team-b-client.properties
Message 4
Message 5
```

   This request returns a **Not authorized to access topics: [x_messages]** error.

5. Use the **team-b-client** to consume messages from the **x_messages** topic:

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-b-client.properties --group
x_consumer_group_b
```

   The consumer receives all the messages from the **x_messages** topic.

6. Use the **team-a-client** to consume messages from the **x_messages** topic.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
x_consumer_group_a
```

   This request returns a **Not authorized to access topics: [x_messages]** error.

7. Use the **team-a-client** to consume messages from a consumer group that begins with **a_**.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
a_consumer_group_a
```

   This request returns a **Not authorized to access topics: [x_messages]** error.

   Dev Team A has no **Read** access on topics that start with a **x_**.

8. Use **alice** to produce messages to the **x_messages** topic.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/alice.properties
```

Messages are produced to Kafka successfully.

alice can read from or write to any topic.

9. Use **alice** to read the cluster configuration.

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/alice.properties \
  --entity-type brokers --describe --entity-default
```

The cluster configuration for this example is empty.

Additional resources

- Server Installation and Configuration
- Map Red Hat Single Sign-On Authorization Services to the Kafka authorization model

CHAPTER 9. MANAGING TLS CERTIFICATES

AMQ Streams supports TLS for encrypted communication between Kafka and AMQ Streams components.

Communication is always encrypted between the following components:

- Communication between Kafka and ZooKeeper
- Interbroker communication between Kafka brokers
- Internodal communication between ZooKeeper nodes
- AMQ Streams operator communication with Kafka brokers and ZooKeeper nodes

Communication between Kafka clients and Kafka brokers is encrypted according to how the cluster is configured. For the Kafka and AMQ Streams components, TLS certificates are also used for authentication.

The Cluster Operator automatically sets up and renews TLS certificates to enable encryption and authentication within your cluster. It also sets up other TLS certificates if you want to enable encryption or mTLS authentication between Kafka brokers and clients.

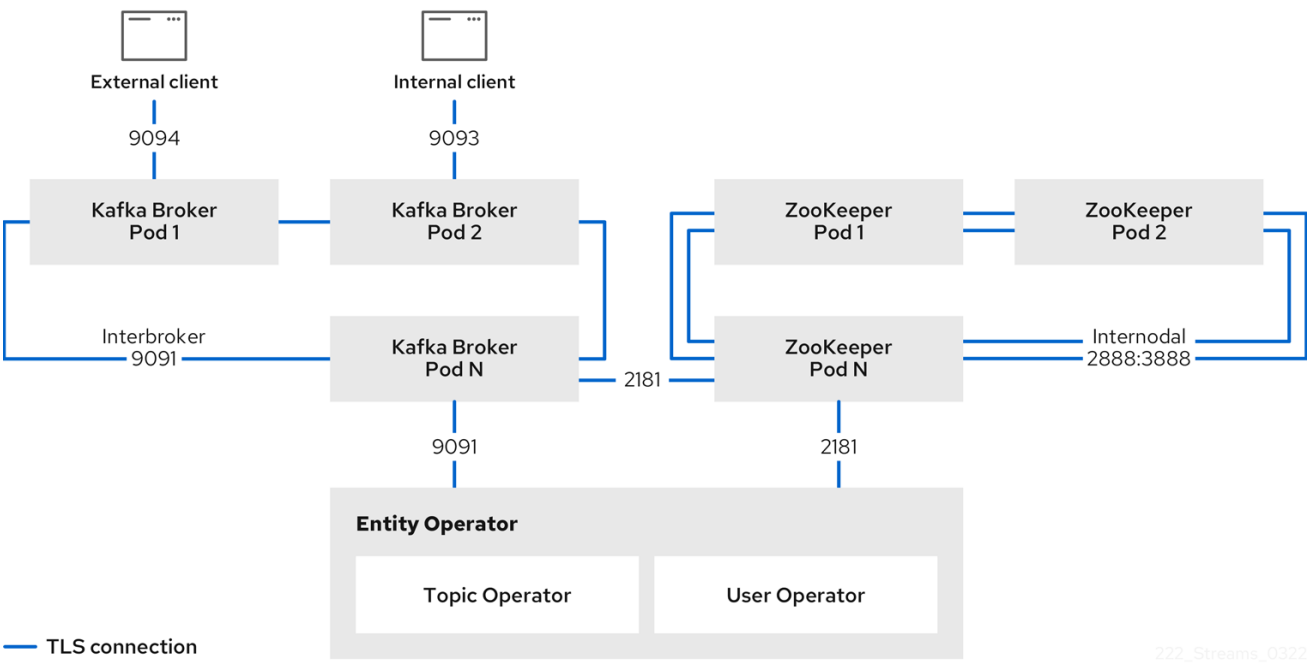
CA (certificate authority) certificates are generated by the Cluster Operator to verify the identities of components and clients. If you don't want to use the CAs generated by the Cluster Operator, you can install your own cluster and clients CA certificates .

You can also provide *Kafka listener certificates* for TLS listeners or external listeners that have TLS encryption enabled. Use Kafka listener certificates to incorporate the security infrastructure you already have in place.

NOTE

Any certificates you provide are not renewed by the Cluster Operator.

Figure 9.1. Example architecture of the communication secured by TLS



9.1. INTERNAL CLUSTER CA AND CLIENTS CA

To support encryption, each AMQ Streams component needs its own private keys and public key certificates. All component certificates are signed by an internal CA (certificate authority) called the *cluster CA*.

Similarly, each Kafka client application connecting to AMQ Streams using mTLS needs to use private keys and certificates. A second internal CA, named the *clients CA*, is used to sign certificates for the Kafka clients.

Both the cluster CA and clients CA have a self-signed public key certificate.

Kafka brokers are configured to trust certificates signed by either the cluster CA or clients CA. Components that clients do not need to connect to, such as ZooKeeper, only trust certificates signed by the cluster CA. Unless TLS encryption for external listeners is disabled, client applications must trust certificates signed by the cluster CA. This is also true for client applications that perform mTLS authentication.

By default, AMQ Streams automatically generates and renews CA certificates issued by the cluster CA or clients CA. You can configure the management of these CA certificates in the **Kafka.spec.clusterCa** and **Kafka.spec.clientsCa** objects.

You can replace the CA certificates for the cluster CA or clients CA with your own. For more information, see Section 9.7.1, “Installing your own CA certificates and private keys” . If you provide your own CA certificates, you must renew them before they expire.

9.2. SECRETS GENERATED BY THE OPERATORS

Secrets are created when custom resources are deployed, such as **Kafka** and **KafkaUser**. AMQ Streams uses these secrets to store private and public key certificates for Kafka clusters, clients, and users. The secrets are used for establishing TLS encrypted connections between Kafka brokers, and between brokers and clients. They are also used for mTLS authentication.

Cluster and clients secrets are always pairs: one contains the public key and one contains the private key.

Cluster secret

A cluster secret contains the *cluster CA* to sign Kafka broker certificates. Connecting clients use the certificate to establish a TLS encrypted connection with a Kafka cluster. The certificate verifies broker identity.

Client secret

A client secret contains the *clients CA* for a user to sign its own client certificate. This allows mutual authentication against the Kafka cluster. The broker validates a client's identity through the certificate.

User secret

A user secret contains a private key and certificate. The secret is created and signed by the clients CA when a new user is created. The key and certificate are used to authenticate and authorize the user when accessing the cluster.

9.2.1. TLS authentication using keys and certificates in PEM or PKCS #12 format

The secrets created by AMQ Streams provide private keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12 (Public-Key Cryptography Standards) formats. PEM and PKCS #12 are OpenSSL-generated key formats for TLS communications using the SSL protocol.

You can configure mutual TLS (mTLS) authentication that uses the credentials contained in the secrets generated for a Kafka cluster and user.

To set up mTLS, you must first do the following:

- Configure your Kafka cluster with a listener that uses mTLS
- Create a **KafkaUser** that provides client credentials for mTLs

When you deploy a Kafka cluster, a **<cluster_name>-cluster-ca-cert** secret is created with public key to verify the cluster. You use the public key to configure a truststore for the client.

When you create a **KafkaUser**, a **<kafka_user_name>** secret is created with the keys and certificates to verify the user (client). Use these credentials to configure a keystore for the client.

With the Kafka cluster and client set up to use mTLS, you extract credentials from the secrets and add them to your client configuration.

PEM keys and certificates

For PEM, you add the following to your client configuration:

Truststore

- **ca.crt** from the **<cluster_name>-cluster-ca-cert** secret, which is the CA certificate for the cluster.

Keystore

- **user.crt** from the **<kafka_user_name>** secret, which is the public certificate of the user.
- **user.key** from the **<kafka_user_name>** secret, which is the private key of the user.

PKCS #12 keys and certificates

For PKCS #12, you add the following to your client configuration:

Truststore

- **ca.p12** from the **<cluster_name>-cluster-ca-cert** secret, which is the CA certificate for the cluster.
- **ca.password** from the **<cluster_name>-cluster-ca-cert** secret, which is the password to access the public cluster CA certificate.

Keystore

- **user.p12** from the **<kafka_user_name>** secret, which is the public key certificate of the user.
- **user.password** from the **<kafka_user_name>** secret, which is the password to access the public key certificate of the Kafka user.

PKCS #12 is supported by Java, so you can add the values of the certificates directly to your Java client configuration. You can also reference the certificates from a secure storage location. With PEM files, you must add the certificates directly to the client configuration in single-line format. Choose a format that's suitable for establishing TLS connections between your Kafka cluster and client. Use PKCS #12 if you are unfamiliar with PEM.

NOTE

All keys are 2048 bits in size and, by default, are valid for 365 days from the initial generation. You can change the validity period .

9.2.2. Secrets generated by the Cluster Operator

The Cluster Operator generates the following certificates, which are saved as secrets in the OpenShift cluster. AMQ Streams uses these secrets by default.

The cluster CA and clients CA have separate secrets for the private key and public key.

***<cluster_name>*-cluster-ca**

Contains the private key of the cluster CA. AMQ Streams and Kafka components use the private key to sign server certificates.

***<cluster_name>*-cluster-ca-cert**

Contains the public key of the cluster CA. Kafka clients use the public key to verify the identity of the Kafka brokers they are connecting to with TLS server authentication.

***<cluster_name>*-clients-ca**

Contains the private key of the clients CA. Kafka clients use the private key to sign new user certificates for mTLS authentication when connecting to Kafka brokers.

***<cluster_name>*-clients-ca-cert**

Contains the public key of the clients CA. Kafka brokers use the public key to verify the identity of clients accessing the Kafka brokers when mTLS authentication is used.

Secrets for communication between AMQ Streams components contain a private key and a public key certificate signed by the cluster CA.

***<cluster_name>*-kafka-brokers**

Contains the private and public keys for Kafka brokers.

***<cluster_name>*-zookeeper-nodes**

Contains the private and public keys for ZooKeeper nodes.

***<cluster_name>*-cluster-operator-certs**

Contains the private and public keys for encrypting communication between the Cluster Operator and Kafka or ZooKeeper.

***<cluster_name>*-entity-topic-operator-certs**

Contains the private and public keys for encrypting communication between the Topic Operator and Kafka or ZooKeeper.

***<cluster_name>*-entity-user-operator-certs**

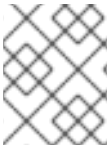
Contains the private and public keys for encrypting communication between the User Operator and Kafka or ZooKeeper.

***<cluster_name>*-cruise-control-certs**

Contains the private and public keys for encrypting communication between Cruise Control and Kafka or ZooKeeper.

***<cluster_name>*-kafka-exporter-certs**

Contains the private and public keys for encrypting communication between Kafka Exporter and Kafka or ZooKeeper.



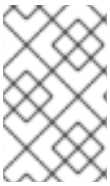
NOTE

You can provide your own server certificates and private keys to connect to Kafka brokers using *Kafka listener certificates* rather than certificates signed by the cluster CA.

9.2.3. Cluster CA secrets

Cluster CA secrets are managed by the Cluster Operator in a Kafka cluster.

Only the ***<cluster_name>*-cluster-ca-cert** secret is required by clients. All other cluster secrets are accessed by AMQ Streams components. You can enforce this using OpenShift role-based access controls, if necessary.



NOTE

The CA certificates in ***<cluster_name>*-cluster-ca-cert** must be trusted by Kafka client applications so that they validate the Kafka broker certificates when connecting to Kafka brokers over TLS.

Table 9.1. Fields in the *<cluster_name>*-cluster-ca secret

Field	Description
ca.key	The current private key for the cluster CA.

Table 9.2. Fields in the *<cluster_name>*-cluster-ca-cert secret

Field	Description
ca.p12	PKCS #12 store for storing certificates and keys.
ca.password	Password for protecting the PKCS #12 store.
ca.crt	The current certificate for the cluster CA.

Table 9.3. Fields in the ***<cluster_name>*-kafka-brokers** secret

Field	Description
***<cluster_name>*-kafka-*<num>*.p12**	PKCS #12 store for storing certificates and keys.
***<cluster_name>*-kafka-*<num>*.password**	Password for protecting the PKCS #12 store.
***<cluster_name>*-kafka-*<num>*.crt**	Certificate for a Kafka broker pod *<num>*. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**.
***<cluster_name>*-kafka-*<num>*.key**	Private key for a Kafka broker pod ***<num>***.

Table 9.4. Fields in the ***<cluster_name>*-zookeeper-nodes** secret

Field	Description
***<cluster_name>*-zookeeper-*<num>*.p12**	PKCS #12 store for storing certificates and keys.
***<cluster_name>*-zookeeper-*<num>*.password**	Password for protecting the PKCS #12 store.
***<cluster_name>*-zookeeper-*<num>*.crt**	Certificate for ZooKeeper node *<num>*. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**.
***<cluster_name>*-zookeeper-*<num>*.key**	Private key for ZooKeeper pod ***<num>***.

Table 9.5. Fields in the ***<cluster_name>*-cluster-operator-certs** secret

Field	Description
cluster-operator.p12	PKCS #12 store for storing certificates and keys.
cluster-operator.password	Password for protecting the PKCS #12 store.

Field	Description
cluster-operator.crt	Certificate for mTLS communication between the Cluster Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**.
cluster-operator.key	Private key for mTLS communication between the Cluster Operator and Kafka or ZooKeeper.

Table 9.6. Fields in the ***<cluster_name>*-entity-topic-operator-certs** secret

Field	Description
entity-operator.p12	PKCS #12 store for storing certificates and keys.
entity-operator.password	Password for protecting the PKCS #12 store.
entity-operator.crt	Certificate for mTLS communication between the Topic Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**.
entity-operator.key	Private key for mTLS communication between the Topic Operator and Kafka or ZooKeeper.

Table 9.7. Fields in the ***<cluster_name>*-entity-user-operator-certs** secret

Field	Description
entity-operator.p12	PKCS #12 store for storing certificates and keys.
entity-operator.password	Password for protecting the PKCS #12 store.
entity-operator.crt	Certificate for mTLS communication between the User Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**.
entity-operator.key	Private key for mTLS communication between the User Operator and Kafka or ZooKeeper.

Table 9.8. Fields in the ***<cluster_name>*-cruise-control-certs** secret

Field	Description
cruise-control.p12	PKCS #12 store for storing certificates and keys.
cruise-control.password	Password for protecting the PKCS #12 store.

Field	Description
cruise-control.crt	Certificate for mTLS communication between Cruise Control and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**.
cruise-control.key	Private key for mTLS communication between the Cruise Control and Kafka or ZooKeeper.

Table 9.9. Fields in the ***<cluster_name>*-kafka-exporter-certs** secret

Field	Description
kafka-exporter.p12	PKCS #12 store for storing certificates and keys.
kafka-exporter.password	Password for protecting the PKCS #12 store.
kafka-exporter.crt	Certificate for mTLS communication between Kafka Exporter and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**.
kafka-exporter.key	Private key for mTLS communication between the Kafka Exporter and Kafka or ZooKeeper.

9.2.4. Clients CA secrets

Clients CA secrets are managed by the Cluster Operator in a Kafka cluster.

The certificates in ***<cluster_name>*-clients-ca-cert** are those which the Kafka brokers trust.

The ***<cluster_name>*-clients-ca** secret is used to sign the certificates of client applications. This secret must be accessible to the AMQ Streams components and for administrative access if you are intending to issue application certificates without using the User Operator. You can enforce this using OpenShift role-based access controls, if necessary.

Table 9.10. Fields in the ***<cluster_name>*-clients-ca** secret

Field	Description
ca.key	The current private key for the clients CA.

Table 9.11. Fields in the ***<cluster_name>*-clients-ca-cert** secret

Field	Description
ca.p12	PKCS #12 store for storing certificates and keys.
ca.password	Password for protecting the PKCS #12 store.

Field	Description
ca.crt	The current certificate for the clients CA.

9.2.5. User secrets generated by the User Operator

User secrets are managed by the User Operator.

When a user is created using the User Operator, a secret is generated using the name of the user.

Table 9.12. Fields in the *user_name* secret

Secret name	Field within secret	Description
<user_name>	**user.p12**	PKCS #12 store for storing certificates and keys.
	user.password	Password for protecting the PKCS #12 store.
	user.crt	Certificate for the user, signed by the clients CA
	user.key	Private key for the user

9.2.6. Adding labels and annotations to cluster CA secrets

By configuring the **clusterCaCert** template property in the **Kafka** custom resource, you can add custom labels and annotations to the Cluster CA secrets created by the Cluster Operator. Labels and annotations are useful for identifying objects and adding contextual information. You configure template properties in AMQ Streams custom resources.

Example template customization to add labels and annotations to secrets

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      clusterCaCert:
        metadata:
          labels:
            label1: value1
            label2: value2
          annotations:
            annotation1: value1
            annotation2: value2
    # ...
```

9.2.7. Disabling ownerReference in the CA secrets

By default, the cluster and clients CA secrets are created with an **ownerReference** property that is set to the **Kafka** custom resource. This means that, when the **Kafka** custom resource is deleted, the CA secrets are also deleted (garbage collected) by OpenShift.

If you want to reuse the CA for a new cluster, you can disable the **ownerReference** by setting the **generateSecretOwnerReference** property for the cluster and clients CA secrets to **false** in the **Kafka** configuration. When the **ownerReference** is disabled, CA secrets are not deleted by OpenShift when the corresponding **Kafka** custom resource is deleted.

Example Kafka configuration with disabled ownerReference for cluster and clients CAs

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    generateSecretOwnerReference: false
  clientsCa:
    generateSecretOwnerReference: false
# ...
```

Additional resources

- **CertificateAuthority** schema reference

9.3. CERTIFICATE RENEWAL AND VALIDITY PERIODS

Cluster CA and clients CA certificates are only valid for a limited time period, known as the validity period. This is usually defined as a number of days since the certificate was generated.

For CA certificates automatically created by the Cluster Operator, you can configure the validity period of:

- Cluster CA certificates in **Kafka.spec.clusterCa.validityDays**
- Clients CA certificates in **Kafka.spec.clientsCa.validityDays**

The default validity period for both certificates is 365 days. Manually-installed CA certificates should have their own validity periods defined.

When a CA certificate expires, components and clients that still trust that certificate will not accept connections from peers whose certificates were signed by the CA private key. The components and clients need to trust the *new* CA certificate instead.

To allow the renewal of CA certificates without a loss of service, the Cluster Operator initiates certificate renewal before the old CA certificates expire.

You can configure the renewal period of the certificates created by the Cluster Operator:

- Cluster CA certificates in **Kafka.spec.clusterCa.renewalDays**
- Clients CA certificates in **Kafka.spec.clientsCa.renewalDays**

The default renewal period for both certificates is 30 days.

The renewal period is measured backwards, from the expiry date of the current certificate.

Validity period against renewal period

```
Not Before                          Not After
   |                                    |
   |<--------------- validityDays ---------------->|
                         <--- renewalDays --->|
```

To make a change to the validity and renewal periods after creating the Kafka cluster, you configure and apply the **Kafka** custom resource, and manually renew the CA certificates. If you do not manually renew the certificates, the new periods will be used the next time the certificate is renewed automatically.

Example Kafka configuration for certificate validity and renewal periods

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
  clientsCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
# ...
```

The behavior of the Cluster Operator during the renewal period depends on the settings for the **generateCertificateAuthority** certificate generation properties for the cluster CA and clients CA.

true

If the properties are set to **true**, a CA certificate is generated automatically by the Cluster Operator, and renewed automatically within the renewal period.

false

If the properties are set to **false**, a CA certificate is not generated by the Cluster Operator. Use this option if you are installing your own certificates.

9.3.1. Renewal process with automatically generated CA certificates

The Cluster Operator performs the following processes in this order when renewing CA certificates:

1. Generates a new CA certificate, but retains the existing key.
   The new certificate replaces the old one with the name **ca.crt** within the corresponding **Secret**.

2. Generates new client certificates (for ZooKeeper nodes, Kafka brokers, and the Entity Operator).
   This is not strictly necessary because the signing key has not changed, but it keeps the validity period of the client certificate in sync with the CA certificate.

3. Restarts ZooKeeper nodes so that they will trust the new CA certificate and use the new client certificates.

4. Restarts Kafka brokers so that they will trust the new CA certificate and use the new client certificates.

5. Restarts the Topic and User Operators so that they will trust the new CA certificate and use the new client certificates.
   User certificates are signed by the clients CA. User certificates generated by the User Operator are renewed when the clients CA is renewed.

9.3.2. Client certificate renewal

The Cluster Operator is not aware of the client applications using the Kafka cluster.

When connecting to the cluster, and to ensure they operate correctly, client applications must:

- Trust the cluster CA certificate published in the *<cluster>*-cluster-ca-cert Secret.

- Use the credentials published in their *<user-name>* Secret to connect to the cluster.
  The User Secret provides credentials in PEM and PKCS #12 format, or it can provide a password when using SCRAM-SHA authentication. The User Operator creates the user credentials when a user is created.

You must ensure clients continue to work after certificate renewal. The renewal process depends on how the clients are configured.

If you are provisioning client certificates and keys manually, you must generate new client certificates and ensure the new certificates are used by clients within the renewal period. Failure to do this by the end of the renewal period could result in client applications being unable to connect to the cluster.

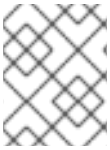
NOTE

For workloads running inside the same OpenShift cluster and namespace, Secrets can be mounted as a volume so the client Pods construct their keystores and truststores from the current state of the Secrets. For more details on this procedure, see Configuring internal clients to trust the cluster CA.

9.3.3. Manually renewing the CA certificates generated by the Cluster Operator

Cluster and clients CA certificates generated by the Cluster Operator auto-renew at the start of their respective certificate renewal periods. However, you can use the **strimzi.io/force-renew** annotation to manually renew one or both of these certificates before the certificate renewal period starts. You might do this for security reasons, or if you have changed the renewal or validity periods for the certificates .

A renewed certificate uses the same private key as the old certificate.



NOTE

If you are using your own CA certificates, the **force-renew** annotation cannot be used. Instead, follow the procedure for renewing your own CA certificates .

Prerequisites

- The Cluster Operator is running.

- A Kafka cluster in which CA certificates and private keys are installed.

Procedure

1. Apply the **strimzi.io/force-renew** annotation to the **Secret** that contains the CA certificate that you want to renew.

   Table 9.13. Annotation for the Secret that forces renewal of certificates

Certificate	Secret	Annotate command
Cluster CA	*KAFKA-CLUSTER-NAME*-cluster-ca-cert	**oc annotate secret *KAFKA-CLUSTER-NAME*-cluster-ca-cert strimzi.io/force-renew=true**
Clients CA	*KAFKA-CLUSTER-NAME*-clients-ca-cert	**oc annotate secret *KAFKA-CLUSTER-NAME*-clients-ca-cert strimzi.io/force-renew=true**

   At the next reconciliation the Cluster Operator will generate a new CA certificate for the **Secret** that you annotated. If maintenance time windows are configured, the Cluster Operator will generate the new CA certificate at the first reconciliation within the next maintenance time window.

   Client applications must reload the cluster and clients CA certificates that were renewed by the Cluster Operator.

2. Check the period the CA certificate is valid:
   For example, using an **openssl** command:

```
oc get secret CA-CERTIFICATE-SECRET -o 'jsonpath={.data.CA-CERTIFICATE}' | base64 -d | openssl x509 -subject -issuer -startdate -enddate -noout
```

   CA-CERTIFICATE-SECRET is the name of the **Secret**, which is ***KAFKA-CLUSTER-NAME*-cluster-ca-cert** for the cluster CA certificate and ***KAFKA-CLUSTER-NAME*-clients-ca-cert** for the clients CA certificate.

   CA-CERTIFICATE is the name of the CA certificate, such as **jsonpath={.data.ca\.crt}**.

   The command returns a **notBefore** and **notAfter** date, which is the validity period for the CA certificate.

   For example, for a cluster CA certificate:

```
subject=O = io.strimzi, CN = cluster-ca v0
issuer=O = io.strimzi, CN = cluster-ca v0
notBefore=Jun 30 09:43:54 2020 GMT
notAfter=Jun 30 09:43:54 2021 GMT
```

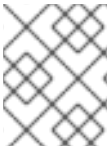
3. Delete old certificates from the Secret.
   When components are using the new certificates, older certificates might still be active. Delete the old certificates to remove any potential security risk.

Additional resources

- Section 9.2, "Secrets generated by the operators"
- Section 20.3, "Maintenance time windows for rolling updates"
- **CertificateAuthority** schema reference

9.3.4. Replacing private keys used by the CA certificates generated by the Cluster Operator

You can replace the private keys used by the cluster CA and clients CA certificates generated by the Cluster Operator. When a private key is replaced, the Cluster Operator generates a new CA certificate for the new private key.



NOTE

If you are using your own CA certificates, the **force-replace** annotation cannot be used. Instead, follow the procedure for renewing your own CA certificates.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which CA certificates and private keys are installed.

Procedure

- Apply the **strimzi.io/force-replace** annotation to the **Secret** that contains the private key that you want to renew.

  Table 9.14. Commands for replacing private keys

Private key for	Secret	Annotate command
Cluster CA	*CLUSTER-NAME*-cluster-ca	**oc annotate secret *CLUSTER-NAME*-cluster-ca strimzi.io/force-replace=true**
Clients CA	*CLUSTER-NAME*-clients-ca	**oc annotate secret *CLUSTER-NAME*-clients-ca strimzi.io/force-replace=true**

At the next reconciliation the Cluster Operator will:

- Generate a new private key for the **Secret** that you annotated

- Generate a new CA certificate

If maintenance time windows are configured, the Cluster Operator will generate the new private key and CA certificate at the first reconciliation within the next maintenance time window.

Client applications must reload the cluster and clients CA certificates that were renewed by the Cluster Operator.

Additional resources

- Section 9.2, "Secrets generated by the operators"
- Section 20.3, "Maintenance time windows for rolling updates"

9.4. TLS CONNECTIONS

9.4.1. ZooKeeper communication

Communication between the ZooKeeper nodes on all ports, as well as between clients and ZooKeeper, is encrypted using TLS.

Communication between Kafka brokers and ZooKeeper nodes is also encrypted.

9.4.2. Kafka inter-broker communication

Communication between Kafka brokers is always encrypted using TLS. The connections between the Kafka controller and brokers use an internal *control plane listener* on port 9090. Replication of data between brokers, as well as internal connections from AMQ Streams operators, Cruise Control, or the Kafka Exporter use the *replication listener* on port 9091. These internal listeners are not available to Kafka clients.

9.4.3. Topic and User Operators

All Operators use encryption for communication with both Kafka and ZooKeeper. In Topic and User Operators, a TLS sidecar is used when communicating with ZooKeeper.

9.4.4. Cruise Control

Cruise Control uses encryption for communication with both Kafka and ZooKeeper. A TLS sidecar is used when communicating with ZooKeeper.

9.4.5. Kafka Client connections

Encrypted or unencrypted communication between Kafka brokers and clients is configured using the **tls** property for **spec.kafka.listeners**.

9.5. CONFIGURING INTERNAL CLIENTS TO TRUST THE CLUSTER CA

This procedure describes how to configure a Kafka client that resides inside the OpenShift cluster — connecting to a TLS listener — to trust the cluster CA certificate.

The easiest way to achieve this for an internal client is to use a volume mount to access the **Secrets** containing the necessary certificates and keys.

Follow the steps to configure trust certificates that are signed by the cluster CA for Java-based Kafka Producer, Consumer, and Streams APIs.

Choose the steps to follow according to the certificate format of the cluster CA: PKCS #12 (**.p12**) or PEM (**.crt**).

The steps describe how to mount the Cluster Secret that verifies the identity of the Kafka cluster to the client pod.

Prerequisites

- The Cluster Operator must be running.
- There needs to be a **Kafka** resource within the OpenShift cluster.
- You need a Kafka client application inside the OpenShift cluster that will connect using TLS, and needs to trust the cluster CA certificate.
- The client application must be running in the same namespace as the **Kafka** resource.

Using PKCS #12 format (.p12)

1. Mount the cluster Secret as a volume when defining the client pod.
   For example:

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
  - name: client-name
    image: client-name
    volumeMounts:
    - name: secret-volume
      mountPath: /data/p12
    env:
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: my-password
  volumes:
  - name: secret-volume
    secret:
      secretName: my-cluster-cluster-ca-cert
```

   Here we're mounting the following:

   - The PKCS #12 file into an exact path, which can be configured
   - The password into an environment variable, where it can be used for Java configuration

2. Configure the Kafka client with the following properties:

   - A security protocol option:

- **security.protocol: SSL** when using TLS for encryption (with or without mTLS authentication).
- **security.protocol: SASL_SSL** when using SCRAM-SHA authentication over TLS.

- **ssl.truststore.location** with the truststore location where the certificates were imported.
- **ssl.truststore.password** with the password for accessing the truststore.
- **ssl.truststore.type=PKCS12** to identify the truststore type.

Using PEM format (.crt)

1. Mount the cluster Secret as a volume when defining the client pod.
   For example:

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
  - name: client-name
    image: client-name
    volumeMounts:
    - name: secret-volume
      mountPath: /data/crt
  volumes:
  - name: secret-volume
    secret:
      secretName: my-cluster-cluster-ca-cert
```

2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

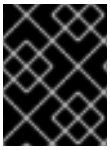
9.6. CONFIGURING EXTERNAL CLIENTS TO TRUST THE CLUSTER CA

This procedure describes how to configure a Kafka client that resides outside the OpenShift cluster – connecting to an **external** listener – to trust the cluster CA certificate. Follow this procedure when setting up the client and during the renewal period, when the old clients CA certificate is replaced.

Follow the steps to configure trust certificates that are signed by the cluster CA for Java-based Kafka Producer, Consumer, and Streams APIs.

Choose the steps to follow according to the certificate format of the cluster CA: PKCS #12 (**.p12**) or PEM (**.crt**).

The steps describe how to obtain the certificate from the Cluster Secret that verifies the identity of the Kafka cluster.



IMPORTANT

The ***<cluster_name>*-cluster-ca-cert** secret contains more than one CA certificate during the CA certificate renewal period. Clients must add *all* of them to their truststores.

Prerequisites

- The Cluster Operator must be running.
- There needs to be a **Kafka** resource within the OpenShift cluster.
- You need a Kafka client application outside the OpenShift cluster that will connect using TLS, and needs to trust the cluster CA certificate.

Using PKCS #12 format (.p12)

1. Extract the cluster CA certificate and password from the ***<cluster_name>*-cluster-ca-cert** Secret of the Kafka cluster.

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' | base64 -d > ca.p12
```

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.password}' | base64 -d > ca.password
```

   Replace *<cluster_name>* with the name of the Kafka cluster.

2. Configure the Kafka client with the following properties:
   - A security protocol option:
     - **security.protocol: SSL** when using TLS.
     - **security.protocol: SASL_SSL** when using SCRAM-SHA authentication over TLS.
   - **ssl.truststore.location** with the truststore location where the certificates were imported.
   - **ssl.truststore.password** with the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.
   - **ssl.truststore.type=PKCS12** to identify the truststore type.

Using PEM format (.crt)

1. Extract the cluster CA certificate from the ***<cluster_name>*-cluster-ca-cert** secret of the Kafka cluster.

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

9.7. USING YOUR OWN CA CERTIFICATES AND PRIVATE KEYS

Install and use your own CA certificates and private keys instead of using the defaults generated by the Cluster Operator. You can replace the cluster and clients CA certificates and private keys.

You can switch to using your own CA certificates and private keys in the following ways:

- Install your own CA certificates and private keys before deploying your Kafka cluster

- Replace the default CA certificates and private keys with your own after deploying a Kafka cluster

The steps to replace the default CA certificates and private keys after deploying a Kafka cluster are the same as those used to renew your own CA certificates and private keys.

If you use your own certificates, they won't be renewed automatically. You need to renew the CA certificates and private keys before they expire.

Renewal options:

- Renew the CA certificates only
- Renew CA certificates and private keys (or replace the defaults)

9.7.1. Installing your own CA certificates and private keys

Install your own CA certificates and private keys instead of using the cluster and clients CA certificates and private keys generated by the Cluster Operator.

By default, AMQ Streams uses the following cluster CA and clients CA secrets, which are renewed automatically.

- Cluster CA secrets
  - ***<cluster_name>*-cluster-ca**
  - ***<cluster_name>*-cluster-ca-cert**
- Clients CA secrets
  - ***<cluster_name>*-clients-ca**
  - ***<cluster_name>*-clients-ca-cert**

To install your own certificates, use the same names.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster is not yet deployed.
  If you have already deployed a Kafka cluster, you can replace the default CA certificates with your own.
- Your own X.509 certificates and keys in PEM format for the cluster CA or clients CA.
  - If you want to use a cluster or clients CA which is not a Root CA, you have to include the whole chain in the certificate file. The chain should be in the following order:
    1. The cluster or clients CA
    2. One or more intermediate CAs
    3. The root CA

- All CAs in the chain should be configured using the X509v3 Basic Constraints extension. Basic Constraints limit the path length of a certificate chain.

- The OpenSSL TLS management tool for converting certificates.

Before you begin

The Cluster Operator generates keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12 (Public-Key Cryptography Standards) formats. You can add your own certificates in either format.

Some applications cannot use PEM certificates and support only PKCS #12 certificates. If you don't have a cluster certificate in PKCS #12 format, use the OpenSSL TLS management tool to generate one from your **ca.crt** file.

Example certificate generation command

```
openssl pkcs12 -export -in ca.crt -nokeys -out ca.p12 -password pass:<P12_password> -caname ca.crt
```

Replace *<P12_password>* with your own password.

Procedure

1. Create a new secret that contains the CA certificate.

   ### Client secret creation with a certificate in PEM format only

   ```
   oc create secret generic <cluster_name>-clients-ca-cert --from-file=ca.crt=ca.crt
   ```

   ### Cluster secret creation with certificates in PEM and PKCS #12 format

   ```
   oc create secret generic <cluster_name>-cluster-ca-cert \
     --from-file=ca.crt=ca.crt \
     --from-file=ca.p12=ca.p12 \
     --from-literal=ca.password=P12-PASSWORD
   ```

   Replace *<cluster_name>* with the name of your Kafka cluster.

2. Create a new secret that contains the private key.

   ```
   oc create secret generic CA-KEY-SECRET --from-file=ca.key=ca.key
   ```

3. Label the secrets.

   ```
   oc label secret CA-CERTIFICATE-SECRET strimzi.io/kind=Kafka
   strimzi.io/cluster=<cluster_name>
   ```

   ```
   oc label secret CA-KEY-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=<cluster_name>
   ```

   - Label **strimzi.io/kind=Kafka** identifies the Kafka custom resource.
   - Label **strimzi.io/cluster=*<cluster_name>*** identifies the Kafka cluster.

4. Annotate the secrets

```
oc annotate secret CA-CERTIFICATE-SECRET strimzi.io/ca-cert-generation=CA-CERTIFICATE-GENERATION
```

```
oc annotate secret CA-KEY-SECRET strimzi.io/ca-key-generation=CA-KEY-GENERATION
```

- Annotation **strimzi.io/ca-cert-generation=*CA-CERTIFICATE-GENERATION*** defines the generation of a new CA certificate.
- Annotation **strimzi.io/ca-key-generation=*CA-KEY-GENERATION*** defines the generation of a new CA key.
  Start from 0 (zero) as the incremental value (**strimzi.io/ca-cert-generation=0**) for your own CA certificate. Set a higher incremental value when you renew the certificates.

5. Create the **Kafka** resource for your cluster, configuring either the **Kafka.spec.clusterCa** or the **Kafka.spec.clientsCa** object to *not* use generated CAs.

Example fragment Kafka resource configuring the cluster CA to use certificates you supply for yourself

```
kind: Kafka
version: kafka.strimzi.io/v1beta2
spec:
  # ...
  clusterCa:
    generateCertificateAuthority: false
```

Additional resources

- Section 9.7.2, "Renewing your own CA certificates"
- Section 9.7.3, "Renewing or replacing CA certificates and private keys with your own"
- Section 8.3.4, "Providing your own Kafka listener certificates for TLS encryption"

9.7.2. Renewing your own CA certificates

If you are using your own CA certificates, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire.

Perform the steps in this procedure when you are renewing CA certificates and continuing with the same private key. If you are renewing your own CA certificates *and* private keys, see Section 9.7.3, "Renewing or replacing CA certificates and private keys with your own".

The procedure describes the renewal of CA certificates in PEM format.

Prerequisites

- The Cluster Operator is running.
- You have new cluster or clients X.509 certificates in PEM format.

Procedure

1. Update the **Secret** for the CA certificate.

Edit the existing secret to add the new CA certificate and update the certificate generation annotation value.

```
oc edit secret <ca_certificate_secret_name>
```

<ca_certificate_secret_name> is the name of the **Secret**, which is ***<kafka_cluster_name>*-cluster-ca-cert** for the cluster CA certificate and ***<kafka_cluster_name>*-clients-ca-cert** for the clients CA certificate.

The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named **my-cluster**.

Example secret configuration for a cluster CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... (1)
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" (2)
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

(1) Current base64-encoded CA certificate

(2) Current CA certificate generation annotation value

2. Encode your new CA certificate into base64.

```
cat <path_to_new_certificate> | base64
```

3. Update the CA certificate.
Copy the base64-encoded CA certificate from the previous step as the value for the **ca.crt** property under **data**.

4. Increase the value of the CA certificate generation annotation.
Update the **strimzi.io/ca-cert-generation** annotation with a higher incremental value. For example, change **strimzi.io/ca-cert-generation=0** to **strimzi.io/ca-cert-generation=1**. If the **Secret** is missing the annotation, the value is treated as **0**, so add the annotation with a value of **1**.

   When AMQ Streams generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The **strimzi.io/ca-cert-generation** has to be incremented on each CA certificate renewal.

5. Save the secret with the new CA certificate and certificate generation annotation value.

Example secret configuration updated with a new CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F... 1
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" 2
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

1. New base64-encoded CA certificate
2. New CA certificate generation annotation value

On the next reconciliation, the Cluster Operator performs a rolling update of ZooKeeper, Kafka, and other components to trust the new CA certificate.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

9.7.3. Renewing or replacing CA certificates and private keys with your own

If you are using your own CA certificates and private keys, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire. You can also use the same procedure to replace the CA certificates and private keys generated by the AMQ Streams operators with your own.

Perform the steps in this procedure when you are renewing or replacing CA certificates and private keys. If you are only renewing your own CA certificates, see Section 9.7.2, “Renewing your own CA certificates”.

The procedure describes the renewal of CA certificates and private keys in PEM format.

Before going through the following steps, make sure that the CN (Common Name) of the new CA certificate is different from the current one. For example, when the Cluster Operator renews certificates automatically it adds a *v<version_number>* suffix to identify a version. Do the same with your own CA certificate by adding a different suffix on each renewal. By using a different key to generate a new CA certificate, you retain the current CA certificate stored in the **Secret**.

Prerequisites

- The Cluster Operator is running.
- You have new cluster or clients X.509 certificates and keys in PEM format.

Procedure

1. Pause the reconciliation of the **Kafka** custom resource.

a. Annotate the custom resource in OpenShift, setting the **pause-reconciliation** annotation to **true**:

```
oc annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="true"
```

For example, for a **Kafka** custom resource named **my-cluster**:

```
oc annotate Kafka my-cluster strimzi.io/pause-reconciliation="true"
```

b. Check that the status conditions of the custom resource show a change to **ReconciliationPaused**:

```
oc describe Kafka <name_of_custom_resource>
```

The **type** condition changes to **ReconciliationPaused** at the **lastTransitionTime**.

2. Update the **Secret** for the CA certificate.

a. Edit the existing secret to add the new CA certificate and update the certificate generation annotation value.

```
oc edit secret <ca_certificate_secret_name>
```

<ca_certificate_secret_name> is the name of the **Secret**, which is ***KAFKA-CLUSTER-NAME*-cluster-ca-cert** for the cluster CA certificate and ***KAFKA-CLUSTER-NAME*-clients-ca-cert** for the clients CA certificate.

The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named **my-cluster**.

Example secret configuration for a cluster CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... (1)
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" (2)
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

(1) Current base64-encoded CA certificate

(2) Current CA certificate generation annotation value

b. Rename the current CA certificate to retain it.
Rename the current **ca.crt** property under **data** as **ca-*<date>*.crt**, where *<date>* is the certificate expiry date in the format *YEAR-MONTH-DAYTHOUR-MINUTE-SECONDZ*. For

example **ca-2022-01-26T17-32-00Z.crt:**. Leave the value for the property as it is to retain the current CA certificate.

c. Encode your new CA certificate into base64.

```
cat <path_to_new_certificate> | base64
```

d. Update the CA certificate.
Create a new **ca.crt** property under **data** and copy the base64-encoded CA certificate from the previous step as the value for **ca.crt** property.

e. Increase the value of the CA certificate generation annotation.
Update the **strimzi.io/ca-cert-generation** annotation with a higher incremental value. For example, change **strimzi.io/ca-cert-generation=0** to **strimzi.io/ca-cert-generation=1**. If the **Secret** is missing the annotation, the value is treated as **0**, so add the annotation with a value of **1**.

When AMQ Streams generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The **strimzi.io/ca-cert-generation** has to be incremented on each CA certificate renewal.

f. Save the secret with the new CA certificate and certificate generation annotation value.

Example secret configuration updated with a new CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F... 1
  ca-2022-01-26T17-32-00Z.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... 2
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" 3
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

1 New base64-encoded CA certificate

2 Old base64-encoded CA certificate

3 New CA certificate generation annotation value

3. Update the **Secret** for the CA key used to sign your new CA certificate.

a. Edit the existing secret to add the new CA key and update the key generation annotation value.

```
oc edit secret <ca_key_name>
```

<ca_key_name> is the name of CA key, which is ***<kafka_cluster_name>*-cluster-ca** for the cluster CA key and ***<kafka_cluster_name>*-clients-ca** for the clients CA key.

The following example shows a secret for a cluster CA key that's associated with a Kafka cluster named **my-cluster**.

Example secret configuration for a cluster CA key

```
apiVersion: v1
kind: Secret
data:
  ca.key: SA1cKF1GFDzOIiPOIUQBHDNFGDFS... (1)
metadata:
  annotations:
    strimzi.io/ca-key-generation: "0" (2)
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque
```

1. Current base64-encoded CA key
2. Current CA key generation annotation value

b. Encode the CA key into base64.

```
cat <path_to_new_key> | base64
```

c. Update the CA key.
Copy the base64-encoded CA key from the previous step as the value for the **ca.key** property under **data**.

d. Increase the value of the CA key generation annotation.
Update the **strimzi.io/ca-key-generation** annotation with a higher incremental value. For example, change **strimzi.io/ca-key-generation=0** to **strimzi.io/ca-key-generation=1**. If the **Secret** is missing the annotation, it is treated as **0**, so add the annotation with a value of **1**.

When AMQ Streams generates certificates, the key generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates together with a new CA key, set the annotation with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates and keys. The **strimzi.io/ca-key-generation** has to be incremented on each CA certificate renewal.

4. Save the secret with the new CA key and key generation annotation value.

Example secret configuration updated with a new CA key

```
apiVersion: v1
```

```
kind: Secret
data:
  ca.key: AB0cKF1GFDzOIiPOIUQWERZJQ0F... (1)
metadata:
  annotations:
    strimzi.io/ca-key-generation: "1" (2)
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque
```

1. New base64-encoded CA key
2. New CA key generation annotation value

5. Resume from the pause.
   To resume the **Kafka** custom resource reconciliation, set the **pause-reconciliation** annotation to **false**.

```
oc annotate --overwrite Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="false"
```

   You can also do the same by removing the **pause-reconciliation** annotation.

```
oc annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation-
```

On the next reconciliation, the Cluster Operator performs a rolling update of ZooKeeper, Kafka, and other components to trust the new CA certificate. When the rolling update is complete, the Cluster Operator will start a new one to generate new server certificates signed by the new CA key.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

CHAPTER 10. SCALING CLUSTERS BY ADDING OR REMOVING BROKERS

Scaling Kafka clusters by adding brokers can increase the performance and reliability of the cluster. Adding more brokers increases available resources, allowing the cluster to handle larger workloads and process more messages. It can also improve fault tolerance by providing more replicas and backups. Conversely, removing underutilized brokers can reduce resource consumption and improve efficiency. Scaling must be done carefully to avoid disruption or data loss. By redistributing partitions across all brokers in the cluster, the resource utilization of each broker is reduced, which can increase the overall throughput of the cluster.

NOTE

To increase the throughput of a Kafka topic, you can increase the number of partitions for that topic. This allows the load of the topic to be shared between different brokers in the cluster. However, if every broker is constrained by a specific resource (such as I/O), adding more partitions will not increase the throughput. In this case, you need to add more brokers to the cluster.

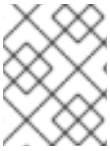
Adjusting the **Kafka.spec.kafka.replicas** configuration affects the number of brokers in the cluster that act as replicas. The actual replication factor for topics is determined by settings for the **default.replication.factor** and **min.insync.replicas**, and the number of available brokers. For example, a replication factor of 3 means that each partition of a topic is replicated across three brokers, ensuring fault tolerance in the event of a broker failure.

Example replica configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    # ...
  config: 1
      # ...
      default.replication.factor: 3
      min.insync.replicas: 2
  # ...
```

When you add add or remove brokers, Kafka does not automatically reassign partitions. The best way to do this is using Cruise Control. You can use Cruise Control's **add-brokers** and **remove-brokers** modes when scaling a cluster up or down.

- Use the **add-brokers** mode after scaling up a Kafka cluster to move partition replicas from existing brokers to the newly added brokers.
- Use the **remove-brokers** mode before scaling down a Kafka cluster to move partition replicas off the brokers that are going to be removed.



NOTE

When scaling down brokers, you cannot specify which specific pod to remove from the cluster. Instead, the broker removal process starts from the highest numbered pod.

CHAPTER 11. REBALANCING CLUSTERS USING CRUISE CONTROL

Cruise Control is an open source system that supports the following Kafka operations:

- Monitoring cluster workload
- Rebalancing a cluster based on predefined constraints

The operations help with running a more balanced Kafka cluster that uses broker pods more efficiently.

A typical cluster can become unevenly loaded over time. Partitions that handle large amounts of message traffic might not be evenly distributed across the available brokers. To rebalance the cluster, administrators must monitor the load on brokers and manually reassign busy partitions to brokers with spare capacity.

Cruise Control automates the cluster rebalancing process. It constructs a *workload model* of resource utilization for the cluster—based on CPU, disk, and network load—and generates optimization proposals (that you can approve or reject) for more balanced partition assignments. A set of configurable optimization goals is used to calculate these proposals.

You can generate optimization proposals in specific modes. The default **full** mode rebalances partitions across all brokers. You can also use the **add-brokers** and **remove-brokers** modes to accommodate changes when scaling a cluster up or down.

When you approve an optimization proposal, Cruise Control applies it to your Kafka cluster. You configure and generate optimization proposals using a **KafkaRebalance** resource. You can configure the resource using an annotation so that optimization proposals are approved automatically or manually.

NOTE

AMQ Streams provides example configuration files for Cruise Control .

11.1. CRUISE CONTROL COMPONENTS AND FEATURES

Cruise Control consists of four main components—the Load Monitor, the Analyzer, the Anomaly Detector, and the Executor—and a REST API for client interactions. AMQ Streams utilizes the REST API to support the following Cruise Control features:

- Generating optimization proposals from optimization goals.
- Rebalancing a Kafka cluster based on an optimization proposal.

Optimization goals

An optimization goal describes a specific objective to achieve from a rebalance. For example, a goal might be to distribute topic replicas across brokers more evenly. You can change what goals to include through configuration. A goal is defined as a hard goal or soft goal. You can add hard goals through Cruise Control deployment configuration. You also have main, default, and user-provided goals that fit into each of these categories.

- **Hard goals** are preset and must be satisfied for an optimization proposal to be successful.
- **Soft goals** do not need to be satisfied for an optimization proposal to be successful. They can be set aside if it means that all hard goals are met.

- **Main goals** are inherited from Cruise Control. Some are preset as hard goals. Main goals are used in optimization proposals by default.
- **Default goals** are the same as the main goals by default. You can specify your own set of default goals.
- **User-provided goals** are a subset of default goals that are configured for generating a specific optimization proposal.

Optimization proposals

Optimization proposals comprise the goals you want to achieve from a rebalance. You generate an optimization proposal to create a summary of proposed changes and the results that are possible with the rebalance. The goals are assessed in a specific order of priority. You can then choose to approve or reject the proposal. You can reject the proposal to run it again with an adjusted set of goals.
You can generate an optimization proposal in one of three modes.

- **full** is the default mode and runs a full rebalance.
- **add-brokers** is the mode you use after adding brokers when scaling up a Kafka cluster.
- **remove-brokers** is the mode you use before removing brokers when scaling down a Kafka cluster.

Other Cruise Control features are not currently supported, including self healing, notifications, write-your-own goals, and changing the topic replication factor.

Additional resources

- Cruise Control documentation

11.2. OPTIMIZATION GOALS OVERVIEW

Optimization goals are constraints on workload redistribution and resource utilization across a Kafka cluster. To rebalance a Kafka cluster, Cruise Control uses optimization goals to generate optimization proposals, which you can approve or reject.

11.2.1. Goals order of priority

AMQ Streams supports most of the optimization goals developed in the Cruise Control project. The supported goals, in the default descending order of priority, are as follows:

1. Rack-awareness
2. Minimum number of leader replicas per broker for a set of topics
3. Replica capacity
4. Capacity goals
    - Disk capacity
    - Network inbound capacity

- Network outbound capacity
- CPU capacity

5. Replica distribution
6. Potential network output
7. Resource distribution goals
   - Disk utilization distribution
   - Network inbound utilization distribution
   - Network outbound utilization distribution
   - CPU utilization distribution
8. Leader bytes-in rate distribution
9. Topic replica distribution
10. Leader replica distribution
11. Preferred leader election
12. Intra-broker disk capacity
13. Intra-broker disk usage distribution

For more information on each optimization goal, see Goals in the Cruise Control Wiki.

NOTE

"Write your own" goals and Kafka assigner goals are not yet supported.

11.2.2. Goals configuration in AMQ Streams custom resources

You configure optimization goals in **Kafka** and **KafkaRebalance** custom resources. Cruise Control has configurations for hard optimization goals that must be satisfied, as well as main, default, and user-provided optimization goals.

You can specify optimization goals in the following configuration:

- Main goals – **Kafka.spec.cruiseControl.config.goals**
- Hard goals – **Kafka.spec.cruiseControl.config.hard.goals**
- Default goals – **Kafka.spec.cruiseControl.config.default.goals**
- User-provided goals – **KafkaRebalance.spec.goals**

NOTE

Resource distribution goals are subject to capacity limits on broker resources.

11.2.3. Hard and soft optimization goals

Hard goals are goals that *must* be satisfied in optimization proposals. Goals that are not configured as hard goals are known as *soft goals*. You can think of soft goals as *best effort* goals: they do *not* need to be satisfied in optimization proposals, but are included in optimization calculations. An optimization proposal that violates one or more soft goals, but satisfies all hard goals, is valid.

Cruise Control will calculate optimization proposals that satisfy all the hard goals and as many soft goals as possible (in their priority order). An optimization proposal that does *not* satisfy all the hard goals is rejected by Cruise Control and not sent to the user for approval.

NOTE

For example, you might have a soft goal to distribute a topic's replicas evenly across the cluster (the topic replica distribution goal). Cruise Control will ignore this goal if doing so enables all the configured hard goals to be met.

In Cruise Control, the following main optimization goals are preset as hard goals:

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal
```

You configure hard goals in the Cruise Control deployment configuration, by editing the **hard.goals** property in **Kafka.spec.cruiseControl.config**.

- To inherit the preset hard goals from Cruise Control, do not specify the **hard.goals** property in **Kafka.spec.cruiseControl.config**
- To change the preset hard goals, specify the desired goals in the **hard.goals** property, using their fully-qualified domain names.

Example Kafka configuration for hard optimization goals

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      # Note that `default.goals` (superset) must also include all `hard.goals` (subset)
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
      hard.goals: >
```

```
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
  # ...
```

Increasing the number of configured hard goals will reduce the likelihood of Cruise Control generating valid optimization proposals.

If **skipHardGoalCheck: true** is specified in the **KafkaRebalance** custom resource, Cruise Control does *not* check that the list of user-provided optimization goals (in **KafkaRebalance.spec.goals**) contains *all* the configured hard goals (**hard.goals**). Therefore, if some, but not all, of the user-provided optimization goals are in the **hard.goals** list, Cruise Control will still treat them as hard goals even if **skipHardGoalCheck: true** is specified.

11.2.4. Main optimization goals

The *main optimization goals* are available to all users. Goals that are not listed in the main optimization goals are not available for use in Cruise Control operations.

Unless you change the Cruise Control deployment configuration, AMQ Streams will inherit the following main optimization goals from Cruise Control, in descending priority order:

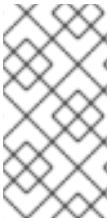
```
RackAwareGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal;
NetworkOutboundCapacityGoal; CpuCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal;
DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal;
NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal;
LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal
```

Some of these goals are preset as hard goals.

To reduce complexity, we recommend that you use the inherited main optimization goals, unless you need to *completely* exclude one or more goals from use in **KafkaRebalance** resources. The priority order of the main optimization goals can be modified, if desired, in the configuration for default optimization goals.

You configure main optimization goals, if necessary, in the Cruise Control deployment configuration: **Kafka.spec.cruiseControl.config.goals**

- To accept the inherited main optimization goals, do not specify the **goals** property in **Kafka.spec.cruiseControl.config**.
- If you need to modify the inherited main optimization goals, specify a list of goals, in descending priority order, in the **goals** configuration option.



NOTE

To avoid errors when generating optimization proposals, make sure that any changes you make to the **goals** or **default.goals** in **Kafka.spec.cruiseControl.config** include all of the hard goals specified for the **hard.goals** property. To clarify, the hard goals must also be specified (as a subset) for the main optimization goals and default goals.

11.2.5. Default optimization goals

Cruise Control uses the *default optimization goals* to generate the *cached optimization proposal*. For more information about the cached optimization proposal, see Section 11.3, “Optimization proposals overview”.

You can override the default optimization goals by setting user-provided optimization goals in a **KafkaRebalance** custom resource.

Unless you specify **default.goals** in the Cruise Control deployment configuration, the main optimization goals are used as the default optimization goals. In this case, the cached optimization proposal is generated using the main optimization goals.

- To use the main optimization goals as the default goals, do not specify the **default.goals** property in **Kafka.spec.cruiseControl.config**.
- To modify the default optimization goals, edit the **default.goals** property in **Kafka.spec.cruiseControl.config**. You must use a subset of the main optimization goals.

Example Kafka configuration for default optimization goals

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      # Note that `default.goals` (superset) must also include all `hard.goals` (subset)
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
      hard.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal
      # ...
```

If no default optimization goals are specified, the cached proposal is generated using the main optimization goals.

11.2.6. User-provided optimization goals

User-provided optimization goals narrow down the configured default goals for a particular optimization proposal. You can set them, as required, in **spec.goals** in a **KafkaRebalance** custom resource:

```
KafkaRebalance.spec.goals
```

User-provided optimization goals can generate optimization proposals for different scenarios. For example, you might want to optimize leader replica distribution across the Kafka cluster without considering disk capacity or disk utilization. So, you create a **KafkaRebalance** custom resource containing a single user-provided goal for leader replica distribution.

User-provided optimization goals must:

- Include all configured hard goals, or an error occurs
- Be a subset of the main optimization goals

To ignore the configured hard goals when generating an optimization proposal, add the **skipHardGoalCheck: true** property to the **KafkaRebalance** custom resource. See Section 11.6, "Generating optimization proposals".

Additional resources

- Configuring and deploying Cruise Control with Kafka
- Configurations in the Cruise Control Wiki.

11.3. OPTIMIZATION PROPOSALS OVERVIEW

Configure a **KafkaRebalance** resource to generate optimization proposals and apply the suggested changes. An *optimization proposal* is a summary of proposed changes that would produce a more balanced Kafka cluster, with partition workloads distributed more evenly among the brokers.

Each optimization proposal is based on the set of optimization goals that was used to generate it, subject to any configured capacity limits on broker resources.

All optimization proposals are *estimates* of the impact of a proposed rebalance. You can approve or reject a proposal. You cannot approve a cluster rebalance without first generating the optimization proposal.

You can run optimization proposals in one of the following rebalancing modes:

- **full**
- **add-brokers**
- **remove-brokers**

11.3.1. Rebalancing modes

You specify a rebalancing mode using the **spec.mode** property of the **KafkaRebalance** custom resource.

full

The **full** mode runs a full rebalance by moving replicas across all the brokers in the cluster. This is the default mode if the **spec.mode** property is not defined in the **KafkaRebalance** custom resource.

add-brokers

The **add-brokers** mode is used after scaling up a Kafka cluster by adding one or more brokers. Normally, after scaling up a Kafka cluster, new brokers are used to host only the partitions of newly created topics. If no new topics are created, the newly added brokers are not used and the existing brokers remain under the same load. By using the **add-brokers** mode immediately after adding brokers to the cluster, the rebalancing operation moves replicas from existing brokers to the newly added brokers. You specify the new brokers as a list using the **spec.brokers** property of the **KafkaRebalance** custom resource.

remove-brokers

The **remove-brokers** mode is used before scaling down a Kafka cluster by removing one or more brokers. If you scale down a Kafka cluster, brokers are shut down even if they host replicas. This can lead to under-replicated partitions and possibly result in some partitions being under their minimum ISR (in-sync replicas). To avoid this potential problem, the **remove-brokers** mode moves replicas off the brokers that are going to be removed. When these brokers are not hosting replicas anymore, you can safely run the scaling down operation. You specify the brokers you're removing as a list in the **spec.brokers** property in the **KafkaRebalance** custom resource.

In general, use the **full** rebalance mode to rebalance a Kafka cluster by spreading the load across brokers. Use the **add-brokers** and **remove-brokers** modes only if you want to scale your cluster up or down and rebalance the replicas accordingly.

The procedure to run a rebalance is actually the same across the three different modes. The only difference is with specifying a mode through the **spec.mode** property and, if needed, listing brokers that have been added or will be removed through the **spec.brokers** property.

11.3.2. The results of an optimization proposal

When an optimization proposal is generated, a summary and broker load is returned.

Summary

The summary is contained in the **KafkaRebalance** resource. The summary provides an overview of the proposed cluster rebalance and indicates the scale of the changes involved. A summary of a successfully generated optimization proposal is contained in the **Status.OptimizationResult** property of the **KafkaRebalance** resource. The information provided is a summary of the full optimization proposal.

Broker load

The broker load is stored in a ConfigMap that contains data as a JSON string. The broker load shows before and after values for the proposed rebalance, so you can see the impact on each of the brokers in the cluster.

11.3.3. Manually approving or rejecting an optimization proposal

An optimization proposal summary shows the proposed scope of changes.

You can use the name of the **KafkaRebalance** resource to return a summary from the command line.

Returning an optimization proposal summary

```
oc describe kafkarebalance <kafka_rebalance_resource_name> -n <namespace>
```

You can also use the **jq** command line JSON parser tool.

Returning an optimization proposal summary using jq

```
oc get kafkarebalance -o json | jq <jq_query>.
```

Use the summary to decide whether to approve or reject an optimization proposal.

Approving an optimization proposal

You approve the optimization proposal by setting the **strimzi.io/rebalance** annotation of the **KafkaRebalance** resource to **approve**. Cruise Control applies the proposal to the Kafka cluster and starts a cluster rebalance operation.

Rejecting an optimization proposal

If you choose not to approve an optimization proposal, you can change the optimization goals or update any of the rebalance performance tuning options , and then generate another proposal. You can use the **strimzi.io/refresh** annotation to generate a new optimization proposal for a **KafkaRebalance** resource.

Use optimization proposals to assess the movements required for a rebalance. For example, a summary describes inter-broker and intra-broker movements. Inter-broker rebalancing moves data between separate brokers. Intra-broker rebalancing moves data between disks on the same broker when you are using a JBOD storage configuration. Such information can be useful even if you don't go ahead and approve the proposal.

You might reject an optimization proposal, or delay its approval, because of the additional load on a Kafka cluster when rebalancing.

In the following example, the proposal suggests the rebalancing of data between separate brokers. The rebalance involves the movement of 55 partition replicas, totaling 12MB of data, across the brokers. Though the inter-broker movement of partition replicas has a high impact on performance, the total amount of data is not large. If the total data was much larger, you could reject the proposal, or time when to approve the rebalance to limit the impact on the performance of the Kafka cluster.

Rebalance performance tuning options can help reduce the impact of data movement. If you can extend the rebalance period, you can divide the rebalance into smaller batches. Fewer data movements at a single time reduces the load on the cluster.

Example optimization proposal summary

```
Name:         my-rebalance
Namespace:    myproject
Labels:       strimzi.io/cluster=my-cluster
Annotations:  API Version:  kafka.strimzi.io/v1alpha1
Kind:         KafkaRebalance
Metadata:
# ...
Status:
  Conditions:
    Last Transition Time:  2022-04-05T14:36:11.900Z
    Status:                ProposalReady
    Type:                  State
  Observed Generation:     1
  Optimization Result:
    Data To Move MB:  0
    Excluded Brokers For Leadership:
    Excluded Brokers For Replica Move:
    Excluded Topics:
    Intra Broker Data To Move MB:         12
    Monitored Partitions Percentage:      100
    Num Intra Broker Replica Movements:   0
    Num Leader Movements:                 24
    Num Replica Movements:                55
    On Demand Balancedness Score After:   82.91290759174306
    On Demand Balancedness Score Before:  78.01176356230222
    Recent Windows:                       5
  Session Id:                             a4f833bd-2055-4213-bfdd-ad21f95bf184
```

The proposal will also move 24 partition leaders to different brokers. This requires a change to the ZooKeeper configuration, which has a low impact on performance.

The balancedness scores are measurements of the overall balance of the Kafka cluster before and after the optimization proposal is approved. A balancedness score is based on optimization goals. If all goals are satisfied, the score is 100. The score is reduced for each goal that will not be met. Compare the balancedness scores to see whether the Kafka cluster is less balanced than it could be following a rebalance.

11.3.4. Automatically approving an optimization proposal

To save time, you can automate the process of approving optimization proposals. With automation, when you generate an optimization proposal it goes straight into a cluster rebalance.

To enable the optimization proposal auto-approval mechanism, create the **KafkaRebalance** resource with the **strimzi.io/rebalance-auto-approval** annotation set to **true**. If the annotation is not set or set to **false**, the optimization proposal requires manual approval.

Example rebalance request with auto-approval mechanism enabled

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: "true"
spec:
  mode: # any mode
  # ...
```

You can still check the status when automatically approving an optimization proposal. The status of the **KafkaRebalance** resource moves to **Ready** when the rebalance is complete.

11.3.5. Optimization proposal summary properties

The following table explains the properties contained in the optimization proposal's summary section.

Table 11.1. Properties contained in an optimization proposal summary

JSON property	Description
numIntraBrokerReplicaMovements	The total number of partition replicas that will be transferred between the disks of the cluster's brokers. **Performance impact during rebalance operation:** Relatively high, but lower than **numReplicaMovements**.
excludedBrokersForLeadership	Not yet supported. An empty list is returned.

JSON property	Description
numReplicaMovements	The number of partition replicas that will be moved between separate brokers. **Performance impact during rebalance operation**: Relatively high.
onDemandBalancednessScoreBefore, onDemandBalancednessScoreAfter	A measurement of the overall *balancedness* of a Kafka Cluster, before and after the optimization proposal was generated. The score is calculated by subtracting the sum of the **BalancednessScore** of each violated soft goal from 100. Cruise Control assigns a **BalancednessScore** to every optimization goal based on several factors, including priority—the goal's position in the list of **default.goals** or user-provided goals. The **Before** score is based on the current configuration of the Kafka cluster. The **After** score is based on the generated optimization proposal.
intraBrokerDataToMoveMB	The sum of the size of each partition replica that will be moved between disks on the same broker (see also **numIntraBrokerReplicaMovements**). **Performance impact during rebalance operation**: Variable. The larger the number, the longer the cluster rebalance will take to complete. Moving a large amount of data between disks on the same broker has less impact than between separate brokers (see **dataToMoveMB**).
recentWindows	The number of metrics windows upon which the optimization proposal is based.
dataToMoveMB	The sum of the size of each partition replica that will be moved to a separate broker (see also **numReplicaMovements**). **Performance impact during rebalance operation**: Variable. The larger the number, the longer the cluster rebalance will take to complete.
monitoredPartitionsPercentage	The percentage of partitions in the Kafka cluster covered by the optimization proposal. Affected by the number of **excludedTopics**.
excludedTopics	If you specified a regular expression in the **spec.excludedTopicsRegex** property in the **KafkaRebalance** resource, all topic names matching that expression are listed here. These topics are excluded from the calculation of partition replica/leader movements in the optimization proposal.
numLeaderMovements	The number of partitions whose leaders will be switched to different replicas. This involves a change to ZooKeeper configuration. **Performance impact during rebalance operation**: Relatively low.

JSON property	Description
excludedBrokersForReplicaMove	Not yet supported. An empty list is returned.

11.3.6. Broker load properties

The broker load is stored in a ConfigMap (with the same name as the KafkaRebalance custom resource) as a JSON formatted string. This JSON string consists of a JSON object with keys for each broker IDs linking to a number of metrics for each broker. Each metric consist of three values. The first is the metric value before the optimization proposal is applied, the second is the expected value of the metric after the proposal is applied, and the third is the difference between the first two values (after minus before).

NOTE

The ConfigMap appears when the KafkaRebalance resource is in the **ProposalReady** state and remains after the rebalance is complete.

You can use the name of the ConfigMap to view its data from the command line.

Returning ConfigMap data

```
oc describe configmaps <my_rebalance_configmap_name> -n <namespace>
```

You can also use the **jq** command line JSON parser tool to extract the JSON string from the ConfigMap.

Extracting the JSON string from the ConfigMap using jq

```
oc get configmaps <my_rebalance_configmap_name> -o json | jq '.["data"]
["brokerLoad.json"]|fromjson|.'
```

The following table explains the properties contained in the optimization proposal's broker load ConfigMap:

JSON property	Description
leaders	The number of replicas on this broker that are partition leaders.
replicas	The number of replicas on this broker.
cpuPercentage	The CPU utilization as a percentage of the defined capacity.
diskUsedPercentage	The disk utilization as a percentage of the defined capacity.
diskUsedMB	The absolute disk usage in MB.
networkOutRate	The total network output rate for the broker.

JSON property	Description
leaderNetworkInRate	The network input rate for all partition leader replicas on this broker.
followerNetworkInRate	The network input rate for all follower replicas on this broker.
potentialMaxNetworkOutRate	The hypothetical maximum network output rate that would be realized if this broker became the leader of all the replicas it currently hosts.

11.3.7. Cached optimization proposal

Cruise Control maintains a *cached optimization proposal* based on the configured default optimization goals. Generated from the workload model, the cached optimization proposal is updated every 15 minutes to reflect the current state of the Kafka cluster. If you generate an optimization proposal using the default optimization goals, Cruise Control returns the most recent cached proposal.

To change the cached optimization proposal refresh interval, edit the **proposal.expiration.ms** setting in the Cruise Control deployment configuration. Consider a shorter interval for fast changing clusters, although this increases the load on the Cruise Control server.

Additional resources

- Section 11.2, "Optimization goals overview"
- Section 11.6, "Generating optimization proposals"
- Section 11.7, "Approving an optimization proposal"

11.4. REBALANCE PERFORMANCE TUNING OVERVIEW

You can adjust several performance tuning options for cluster rebalances. These options control how partition replica and leadership movements in a rebalance are executed, as well as the bandwidth that is allocated to a rebalance operation.

11.4.1. Partition reassignment commands

Optimization proposals are comprised of separate partition reassignment commands. When you approve a proposal, the Cruise Control server applies these commands to the Kafka cluster.

A partition reassignment command consists of either of the following types of operations:

- Partition movement: Involves transferring the partition replica and its data to a new location. Partition movements can take one of two forms:
  - Inter-broker movement: The partition replica is moved to a log directory on a different broker.
  - Intra-broker movement: The partition replica is moved to a different log directory on the same broker.
- Leadership movement: This involves switching the leader of the partition's replicas.

Cruise Control issues partition reassignment commands to the Kafka cluster in batches. The performance of the cluster during the rebalance is affected by the number of each type of movement contained in each batch.

11.4.2. Replica movement strategies

Cluster rebalance performance is also influenced by the *replica movement strategy* that is applied to the batches of partition reassignment commands. By default, Cruise Control uses the **BaseReplicaMovementStrategy**, which simply applies the commands in the order they were generated. However, if there are some very large partition reassignments early in the proposal, this strategy can slow down the application of the other reassignments.

Cruise Control provides four alternative replica movement strategies that can be applied to optimization proposals:

- **PrioritizeSmallReplicaMovementStrategy**: Order reassignments in order of ascending size.
- **PrioritizeLargeReplicaMovementStrategy**: Order reassignments in order of descending size.
- **PostponeUrpReplicaMovementStrategy**: Prioritize reassignments for replicas of partitions which have no out-of-sync replicas.
- **PrioritizeMinIsrWithOfflineReplicasStrategy**: Prioritize reassignments with (At/Under)MinISR partitions with offline replicas. This strategy will only work if **cruiseControl.config.concurrency.adjuster.min.isr.check.enabled** is set to **true** in the **Kafka** custom resource's spec.

These strategies can be configured as a sequence. The first strategy attempts to compare two partition reassignments using its internal logic. If the reassignments are equivalent, then it passes them to the next strategy in the sequence to decide the order, and so on.

11.4.3. Intra-broker disk balancing

Moving a large amount of data between disks on the same broker has less impact than between separate brokers. If you are running a Kafka deployment that uses JBOD storage with multiple disks on the same broker, Cruise Control can balance partitions between the disks.

NOTE

If you are using JBOD storage with a single disk, intra-broker disk balancing will result in a proposal with 0 partition movements since there are no disks to balance between.

To perform an intra-broker disk balance, set **rebalanceDisk** to **true** under the **KafkaRebalance.spec**. When setting **rebalanceDisk** to **true**, do not set a **goals** field in the **KafkaRebalance.spec**, as Cruise Control will automatically set the intra-broker goals and ignore the inter-broker goals. Cruise Control does not perform inter-broker and intra-broker balancing at the same time.

11.4.4. Rebalance tuning options

Cruise Control provides several configuration options for tuning the rebalance parameters discussed above. You can set these tuning options when configuring and deploying Cruise Control with Kafka or optimization proposal levels:

- The Cruise Control server setting can be set in the Kafka custom resource under **Kafka.spec.cruiseControl.config**.

- The individual rebalance performance configurations can be set under **KafkaRebalance.spec**.

The relevant configurations are summarized in the following table.

Table 11.2. Rebalance performance tuning configuration

Cruise Control properties	KafkaRebalance properties	Default	Description
num.concurrent.partition.movements.per.broker	**concurrentPartitionMovementsPerBroker**	5	The maximum number of inter-broker partition movements in each partition reassignment batch
num.concurrent.intra.broker.partition.movements	**concurrentIntraBrokerPartitionMovements**	2	The maximum number of intra-broker partition movements in each partition reassignment batch
num.concurrent.leader.movements	**concurrentLeaderMovements**	1000	The maximum number of partition leadership changes in each partition reassignment batch
default.replication.throttle	**replicationThrottle**	Null (no limit)	The bandwidth (in bytes per second) to assign to partition reassignment

Cruise Control properties	KafkaRebalance properties	Default	Description
default.replica.movement.strategies	**replicaMovementStrategies**	**BaseReplicaMovementStrategy**	The list of strategies (in priority order) used to determine the order in which partition reassignment commands are executed for generated proposals. For the server setting, use a comma separated string with the fully qualified names of the strategy class (add **com.linkedin.kafka.cruisecontrol.executor.strategy.** to the start of each class name). For the **KafkaRebalance** resource setting use a YAML array of strategy class names.
-	**rebalanceDisk**	false	Enables intra-broker disk balancing, which balances disk space utilization between disks on the same broker. Only applies to Kafka deployments that use JBOD storage with multiple disks.

Changing the default settings affects the length of time that the rebalance takes to complete, as well as the load placed on the Kafka cluster during the rebalance. Using lower values reduces the load but increases the amount of time taken, and vice versa.

Additional resources

- **CruiseControlSpec** schema reference
- **KafkaRebalanceSpec** schema reference

11.5. CONFIGURING AND DEPLOYING CRUISE CONTROL WITH KAFKA

Configure a **Kafka** resource to deploy Cruise Control alongside a Kafka cluster. You can use the **cruiseControl** properties of the **Kafka** resource to configure the deployment. Deploy one instance of Cruise Control per Kafka cluster.

Use **goals** configuration in the Cruise Control **config** to specify optimization goals for generating optimization proposals. You can use **brokerCapacity** to change the default capacity limits for goals related to resource distribution. If brokers are running on nodes with heterogeneous network resources, you can use **overrides** to set network capacity limits for each broker.

If an empty object (**{}**) is used for the **cruiseControl** configuration, all properties use their default values.

For more information on the configuration options for Cruise Control, see the Custom resource API reference.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **cruiseControl** property for the **Kafka** resource.
   The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    brokerCapacity: 1
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
      overrides: 2
      - brokers: [0]
        inboundNetwork: 20000KiB/s
        outboundNetwork: 20000KiB/s
      - brokers: [1, 2]
        inboundNetwork: 30000KiB/s
        outboundNetwork: 30000KiB/s
```

```
    # ...
  config: 3
    # Note that `default.goals` (superset) must also include all `hard.goals` (subset)
    default.goals: > 4
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
      # ...
    hard.goals: >
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal
      # ...
    cpu.balance.threshold: 1.1
    metadata.max.age.ms: 300000
    send.buffer.bytes: 131072
    webserver.http.cors.enabled: true 5
    webserver.http.cors.origin: "*"
    webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
    # ...
  resources: 6
    requests:
      cpu: 1
      memory: 512Mi
    limits:
      cpu: 2
      memory: 2Gi
  logging: 7
      type: inline
      loggers:
        rootLogger.level: "INFO"
  template: 8
    pod:
      metadata:
        labels:
          label1: value1
      securityContext:
        runAsUser: 1000001
        fsGroup: 0
      terminationGracePeriodSeconds: 120
  readinessProbe: 9
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  metricsConfig: 10
    type: jmxPrometheusExporter
    valueFrom:
      configMapKeyRef:
        name: cruise-control-metrics
        key: metrics-config.yml
# ...
```

1 Capacity limits for broker resources.

2 Overrides set network capacity limits for specific brokers when running on nodes with heterogeneous network resources.

heterogeneous network resources.

3. Cruise Control configuration. Standard Cruise Control configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

4. Optimization goals configuration, which can include configuration for default optimization goals (**default.goals**), main optimization goals (**goals**), and hard goals (**hard.goals**).

5. CORS enabled and configured for read-only access to the Cruise Control API.

6. Requests for reservation of supported resources, currently **cpu** and **memory**, and limits to specify the maximum resources that can be consumed.

7. Cruise Control loggers and log levels added directly (**inline**) or indirectly (**external**) through a ConfigMap. A custom ConfigMap must be placed under the **log4j.properties** key. Cruise Control has a single logger named **rootLogger.level**. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

8. Template customization. Here a pod is scheduled with additional security attributes.

9. Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).

10. Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter (the default metrics exporter).

2. Create or update the resource:

```
oc apply -f <kafka_configuration_file>
```

3. Check the status of the deployment:

```
oc get deployments -n <my_cluster_operator_namespace>
```

Output shows the deployment name and readiness

```
NAME                       READY  UP-TO-DATE  AVAILABLE
my-cluster-cruise-control 1/1    1           1
```

my-cluster is the name of the Kafka cluster.

READY shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows **1**.

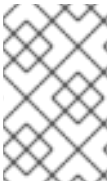
Auto-created topics

The following table shows the three topics that are automatically created when Cruise Control is deployed. These topics are required for Cruise Control to work properly and must not be deleted or changed. You can change the name of the topic using the specified configuration option.

Table 11.3. Auto-created topics

Auto-created topic configuration	Default topic name	Created by	Function
metric.reporter.topic	**strimzi.cruisecontrol.metrics**	AMQ Streams Metrics Reporter	Stores the raw metrics from the Metrics Reporter in each Kafka broker.
partition.metric.sample.store.topic	**strimzi.cruisecontrol.partitionmetricsamples**	Cruise Control	Stores the derived metrics for each partition. These are created by the Metric Sample Aggregator.
broker.metric.sample.store.topic	**strimzi.cruisecontrol.modeltrainingsamples**	Cruise Control	Stores the metrics samples used to create the Cluster Workload Model.

To prevent the removal of records that are needed by Cruise Control, log compaction is disabled in the auto-created topics.



NOTE

If the names of the auto-created topics are changed in a Kafka cluster that already has Cruise Control enabled, the old topics will not be deleted and should be manually removed.

What to do next

After configuring and deploying Cruise Control, you can generate optimization proposals.

Additional resources

- Optimization goals overview

11.6. GENERATING OPTIMIZATION PROPOSALS

When you create or update a **KafkaRebalance** resource, Cruise Control generates an optimization proposal for the Kafka cluster based on the configured optimization goals. Analyze the information in the optimization proposal and decide whether to approve it. You can use the results of the optimization proposal to rebalance your Kafka cluster.

You can run the optimization proposal in one of the following modes:

- **full** (default)
- **add-brokers**
- **remove-brokers**

The mode you use depends on whether you are rebalancing across all the brokers already running in the Kafka cluster; or you want to rebalance after scaling up or before scaling down your Kafka cluster. For more information, see Rebalancing modes with broker scaling.

Prerequisites

- You have deployed Cruise Control to your AMQ Streams cluster.
- You have configured optimization goals and, optionally, capacity limits on broker resources.

For more information on configuring Cruise Control, see Section 11.5, "Configuring and deploying Cruise Control with Kafka".

Procedure

1. Create a **KafkaRebalance** resource and specify the appropriate mode.

   full mode (default)

   To use the *default optimization goals* defined in the **Kafka** resource, leave the **spec** property empty. Cruise Control rebalances a Kafka cluster in **full** mode by default.

   Example configuration with full rebalancing by default

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec: {}
   ```

   You can also run a full rebalance by specifying the **full** mode through the **spec.mode** property.

   Example configuration specifying full mode

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     mode: full
   ```

   add-brokers mode

   If you want to rebalance a Kafka cluster after scaling up, specify the **add-brokers** mode. In this mode, existing replicas are moved to the newly added brokers. You need to specify the brokers as a list.

   Example configuration specifying add-brokers mode

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   ```

```
spec:
  mode: add-brokers
  brokers: [3, 4] 1
```

1 List of newly added brokers added by the scale up operation. This property is mandatory.

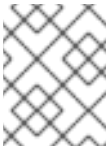
remove-brokers mode

If you want to rebalance a Kafka cluster before scaling down, specify the **remove-brokers** mode.
In this mode, replicas are moved off the brokers that are going to be removed. You need to specify the brokers that are being removed as a list.

Example configuration specifying remove-brokers mode

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: remove-brokers
  brokers: [3, 4] 1
```

1 List of brokers to be removed by the scale down operation. This property is mandatory.



NOTE

The following steps and the steps to approve or stop a rebalance are the same regardless of the rebalance mode you are using.

2. To configure *user-provided optimization goals* instead of using the default goals, add the **goals** property and enter one or more goals.
   In the following example, rack awareness and replica capacity are configured as user-provided optimization goals:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
```

3. To ignore the configured hard goals, add the **skipHardGoalCheck: true** property:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

4. (Optional) To approve the optimization proposal automatically, set the **strimzi.io/rebalance-auto-approval** annotation to **true**:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: "true"
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

5. Create or update the resource:

```
oc apply -f <kafka_rebalance_configuration_file>
```

The Cluster Operator requests the optimization proposal from Cruise Control. This might take a few minutes depending on the size of the Kafka cluster.

6. If you used the automatic approval mechanism, wait for the status of the optimization proposal to change to **Ready**. If you haven't enabled the automatic approval mechanism, wait for the status of the optimization proposal to change to **ProposalReady**:

```
oc get kafkarebalance -o wide -w -n <namespace>
```

PendingProposal

A **PendingProposal** status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

ProposalReady

A **ProposalReady** status means the optimization proposal is ready for review and approval.

When the status changes to **ProposalReady**, the optimization proposal is ready to approve.

7. Review the optimization proposal.
The optimization proposal is contained in the **Status.Optimization Result** property of the **KafkaRebalance** resource.

```
oc describe kafkarebalance <kafka_rebalance_resource_name>
```

Example optimization proposal

```
Status:
  Conditions:
    Last Transition Time:  2020-05-19T13:50:12.533Z
    Status:                ProposalReady
    Type:                  State
  Observed Generation:     1
  Optimization Result:
    Data To Move MB:  0
    Excluded Brokers For Leadership:
    Excluded Brokers For Replica Move:
    Excluded Topics:
    Intra Broker Data To Move MB:         0
    Monitored Partitions Percentage:      100
    Num Intra Broker Replica Movements:   0
    Num Leader Movements:                 0
    Num Replica Movements:                26
    On Demand Balancedness Score After:   81.8666802863978
    On Demand Balancedness Score Before:  78.01176356230222
    Recent Windows:                       1
  Session Id:                             05539377-ca7b-45ef-b359-e13564f1458c
```

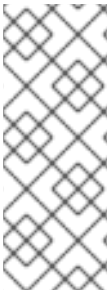
The properties in the **Optimization Result** section describe the pending cluster rebalance operation. For descriptions of each property, see Contents of optimization proposals.

Insufficient CPU capacity

If a Kafka cluster is overloaded in terms of CPU utilization, you might see an insufficient CPU capacity error in the **KafkaRebalance** status. It's worth noting that this utilization value is unaffected by the **excludedTopics** configuration. Although optimization proposals will not reassign replicas of excluded topics, their load is still considered in the utilization calculation.

Example CPU utilization error

```
com.linkedin.kafka.cruisecontrol.exception.OptimizationFailureException:
    [CpuCapacityGoal] Insufficient capacity for cpu (Utilization 615.21,
    Allowed Capacity 420.00, Threshold: 0.70). Add at least 3 brokers with
    the same cpu capacity (100.00) as broker-0. Add at least 3 brokers with
    the same cpu capacity (100.00) as broker-0.
```



NOTE

The error shows CPU capacity as a percentage rather than the number of CPU cores. For this reason, it does not directly map to the number of CPUs configured in the Kafka custom resource. It is like having a single *virtual* CPU per broker, which has the cycles of the CPUs configured in **Kafka.spec.kafka.resources.limits.cpu**. This has no effect on the rebalance behavior, since the ratio between CPU utilization and capacity remains the same.

What to do next

Section 11.7, “Approving an optimization proposal”

Additional resources

- Section 11.3, “Optimization proposals overview”

11.7. APPROVING AN OPTIMIZATION PROPOSAL

You can approve an optimization proposal generated by Cruise Control, if its status is **ProposalReady**. Cruise Control will then apply the optimization proposal to the Kafka cluster, reassigning partitions to brokers and changing partition leadership.

CAUTION

This is not a dry run. Before you approve an optimization proposal, you must:

- Refresh the proposal in case it has become out of date.
- Carefully review the contents of the proposal.

Prerequisites

- You have generated an optimization proposal from Cruise Control.
- The **KafkaRebalance** custom resource status is **ProposalReady**.

Procedure

Perform these steps for the optimization proposal that you want to approve.

1. Unless the optimization proposal is newly generated, check that it is based on current information about the state of the Kafka cluster. To do so, refresh the optimization proposal to make sure it uses the latest cluster metrics:
   a. Annotate the **KafkaRebalance** resource in OpenShift with **strimzi.io/rebalance=refresh**:

```
oc annotate kafkarebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance=refresh
```

2. Wait for the status of the optimization proposal to change to **ProposalReady**:

```
oc get kafkarebalance -o wide -w -n <namespace>
```

   PendingProposal

   A **PendingProposal** status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

   ProposalReady

   A **ProposalReady** status means the optimization proposal is ready for review and approval.

   When the status changes to **ProposalReady**, the optimization proposal is ready to approve.

3. Approve the optimization proposal that you want Cruise Control to apply.
   Annotate the **KafkaRebalance** resource in OpenShift with **strimzi.io/rebalance=approve**:

```
oc annotate kafkarebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance=approve
```

4. The Cluster Operator detects the annotated resource and instructs Cruise Control to rebalance the Kafka cluster.

5. Wait for the status of the optimization proposal to change to **Ready**:

```
oc get kafkarebalance -o wide -w -n <namespace>
```

Rebalancing

A **Rebalancing** status means the rebalancing is in progress.

Ready

A **Ready** status means the rebalance is complete.

NotReady

A **NotReady** status means an error occurred—see Fixing problems with a **KafkaRebalance** resource.

When the status changes to **Ready**, the rebalance is complete.

To use the same **KafkaRebalance** custom resource to generate another optimization proposal, apply the **refresh** annotation to the custom resource. This moves the custom resource to the **PendingProposal** or **ProposalReady** state. You can then review the optimization proposal and approve it, if desired.

Additional resources

- Section 11.3, "Optimization proposals overview"
- Section 11.8, "Stopping a cluster rebalance"

11.8. STOPPING A CLUSTER REBALANCE

Once started, a cluster rebalance operation might take some time to complete and affect the overall performance of the Kafka cluster.

If you want to stop a cluster rebalance operation that is in progress, apply the **stop** annotation to the **KafkaRebalance** custom resource. This instructs Cruise Control to finish the current batch of partition reassignments and then stop the rebalance. When the rebalance has stopped, completed partition reassignments have already been applied; therefore, the state of the Kafka cluster is different when compared to prior to the start of the rebalance operation. If further rebalancing is required, you should generate a new optimization proposal.

NOTE

The performance of the Kafka cluster in the intermediate (stopped) state might be worse than in the initial state.

Prerequisites

- You have approved the optimization proposal by annotating the **KafkaRebalance** custom resource with **approve**.

- The status of the **KafkaRebalance** custom resource is **Rebalancing**.

Procedure

1. Annotate the **KafkaRebalance** resource in OpenShift:

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=stop
```

2. Check the status of the **KafkaRebalance** resource:

```
oc describe kafkarebalance rebalance-cr-name
```

3. Wait until the status changes to **Stopped**.

Additional resources

- Section 11.3, "Optimization proposals overview"

11.9. FIXING PROBLEMS WITH A KAFKAREBALANCE RESOURCE

If an issue occurs when creating a **KafkaRebalance** resource or interacting with Cruise Control, the error is reported in the resource status, along with details of how to fix it. The resource also moves to the **NotReady** state.

To continue with the cluster rebalance operation, you must fix the problem in the **KafkaRebalance** resource itself or with the overall Cruise Control deployment. Problems might include the following:

- A misconfigured parameter in the **KafkaRebalance** resource.
- The **strimzi.io/cluster** label for specifying the Kafka cluster in the **KafkaRebalance** resource is missing.
- The Cruise Control server is not deployed as the **cruiseControl** property in the **Kafka** resource is missing.
- The Cruise Control server is not reachable.

After fixing the issue, you need to add the **refresh** annotation to the **KafkaRebalance** resource. During a "refresh", a new optimization proposal is requested from the Cruise Control server.

Prerequisites

- You have approved an optimization proposal .
- The status of the **KafkaRebalance** custom resource for the rebalance operation is **NotReady**.

Procedure

1. Get information about the error from the **KafkaRebalance** status:

```
oc describe kafkarebalance rebalance-cr-name
```

2. Attempt to resolve the issue in the **KafkaRebalance** resource.

3. Annotate the **KafkaRebalance** resource in OpenShift:

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=refresh
```

4. Check the status of the **KafkaRebalance** resource:

```
oc describe kafkarebalance rebalance-cr-name
```

5. Wait until the status changes to **PendingProposal**, or directly to **ProposalReady**.

Additional resources

- Section 11.3, “Optimization proposals overview”

CHAPTER 12. USING THE PARTITION REASSIGNMENT TOOL

When scaling a Kafka cluster, you may need to add or remove brokers and update the distribution of partitions or the replication factor of topics. To update partitions and topics, you can use the **kafka-reassign-partitions.sh** tool.

Neither the AMQ Streams Cruise Control integration nor the Topic Operator support changing the replication factor of a topic. However, you can change the replication factor of a topic using the **kafka-reassign-partitions.sh** tool.

The tool can also be used to reassign partitions and balance the distribution of partitions across brokers to improve performance. However, it is recommended to use Cruise Control for automated partition reassignments and cluster rebalancing. Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

It is recommended to run the **kafka-reassign-partitions.sh** tool as a separate interactive pod rather than within the broker container. Running the Kafka **bin/** scripts within the broker container may cause a JVM to start with the same settings as the Kafka broker, which can potentially cause disruptions. By running the **kafka-reassign-partitions.sh** tool in a separate pod, you can avoid this issue. Running a pod with the **-ti** option creates an interactive pod with a terminal for running shell commands inside the pod.

Running an interactive pod with a terminal

```
oc run helper-pod -ti --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 --rm=true --
restart=Never -- bash
```

12.1. PARTITION REASSIGNMENT TOOL OVERVIEW

The partition reassignment tool provides the following capabilities for managing Kafka partitions and brokers:

Redistributing partition replicas

Scale your cluster up and down by adding or removing brokers, and move Kafka partitions from heavily loaded brokers to under-utilized brokers. To do this, you must create a partition reassignment plan that identifies which topics and partitions to move and where to move them. Cruise Control is recommended for this type of operation as it automates the cluster rebalancing process.

Scaling topic replication factor up and down

Increase or decrease the replication factor of your Kafka topics. To do this, you must create a partition reassignment plan that identifies the existing replication assignment across partitions and an updated assignment with the replication factor changes.

Changing the preferred leader

Change the preferred leader of a Kafka partition. This can be useful if the current preferred leader is unavailable or if you want to redistribute load across the brokers in the cluster. To do this, you must create a partition reassignment plan that specifies the new preferred leader for each partition by changing the order of replicas.

Changing the log directories to use a specific JBOD volume

Change the log directories of your Kafka brokers to use a specific JBOD volume. This can be useful if you want to move your Kafka data to a different disk or storage device. To do this, you must create a partition reassignment plan that specifies the new log directory for each topic.

12.1.1. Generating a partition reassignment plan

The partition reassignment tool (**kafka-reassign-partitions.sh**) works by generating a partition assignment plan that specifies which partitions should be moved from their current broker to a new broker.

If you are satisfied with the plan, you can execute it. The tool then does the following:

- Migrates the partition data to the new broker
- Updates the metadata on the Kafka brokers to reflect the new partition assignments
- Triggers a rolling restart of the Kafka brokers to ensure that the new assignments take effect

The partition reassignment tool has three different modes:

--generate

Takes a set of topics and brokers and generates a *reassignment JSON file* which will result in the partitions of those topics being assigned to those brokers. Because this operates on whole topics, it cannot be used when you only want to reassign some partitions of some topics.

--execute

Takes a *reassignment JSON file* and applies it to the partitions and brokers in the cluster. Brokers that gain partitions as a result become followers of the partition leader. For a given partition, once the new broker has caught up and joined the ISR (in-sync replicas) the old broker will stop being a follower and will delete its replica.

--verify

Using the same *reassignment JSON file* as the **--execute** step, **--verify** checks whether all the partitions in the file have been moved to their intended brokers. If the reassignment is complete, **--verify** also removes any traffic throttles (**--throttle**) that are in effect. Unless removed, throttles will continue to affect the cluster even after the reassignment has finished.

It is only possible to have one reassignment running in a cluster at any given time, and it is not possible to cancel a running reassignment. If you must cancel a reassignment, wait for it to complete and then perform another reassignment to revert the effects of the first reassignment. The **kafka-reassign-partitions.sh** will print the reassignment JSON for this reversion as part of its output. Very large reassignments should be broken down into a number of smaller reassignments in case there is a need to stop in-progress reassignment.

12.1.2. Specifying topics in a partition reassignment JSON file

The tool uses a reassignment JSON file that specifies the topics to reassign. You can generate a reassignment JSON file or create a file manually if you want to move specific partitions.

The reassignment JSON file has the following structure:

```
{
  "version": 1,
  "partitions": [
    <PartitionObjects>
  ]
}
```

Where *<PartitionObjects>* is a comma-separated list of objects like:

```
{
  "topic": <TopicName>,
```

```
  "partition": <Partition>,
  "replicas": [ <AssignedBrokerIds> ]
}
```

The following is an example reassignment JSON file that assigns partition **4** of topic **topic-a** to brokers **2**, **4** and **7**, and partition **2** of topic **topic-b** to brokers **1**, **5** and **7**:

Example partition reassignment file

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "topic-a",
      "partition": 4,
      "replicas": [2,4,7]
    },
    {
      "topic": "topic-b",
      "partition": 2,
      "replicas": [1,5,7]
    }
  ]
}
```

Partitions not included in the JSON are not changed.

12.1.3. Reassigning partitions between JBOD volumes

When using JBOD storage in your Kafka cluster, you can choose to reassign the partitions between specific volumes and their log directories (each volume has a single log directory). To reassign a partition to a specific volume, add the **log_dirs** option to *<PartitionObjects>* in the reassignment JSON file.

```
{
  "topic": <TopicName>,
  "partition": <Partition>,
  "replicas": [ <AssignedBrokerIds> ],
  "log_dirs": [ <AssignedLogDirs> ]
}
```

The **log_dirs** object should contain the same number of log directories as the number of replicas specified in the **replicas** object. The value should be either an absolute path to the log directory, or the **any** keyword.

Example partition reassignment file specifying log directories

```
{
      "topic": "topic-a",
      "partition": 4,
      "replicas": [2,4,7].
      "log_dirs": [ "/var/lib/kafka/data-0/kafka-log2", "/var/lib/kafka/data-0/kafka-log4",
"/var/lib/kafka/data-0/kafka-log7" ]
}
```

12.1.4. Throttling partition reassignment

Partition reassignment can be a slow process because it involves transferring large amounts of data between brokers. To avoid a detrimental impact on clients, you can throttle the reassignment process. Use the **--throttle** parameter with the **kafka-reassign-partitions.sh** tool to throttle a reassignment. You specify a maximum threshold in bytes per second for the movement of partitions between brokers. For example, **--throttle 5000000** sets a maximum threshold for moving partitions of 50 MBps.

Throttling might cause the reassignment to take longer to complete.

- If the throttle is too low, the newly assigned brokers will not be able to keep up with records being published and the reassignment will never complete.
- If the throttle is too high, clients will be impacted.

For example, for producers, this could manifest as higher than normal latency waiting for acknowledgment. For consumers, this could manifest as a drop in throughput caused by higher latency between polls.

12.2. GENERATING A REASSIGNMENT JSON FILE TO REASSIGN PARTITIONS

Generate a reassignment JSON file with the **kafka-reassign-partitions.sh** tool to reassign partitions after scaling a Kafka cluster. Adding or removing brokers does not automatically redistribute the existing partitions. To balance the partition distribution and take full advantage of the new brokers, you can reassign the partitions using the **kafka-reassign-partitions.sh** tool.

You run the tool from an interactive pod container connected to the Kafka cluster.

The following procedure describes a secure reassignment process that uses mTLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

You'll need the following to establish a connection:

- The cluster CA certificate and password generated by the Cluster Operator when the Kafka cluster is created
- The user CA certificate and password generated by the User Operator when a user is created for client access to the Kafka cluster

In this procedure, the CA certificates and corresponding passwords are extracted from the cluster and user secrets that contain them in PKCS #12 (**.p12** and **.password**) format. The passwords allow access to the **.p12** stores that contain the certificates. You use the **.p12** stores to specify a truststore and keystore to authenticate connection to the Kafka cluster.

Prerequisites

- You have a running Cluster Operator.
- You have a running Kafka cluster based on a **Kafka** resource configured with internal TLS encryption and mTLS authentication.

  Kafka configuration with TLS encryption and mTLS authentication

  ```
  apiVersion: kafka.strimzi.io/v1beta2
  kind: Kafka
  ```

```
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      # ...
      - name: tls
        port: 9093
        type: internal
        tls: true (1)
        authentication:
          type: tls (2)
    # ...
```

(1) Enables TLS encryption for the internal listener.

(2) Listener authentication mechanism specified as mutual **tls**.

- The running Kafka cluster contains a set of topics and partitions to reassign.

  Example topic configuration for my-topic

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 3
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
    # ...
```

- You have a **KafkaUser** configured with ACL rules that specify permission to produce and consume topics from the Kafka brokers.

  Example Kafka user configuration with ACL rules to allow operations on my-topic and my-cluster

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication: (1)
    type: tls
  authorization:
```

```
    type: simple (2)
    acls:
      # access to the topic
      - resource:
          type: topic
          name: my-topic
        operations:
          - Create
          - Describe
          - Read
          - AlterConfigs
        host: "*"
      # access to the cluster
      - resource:
          type: cluster
        operations:
          - Alter
          - AlterConfigs
        host: "*"
      # ...
  # ...
```

(1) User authentication mechanism defined as mutual **tls**.

(2) Simple authorization and accompanying list of ACL rules.

Procedure

1. Extract the cluster CA certificate and password from the ***<cluster_name>*-cluster-ca-cert** secret of the Kafka cluster.

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' | base64 -d > ca.p12
```

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.password}' | base64 -d > ca.password
```

   Replace *<cluster_name>* with the name of the Kafka cluster. When you deploy Kafka using the **Kafka** resource, a secret with the cluster CA certificate is created with the Kafka cluster name (***<cluster_name>*-cluster-ca-cert**). For example, **my-cluster-cluster-ca-cert**.

2. Run a new interactive pod container using the AMQ Streams Kafka image to connect to a running Kafka broker.

```
oc run --restart=Never --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 <interactive_pod_name> -- /bin/sh -c "sleep 3600"
```

   Replace *<interactive_pod_name>* with the name of the pod.

3. Copy the cluster CA certificate to the interactive pod container.

```
oc cp ca.p12 <interactive_pod_name>:/tmp
```

4. Extract the user CA certificate and password from the secret of the Kafka user that has permission to access the Kafka brokers.

```
oc get secret <kafka_user> -o jsonpath='{.data.user\.p12}' | base64 -d > user.p12
```

```
oc get secret <kafka_user> -o jsonpath='{.data.user\.password}' | base64 -d > user.password
```

Replace *<kafka_user>* with the name of the Kafka user. When you create a Kafka user using the **KafkaUser** resource, a secret with the user CA certificate is created with the Kafka user name. For example, **my-user**.

5. Copy the user CA certificate to the interactive pod container.

```
oc cp user.p12 <interactive_pod_name>:/tmp
```

The CA certificates allow the interactive pod container to connect to the Kafka broker using TLS.

6. Create a **config.properties** file to specify the truststore and keystore used to authenticate connection to the Kafka cluster.
Use the certificates and passwords you extracted in the previous steps.

```
bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093 1
security.protocol=SSL 2
ssl.truststore.location=/tmp/ca.p12 3
ssl.truststore.password=<truststore_password> 4
ssl.keystore.location=/tmp/user.p12 5
ssl.keystore.password=<keystore_password> 6
```

1. The bootstrap server address to connect to the Kafka cluster. Use your own Kafka cluster name to replace *<kafka_cluster_name>*.
2. The security protocol option when using TLS for encryption.
3. The truststore location contains the public key certificate (**ca.p12**) for the Kafka cluster.
4. The password (**ca.password**) for accessing the truststore.
5. The keystore location contains the public key certificate (**user.p12**) for the Kafka user.
6. The password (**user.password**) for accessing the keystore.

7. Copy the **config.properties** file to the interactive pod container.

```
oc cp config.properties <interactive_pod_name>:/tmp/config.properties
```

8. Prepare a JSON file named **topics.json** that specifies the topics to move.
Specify topic names as a comma-separated list.

Example JSON file to reassign all the partitions of my-topic

```
{
  "version": 1,
```

```
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

You can also use this file to change the replication factor of a topic .

9. Copy the ***topics.json*** file to the interactive pod container.

```
oc cp topics.json <interactive_pod_name>:/tmp/topics.json
```

10. Start a shell process in the interactive pod container.

```
oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

Replace *<namespace>* with the OpenShift namespace where the pod is running.

11. Use the **kafka-reassign-partitions.sh** command to generate the reassignment JSON.

Example command to move the partitions of my-topic to specified brokers

```
bin/kafka-reassign-partitions.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --topics-to-move-json-file /tmp/topics.json \
  --broker-list 0,1,2,3,4 \
  --generate
```

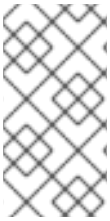
Additional resources

- Configuring Kafka
- Section 13.3.3, “Configuring Kafka topics”
- Section 13.4.1, “Configuring Kafka users”

12.3. REASSIGNING PARTITIONS AFTER ADDING BROKERS

Use a reassignment file generated by the **kafka-reassign-partitions.sh** tool to reassign partitions after increasing the number of brokers in a Kafka cluster. The reassignment file should describe how partitions are reassigned to brokers in the enlarged Kafka cluster. You apply the reassignment specified in the file to the brokers and then verify the new partition assignments.

This procedure describes a secure scaling process that uses TLS. You’ll need a Kafka cluster that uses TLS encryption and mTLS authentication.



NOTE

Though you can use the **kafka-reassign-partitions.sh** tool, Cruise Control is recommended for automated partition reassignments and cluster rebalancing . Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

Prerequisites

- You have a running Kafka cluster based on a **Kafka** resource configured with internal TLS encryption and mTLS authentication.
- You have generated a reassignment JSON file named **reassignment.json**.
- You are running an interactive pod container that is connected to the running Kafka broker.
- You are connected as a **KafkaUser** configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

See Generating reassignment JSON files.

Procedure

1. Add as many new brokers as you need by increasing the **Kafka.spec.kafka.replicas** configuration option.
2. Verify that the new broker pods have started.
3. If you haven't done so, run an interactive pod container to generate a reassignment JSON file named **reassignment.json**.
4. Copy the **reassignment.json** file to the interactive pod container.

   ```
   oc cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
   ```

   Replace *<interactive_pod_name>* with the name of the pod.
5. Start a shell process in the interactive pod container.

   ```
   oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
   ```

   Replace *<namespace>* with the OpenShift namespace where the pod is running.
6. Run the partition reassignment using the **kafka-reassign-partitions.sh** script from the interactive pod container.

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --execute
   ```

   Replace *<cluster_name>* with the name of your Kafka cluster. For example, **my-cluster-kafka-bootstrap:9093**

   If you are going to throttle replication, you can also pass the **--throttle** option with an inter-broker throttled rate in bytes per second. For example:

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
     <cluster_name>-kafka-bootstrap:9093 \
     --command-config /tmp/config.properties \
     --reassignment-json-file /tmp/reassignment.json \
     --throttle 5000000 \
     --execute
   ```

This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

If you need to change the throttle during reassignment, you can use the same command with a different throttled rate. For example:

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

7. Verify that the reassignment has completed using the **kafka-reassign-partitions.sh** command line tool from any of the broker pods. This is the same command as the previous step, but with the **--verify** option instead of the **--execute** option.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

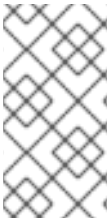
   The reassignment has finished when the **--verify** command reports that each of the partitions being moved has completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles.

8. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

12.4. REASSIGNING PARTITIONS BEFORE REMOVING BROKERS

Use a reassignment file generated by the **kafka-reassign-partitions.sh** tool to reassign partitions before decreasing the number of brokers in a Kafka cluster. The reassignment file must describe how partitions are reassigned to the remaining brokers in the Kafka cluster. You apply the reassignment specified in the file to the brokers and then verify the new partition assignments. Brokers in the highest numbered pods are removed first.

This procedure describes a secure scaling process that uses TLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.



NOTE

Though you can use the **kafka-reassign-partitions.sh** tool, Cruise Control is recommended for automated partition reassignments and cluster rebalancing . Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

Prerequisites

- You have a running Kafka cluster based on a **Kafka** resource configured with internal TLS encryption and mTLS authentication.

- You have generated a reassignment JSON file named **reassignment.json**.
- You are running an interactive pod container that is connected to the running Kafka broker.
- You are connected as a **KafkaUser** configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

See Generating reassignment JSON files.

Procedure

1. If you haven't done so, run an interactive pod container to generate a reassignment JSON file named **reassignment.json**.
2. Copy the **reassignment.json** file to the interactive pod container.

   ```
   oc cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
   ```

   Replace *<interactive_pod_name>* with the name of the pod.
3. Start a shell process in the interactive pod container.

   ```
   oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
   ```

   Replace *<namespace>* with the OpenShift namespace where the pod is running.
4. Run the partition reassignment using the **kafka-reassign-partitions.sh** script from the interactive pod container.

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --execute
   ```

   Replace *<cluster_name>* with the name of your Kafka cluster. For example, **my-cluster-kafka-bootstrap:9093**

   If you are going to throttle replication, you can also pass the **--throttle** option with an inter-broker throttled rate in bytes per second. For example:

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
     <cluster_name>-kafka-bootstrap:9093 \
     --command-config /tmp/config.properties \
     --reassignment-json-file /tmp/reassignment.json \
     --throttle 5000000 \
     --execute
   ```

   This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

   If you need to change the throttle during reassignment, you can use the same command with a different throttled rate. For example:

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

5. Verify that the reassignment has completed using the **kafka-reassign-partitions.sh** command line tool from any of the broker pods. This is the same command as the previous step, but with the **--verify** option instead of the **--execute** option.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

   The reassignment has finished when the **--verify** command reports that each of the partitions being moved has completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles.

6. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

7. When all the partition reassignments have finished, the brokers being removed should not have responsibility for any of the partitions in the cluster. You can verify this by checking that the broker's data log directory does not contain any live partition logs. If the log directory on the broker contains a directory that does not match the extended regular expression **\.[a-z0-9]-delete$**, the broker still has live partitions and should not be stopped.
   You can check this by executing the command:

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  /bin/bash -c \
  "ls -l /var/lib/kafka/kafka-log_<n>_ | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-
delete$'"
```

   where *n* is the number of the pods being deleted.

   If the above command prints any output then the broker still has live partitions. In this case, either the reassignment has not finished or the reassignment JSON file was incorrect.

8. When you have confirmed that the broker has no live partitions, you can edit the **Kafka.spec.kafka.replicas** property of your **Kafka** resource to reduce the number of brokers.

12.5. CHANGING THE REPLICATION FACTOR OF TOPICS

To change the replication factor of topics in a Kafka cluster, use the **kafka-reassign-partitions.sh** tool. This can be done by running the tool from an interactive pod container that is connected to the Kafka cluster, and using a reassignment file to describe how the topic replicas should be changed.

This procedure describes a secure process that uses TLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

Prerequisites

- You have a running Kafka cluster based on a **Kafka** resource configured with internal TLS encryption and mTLS authentication.
- You are running an interactive pod container that is connected to the running Kafka broker.
- You have generated a reassignment JSON file named **reassignment.json**.
- You are connected as a **KafkaUser** configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

See Generating reassignment JSON files.

In this procedure, a topic called **my-topic** has 4 replicas and we want to reduce it to 3. A JSON file named **topics.json** specifies the topic, and was used to generate the **reassignment.json** file.

Example JSON file specifies my-topic

```
{
  "version": 1,
  "topics": [
    { "topic": "my-topic"}
  ]
}
```

Procedure

1. If you haven't done so, run an interactive pod container to generate a reassignment JSON file named **reassignment.json**.

   #### Example reassignment JSON file showing the current and proposed replica assignment

   ```
   Current partition replica assignment
   {"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[3,4,2,0],"log_dirs":
   ["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[0,2,3,1],"log_dirs":
   ["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[1,3,0,4],"log_dirs":
   ["any","any","any","any"]}]}

   Proposed partition reassignment configuration
   {"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2,3],"log_dirs":
   ["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3,4],"log_dirs":
   ["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,4,0],"log_dirs":
   ["any","any","any","any"]}]}
   ```

   Save a copy of this file locally in case you need to revert the changes later on.

2. Edit the **reassignment.json** to remove a replica from each partition.
   For example use **jq** to remove the last replica in the list for each partition of the topic:

   #### Removing the last topic replica for each partition

   ```
   jq '.partitions[].replicas |= del(.[-1])' reassignment.json > reassignment.json
   ```

   #### Example reassignment file showing the updated replicas

```
{"version":1,"partitions":[{"topic":"my-topic","partition":0,"replicas":[0,1,2],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":
["any","any","any","any"]},{"topic":"my-topic","partition":2,"replicas":[2,3,4],"log_dirs":
["any","any","any","any"]}]}
```

3. Copy the **reassignment.json** file to the interactive pod container.

```
oc cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
```

Replace *<interactive_pod_name>* with the name of the pod.

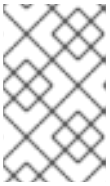
4. Start a shell process in the interactive pod container.

```
oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

Replace *<namespace>* with the OpenShift namespace where the pod is running.

5. Make the topic replica change using the **kafka-reassign-partitions.sh** script from the interactive pod container.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
 <cluster_name>-kafka-bootstrap:9093 \
 --command-config /tmp/config.properties \
 --reassignment-json-file /tmp/reassignment.json \
 --execute
```



NOTE

Removing replicas from a broker does not require any inter-broker data movement, so there is no need to throttle replication. If you are adding replicas, then you may want to change the throttle rate.

6. Verify that the change to the topic replicas has completed using the **kafka-reassign-partitions.sh** command line tool from any of the broker pods. This is the same command as the previous step, but with the **--verify** option instead of the **--execute** option.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

The reassignment has finished when the **--verify** command reports that each of the partitions being moved has completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles.

7. Run the **bin/kafka-topics.sh** command with the **--describe** option to see the results of the change to the topics.

```
bin/kafka-topics.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --describe
```

Results of reducing the number of replicas for a topic

```
my-topic  Partition: 0  Leader: 0  Replicas: 0,1,2 Isr: 0,1,2
my-topic  Partition: 1  Leader: 2  Replicas: 1,2,3 Isr: 1,2,3
my-topic  Partition: 2  Leader: 3  Replicas: 2,3,4 Isr: 2,3,4
```

CHAPTER 13. USING AMQ STREAMS OPERATORS

Use the AMQ Streams operators to manage your Kafka cluster, and Kafka topics and users.

13.1. WATCHING NAMESPACES WITH AMQ STREAMS OPERATORS

Operators watch and manage AMQ Streams resources in namespaces. The Cluster Operator can watch a single namespace, multiple namespaces, or all namespaces in an OpenShift cluster. The Topic Operator and User Operator can watch a single namespace.

- The Cluster Operator watches for **Kafka** resources
- The Topic Operator watches for **KafkaTopic** resources
- The User Operator watches for **KafkaUser** resources

The Topic Operator and the User Operator can only watch a single Kafka cluster in a namespace. And they can only be connected to a single Kafka cluster.

If multiple Topic Operators watch the same namespace, name collisions and topic deletion can occur. This is because each Kafka cluster uses Kafka topics that have the same name (such as **__consumer_offsets**). Make sure that only one Topic Operator watches a given namespace.

When using multiple User Operators with a single namespace, a user with a given username can exist in more than one Kafka cluster.

If you deploy the Topic Operator and User Operator using the Cluster Operator, they watch the Kafka cluster deployed by the Cluster Operator by default. You can also specify a namespace using **watchedNamespace** in the operator configuration.

For a standalone deployment of each operator, you specify a namespace and connection to the Kafka cluster to watch in the configuration.

13.2. USING THE CLUSTER OPERATOR

Use the Cluster Operator to deploy a Kafka cluster and other Kafka components.

13.2.1. Role-Based Access Control (RBAC) resources

The Cluster Operator creates and manages RBAC resources for AMQ Streams components that need access to OpenShift resources.

For the Cluster Operator to function, it needs permission within the OpenShift cluster to interact with Kafka resources, such as **Kafka** and **KafkaConnect**, as well as managed resources like **ConfigMap**, **Pod**, **Deployment**, **StatefulSet**, and **Service**.

Permission is specified through OpenShift role-based access control (RBAC) resources:

- **ServiceAccount**
- **Role** and **ClusterRole**
- **RoleBinding** and **ClusterRoleBinding**

13.2.1.1. Delegating privileges to AMQ Streams components

The Cluster Operator runs under a service account called **strimzi-cluster-operator**. It is assigned cluster roles that give it permission to create the RBAC resources for AMQ Streams components. Role bindings associate the cluster roles with the service account.

OpenShift prevents components operating under one **ServiceAccount** from granting another **ServiceAccount** privileges that the granting **ServiceAccount** does not have. Because the Cluster Operator creates the **RoleBinding** and **ClusterRoleBinding** RBAC resources needed by the resources it manages, it requires a role that gives it the same privileges.

The following tables describe the RBAC resources created by the Cluster Operator.

Table 13.1. **ServiceAccount** resources

Name	Used by
***<cluster_name>*-kafka**	Kafka broker pods
***<cluster_name>*-zookeeper**	ZooKeeper pods
***<cluster_name>*-cluster-connect**	Kafka Connect pods
***<cluster_name>*-mirror-maker**	MirrorMaker pods
***<cluster_name>*-mirrormaker2**	MirrorMaker 2 pods
***<cluster_name>*-bridge**	Kafka Bridge pods
***<cluster_name>*-entity-operator**	Entity Operator

Table 13.2. **ClusterRole** resources

Name	Used by
strimzi-cluster-operator-namespaced	Cluster Operator
strimzi-cluster-operator-global	Cluster Operator
strimzi-cluster-operator-leader-election	Cluster Operator
strimzi-kafka-broker	Cluster Operator, rack feature (when used)
strimzi-entity-operator	Cluster Operator, Topic Operator, User Operator
strimzi-kafka-client	Cluster Operator, Kafka clients for rack awareness

Table 13.3. **ClusterRoleBinding** resources

Name	Used by
strimzi-cluster-operator	Cluster Operator
strimzi-cluster-operator-kafka-broker-delegation	Cluster Operator, Kafka brokers for rack awareness
strimzi-cluster-operator-kafka-client-delegation	Cluster Operator, Kafka clients for rack awareness

Table 13.4. **RoleBinding** resources

Name	Used by
strimzi-cluster-operator	Cluster Operator
strimzi-cluster-operator-kafka-broker-delegation	Cluster Operator, Kafka brokers for rack awareness

13.2.1.2. Running the Cluster Operator using a ServiceAccount

The Cluster Operator is best run using a **ServiceAccount**:

Example ServiceAccount for the Cluster Operator

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
```

The **Deployment** of the operator then needs to specify this in its **spec.template.spec.serviceAccountName**:

Partial example of Deployment for the Cluster Operator

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 1
  selector:
    matchLabels:
      name: strimzi-cluster-operator
```

```
      strimzi.io/kind: cluster-operator
  template:
    # ...
```

Note line 12, where **strimzi-cluster-operator** is specified as the **serviceAccountName**.

13.2.1.3. ClusterRole resources

The Cluster Operator uses **ClusterRole** resources to provide the necessary access to resources. Depending on the OpenShift cluster setup, a cluster administrator might be needed to create the cluster roles.

NOTE

Cluster administrator rights are only needed for the creation of **ClusterRole** resources. The Cluster Operator will not run under a cluster admin account.

ClusterRole resources follow the *principle of least privilege* and contain only those privileges needed by the Cluster Operator to operate the cluster of the Kafka component. The first set of assigned privileges allow the Cluster Operator to manage OpenShift resources such as **StatefulSet**, **Deployment**, **Pod**, and **ConfigMap**.

All cluster roles are required by the Cluster Operator in order to delegate privileges.

The Cluster Operator uses the **strimzi-cluster-operator-namespaced** and **strimzi-cluster-operator-global** cluster roles to grant permission at the namespace-scoped resources level and cluster-scoped resources level.

ClusterRole with namespaced resources for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-namespaced
  labels:
    app: strimzi
rules:
  # Resources in this role are used by the operator based on an operand being deployed in some
namespace. When needed, you
  # can deploy the operator as a cluster-wide operator. But grant the rights listed in this role only on
the namespaces
  # where the operands will be deployed. That way, you can limit the access the operator has to other
namespaces where it
  # does not manage any clusters.
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to access and manage rolebindings to grant Strimzi components
cluster permissions
      - rolebindings
    verbs:
      - get
      - list
      - watch
      - create
```

```
    - delete
    - patch
    - update
  - apiGroups:
    - "rbac.authorization.k8s.io"
    resources:
    # The cluster operator needs to access and manage roles to grant the entity operator permissions
    - roles
    verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - ""
    resources:
    # The cluster operator needs to access and delete pods, this is to allow it to monitor pod health and coordinate rolling updates
    - pods
    # The cluster operator needs to access and manage service accounts to grant Strimzi components cluster permissions
    - serviceaccounts
    # The cluster operator needs to access and manage config maps for Strimzi components configuration
    - configmaps
    # The cluster operator needs to access and manage services and endpoints to expose Strimzi components to network traffic
    - services
    - endpoints
    # The cluster operator needs to access and manage secrets to handle credentials
    - secrets
    # The cluster operator needs to access and manage persistent volume claims to bind them to Strimzi components for persistent data
    - persistentvolumeclaims
    verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - "apps"
    resources:
    # The cluster operator needs to access and manage deployments to run deployment based Strimzi components
    - deployments
    - deployments/scale
    - deployments/status
    # The cluster operator needs to access and manage stateful sets to run stateful sets based Strimzi components
    - statefulsets
```

```
    # The cluster operator needs to access replica-sets to manage Strimzi components and to
determine error states
    - replicasets
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - "" # legacy core events api, used by topic operator
    - "events.k8s.io" # new events api, used by cluster operator
  resources:
    # The cluster operator needs to be able to create events and delegate permissions to do so
    - events
  verbs:
    - create
- apiGroups:
    # Kafka Connect Build on OpenShift requirement
    - build.openshift.io
  resources:
    - buildconfigs
    - buildconfigs/instantiate
    - builds
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - networking.k8s.io
  resources:
    # The cluster operator needs to access and manage network policies to lock down
communication between Strimzi components
    - networkpolicies
    # The cluster operator needs to access and manage ingresses which allow external access to the
services in a cluster
    - ingresses
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - route.openshift.io
  resources:
    # The cluster operator needs to access and manage routes to expose Strimzi components for
external access
```

```
      - routes
      - routes/custom-host
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - image.openshift.io
    resources:
      # The cluster operator needs to verify the image stream when used for Kafka Connect image build
      - imagestreams
    verbs:
      - get
  - apiGroups:
      - policy
    resources:
      # The cluster operator needs to access and manage pod disruption budgets this limits the number
of concurrent disruptions
      # that a Strimzi component experiences, allowing for higher availability
      - poddisruptionbudgets
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
```

ClusterRole with cluster-scoped resources for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-global
  labels:
    app: strimzi
rules:
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to create and manage cluster role bindings in the case of an install
where a user
      # has specified they want their cluster role bindings generated
      - clusterrolebindings
    verbs:
      - get
      - list
      - watch
      - create
      - delete
```

```
    - patch
    - update
  - apiGroups:
    - storage.k8s.io
    resources:
    # The cluster operator requires "get" permissions to view storage class details
    # This is because only a persistent volume of a supported storage class type can be resized
    - storageclasses
    verbs:
    - get
  - apiGroups:
    - ""
    resources:
    # The cluster operator requires "list" permissions to view all nodes in a cluster
    # The listing is used to determine the node addresses when NodePort access is configured
    # These addresses are then exposed in the custom resource states
    - nodes
    verbs:
    - list
```

The **strimzi-cluster-operator-leader-election** cluster role represents the permissions needed for the leader election.

ClusterRole with leader election permissions

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
rules:
  - apiGroups:
    - coordination.k8s.io
    resources:
    # The cluster operator needs to access and manage leases for leader election
    # The "create" verb cannot be used with "resourceNames"
    - leases
    verbs:
    - create
  - apiGroups:
    - coordination.k8s.io
    resources:
    # The cluster operator needs to access and manage leases for leader election
    - leases
    resourceNames:
    # The default RBAC files give the operator only access to the Lease resource names strimzi-cluster-operator
    # If you want to use another resource name or resource namespace, you have to configure the RBAC resources accordingly
    - strimzi-cluster-operator
    verbs:
    - get
    - list
    - watch
```

```
    - delete
    - patch
    - update
```

The **strimzi-kafka-broker** cluster role represents the access needed by the init container in Kafka pods that use rack awareness.

A role binding named **strimzi-*<cluster_name>*-kafka-init** grants the ***<cluster_name>*-kafka** service account access to nodes within a cluster using the **strimzi-kafka-broker** role. If the rack feature is not used and the cluster is not exposed through **nodeport**, no binding is created.

ClusterRole for the Cluster Operator allowing it to delegate access to OpenShift nodes to the Kafka broker pods

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-kafka-broker
  labels:
    app: strimzi
rules:
  - apiGroups:
      - ""
    resources:
      # The Kafka Brokers require "get" permissions to view the node they are on
      # This information is used to generate a Rack ID that is used for High Availability configurations
      - nodes
    verbs:
      - get
```

The **strimzi-entity-operator** cluster role represents the access needed by the Topic Operator and User Operator.

The Topic Operator produces OpenShift events with status information, so the ***<cluster_name>*-entity-operator** service account is bound to the **strimzi-entity-operator** role, which grants this access via the **strimzi-entity-operator** role binding.

ClusterRole for the Cluster Operator allowing it to delegate access to events to the Topic and User Operators

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-entity-operator
  labels:
    app: strimzi
rules:
  - apiGroups:
      - "kafka.strimzi.io"
    resources:
      # The entity operator runs the KafkaTopic assembly operator, which needs to access and manage
KafkaTopic resources
      - kafkatopics
      - kafkatopics/status
      # The entity operator runs the KafkaUser assembly operator, which needs to access and manage
```

```
  KafkaUser resources
      - kafkausers
      - kafkausers/status
    verbs:
      - get
      - list
      - watch
      - create
      - patch
      - update
      - delete
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      # The entity operator needs to be able to create events
      - create
  - apiGroups:
      - ""
    resources:
      # The entity operator user-operator needs to access and manage secrets to store generated
credentials
      - secrets
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
```

The **strimzi-kafka-client** cluster role represents the access needed by Kafka clients that use rack awareness.

ClusterRole for the Cluster Operator allowing it to delegate access to OpenShift nodes to the Kafka client-based pods

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-kafka-client
  labels:
    app: strimzi
rules:
  - apiGroups:
      - ""
    resources:
      # The Kafka clients (Connect, Mirror Maker, etc.) require "get" permissions to view the node they
are on
      # This information is used to generate a Rack ID (client.rack option) that is used for consuming
from the closest
      # replicas when enabled
```

```
    - nodes
  verbs:
    - get
```

13.2.1.4. ClusterRoleBinding resources

The Cluster Operator uses **ClusterRoleBinding** and **RoleBinding** resources to associate its **ClusterRole** with its **ServiceAccount**: Cluster role bindings are required by cluster roles containing cluster-scoped resources.

Example ClusterRoleBinding for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-cluster-operator-global
  apiGroup: rbac.authorization.k8s.io
```

Cluster role bindings are also needed for the cluster roles used in delegating privileges:

Example ClusterRoleBinding for the Cluster Operator and Kafka broker rack awareness

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator-kafka-broker-delegation
  labels:
    app: strimzi
# The Kafka broker cluster role must be bound to the cluster operator service account so that it can delegate the cluster role to the Kafka brokers.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-broker
  apiGroup: rbac.authorization.k8s.io
```

Example ClusterRoleBinding for the Cluster Operator and Kafka client rack awareness

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

```
  name: strimzi-cluster-operator-kafka-client-delegation
  labels:
    app: strimzi
# The Kafka clients cluster role must be bound to the cluster operator service account so that it can delegate the
# cluster role to the Kafka clients using it for consuming from closest replica.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-client
  apiGroup: rbac.authorization.k8s.io
```

Cluster roles containing only namespaced resources are bound using role bindings only.

Example RoleBinding for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-cluster-operator-namespaced
  apiGroup: rbac.authorization.k8s.io
```

Example RoleBinding for the Cluster Operator and Kafka broker rack awareness

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-entity-operator-delegation
  labels:
    app: strimzi
# The Entity Operator cluster role must be bound to the cluster operator service account so that it can delegate the cluster role to the Entity Operator.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-entity-operator
  apiGroup: rbac.authorization.k8s.io
```

13.2.2. ConfigMap for Cluster Operator logging

Cluster Operator logging is configured through a **ConfigMap** named **strimzi-cluster-operator**.

A **ConfigMap** containing logging configuration is created when installing the Cluster Operator. This **ConfigMap** is described in the file **install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml**. You configure Cluster Operator logging by changing the data field **log4j2.properties** in this **ConfigMap**.

To update the logging configuration, you can edit the **050-ConfigMap-strimzi-cluster-operator.yaml** file and then run the following command:

```
oc create -f install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml
```

Alternatively, edit the **ConfigMap** directly:

```
oc edit configmap strimzi-cluster-operator
```

To change the frequency of the reload interval, set a time in seconds in the **monitorInterval** option in the created **ConfigMap**.

If the **ConfigMap** is missing when the Cluster Operator is deployed, the default logging values are used.

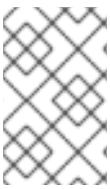
If the **ConfigMap** is accidentally deleted after the Cluster Operator is deployed, the most recently loaded logging configuration is used. Create a new **ConfigMap** to load a new logging configuration.

NOTE

Do not remove the **monitorInterval** option from the **ConfigMap**.

13.2.3. Configuring the Cluster Operator with environment variables

You can configure the Cluster Operator using environment variables. The supported environment variables are listed here.



NOTE

The environment variables are specified for the container image of the Cluster Operator in its **Deployment** configuration file. (**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml**)

STRIMZI_NAMESPACE

A comma-separated list of namespaces that the operator operates in. When not set, set to empty string, or set to *, the Cluster Operator operates in all namespaces.
The Cluster Operator deployment might use the downward API to set this automatically to the namespace the Cluster Operator is deployed in.

Example configuration for Cluster Operator namespaces

```
env:
  - name: STRIMZI_NAMESPACE
    valueFrom:
```

```
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_FULL_RECONCILIATION_INTERVAL_MS

Optional, default is 120000 ms. The interval between periodic reconciliations, in milliseconds.

STRIMZI_OPERATION_TIMEOUT_MS

Optional, default 300000 ms. The timeout for internal operations, in milliseconds. Increase this value when using AMQ Streams on clusters where regular OpenShift operations take longer than usual (because of slow downloading of Docker images, for example).

STRIMZI_ZOOKEEPER_ADMIN_SESSION_TIMEOUT_MS

Optional, default 10000 ms. The session timeout for the Cluster Operator's ZooKeeper admin client, in milliseconds. Increase the value if ZooKeeper requests from the Cluster Operator are regularly failing due to timeout issues. There is a maximum allowed session time set on the ZooKeeper server side via the **maxSessionTimeout** config. By default, the maximum session timeout value is 20 times the default **tickTime** (whose default is 2000) at 40000 ms. If you require a higher timeout, change the **maxSessionTimeout** ZooKeeper server configuration value.

STRIMZI_OPERATIONS_THREAD_POOL_SIZE

Optional, default 10. The worker thread pool size, which is used for various asynchronous and blocking operations that are run by the Cluster Operator.

STRIMZI_OPERATOR_NAME

Optional, defaults to the pod's hostname. The operator name identifies the AMQ Streams instance when emitting OpenShift events.

STRIMZI_OPERATOR_NAMESPACE

The name of the namespace where the Cluster Operator is running. Do not configure this variable manually. Use the downward API.

```
env:
  - name: STRIMZI_OPERATOR_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_OPERATOR_NAMESPACE_LABELS

Optional. The labels of the namespace where the AMQ Streams Cluster Operator is running. Use namespace labels to configure the namespace selector in network policies. Network policies allow the AMQ Streams Cluster Operator access only to the operands from the namespace with these labels. When not set, the namespace selector in network policies is configured to allow access to the Cluster Operator from any namespace in the OpenShift cluster.

```
env:
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS
    value: label1=value1,label2=value2
```

STRIMZI_LABELS_EXCLUSION_PATTERN

Optional, default regex pattern is **^app.kubernetes.io/(?!part-of).***. The regex exclusion pattern used to filter labels propagation from the main custom resource to its subresources. The labels exclusion filter is not applied to labels in template sections such as **spec.kafka.template.pod.metadata.labels**.

```
env:
  - name: STRIMZI_LABELS_EXCLUSION_PATTERN
    value: "^key1.*"
```

STRIMZI_CUSTOM_{COMPONENT_NAME}_LABELS

Optional. One or more custom labels to apply to all the pods created by the **{COMPONENT_NAME}** custom resource. The Cluster Operator labels the pods when the custom resource is created or is next reconciled.
Labels can be applied to the following components:

- **KAFKA**
- **KAFKA_CONNECT**
- **KAFKA_CONNECT_BUILD**
- **ZOOKEEPER**
- **ENTITY_OPERATOR**
- **KAFKA_MIRROR_MAKER2**
- **KAFKA_MIRROR_MAKER**
- **CRUISE_CONTROL**
- **KAFKA_BRIDGE**
- **KAFKA_EXPORTER**

STRIMZI_CUSTOM_RESOURCE_SELECTOR

Optional. The label selector to filter the custom resources handled by the Cluster Operator. The operator will operate only on those custom resources that have the specified labels set. Resources without these labels will not be seen by the operator. The label selector applies to **Kafka**, **KafkaConnect**, **KafkaBridge**, **KafkaMirrorMaker**, and **KafkaMirrorMaker2** resources. **KafkaRebalance** and **KafkaConnector** resources are operated only when their corresponding Kafka and Kafka Connect clusters have the matching labels.

```
env:
  - name: STRIMZI_CUSTOM_RESOURCE_SELECTOR
    value: label1=value1,label2=value2
```

STRIMZI_KAFKA_IMAGES

Required. The mapping from the Kafka version to the corresponding Docker image containing a Kafka broker for that version. The required syntax is whitespace or comma-separated ***<version>=<image>*** pairs. For example **3.3.1=registry.redhat.io/amq-streams/kafka-33-rhel8:2.4.0, 3.4.0=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0**. This is used when a **Kafka.spec.kafka.version** property is specified but not the **Kafka.spec.kafka.image** in the **Kafka** resource.

STRIMZI_DEFAULT_KAFKA_INIT_IMAGE

Optional, default **registry.redhat.io/amq-streams/strimzi-rhel8-operator:2.4.0**. The image name to use as default for the init container if no image is specified as the **kafka-init-image** in the **Kafka** resource. The init container is started before the broker for initial configuration work, such as rack

support.

STRIMZI_KAFKA_CONNECT_IMAGES

Required. The mapping from the Kafka version to the corresponding Docker image of Kafka Connect for that version. The required syntax is whitespace or comma-separated ***<version>=<image>*** pairs. For example **3.3.1=registry.redhat.io/amq-streams/kafka-33-rhel8:2.4.0, 3.4.0=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0**. This is used when a **KafkaConnect.spec.version** property is specified but not the **KafkaConnect.spec.image**.

STRIMZI_KAFKA_MIRROR_MAKER_IMAGES

Required. The mapping from the Kafka version to the corresponding Docker image of MirrorMaker for that version. The required syntax is whitespace or comma-separated ***<version>=<image>*** pairs. For example **3.3.1=registry.redhat.io/amq-streams/kafka-33-rhel8:2.4.0, 3.4.0=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0**. This is used when a **KafkaMirrorMaker.spec.version** property is specified but not the **KafkaMirrorMaker.spec.image**.

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq-streams/strimzi-rhel8-operator:2.4.0**. The image name to use as the default when deploying the Topic Operator if no image is specified as the **Kafka.spec.entityOperator.topicOperator.image** in the **Kafka** resource.

STRIMZI_DEFAULT_USER_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq-streams/strimzi-rhel8-operator:2.4.0**. The image name to use as the default when deploying the User Operator if no image is specified as the **Kafka.spec.entityOperator.userOperator.image** in the **Kafka** resource.

STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0**. The image name to use as the default when deploying the sidecar container for the Entity Operator if no image is specified as the **Kafka.spec.entityOperator.tlsSidecar.image** in the **Kafka** resource. The sidecar provides TLS support.

STRIMZI_IMAGE_PULL_POLICY

Optional. The **ImagePullPolicy** that is applied to containers in all pods managed by the Cluster Operator. The valid values are **Always**, **IfNotPresent**, and **Never**. If not specified, the OpenShift defaults are used. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

STRIMZI_IMAGE_PULL_SECRETS

Optional. A comma-separated list of **Secret** names. The secrets referenced here contain the credentials to the container registries where the container images are pulled from. The secrets are specified in the **imagePullSecrets** property for all pods created by the Cluster Operator. Changing this list results in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

STRIMZI_KUBERNETES_VERSION

Optional. Overrides the OpenShift version information detected from the API server.

Example configuration for OpenShift version override

```
env:
  - name: STRIMZI_KUBERNETES_VERSION
    value: |
           major=1
           minor=16
           gitVersion=v1.16.2
           gitCommit=c97fe5036ef3df2967d086711e6c0c405941e14b
           gitTreeState=clean
           buildDate=2019-10-15T19:09:08Z
```

```
goVersion=go1.12.10
compiler=gc
platform=linux/amd64
```

KUBERNETES_SERVICE_DNS_DOMAIN

Optional. Overrides the default OpenShift DNS domain name suffix.
By default, services assigned in the OpenShift cluster have a DNS domain name that uses the default suffix **cluster.local**.

For example, for broker *kafka-0*:

```
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers.<namespace>.svc.cluster.local
```

The DNS domain name is added to the Kafka broker certificates used for hostname verification.

If you are using a different DNS domain name suffix in your cluster, change the **KUBERNETES_SERVICE_DNS_DOMAIN** environment variable from the default to the one you are using in order to establish a connection with the Kafka brokers.

STRIMZI_CONNECT_BUILD_TIMEOUT_MS

Optional, default 300000 ms. The timeout for building new Kafka Connect images with additional connectors, in milliseconds. Consider increasing this value when using AMQ Streams to build container images containing many connectors or using a slow container registry.

STRIMZI_NETWORK_POLICY_GENERATION

Optional, default **true**. Network policy for resources. Network policies allow connections between Kafka components.
Set this environment variable to **false** to disable network policy generation. You might do this, for example, if you want to use custom network policies. Custom network policies allow more control over maintaining the connections between components.

STRIMZI_DNS_CACHE_TTL

Optional, default **30**. Number of seconds to cache successful name lookups in local DNS resolver. Any negative value means cache forever. Zero means do not cache, which can be useful for avoiding connection errors due to long caching policies being applied.

STRIMZI_POD_SET_RECONCILIATION_ONLY

Optional, default **false**. When set to **true**, the Cluster Operator reconciles only the **StrimziPodSet** resources and any changes to the other custom resources (**Kafka**, **KafkaConnect**, and so on) are ignored. This mode is useful for ensuring that your pods are recreated if needed, but no other changes happen to the clusters.

STRIMZI_FEATURE_GATES

Optional. Enables or disables the features and functionality controlled by feature gates.

STRIMZI_POD_SECURITY_PROVIDER_CLASS

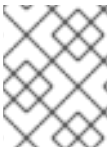
Optional. Configuration for the pluggable **PodSecurityProvider** class, which can be used to provide the security context configuration for Pods and containers.

13.2.3.1. Leader election environment variables

Use leader election environment variables when running additional Cluster Operator replicas. You might run additional replicas to safeguard against disruption caused by major failure.

STRIMZI_LEADER_ELECTION_ENABLED

Optional, disabled (**false**) by default. Enables or disables leader election, which allows additional Cluster Operator replicas to run on standby.



NOTE

Leader election is disabled by default. It is only enabled when applying this environment variable on installation.

STRIMZI_LEADER_ELECTION_LEASE_NAME

Required when leader election is enabled. The name of the OpenShift **Lease** resource that is used for the leader election.

STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE

Required when leader election is enabled. The namespace where the OpenShift **Lease** resource used for leader election is created. You can use the downward API to configure it to the namespace where the Cluster Operator is deployed.

```
env:
  - name: STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_LEADER_ELECTION_IDENTITY

Required when leader election is enabled. Configures the identity of a given Cluster Operator instance used during the leader election. The identity must be unique for each operator instance. You can use the downward API to configure it to the name of the pod where the Cluster Operator is deployed.

```
env:
  - name: STRIMZI_LEADER_ELECTION_IDENTITY
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
```

STRIMZI_LEADER_ELECTION_LEASE_DURATION_MS

Optional, default 15000 ms. Specifies the duration the acquired lease is valid.

STRIMZI_LEADER_ELECTION_RENEW_DEADLINE_MS

Optional, default 10000 ms. Specifies the period the leader should try to maintain leadership.

STRIMZI_LEADER_ELECTION_RETRY_PERIOD_MS

Optional, default 2000 ms. Specifies the frequency of updates to the lease lock by the leader.

13.2.3.2. Restricting Cluster Operator access with network policy

Use the **STRIMZI_OPERATOR_NAMESPACE_LABELS** environment variable to establish network policy for the Cluster Operator using namespace labels.

The Cluster Operator can run in the same namespace as the resources it manages, or in a separate namespace. By default, the **STRIMZI_OPERATOR_NAMESPACE** environment variable is configured to use the downward API to find the namespace the Cluster Operator is running in. If the Cluster

Operator is running in the same namespace as the resources, only local access is required and allowed by AMQ Streams.

If the Cluster Operator is running in a separate namespace to the resources it manages, any namespace in the OpenShift cluster is allowed access to the Cluster Operator unless network policy is configured. By adding namespace labels, access to the Cluster Operator is restricted to the namespaces specified.

Network policy configured for the Cluster Operator deployment

```
#...
env:
  # ...
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS
    value: label1=value1,label2=value2
  #...
```

13.2.3.3. Setting the time interval for periodic reconciliation

Use the **STRIMZI_FULL_RECONCILIATION_INTERVAL_MS** variable to set the time interval for periodic reconciliations.

The Cluster Operator reacts to all notifications about applicable cluster resources received from the OpenShift cluster. If the operator is not running, or if a notification is not received for any reason, resources will get out of sync with the state of the running OpenShift cluster. In order to handle failovers properly, a periodic reconciliation process is executed by the Cluster Operator so that it can compare the state of the resources with the current cluster deployments in order to have a consistent state across all of them.

Additional resources

- Downward API

13.2.4. Configuring the Cluster Operator with default proxy settings

If you are running a Kafka cluster behind a HTTP proxy, you can still pass data in and out of the cluster. For example, you can run Kafka Connect with connectors that push and pull data from outside the proxy. Or you can use a proxy to connect with an authorization server.

Configure the Cluster Operator deployment to specify the proxy environment variables. The Cluster Operator accepts standard proxy configuration (**HTTP_PROXY**, **HTTPS_PROXY** and **NO_PROXY**) as environment variables. The proxy settings are applied to all AMQ Streams containers.

The format for a proxy address is *http://IP-ADDRESS:PORT-NUMBER*. To set up a proxy with a name and password, the format is *http://USERNAME:PASSWORD@IP-ADDRESS:PORT-NUMBER*.

Prerequisites

- You need an account with permission to create and manage **CustomResourceDefinition** and RBAC (**ClusterRole**, and **RoleBinding**) resources.

Procedure

1. To add proxy environment variables to the Cluster Operator, update its **Deployment** configuration (**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml**).

Example proxy configuration for the Cluster Operator

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
        # ...
        - name: "HTTP_PROXY"
          value: "http://proxy.com" (1)
        - name: "HTTPS_PROXY"
          value: "https://proxy.com" (2)
        - name: "NO_PROXY"
          value: "internal.com, other.domain.com" (3)
  # ...
```

1. Address of the proxy server.
2. Secure address of the proxy server.
3. Addresses for servers that are accessed directly as exceptions to the proxy server. The URLs are comma-separated.

Alternatively, edit the **Deployment** directly:

```
oc edit deployment strimzi-cluster-operator
```

2. If you updated the YAML file instead of editing the **Deployment** directly, apply the changes:

```
oc create -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
```

Additional resources

- Host aliases
- Designating AMQ Streams administrators

13.2.5. Running multiple Cluster Operator replicas with leader election

The default Cluster Operator configuration enables leader election. Use leader election to run multiple parallel replicas of the Cluster Operator. One replica is elected as the active leader and operates the deployed resources. The other replicas run in standby mode. When the leader stops or fails, one of the standby replicas is elected as the new leader and starts operating the deployed resources.

By default, AMQ Streams runs with a single Cluster Operator replica that is always the leader replica. When a single Cluster Operator replica stops or fails, OpenShift starts a new replica.

Running the Cluster Operator with multiple replicas is not essential. But it's useful to have replicas on

standby in case of large-scale disruptions. For example, suppose multiple worker nodes or an entire availability zone fails. This failure might cause the Cluster Operator pod and many Kafka pods to go down at the same time. If subsequent pod scheduling causes congestion through lack of resources, this can delay operations when running a single Cluster Operator.

13.2.5.1. Configuring Cluster Operator replicas

To run additional Cluster Operator replicas in standby mode, you will need to increase the number of replicas and enable leader election. To configure leader election, use the leader election environment variables.

To make the required changes, configure the following Cluster Operator installation files located in **install/cluster-operator/**:

- 060-Deployment-strimzi-cluster-operator.yaml
- 022-ClusterRole-strimzi-cluster-operator-role.yaml
- 022-RoleBinding-strimzi-cluster-operator.yaml

Leader election has its own **ClusterRole** and **RoleBinding** RBAC resources that target the namespace where the Cluster Operator is running, rather than the namespace it is watching.

The default deployment configuration creates a **Lease** resource called **strimzi-cluster-operator** in the same namespace as the Cluster Operator. The Cluster Operator uses leases to manage leader election. The RBAC resources provide the permissions to use the **Lease** resource. If you use a different **Lease** name or namespace, update the **ClusterRole** and **RoleBinding** files accordingly.

Prerequisites

- You need an account with permission to create and manage **CustomResourceDefinition** and RBAC (**ClusterRole**, and **RoleBinding**) resources.

Procedure

Edit the **Deployment** resource that is used to deploy the Cluster Operator, which is defined in the **060-Deployment-strimzi-cluster-operator.yaml** file.

1. Change the **replicas** property from the default (1) to a value that matches the required number of replicas.

   Increasing the number of Cluster Operator replicas

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: strimzi-cluster-operator
     labels:
       app: strimzi
   spec:
     replicas: 3
   ```

2. Check that the leader election **env** properties are set.
   If they are not set, configure them.

To enable leader election, **STRIMZI_LEADER_ELECTION_ENABLED** must be set to **true** (default).

In this example, the name of the lease is changed to **my-strimzi-cluster-operator**.

Configuring leader election environment variables for the Cluster Operator

```
# ...
spec
  containers:
    - name: strimzi-cluster-operator
      # ...
      env:
        - name: STRIMZI_LEADER_ELECTION_ENABLED
          value: "true"
        - name: STRIMZI_LEADER_ELECTION_LEASE_NAME
          value: "my-strimzi-cluster-operator"
        - name: STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
        - name: STRIMZI_LEADER_ELECTION_IDENTITY
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
```

For a description of the available environment variables, see Section 13.2.3.1, “Leader election environment variables”.

If you specified a different name or namespace for the **Lease** resource used in leader election, update the RBAC resources.

3. (optional) Edit the **ClusterRole** resource in the **022-ClusterRole-strimzi-cluster-operator-role.yaml** file.
   Update **resourceNames** with the name of the **Lease** resource.

   Updating the ClusterRole references to the lease

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
rules:
  - apiGroups:
      - coordination.k8s.io
    resourceNames:
      - my-strimzi-cluster-operator
# ...
```

4. (optional) Edit the **RoleBinding** resource in the **022-RoleBinding-strimzi-cluster-operator.yaml** file.
   Update **subjects.name** and **subjects.namespace** with the name of the **Lease** resource and the namespace where it was created.

Updating the RoleBinding references to the lease

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: my-strimzi-cluster-operator
    namespace: myproject
# ...
```

5. Deploy the Cluster Operator:

```
oc create -f install/cluster-operator -n myproject
```

6. Check the status of the deployment:

```
oc get deployments -n myproject
```

Output shows the deployment name and readiness

```
NAME                      READY  UP-TO-DATE  AVAILABLE
strimzi-cluster-operator  3/3    3           3
```

READY shows the number of replicas that are ready/expected. The deployment is successful when the **AVAILABLE** output shows the correct number of replicas.

13.2.6. FIPS support

Federal Information Processing Standards (FIPS) are standards for computer security and interoperability. When running AMQ Streams on a FIPS-enabled OpenShift cluster, the OpenJDK used in AMQ Streams container images automatically switches to FIPS mode. From version 2.4, AMQ Streams can run on FIPS-enabled OpenShift clusters without any changes or special configuration. It uses only the FIPS-compliant security libraries from the OpenJDK.

Minimum password length

When running in the FIPS mode, SCRAM-SHA-512 passwords need to be at least 32 characters long. From AMQ Streams 2.4, the default password length in AMQ Streams User Operator is set to 32 characters as well. If you have a Kafka cluster with custom configuration that uses a password length that is less than 32 characters, you need to update your configuration. If you have any users with passwords shorter than 32 characters, you need to regenerate a password with the required length. You can do that, for example, by deleting the user secret and waiting for the User Operator to create a new password with the appropriate length.

IMPORTANT

If you are using FIPS-enabled OpenShift clusters, you may experience higher memory consumption compared to regular OpenShift clusters. To avoid any issues, we suggest increasing the memory request to at least 512Mi.

Additional resources

- What are Federal Information Processing Standards (FIPS)

13.2.6.1. Disabling FIPS mode

AMQ Streams automatically switches to FIPS mode when running on a FIPS-enabled OpenShift cluster. Disable FIPS mode by setting the **FIPS_MODE** environment variable to **disabled** in the deployment configuration for the Cluster Operator. With FIPS mode disabled, AMQ Streams automatically disables FIPS in the OpenJDK for all components. With FIPS mode disabled, AMQ Streams is not FIPS compliant. The AMQ Streams operators, as well as all operands, run in the same way as if they were running on an OpenShift cluster without FIPS enabled.

Procedure

1. To disable the FIPS mode in the Cluster Operator, update its **Deployment** configuration (**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml**) and add the **FIPS_MODE** environment variable.

   Example FIPS configuration for the Cluster Operator

   ```
   apiVersion: apps/v1
   kind: Deployment
   spec:
     # ...
     template:
       spec:
         serviceAccountName: strimzi-cluster-operator
         containers:
           # ...
           env:
           # ...
           - name: "FIPS_MODE"
             value: "disabled" 1
     # ...
   ```

   1 Disables the FIPS mode.

   Alternatively, edit the **Deployment** directly:

   ```
   oc edit deployment strimzi-cluster-operator
   ```

2. If you updated the YAML file instead of editing the **Deployment** directly, apply the changes:

   ```
   oc apply -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
   ```

13.3. USING THE TOPIC OPERATOR

When you create, modify or delete a topic using the **KafkaTopic** resource, the Topic Operator ensures those changes are reflected in the Kafka cluster.

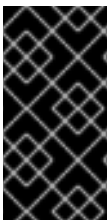
For more information on the **KafkaTopic** resource, see the **KafkaTopic** schema reference.

Deploying the Topic Operator

You can deploy the Topic Operator using the Cluster Operator or as a standalone operator. You would use a standalone Topic Operator with a Kafka cluster that is not managed by the Cluster Operator.

For deployment instructions, see the following:

- Deploying the Topic Operator using the Cluster Operator (recommended)
- Deploying the standalone Topic Operator



IMPORTANT

To deploy the standalone Topic Operator, you need to set environment variables to connect to a Kafka cluster. These environment variables do not need to be set if you are deploying the Topic Operator using the Cluster Operator as they will be set by the Cluster Operator.

13.3.1. Kafka topic resource

The **KafkaTopic** resource is used to configure topics, including the number of partitions and replicas.

The full schema for **KafkaTopic** is described in **KafkaTopic** schema reference.

13.3.1.1. Identifying a Kafka cluster for topic handling

A **KafkaTopic** resource includes a label that specifies the name of the Kafka cluster (derived from the name of the **Kafka** resource) to which it belongs.

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
  labels:
    strimzi.io/cluster: my-cluster
```

The label is used by the Topic Operator to identify the **KafkaTopic** resource and create a new topic, and also in subsequent handling of the topic.

If the label does not match the Kafka cluster, the Topic Operator cannot identify the **KafkaTopic** and the topic is not created.

13.3.1.2. Kafka topic usage recommendations

When working with topics, be consistent. Always operate on either **KafkaTopic** resources or topics directly in OpenShift. Avoid routinely switching between both methods for a given topic.

Use topic names that reflect the nature of the topic, and remember that names cannot be changed later.

If creating a topic in Kafka, use a name that is a valid OpenShift resource name, otherwise the Topic Operator will need to create the corresponding **KafkaTopic** with a name that conforms to the OpenShift rules.

NOTE

For information on the requirements for identifiers and names in OpenShift, refer to Object Names and IDs.

13.3.1.3. Kafka topic naming conventions

Kafka and OpenShift impose their own validation rules for the naming of topics in Kafka and **KafkaTopic.metadata.name** respectively. There are valid names for each which are invalid in the other.

Using the **spec.topicName** property, it is possible to create a valid topic in Kafka with a name that would be invalid for the Kafka topic in OpenShift.

The **spec.topicName** property inherits Kafka naming validation rules:

- The name must not be longer than 249 characters.
- Valid characters for Kafka topics are ASCII alphanumerics, **.**, _, and **-**.
- The name cannot be **.** or **..**, though **.** can be used in a name, such as **exampleTopic.** or **.exampleTopic**.

spec.topicName must not be changed.

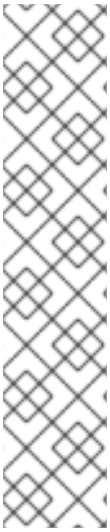
For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: topicName-1 1
  # ...
```

1 Upper case is invalid in OpenShift.

cannot be changed to:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: name-2
  # ...
```



NOTE

Some Kafka client applications, such as Kafka Streams, can create topics in Kafka programmatically. If those topics have names that are invalid OpenShift resource names, the Topic Operator gives them a valid **metadata.name** based on the Kafka name. Invalid characters are replaced and a hash is appended to the name. For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: mytopic---c55e57fe2546a33f9e603caf57165db4072e827e
spec:
  topicName: myTopic
  # ...
```

13.3.2. Topic Operator topic store

The Topic Operator uses Kafka to store topic metadata describing topic configuration as key-value pairs. The *topic store* is based on the Kafka Streams key-value mechanism, which uses Kafka topics to persist the state.

Topic metadata is cached in-memory and accessed locally within the Topic Operator. Updates from operations applied to the local in-memory cache are persisted to a backup topic store on disk. The topic store is continually synchronized with updates from Kafka topics or OpenShift **KafkaTopic** custom resources. Operations are handled rapidly with the topic store set up this way, but should the in-memory cache crash it is automatically repopulated from the persistent storage.

13.3.2.1. Internal topic store topics

Internal topics support the handling of topic metadata in the topic store.

__strimzi_store_topic

Input topic for storing the topic metadata

__strimzi-topic-operator-kstreams-topic-store-changelog

Retains a log of compacted topic store values

WARNING

Do not delete these topics, as they are essential to the running of the Topic Operator.

13.3.2.2. Migrating topic metadata from ZooKeeper

In previous releases of AMQ Streams, topic metadata was stored in ZooKeeper. The new process removes this requirement, bringing the metadata into the Kafka cluster, and under the control of the Topic Operator.

When upgrading to AMQ Streams 2.4, the transition to Topic Operator control of the topic store is seamless. Metadata is found and migrated from ZooKeeper, and the old store is deleted.

13.3.2.3. Downgrading to an AMQ Streams version that uses ZooKeeper to store topic metadata

If you are reverting back to a version of AMQ Streams earlier than 1.7, which uses ZooKeeper for the storage of topic metadata, you still downgrade your Cluster Operator to the previous version, then downgrade Kafka brokers and client applications to the previous Kafka version as standard.

However, you must also delete the topics that were created for the topic store using a **kafka-admin** command, specifying the bootstrap address of the Kafka cluster. For example:

```
oc run kafka-admin -ti --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

The command must correspond to the type of listener and authentication used to access the Kafka cluster.

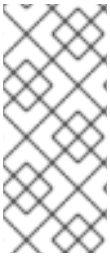
The Topic Operator will reconstruct the ZooKeeper topic metadata from the state of the topics in Kafka.

13.3.2.4. Topic Operator topic replication and scaling

The recommended configuration for topics managed by the Topic Operator is a topic replication factor of 3, and a minimum of 2 in-sync replicas.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10 (1)
  replicas: 3 (2)
  config:
    min.insync.replicas: 2 (3)
  #...
```

1. The number of partitions for the topic.
2. The number of replica topic partitions. Currently, this cannot be changed in the **KafkaTopic** resource, but it can be changed using the **kafka-reassign-partitions.sh** tool.
3. The minimum number of replica partitions that a message must be successfully written to, or an exception is raised.



NOTE

In-sync replicas are used in conjunction with the **acks** configuration for producer applications. The **acks** configuration determines the number of follower partitions a message must be replicated to before the message is acknowledged as successfully received. The Topic Operator runs with **acks=all**, whereby messages must be acknowledged by all in-sync replicas.

When scaling Kafka clusters by adding or removing brokers, replication factor configuration is not changed and replicas are not reassigned automatically. However, you can use the **kafka-reassign-partitions.sh** tool to change the replication factor, and manually reassign replicas to brokers.

Alternatively, though the integration of Cruise Control for AMQ Streams cannot change the replication factor for topics, the optimization proposals it generates for rebalancing Kafka include commands that transfer partition replicas and change partition leadership.

13.3.2.5. Handling changes to topics

A fundamental problem that the Topic Operator needs to solve is that there is no single source of truth: both the **KafkaTopic** resource and the Kafka topic can be modified independently of the Topic Operator. Complicating this, the Topic Operator might not always be able to observe changes at each end in real time. For example, when the Topic Operator is down.

To resolve this, the Topic Operator maintains information about each topic in the topic store. When a change happens in the Kafka cluster or OpenShift, it looks at both the state of the other system and the topic store in order to determine what needs to change to keep everything in sync. The same thing happens whenever the Topic Operator starts, and periodically while it is running.

For example, suppose the Topic Operator is not running, and a **KafkaTopic** called *my-topic* is created. When the Topic Operator starts, the topic store does not contain information on my-topic, so it can infer that the **KafkaTopic** was created after it was last running. The Topic Operator creates the topic corresponding to my-topic, and also stores metadata for my-topic in the topic store.

If you update Kafka topic configuration or apply a change through the **KafkaTopic** custom resource, the topic store is updated after the Kafka cluster is reconciled.

The topic store also allows the Topic Operator to manage scenarios where the topic configuration is changed in Kafka topics *and* updated through OpenShift **KafkaTopic** custom resources, as long as the changes are not incompatible. For example, it is possible to make changes to the same topic config key, but to different values. For incompatible changes, the Kafka configuration takes priority, and the **KafkaTopic** is updated accordingly.

NOTE

You can also use the **KafkaTopic** resource to delete topics using a **oc delete -f *KAFKA-TOPIC-CONFIG-FILE*** command. To be able to do this, **delete.topic.enable** must be set to **true** (default) in the **spec.kafka.config** of the Kafka resource.

Additional resources

- Downgrading AMQ Streams
- Section 12.1, “Partition reassignment tool overview”
- Chapter 11, *Rebalancing clusters using Cruise Control*

13.3.3. Configuring Kafka topics

Use the properties of the **KafkaTopic** resource to configure Kafka topics.

You can use **oc apply** to create or modify topics, and **oc delete** to delete existing topics.

For example:

- **oc apply -f *<topic_config_file>***
- **oc delete KafkaTopic *<topic_name>***

This procedure shows how to create a topic with 10 partitions and 2 replicas.

Before you start

It is important that you consider the following before making your changes:

- Kafka does not support decreasing the number of partitions.
- Increasing **spec.partitions** for topics with keys will change how records are partitioned, which can be particularly problematic when the topic uses *semantic partitioning*.
- AMQ Streams does not support making the following changes through the **KafkaTopic** resource:
  - Using **spec.replicas** to change the number of replicas that were initially specified
  - Changing topic names using **spec.topicName**

Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.
- A running Topic Operator (typically deployed with the Entity Operator).
- For deleting a topic, **delete.topic.enable=true** (default) in the **spec.kafka.config** of the **Kafka** resource.

Procedure

1. Configure the **KafkaTopic** resource.

   Example Kafka topic configuration

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaTopic
   metadata:
     name: orders
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     partitions: 10
     replicas: 2
   ```

   TIP

   When modifying a topic, you can get the current version of the resource using **oc get kafkatopic orders -o yaml**.

2. Create the **KafkaTopic** resource in OpenShift.

```
oc apply -f <topic_config_file>
```

3. Wait for the ready status of the topic to change to **True**:

```
oc get kafkatopics -o wide -w -n <namespace>
```

Kafka topic status

```
NAME        CLUSTER     PARTITIONS  REPLICATION FACTOR READY
my-topic-1  my-cluster  10          3                  True
my-topic-2  my-cluster  10          3
my-topic-3  my-cluster  10          3                  True
```

Topic creation is successful when the **READY** output shows **True**.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or from the Topic Operator logs.
Messages provide details on the reason for the current status.

```
oc get kafkatopics my-topic-2 -o yaml
```

Details on a topic with a NotReady status

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-13T10:14:43.351550Z"
    message: Number of partitions cannot be decreased
    reason: PartitionDecreaseException
    status: "True"
    type: NotReady
```

In this example, the reason the topic is not ready is because the original number of partitions was reduced in the **KafkaTopic** configuration. Kafka does not support this.

After resetting the topic configuration, the status shows the topic is ready.

```
oc get kafkatopics my-topic-2 -o wide -w -n <namespace>
```

Status update of the topic

```
NAME        CLUSTER     PARTITIONS  REPLICATION FACTOR READY
my-topic-2  my-cluster  10          3                  True
```

Fetching the details shows no messages

```
oc get kafkatopics my-topic-2 -o yaml
```

Details on a topic with a READY status

```
# ...
```

```
status:
  conditions:
  - lastTransitionTime: '2022-06-13T10:15:03.761084Z'
    status: 'True'
    type: Ready
```

13.3.4. Configuring the Topic Operator with resource requests and limits

You can allocate resources, such as CPU and memory, to the Topic Operator and set a limit on the amount of resources it can consume.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Update the Kafka cluster configuration in an editor, as required:

```
oc edit kafka MY-CLUSTER
```

2. In the **spec.entityOperator.topicOperator.resources** property in the **Kafka** resource, set the resource requests and limits for the Topic Operator.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # Kafka and ZooKeeper sections...
  entityOperator:
    topicOperator:
      resources:
        requests:
          cpu: "1"
          memory: 500Mi
        limits:
          cpu: "1"
          memory: 500Mi
```

3. Apply the new configuration to create or update the resource.

```
oc apply -f <kafka_configuration_file>
```

13.4. USING THE USER OPERATOR

When you create, modify or delete a user using the **KafkaUser** resource, the User Operator ensures those changes are reflected in the Kafka cluster.

For more information on the **KafkaUser** resource, see the **KafkaUser** schema reference.

Deploying the User Operator

You can deploy the User Operator using the Cluster Operator or as a standalone operator. You would use a standalone User Operator with a Kafka cluster that is not managed by the Cluster Operator.

For deployment instructions, see the following:

- Deploying the User Operator using the Cluster Operator (recommended)
- Deploying the standalone User Operator

IMPORTANT

To deploy the standalone User Operator, you need to set environment variables to connect to a Kafka cluster. These environment variables do not need to be set if you are deploying the User Operator using the Cluster Operator as they will be set by the Cluster Operator.

13.4.1. Configuring Kafka users

Use the properties of the **KafkaUser** resource to configure Kafka users.

You can use **oc apply** to create or modify users, and **oc delete** to delete existing users.

For example:

- **oc apply -f *<user_config_file>***
- **oc delete KafkaUser *<user_name>***

Users represent Kafka clients. When you configure Kafka users, you enable the user authentication and authorization mechanisms required by clients to access Kafka. The mechanism used must match the equivalent **Kafka** configuration. For more information on using **Kafka** and **KafkaUser** resources to secure access to Kafka brokers, see Securing access to Kafka brokers .

Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.
- A running User Operator (typically deployed with the Entity Operator).

Procedure

1. Configure the **KafkaUser** resource.
   This example specifies mTLS authentication and simple authorization using ACLs.

   Example Kafka user configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
```

```
    acls:
      # Example consumer Acls for topic my-topic using consumer group my-group
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Describe
          - Read
        host: "*"
      - resource:
          type: group
          name: my-group
          patternType: literal
        operations:
          - Read
        host: "*"
      # Example Producer Acls for topic my-topic
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Create
          - Describe
          - Write
        host: "*"
```

2. Create the **KafkaUser** resource in OpenShift.

```
oc apply -f <user_config_file>
```

3. Wait for the ready status of the user to change to **True**:

```
oc get kafkausers -o wide -w -n <namespace>
```

Kafka user status

```
NAME      CLUSTER     AUTHENTICATION  AUTHORIZATION READY
my-user-1  my-cluster  tls           simple       True
my-user-2  my-cluster  tls           simple
my-user-3  my-cluster  tls           simple       True
```

User creation is successful when the **READY** output shows **True**.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or User Operator logs.
Messages provide details on the reason for the current status.

```
oc get kafkausers my-user-2 -o yaml
```

Details on a user with a NotReady status

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:07:37.238065Z"
    message: Simple authorization ACL rules are configured but not supported in the
      Kafka cluster configuration.
    reason: InvalidResourceException
    status: "True"
    type: NotReady
```

In this example, the reason the user is not ready is because simple authorization is not enabled in the **Kafka** configuration.

Kafka configuration for simple authorization

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    authorization:
      type: simple
```

After updating the Kafka configuration, the status shows the user is ready.

```
oc get kafkausers my-user-2 -o wide -w -n <namespace>
```

Status update of the user

```
NAME       CLUSTER     AUTHENTICATION  AUTHORIZATION READY
my-user-2  my-cluster  tls             simple        True
```

Fetching the details shows no messages.

```
oc get kafkausers my-user-2 -o yaml
```

Details on a user with a READY status

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:33:40.166846Z"
    status: "True"
    type: Ready
```

13.4.2. Configuring the User Operator with resource requests and limits

You can allocate resources, such as CPU and memory, to the User Operator and set a limit on the amount of resources it can consume.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Update the Kafka cluster configuration in an editor, as required:

```
oc edit kafka MY-CLUSTER
```

2. In the **spec.entityOperator.userOperator.resources** property in the **Kafka** resource, set the resource requests and limits for the User Operator.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # Kafka and ZooKeeper sections...
  entityOperator:
    userOperator:
      resources:
        requests:
          cpu: "1"
          memory: 500Mi
        limits:
          cpu: "1"
          memory: 500Mi
```

Save the file and exit the editor. The Cluster Operator applies the changes automatically.

13.5. CONFIGURING FEATURE GATES

AMQ Streams operators support *feature gates* to enable or disable certain features and functionality. Enabling a feature gate changes the behavior of the relevant operator and introduces the feature to your AMQ Streams deployment.

Feature gates have a default state of either *enabled* or *disabled*.

To modify a feature gate's default state, use the **STRIMZI_FEATURE_GATES** environment variable in the operator's configuration. You can modify multiple feature gates using this single environment variable. Specify a comma-separated list of feature gate names and prefixes. A **+** prefix enables the feature gate and a **-** prefix disables it.

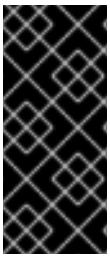
Example feature gate configuration that enables FeatureGate1 and disables FeatureGate2

```
env:
  - name: STRIMZI_FEATURE_GATES
    value: +FeatureGate1,-FeatureGate2
```

13.5.1. ControlPlaneListener feature gate

The **ControlPlaneListener** feature gate has moved to GA, which means it is now permanently enabled and cannot be disabled. With **ControlPlaneListener** enabled, the connections between the Kafka controller and brokers use an internal *control plane listener* on port 9090. Replication of data between

brokers, as well as internal connections from AMQ Streams operators, Cruise Control, or the Kafka Exporter use the *replication listener* on port 9091.



IMPORTANT

With the **ControlPlaneListener** feature gate permanently enabled, it is no longer possible to upgrade or downgrade directly between AMQ Streams 1.7 and earlier and AMQ Streams 2.3 and newer. You have to first upgrade or downgrade through one of the AMQ Streams versions in-between, disable the **ControlPlaneListener** feature gate, and then downgrade or upgrade (with the feature gate enabled) to the target version.

13.5.2. ServiceAccountPatching feature gate

The **ServiceAccountPatching** feature gate has moved to GA, which means it is now permanently enabled and cannot be disabled. With **ServiceAccountPatching** enabled, the Cluster Operator always reconciles service accounts and updates them when needed. For example, when you change service account labels or annotations using the **template** property of a custom resource, the operator automatically updates them on the existing service account resources.

13.5.3. UseStrimziPodSets feature gate

The **UseStrimziPodSets** feature gate has a default state of *enabled*.

The **UseStrimziPodSets** feature gate introduces a resource for managing pods called **StrimziPodSet**. When the feature gate is enabled, this resource is used instead of the StatefulSets. AMQ Streams handles the creation and management of pods instead of OpenShift. Using StrimziPodSets instead of StatefulSets provides more control over the functionality.

When this feature gate is disabled, AMQ Streams relies on StatefulSets to create and manage pods for the ZooKeeper and Kafka clusters. AMQ Streams creates the StatefulSet and OpenShift creates the pods according to the StatefulSet definition. When a pod is deleted, OpenShift is responsible for recreating it. The use of StatefulSets has the following limitations:

- Pods are always created or removed based on their index numbers
- All pods in the StatefulSet need to have a similar configuration
- Changing storage configuration for the Pods in the StatefulSet is complicated

Disabling the UseStrimziPodSets feature gate

To disable the **UseStrimziPodSets** feature gate, specify **-UseStrimziPodSets** in the **STRIMZI_FEATURE_GATES** environment variable in the Cluster Operator configuration.

IMPORTANT

The **UseStrimziPodSets** feature gate must be disabled when downgrading to AMQ Streams 2.0 and earlier versions.

13.5.4. (Preview) UseKRaft feature gate

The **UseKRaft** feature gate has a default state of *disabled*.

The **UseKRaft** feature gate deploys the Kafka cluster in the KRaft (Kafka Raft metadata) mode without ZooKeeper. This feature gate is currently intended only for development and testing.

IMPORTANT

The KRaft mode is not ready for production in Apache Kafka or in AMQ Streams.

When the **UseKRaft** feature gate is enabled, the Kafka cluster is deployed without ZooKeeper. **The .spec.zookeeper properties in the Kafka custom resource will be ignored, but still need to be present.** The **UseKRaft** feature gate provides an API that configures Kafka cluster nodes and their roles. The API is still in development and is expected to change before the KRaft mode is production-ready.

Currently, the KRaft mode in AMQ Streams has the following major limitations:

- Moving from Kafka clusters with ZooKeeper to KRaft clusters or the other way around is not supported.
- Upgrades and downgrades of Apache Kafka versions or the AMQ Streams operator are not supported. Users might need to delete the cluster, upgrade the operator and deploy a new Kafka cluster.
- The Topic Operator is not supported. The **spec.entityOperator.topicOperator** property **must be removed** from the **Kafka** custom resource.
- SCRAM-SHA-512 authentication is not supported.
- JBOD storage is not supported. The **type: jbod** storage can be used, but the JBOD array can contain only one disk.
- All Kafka nodes have both the **controller** and **broker** KRaft roles. Kafka clusters with separate **controller** and **broker** nodes are not supported.

Enabling the UseKRaft feature gate

To enable the **UseKRaft** feature gate, specify **+UseKRaft** in the **STRIMZI_FEATURE_GATES** environment variable in the Cluster Operator configuration.

IMPORTANT

The **UseKRaft** feature gate depends on the **UseStrimziPodSets** feature gate. When enabling the **UseKRaft** feature gate, make sure that the **UseStrimziPodSets** feature gate is enabled as well.

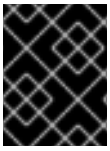
13.5.5. (Preview) StableConnectIdentities feature gate

The **StableConnectIdentities** feature gate has a default state of *disabled*.

The **StableConnectIdentities** feature gate uses **StrimziPodSet** resources to manage Kafka Connect and Kafka MirrorMaker 2 pods instead of using OpenShift **Deployment** resources. **StrimziPodSets** give the pods stable names and stable addresses, which do not change during rolling upgrades. This helps to minimize the number of rebalances of connector tasks.

Enabling the StableConnectIdentities feature gate

To enable the **StableConnectIdentities** feature gate, specify **+StableConnectIdentities** in the **STRIMZI_FEATURE_GATES** environment variable in the Cluster Operator configuration.



IMPORTANT

The **StableConnectIdentities** feature gate must be disabled when downgrading to AMQ Streams 2.3 and earlier versions.

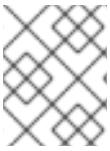
13.5.6. Feature gate releases

Feature gates have three stages of maturity:

- Alpha — typically disabled by default
- Beta — typically enabled by default
- General Availability (GA) — typically always enabled

Alpha stage features might be experimental or unstable, subject to change, or not sufficiently tested for production use. Beta stage features are well tested and their functionality is not likely to change. GA stage features are stable and should not change in the future. Alpha and beta stage features are removed if they do not prove to be useful.

- The **ControlPlaneListener** feature gate moved to GA stage in AMQ Streams 2.3. It is now permanently enabled and cannot be disabled.
- The **ServiceAccountPatching** feature gate moved to GA stage in AMQ Streams 2.3. It is now permanently enabled and cannot be disabled.
- The **UseStrimziPodSets** feature gate moved to beta stage in AMQ Streams 2.3. It moves to GA in a future release of AMQ Streams when the support for StatefulSets is completely removed.
- The **UseKRaft** feature gate is available for development only and does not currently have a planned release for moving to the beta phase.
- The **StableConnectIdentities** feature gate is in alpha stage and is disabled by default.



NOTE

Feature gates might be removed when they reach GA. This means that the feature was incorporated into the AMQ Streams core features and can no longer be disabled.

Table 13.5. Feature gates and the AMQ Streams versions when they moved to alpha, beta, or GA

Feature gate	Alpha	Beta	GA
ControlPlaneListener	1.8	2.0	2.3
ServiceAccountPatching	1.8	2.0	2.3
UseStrimziPodSets	2.1	2.3	future release (planned)
UseKRaft	2.2	-	-

Feature gate	Alpha	Beta	GA
StableConnectIdentities	2.4	future release (planned)	-

If a feature gate is enabled, you may need to disable it before upgrading or downgrading from a specific AMQ Streams version. The following table shows which feature gates you need to disable when upgrading or downgrading AMQ Streams versions.

Table 13.6. Feature gates to disable when upgrading or downgrading AMQ Streams

Disable Feature gate	Upgrading from AMQ Streams version	Downgrading to AMQ Streams version
ControlPlaneListener	1.7 and earlier	1.7 and earlier
UseStrimziPodSets	-	2.0 and earlier
StableConnectIdentities	-	2.3 and earlier

13.6. MONITORING OPERATORS USING PROMETHEUS METRICS

AMQ Streams operators expose Prometheus metrics. The metrics are automatically enabled and contain information about the following:

- Number of reconciliations
- Number of Custom Resources the operator is processing
- Duration of reconciliations
- JVM metrics from the operators

Additionally, AMQ Streams provides an example Grafana dashboard for the operator .

CHAPTER 14. SETTING UP METRICS AND DASHBOARDS FOR AMQ STREAMS

You can use Prometheus and Grafana to monitor your AMQ Streams deployment.

You can monitor your AMQ Streams deployment by viewing key metrics on dashboards and setting up alerts that trigger under certain conditions. Metrics are available for each of the components of AMQ Streams.

You can also collect metrics specific to **oauth** authentication and **opa** or **keycloak** authorization. You do this by setting the **enableMetrics** property to **true** in the listener configuration of the **Kafka** resource. For example, set **enableMetrics** to **true** in **spec.kafka.listeners.authentication** and **spec.kafka.authorization**. Similarly, you can enable metrics for **oauth** authentication in the **KafkaBridge**, **KafkaConnect**, **KafkaMirrorMaker**, and **KafkaMirrorMaker2** custom resources.

To provide metrics information, AMQ Streams uses Prometheus rules and Grafana dashboards.

When configured with a set of rules for each component of AMQ Streams, Prometheus consumes key metrics from the pods that are running in your cluster. Grafana then visualizes those metrics on dashboards. AMQ Streams includes example Grafana dashboards that you can customize to suit your deployment.

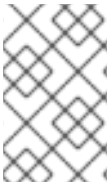
AMQ Streams employs *monitoring for user-defined projects* (an OpenShift feature) to simplify the Prometheus setup process.

Depending on your requirements, you can:

- Set up and deploy Prometheus to expose metrics
- Deploy Kafka Exporter to provide additional metrics
- Use Grafana to present the Prometheus metrics

With Prometheus and Grafana set up, you can use the example Grafana dashboards provided by AMQ Streams for monitoring.

Additionally, you can configure your deployment to track messages end-to-end by setting up distributed tracing.



NOTE

AMQ Streams provides example installation files for Prometheus and Grafana. You can use these files as a starting point when trying out monitoring of AMQ Streams. For further support, try engaging with the Prometheus and Grafana developer communities.

Supporting documentation for metrics and monitoring tools

For more information on the metrics and monitoring tools, refer to the supporting documentation:

- Prometheus
- Prometheus configuration
- Kafka Exporter
- Grafana Labs

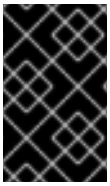
- Apache Kafka Monitoring describes JMX metrics exposed by Apache Kafka
- ZooKeeper JMX describes JMX metrics exposed by Apache ZooKeeper

14.1. MONITORING CONSUMER LAG WITH KAFKA EXPORTER

Kafka Exporter is an open source project to enhance monitoring of Apache Kafka brokers and clients. You can configure the **Kafka** resource to deploy Kafka Exporter with your Kafka cluster . Kafka Exporter extracts additional metrics data from Kafka brokers related to offsets, consumer groups, consumer lag, and topics. The metrics data is used, for example, to help identify slow consumers. Lag data is exposed as Prometheus metrics, which can then be presented in Grafana for analysis.

Kafka Exporter reads from the **__consumer_offsets** topic, which stores information on committed offsets for consumer groups. For Kafka Exporter to be able to work properly, consumer groups needs to be in use.

A Grafana dashboard for Kafka Exporter is one of a number of example Grafana dashboards provided by AMQ Streams.



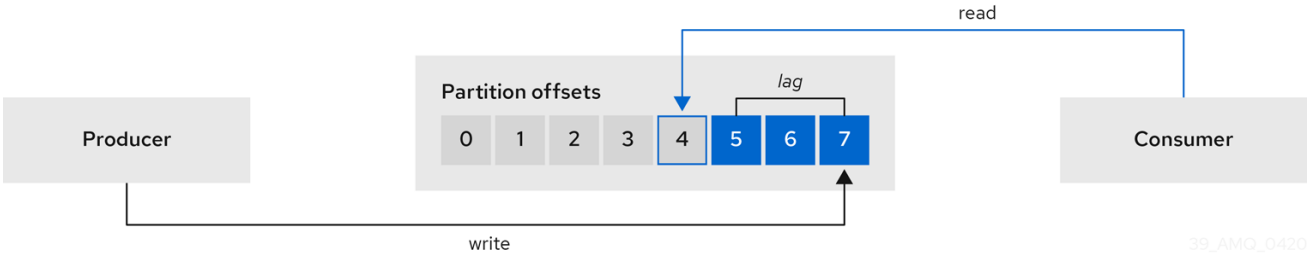
IMPORTANT

Kafka Exporter provides only additional metrics related to consumer lag and consumer offsets. For regular Kafka metrics, you have to configure the Prometheus metrics in Kafka brokers.

Consumer lag indicates the difference in the rate of production and consumption of messages. Specifically, consumer lag for a given consumer group indicates the delay between the last message in the partition and the message being currently picked up by that consumer.

The lag reflects the position of the consumer offset in relation to the end of the partition log.

Consumer lag between the producer and consumer offset



This difference is sometimes referred to as the *delta* between the producer offset and consumer offset: the read and write positions in the Kafka broker topic partitions.

Suppose a topic streams 100 messages a second. A lag of 1000 messages between the producer offset (the topic partition head) and the last offset the consumer has read means a 10-second delay.

The importance of monitoring consumer lag

For applications that rely on the processing of (near) real-time data, it is critical to monitor consumer lag to check that it does not become too big. The greater the lag becomes, the further the process moves from the real-time processing objective.

Consumer lag, for example, might be a result of consuming too much old data that has not been purged, or through unplanned shutdowns.

Reducing consumer lag

Use the Grafana charts to analyze lag and to check if actions to reduce lag are having an impact on an affected consumer group. If, for example, Kafka brokers are adjusted to reduce lag, the dashboard will show the *Lag by consumer group* chart going down and the *Messages consumed per minute* chart going up.

Typical actions to reduce lag include:

- Scaling-up consumer groups by adding new consumers
- Increasing the retention time for a message to remain in a topic
- Adding more disk capacity to increase the message buffer

Actions to reduce consumer lag depend on the underlying infrastructure and the use cases AMQ Streams is supporting. For instance, a lagging consumer is less likely to benefit from the broker being able to service a fetch request from its disk cache. And in certain cases, it might be acceptable to automatically drop messages until a consumer has caught up.

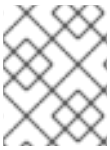
14.2. MONITORING CRUISE CONTROL OPERATIONS

Cruise Control monitors Kafka brokers in order to track the utilization of brokers, topics, and partitions. Cruise Control also provides a set of metrics for monitoring its own performance.

The Cruise Control metrics reporter collects raw metrics data from Kafka brokers. The data is produced to topics that are automatically created by Cruise Control. The metrics are used to generate optimization proposals for Kafka clusters.

Cruise Control metrics are available for real-time monitoring of Cruise Control operations. For example, you can use Cruise Control metrics to monitor the status of rebalancing operations that are running or provide alerts on any anomalies that are detected in an operation's performance.

You expose Cruise Control metrics by enabling the Prometheus JMX Exporter in the Cruise Control configuration.



NOTE

For a full list of available Cruise Control metrics, which are known as *sensors*, see the Cruise Control documentation.

14.2.1. Exposing Cruise Control metrics

If you want to expose metrics on Cruise Control operations, configure the **Kafka** resource to deploy Cruise Control and enable Prometheus metrics in the deployment. You can use your own configuration or use the example **kafka-cruise-control-metrics.yaml** file provided by AMQ Streams.

You add the configuration to the **metricsConfig** of the **CruiseControl** property in the **Kafka** resource. The configuration enables the Prometheus JMX Exporter to expose Cruise Control metrics through an HTTP endpoint. The HTTP endpoint is scraped by the Prometheus server.

Example metrics configuration for Cruise Control

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
```

```
  name: my-cluster
Spec:
  # ...
  cruiseControl:
    # ...
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: cruise-control-metrics
          key: metrics-config.yml
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: cruise-control-metrics
  labels:
    app: strimzi
data:
  metrics-config.yml: |
  # metrics configuration...
```

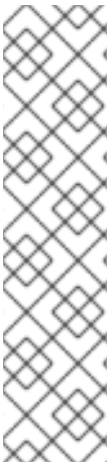
14.2.2. Viewing Cruise Control metrics

After you expose the Cruise Control metrics, you can use Prometheus or another suitable monitoring system to view information on the metrics data. AMQ Streams provides an example Grafana dashboard to display visualizations of Cruise Control metrics. The dashboard is a JSON file called **strimzi-cruise-control.json**. The exposed metrics provide the monitoring data when you enable the Grafana dashboard.

14.2.2.1. Monitoring balancedness scores

Cruise Control metrics include a balancedness score. Balancedness is the measure of how evenly a workload is distributed in a Kafka cluster.

The Cruise Control metric for balancedness score (**balancedness-score**) might differ from the balancedness score in the **KafkaRebalance** resource. Cruise Control calculates each score using **anomaly.detection.goals** which might not be the same as the **default.goals** used in the **KafkaRebalance** resource. The **anomaly.detection.goals** are specified in the **spec.cruiseControl.config** of the **Kafka** custom resource.



NOTE

Refreshing the **KafkaRebalance** resource fetches an optimization proposal. The latest cached optimization proposal is fetched if one of the following conditions applies:

- KafkaRebalance **goals** match the goals configured in the **default.goals** section of the **Kafka** resource
- KafkaRebalance **goals** are not specified

Otherwise, Cruise Control generates a new optimization proposal based on KafkaRebalance **goals**. If new proposals are generated with each refresh, this can impact performance monitoring.

14.2.2.2. Alerts on anomaly detection

Cruise control's *anomaly detector* provides metrics data for conditions that block the generation of optimization goals, such as broker failures. If you want more visibility, you can use the metrics provided by the anomaly detector to set up alerts and send out notifications. You can set up Cruise Control's *anomaly notifier* to route alerts based on these metrics through a specified notification channel. Alternatively, you can set up Prometheus to scrape the metrics data provided by the anomaly detector and generate alerts. Prometheus Alertmanager can then route the alerts generated by Prometheus.

The Cruise Control documentation provides information on **AnomalyDetector** metrics and the anomaly notifier.

14.3. EXAMPLE METRICS FILES

You can find example Grafana dashboards and other metrics configuration files in the example configuration files provided by AMQ Streams.

Example metrics files provided with AMQ Streams

```
metrics
├── grafana-dashboards 1
│   ├── strimzi-cruise-control.json
│   ├── strimzi-kafka-bridge.json
│   ├── strimzi-kafka-connect.json
│   ├── strimzi-kafka-exporter.json
│   ├── strimzi-kafka-mirror-maker-2.json
│   ├── strimzi-kafka.json
│   ├── strimzi-operators.json
│   └── strimzi-zookeeper.json
├── grafana-install
│   └── grafana.yaml 2
├── prometheus-additional-properties
│   └── prometheus-additional.yaml 3
├── prometheus-alertmanager-config
│   └── alert-manager-config.yaml 4
├── prometheus-install
│   ├── alert-manager.yaml 5
│   ├── prometheus-rules.yaml 6
│   ├── prometheus.yaml 7
│   ├── strimzi-pod-monitor.yaml 8
├── kafka-bridge-metrics.yaml 9
├── kafka-connect-metrics.yaml 10
├── kafka-cruise-control-metrics.yaml 11
├── kafka-metrics.yaml 12
└── kafka-mirror-maker-2-metrics.yaml 13
```

1. Example Grafana dashboards for the different AMQ Streams components.
2. Installation file for the Grafana image.
3. Additional configuration to scrape metrics for CPU, memory and disk volume usage, which comes directly from the OpenShift cAdvisor agent and kubelet on the nodes.

4. Hook definitions for sending notifications through Alertmanager.
5. Resources for deploying and configuring Alertmanager.
6. Alerting rules examples for use with Prometheus Alertmanager (deployed with Prometheus).
7. Installation resource file for the Prometheus image.
8. PodMonitor definitions translated by the Prometheus Operator into jobs for the Prometheus server to be able to scrape metrics data directly from pods.
9. Kafka Bridge resource with metrics enabled.
10. Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Connect.
11. Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Cruise Control.
12. Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka and ZooKeeper.
13. Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Mirror Maker 2.0.

14.3.1. Example Prometheus metrics configuration

AMQ Streams uses the Prometheus JMX Exporter to expose metrics through an HTTP endpoint, which can be scraped by the Prometheus server.

Grafana dashboards are dependent on Prometheus JMX Exporter relabeling rules, which are defined for AMQ Streams components in the custom resource configuration.

A label is a name-value pair. Relabeling is the process of writing a label dynamically. For example, the value of a label may be derived from the name of a Kafka server and client ID.

AMQ Streams provides example custom resource configuration YAML files with relabeling rules. When deploying Prometheus metrics configuration, you can can deploy the example custom resource or copy the metrics configuration to your own custom resource definition.

Table 14.1. Example custom resources with metrics configuration

Component	Custom resource	Example YAML file
Kafka and ZooKeeper	**Kafka**	**kafka-metrics.yaml**
Kafka Connect	**KafkaConnect**	**kafka-connect-metrics.yaml**
Kafka MirrorMaker 2	**KafkaMirrorMaker2**	**kafka-mirror-maker-2-metrics.yaml**
Kafka Bridge	**KafkaBridge**	**kafka-bridge-metrics.yaml**
Cruise Control	**Kafka**	**kafka-cruise-control-metrics.yaml**

14.3.2. Example Prometheus rules for alert notifications

Example Prometheus rules for alert notifications are provided with the example metrics configuration files provided by AMQ Streams. The rules are specified in the example **prometheus-rules.yaml** file for use in a Prometheus deployment.

Alerting rules provide notifications about specific conditions observed in metrics. Rules are declared on the Prometheus server, but Prometheus Alertmanager is responsible for alert notifications.

Prometheus alerting rules describe conditions using PromQL expressions that are continuously evaluated.

When an alert expression becomes true, the condition is met and the Prometheus server sends alert data to the Alertmanager. Alertmanager then sends out a notification using the communication method configured for its deployment.

General points about the alerting rule definitions:

- A **for** property is used with the rules to determine the period of time a condition must persist before an alert is triggered.
- A tick is a basic ZooKeeper time unit, which is measured in milliseconds and configured using the **tickTime** parameter of **Kafka.spec.zookeeper.config**. For example, if ZooKeeper **tickTime=3000**, 3 ticks (3 x 3000) equals 9000 milliseconds.
- The availability of the **ZookeeperRunningOutOfSpace** metric and alert is dependent on the OpenShift configuration and storage implementation used. Storage implementations for certain platforms may not be able to supply the information on available space required for the metric to provide an alert.

Alertmanager can be configured to use email, chat messages or other notification methods. Adapt the default configuration of the example rules according to your specific needs.

14.3.2.1. Example altering rules

The **prometheus-rules.yaml** file contains example rules for the following components:

- Kafka
- ZooKeeper
- Entity Operator
- Kafka Connect
- Kafka Bridge
- MirrorMaker
- Kafka Exporter

A description of each of the example rules is provided in the file.

14.3.3. Example Grafana dashboards

If you deploy Prometheus to provide metrics, you can use the example Grafana dashboards provided with AMQ Streams to monitor AMQ Streams components.

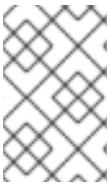
Example dashboards are provided in the **examples/metrics/grafana-dashboards** directory as JSON files.

All dashboards provide JVM metrics, as well as metrics specific to the component. For example, the Grafana dashboard for AMQ Streams operators provides information on the number of reconciliations or custom resources they are processing.

The example dashboards don't show all the metrics supported by Kafka. The dashboards are populated with a representative set of metrics for monitoring.

Table 14.2. Example Grafana dashboard files

Component	Example JSON file
AMQ Streams operators	**strimzi-operators.json**
Kafka	**strimzi-kafka.json**
ZooKeeper	**strimzi-zookeeper.json**
Kafka Connect	**strimzi-kafka-connect.json**
Kafka MirrorMaker 2	**strimzi-kafka-mirror-maker-2.json**
Kafka Bridge	**strimzi-kafka-bridge.json**
Cruise Control	**strimzi-cruise-control.json**
Kafka Exporter	**strimzi-kafka-exporter.json**



NOTE

When metrics are not available to the Kafka Exporter, because there is no traffic in the cluster yet, the Kafka Exporter Grafana dashboard will show **N/A** for numeric fields and **No data to show** for graphs.

14.4. DEPLOYING PROMETHEUS METRICS CONFIGURATION

Deploy Prometheus metrics configuration to use Prometheus with AMQ Streams. Use the **metricsConfig** property to enable and configure Prometheus metrics.

You can use your own configuration or the example custom resource configuration files provided with AMQ Streams.

- **kafka-metrics.yaml**
- **kafka-connect-metrics.yaml**
- **kafka-mirror-maker-2-metrics.yaml**
- **kafka-bridge-metrics.yaml**

- **kafka-cruise-control-metrics.yaml**

The example configuration files have relabeling rules and the configuration required to enable Prometheus metrics. Prometheus scrapes metrics from target HTTP endpoints. The example files are a good way to try Prometheus with AMQ Streams.

To apply the relabeling rules and metrics configuration, do one of the following:

- Copy the example configuration to your own custom resources
- Deploy the custom resource with the metrics configuration

If you want to include Kafka Exporter metrics, add **kafkaExporter** configuration to your **Kafka** resource.

IMPORTANT

Kafka Exporter provides only additional metrics related to consumer lag and consumer offsets. For regular Kafka metrics, you have to configure the Prometheus metrics in Kafka brokers.

This procedure shows how to deploy Prometheus metrics configuration in the **Kafka** resource. The process is the same when using the example files for other resources.

Procedure

1. Deploy the example custom resource with the Prometheus configuration.
   For example, for each **Kafka** resource you apply the **kafka-metrics.yaml** file.

   Deploying the example configuration

   ```
   oc apply -f kafka-metrics.yaml
   ```

   Alternatively, you can copy the example configuration in **kafka-metrics.yaml** to your own **Kafka** resource.

   Copying the example configuration

   ```
   oc edit kafka <kafka-configuration-file>
   ```

   Copy the **metricsConfig** property and the **ConfigMap** it references to your **Kafka** resource.

   Example metrics configuration for Kafka

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       metricsConfig: 1
         type: jmxPrometheusExporter
         valueFrom:
           configMapKeyRef:
   ```

```
        name: kafka-metrics
        key: kafka-metrics-config.yml
---
kind: ConfigMap 2
apiVersion: v1
metadata:
  name: kafka-metrics
  labels:
    app: strimzi
data:
  kafka-metrics-config.yml: |
  # metrics configuration...
```

1 Copy the **metricsConfig** property that references the ConfigMap that contains metrics configuration.

2 Copy the whole **ConfigMap** that specifies the metrics configuration.

NOTE

For Kafka Bridge, you specify the **enableMetrics** property and set it to **true**.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    # ...
  enableMetrics: true
  # ...
```

2. To deploy Kafka Exporter, add **kafkaExporter** configuration. **kafkaExporter** configuration is only specified in the **Kafka** resource.

Example configuration for deploying Kafka Exporter

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    image: my-registry.io/my-org/my-exporter-cluster:latest 1
    groupRegex: ".*" 2
    topicRegex: ".*" 3
    resources: 4
      requests:
        cpu: 200m
        memory: 64Mi
```

```
      limits:
        cpu: 500m
        memory: 128Mi
    logging: debug 5
    enableSaramaLogging: true 6
    template: 7
      pod:
        metadata:
          labels:
            label1: value1
        imagePullSecrets:
          - name: my-docker-credentials
        securityContext:
          runAsUser: 1000001
          fsGroup: 0
        terminationGracePeriodSeconds: 120
    readinessProbe: 8
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe: 9
      initialDelaySeconds: 15
      timeoutSeconds: 5
# ...
```

1. ADVANCED OPTION: Container image configuration, which is recommended only in special situations.
2. A regular expression to specify the consumer groups to include in the metrics.
3. A regular expression to specify the topics to include in the metrics.
4. CPU and memory resources to reserve.
5. Logging configuration, to log messages with a given severity (debug, info, warn, error, fatal) or above.
6. Boolean to enable Sarama logging, a Go client library used by Kafka Exporter.
7. Customization of deployment templates and pods.
8. Healthcheck readiness probes.
9. Healthcheck liveness probes.

NOTE

For Kafka Exporter to be able to work properly, consumer groups need to be in use.

Additional resources

Custom resource API reference.

14.5. VIEWING KAFKA METRICS AND DASHBOARDS IN OPENSHIFT

When AMQ Streams is deployed to OpenShift Container Platform, metrics are provided through *monitoring for user-defined projects*. This OpenShift feature gives developers access to a separate Prometheus instance for monitoring their own projects (for example, a **Kafka** project).

If monitoring for user-defined projects is enabled, the **openshift-user-workload-monitoring** project contains the following components:

- A Prometheus Operator
- A Prometheus instance (automatically deployed by the Prometheus Operator)
- A Thanos Ruler instance

AMQ Streams uses these components to consume metrics.

A cluster administrator must enable monitoring for user-defined projects and then grant developers and other users permission to monitor applications within their own projects.

Grafana deployment

You can deploy a Grafana instance to the project containing your Kafka cluster. The example Grafana dashboards can then be used to visualize Prometheus metrics for AMQ Streams in the Grafana user interface.

IMPORTANT

The **openshift-monitoring** project provides monitoring for core platform components. Do *not* use the Prometheus and Grafana components in this project to configure monitoring for AMQ Streams on OpenShift Container Platform 4.x.

Procedure outline

To set up AMQ Streams monitoring in OpenShift Container Platform, follow these procedures in order:

1. Prerequisite: Deploy the Prometheus metrics configuration
2. Deploy the Prometheus resources
3. Create a service account for Grafana
4. Deploy Grafana with a Prometheus datasource
5. Create a Route to the Grafana Service
6. Import the example Grafana dashboards

14.5.1. Prerequisites

- You have deployed the Prometheus metrics configuration using the example YAML files.
- *Monitoring for user-defined projects* is enabled. A cluster administrator has created a **cluster-monitoring-config** config map in your OpenShift cluster.
- A cluster administrator has assigned you a **monitoring-rules-edit** or **monitoring-edit** role.

For more information on creating a **cluster-monitoring-config** config map and granting users permission to monitor user-defined projects, see OpenShift Container Platform Monitoring.

14.5.2. Additional resources

- OpenShift Container Platform Monitoring

14.5.3. Deploying the Prometheus resources

Use Prometheus to obtain monitoring data in your Kafka cluster.

You can use your own Prometheus deployment or deploy Prometheus using the example metrics configuration files provided by AMQ Streams. To use the example files, you configure and deploy the **PodMonitor** resources. The **PodMonitors** scrape data directly from pods for Apache Kafka, ZooKeeper, Operators, the Kafka Bridge, and Cruise Control.

Then, you deploy the example alerting rules for Alertmanager.

Prerequisites

- A running Kafka cluster.
- Check the example alerting rules provided with AMQ Streams.

Procedure

1. Check that monitoring for user-defined projects is enabled:

```
oc get pods -n openshift-user-workload-monitoring
```

If enabled, pods for the monitoring components are returned. For example:

```
NAME                                  READY   STATUS    RESTARTS   AGE
prometheus-operator-5cc59f9bc6-kgcq8   1/1     Running   0          25s
prometheus-user-workload-0             5/5     Running   1          14s
prometheus-user-workload-1             5/5     Running   1          14s
thanos-ruler-user-workload-0           3/3     Running   0          14s
thanos-ruler-user-workload-1           3/3     Running   0          14s
```

If no pods are returned, monitoring for user-defined projects is disabled. See the Prerequisites in Section 14.5, “Viewing Kafka metrics and dashboards in OpenShift” .

2. Multiple **PodMonitor** resources are defined in **examples/metrics/prometheus-install/strimzi-pod-monitor.yaml**.
For each **PodMonitor** resource, edit the **spec.namespaceSelector.matchNames** property:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cluster-operator-metrics
  labels:
    app: strimzi
spec:
  selector:
    matchLabels:
      strimzi.io/kind: cluster-operator
  namespaceSelector:
    matchNames:
```

```
      - <project-name> 1
  podMetricsEndpoints:
  - path: /metrics
    port: http
# ...
```

1 The project where the pods to scrape the metrics from are running, for example, **Kafka**.

3. Deploy the **strimzi-pod-monitor.yaml** file to the project where your Kafka cluster is running:

```
oc apply -f strimzi-pod-monitor.yaml -n MY-PROJECT
```

4. Deploy the example Prometheus rules to the same project:

```
oc apply -f prometheus-rules.yaml -n MY-PROJECT
```

14.5.4. Creating a service account for Grafana

A Grafana instance for AMQ Streams needs to run with a service account that is assigned the **cluster-monitoring-view** role.

Create a service account if you are using Grafana to present metrics for monitoring.

Prerequisites

- Deploy the Prometheus resources

Procedure

1. Create a **ServiceAccount** for Grafana in the project containing your Kafka cluster:

```
oc create sa grafana-service-account -n my-project
```

In this example, a service account named **grafana-service-account** is created in the **my-project** namespace.

2. Create a **ClusterRoleBinding** resource that assigns the **cluster-monitoring-view** role to the Grafana **ServiceAccount**. Here the resource is named **grafana-cluster-monitoring-binding**.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: grafana-cluster-monitoring-binding
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: grafana-service-account
    namespace: my-project
roleRef:
  kind: ClusterRole
  name: cluster-monitoring-view
  apiGroup: rbac.authorization.k8s.io
```

3. Deploy the **ClusterRoleBinding** to the same project:

```
oc apply -f grafana-cluster-monitoring-binding.yaml -n my-project
```

4. Create a token secret for the service account:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa
  annotations:
    kubernetes.io/service-account.name: "grafana-service-account" (1)
type: kubernetes.io/service-account-token (2)
```

(1) Specifies the service account.

(2) Specifies a service account token secret.

5. Create the **Secret** object and access token:

```
oc create -f <secret_configuration>.yaml
```

You need the access token when deploying Grafana.

14.5.5. Deploying Grafana with a Prometheus datasource

Deploy Grafana to present Prometheus metrics. A Grafana application requires configuration for the OpenShift Container Platform monitoring stack.

OpenShift Container Platform includes a *Thanos Querier* instance in the **openshift-monitoring** project. Thanos Querier is used to aggregate platform metrics.

To consume the required platform metrics, your Grafana instance requires a Prometheus data source that can connect to Thanos Querier. To configure this connection, you create a config map that authenticates, by using a token, to the **oauth-proxy** sidecar that runs alongside Thanos Querier. A **datasource.yaml** file is used as the source of the config map.

Finally, you deploy the Grafana application with the config map mounted as a volume to the project containing your Kafka cluster.

Prerequisites

- You have deployed Prometheus resources .
- You have created a service account for Grafana .

Procedure

1. Get the access token of the Grafana **ServiceAccount**:

```
oc describe sa/grafana-service-account | grep Tokens:
oc describe secret grafana-service-account-token-mmlp9 | grep token:
```

In this example, the service account is named **grafana-service-account**. Copy the access token to use in the next step.

2. Create a **datasource.yaml** file containing the Thanos Querier configuration for Grafana. Paste the access token into the **httpHeaderValue1** property as indicated.

```
apiVersion: 1

datasources:
- name: Prometheus
  type: prometheus
  url: https://thanos-querier.openshift-monitoring.svc.cluster.local:9091
  access: proxy
  basicAuth: false
  withCredentials: false
  isDefault: true
  jsonData:
    timeInterval: 5s
    tlsSkipVerify: true
    httpHeaderName1: "Authorization"
  secureJsonData:
    httpHeaderValue1: "Bearer ${GRAFANA-ACCESS-TOKEN}" (1)
  editable: true
```

(1) ***GRAFANA-ACCESS-TOKEN***: The value of the access token for the Grafana **ServiceAccount**.

3. Create a config map named **grafana-config** from the **datasource.yaml** file:

```
oc create configmap grafana-config --from-file=datasource.yaml -n MY-PROJECT
```

4. Create a Grafana application consisting of a **Deployment** and a **Service**. The **grafana-config** config map is mounted as a volume for the datasource configuration.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
  labels:
    app: strimzi
spec:
  replicas: 1
  selector:
    matchLabels:
      name: grafana
  template:
    metadata:
      labels:
        name: grafana
    spec:
      serviceAccountName: grafana-service-account
      containers:
      - name: grafana
        image: grafana/grafana:9.4.7
```

```
        ports:
        - name: grafana
          containerPort: 3000
          protocol: TCP
        volumeMounts:
        - name: grafana-data
          mountPath: /var/lib/grafana
        - name: grafana-logs
          mountPath: /var/log/grafana
        - name: grafana-config
          mountPath: /etc/grafana/provisioning/datasources/datasource.yaml
          readOnly: true
          subPath: datasource.yaml
        readinessProbe:
          httpGet:
            path: /api/health
            port: 3000
          initialDelaySeconds: 5
          periodSeconds: 10
        livenessProbe:
          httpGet:
            path: /api/health
            port: 3000
          initialDelaySeconds: 15
          periodSeconds: 20
      volumes:
      - name: grafana-data
        emptyDir: {}
      - name: grafana-logs
        emptyDir: {}
      - name: grafana-config
        configMap:
          name: grafana-config
---
apiVersion: v1
kind: Service
metadata:
  name: grafana
  labels:
    app: strimzi
spec:
  ports:
  - name: grafana
    port: 3000
    targetPort: 3000
    protocol: TCP
  selector:
    name: grafana
  type: ClusterIP
```

5. Deploy the Grafana application to the project containing your Kafka cluster:

```
oc apply -f <grafana-application> -n <my-project>
```

14.5.6. Creating a route to the Grafana Service

You can access the Grafana user interface through a Route that exposes the Grafana service.

Prerequisites

- Deploy the Prometheus resources
- Create a service account for Grafana
- Deploy Grafana with a Prometheus datasource

Procedure

- Create an edge route to the **grafana** service:

```
oc create route edge <my-grafana-route> --service=grafana --namespace=KAFKA-NAMESPACE
```

14.5.7. Importing the example Grafana dashboards

Use Grafana to provide visualizations of Prometheus metrics on customizable dashboards.

AMQ Streams provides example dashboard configuration files for Grafana in JSON format.

- **examples/metrics/grafana-dashboards**

This procedure uses the example Grafana dashboards.

The example dashboards are a good starting point for monitoring key metrics, but they don't show all the metrics supported by Kafka. You can modify the example dashboards or add other metrics, depending on your infrastructure.

Prerequisites

- Deploy the Prometheus resources
- Create a service account for Grafana
- Deploy Grafana with a Prometheus datasource
- Create a Route to the Grafana Service

Procedure

1. Get the details of the Route to the Grafana Service. For example:

```
oc get routes

NAME                HOST/PORT                          PATH  SERVICES
MY-GRAFANA-ROUTE    MY-GRAFANA-ROUTE-amq-streams.net         grafana
```

2. In a web browser, access the Grafana login screen using the URL for the Route host and port.
3. Enter your user name and password, and then click **Log In**.
   The default Grafana user name and password are both **admin**. After logging in for the first time, you can change the password.

4. In **Configuration > Data Sources**, check that the **Prometheus** data source was created. The data source was created in Section 14.5.5, "Deploying Grafana with a Prometheus datasource".
5. Click the **+** icon and then click **Import**.
6. In **examples/metrics/grafana-dashboards**, copy the JSON of the dashboard to import.
7. Paste the JSON into the text box, and then click **Load**.
8. Repeat steps 5-7 for the other example Grafana dashboards.

The imported Grafana dashboards are available to view from the **Dashboards** home page.

CHAPTER 15. INTRODUCING DISTRIBUTED TRACING

Distributed tracing tracks the progress of transactions between applications in a distributed system. In a microservices architecture, tracing tracks the progress of transactions between services. Trace data is useful for monitoring application performance and investigating issues with target systems and end-user applications.

In AMQ Streams, tracing facilitates the end-to-end tracking of messages: from source systems to Kafka, and then from Kafka to target systems and applications. Distributed tracing complements the monitoring of metrics in Grafana dashboards, as well as the component loggers.

Support for tracing is built in to the following Kafka components:

- MirrorMaker to trace messages from a source cluster to a target cluster
- Kafka Connect to trace messages consumed and produced by Kafka Connect
- Kafka Bridge to trace messages between Kafka and HTTP client applications

Tracing is not supported for Kafka brokers.

You enable and configure tracing for these components through their custom resources. You add tracing configuration using **spec.template** properties.

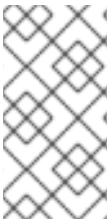
You enable tracing by specifying a tracing type using the **spec.tracing.type** property:

opentelemetry

Specify **type: opentelemetry** to use OpenTelemetry. By Default, OpenTelemetry uses the OTLP (OpenTelemetry Protocol) exporter and endpoint to get trace data. You can specify other tracing systems supported by OpenTelemetry, including Jaeger tracing. To do this, you change the OpenTelemetry exporter and endpoint in the tracing configuration.

jaeger

Specify **type:jaeger** to use OpenTracing and the Jaeger client to get trace data.



NOTE

Support for **type: jaeger** tracing is deprecated. The Jaeger clients are now retired and the OpenTracing project archived. As such, we cannot guarantee their support for future Kafka versions. If possible, we will maintain the support for **type: jaeger** tracing until June 2023 and remove it afterwards. Please migrate to OpenTelemetry as soon as possible.

15.1. TRACING OPTIONS

Use OpenTelemetry or OpenTracing (deprecated) with the Jaeger tracing system.

OpenTelemetry and OpenTracing provide API specifications that are independent from the tracing or monitoring system.

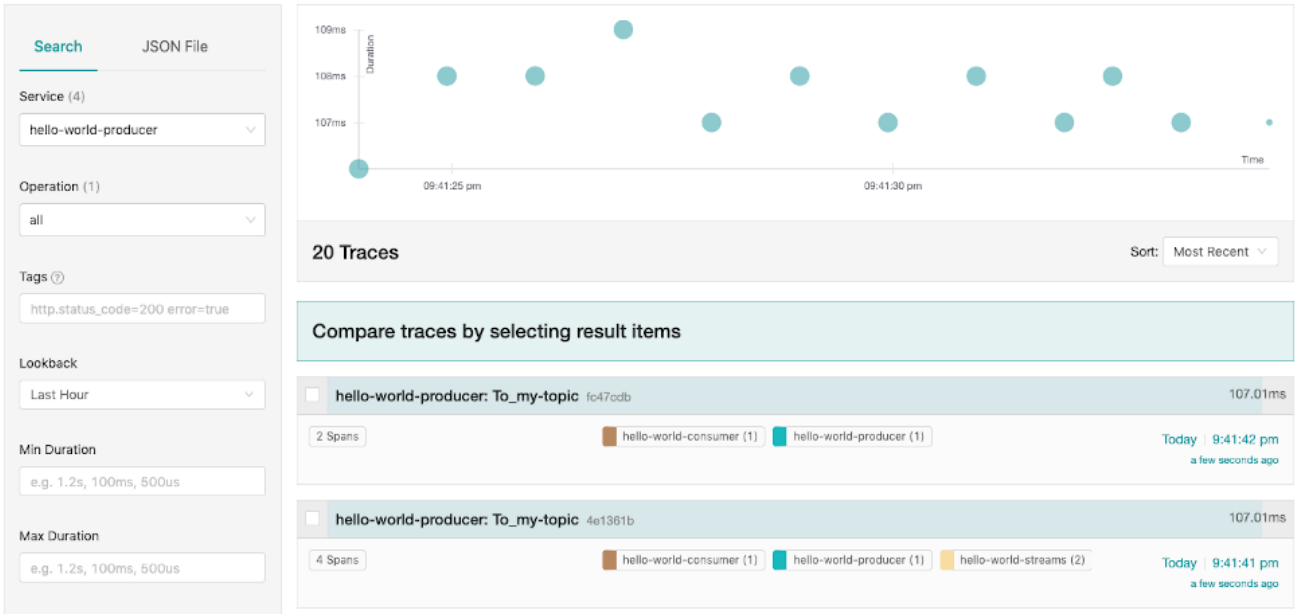
You use the APIs to instrument application code for tracing.

- Instrumented applications generate *traces* for individual requests across the distributed system.
- Traces are composed of *spans* that define specific units of work over time.

Jaeger is a tracing system for microservices-based distributed systems.

- Jaeger implements the tracing APIs and provides client libraries for instrumentation.
- The Jaeger user interface allows you to query, filter, and analyze trace data.

The Jaeger user interface showing a simple query



Additional resources

- Jaeger documentation
- OpenTelemetry documentation
- OpenTracing documentation

15.2. ENVIRONMENT VARIABLES FOR TRACING

Use environment variables when you are enabling tracing for Kafka components or initializing a tracer for Kafka clients.

Tracing environment variables are subject to change. For the latest information, see the OpenTelemetry documentation and OpenTracing documentation.

The following tables describe the key environment variables for setting up a tracer.

Table 15.1. OpenTelemetry environment variables

Property	Required	Description
OTEL_SERVICE_NAME	Yes	The name of the Jaeger tracing service for OpenTelemetry.
OTEL_EXPORTER_JAEGER_ENDPOINT	Yes	The exporter used for tracing.

Property	Required	Description
OTEL_TRACES_EXPORTER	Yes	The exporter used for tracing. Set to **otlp** by default. If using Jaeger tracing, you need to set this environment variable as **jaeger**. If you are using another tracing implementation, specify the exporter used.

Table 15.2. OpenTracing environment variables

Property	Required	Description
JAEGER_SERVICE_NAME	Yes	The name of the Jaeger tracer service.
JAEGER_AGENT_HOST	No	The hostname for communicating with the **jaeger-agent** through the User Datagram Protocol (UDP).
JAEGER_AGENT_PORT	No	The port used for communicating with the **jaeger-agent** through UDP.

15.3. SETTING UP DISTRIBUTED TRACING

Enable distributed tracing in Kafka components by specifying a tracing type in the custom resource. Instrument tracers in Kafka clients for end-to-end tracking of messages.

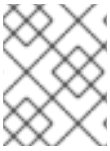
To set up distributed tracing, follow these procedures in order:

- Enable tracing for MirrorMaker, Kafka Connect, and the Kafka Bridge
- Set up tracing for clients:
  - Initialize a Jaeger tracer for Kafka clients
- Instrument clients with tracers:
  - Instrument producers and consumers for tracing
  - Instrument Kafka Streams applications for tracing

15.3.1. Prerequisites

Before setting up distributed tracing, make sure Jaeger backend components are deployed to your OpenShift cluster. We recommend using the Jaeger operator for deploying Jaeger on your OpenShift cluster.

For deployment instructions, see the Jaeger documentation.



NOTE

Setting up tracing for applications and systems beyond AMQ Streams is outside the scope of this content.

15.3.2. Enabling tracing in MirrorMaker, Kafka Connect, and Kafka Bridge resources

Distributed tracing is supported for MirrorMaker, MirrorMaker 2, Kafka Connect, and the AMQ Streams Kafka Bridge. Configure the custom resource of the component to specify and enable a tracer service.

Enabling tracing in a resource triggers the following events:

- Interceptor classes are updated in the integrated consumers and producers of the component.
- For MirrorMaker, MirrorMaker 2, and Kafka Connect, the tracing agent initializes a tracer based on the tracing configuration defined in the resource.
- For the Kafka Bridge, a tracer based on the tracing configuration defined in the resource is initialized by the Kafka Bridge itself.

You can enable tracing that uses OpenTelemetry or OpenTracing.

Tracing in MirrorMaker and MirrorMaker 2

For MirrorMaker and MirrorMaker 2, messages are traced from the source cluster to the target cluster. The trace data records messages entering and leaving the MirrorMaker or MirrorMaker 2 component.

Tracing in Kafka Connect

For Kafka Connect, only messages produced and consumed by Kafka Connect are traced. To trace messages sent between Kafka Connect and external systems, you must configure tracing in the connectors for those systems.

Tracing in the Kafka Bridge

For the Kafka Bridge, messages produced and consumed by the Kafka Bridge are traced. Incoming HTTP requests from client applications to send and receive messages through the Kafka Bridge are also traced. To have end-to-end tracing, you must configure tracing in your HTTP clients.

Procedure

Perform these steps for each **KafkaMirrorMaker**, **KafkaMirrorMaker2**, **KafkaConnect**, and **KafkaBridge** resource.

1. In the **spec.template** property, configure the tracer service.
   - Use the tracing environment variables as template configuration properties.
   - For OpenTelemetry, set the **spec.tracing.type** property to **opentelemetry**.
   - For OpenTracing, set the **spec.tracing.type** property to **jaeger**.

   Example tracing configuration for Kafka Connect using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
```

```
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
  #...
```

Example tracing configuration for MirrorMaker using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  #...
  template:
    mirrorMakerContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
#...
```

Example tracing configuration for MirrorMaker 2 using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
#...
```

Example tracing configuration for the Kafka Bridge using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
```

```
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
#...
```

Example tracing configuration for Kafka Connect using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
  #...
```

Example tracing configuration for MirrorMaker using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  #...
  template:
    mirrorMakerContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
```

```
  tracing:
    type: jaeger
#...
```

Example tracing configuration for MirrorMaker 2 using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

Example tracing configuration for the Kafka Bridge using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

2. Create or update the resource:

```
oc apply -f <resource_configuration_file>
```

15.3.3. Initializing tracing for Kafka clients

Initialize a tracer, then instrument your client applications for distributed tracing. You can instrument Kafka producer and consumer clients, and Kafka Streams API applications. You can initialize a tracer for OpenTracing or OpenTelemetry.

Configure and initialize a tracer using a set of tracing environment variables.

Procedure

In each client application add the dependencies for the tracer:

1. Add the Maven dependencies to the **pom.xml** file for the client application:

   #### Dependencies for OpenTelemetry

   ```
   <dependency>
     <groupId>io.opentelemetry</groupId>
     <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
     <version>1.18.0.redhat-00001</version>
   </dependency>
   <dependency>
    <groupId>io.opentelemetry.instrumentation</groupId>
    <artifactId>opentelemetry-kafka-clients-{OpenTelemetryKafkaClient}</artifactId>
    <version>1.18.0.redhat-00001</version>
   </dependency>
   <dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-otlp</artifactId>
    <version>1.18.0.redhat-00001</version>
   </dependency>
   ```

   #### Dependencies for OpenTracing

   ```
   <dependency>
     <groupId>io.jaegertracing</groupId>
     <artifactId>jaeger-client</artifactId>
     <version>1.8.1.redhat-00002</version>
   </dependency>
   <dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-kafka-client</artifactId>
    <version>0.1.15.redhat-00006</version>
   </dependency>
   ```

2. Define the configuration of the tracer using the tracing environment variables.

3. Create a tracer, which is initialized with the environment variables:

   #### Creating a tracer for OpenTelemetry

   ```
   OpenTelemetry ot = GlobalOpenTelemetry.get();
   ```

   #### Creating a tracer for OpenTracing

   ```
   Tracer tracer = Configuration.fromEnv().getTracer();
   ```

4. Register the tracer as a global tracer:

```
GlobalTracer.register(tracer);
```

5. Instrument your client:

- Section 15.3.4, "Instrumenting producers and consumers for tracing"
- Section 15.3.5, "Instrumenting Kafka Streams applications for tracing"

15.3.4. Instrumenting producers and consumers for tracing

Instrument application code to enable tracing in Kafka producers and consumers. Use a decorator pattern or interceptors to instrument your Java producer and consumer application code for tracing. You can then record traces when messages are produced or retrieved from a topic.

OpenTelemetry and OpenTracing instrumentation projects provide classes that support instrumentation of producers and consumers.

Decorator instrumentation

For decorator instrumentation, create a modified producer or consumer instance for tracing. Decorator instrumentation is different for OpenTelemetry and OpenTracing.

Interceptor instrumentation

For interceptor instrumentation, add the tracing capability to the consumer or producer configuration. Interceptor instrumentation is the same for OpenTelemetry and OpenTracing.

Prerequisites

- You have initialized tracing for the client.
  You enable instrumentation in producer and consumer applications by adding the tracing JARs as dependencies to your project.

Procedure

Perform these steps in the application code of each producer and consumer application. Instrument your client application code using either a decorator pattern or interceptors.

- To use a decorator pattern, create a modified producer or consumer instance to send or receive messages.
  You pass the original **KafkaProducer** or **KafkaConsumer** class.

  Example decorator instrumentation for OpenTelemetry

```
// Producer instance
Producer < String, String > op = new KafkaProducer < > (
   configs,
   new StringSerializer(),
   new StringSerializer()
   );
   Producer < String, String > producer = tracing.wrap(op);
KafkaTracing tracing = KafkaTracing.create(GlobalOpenTelemetry.get());
producer.send(...);

//consumer instance
Consumer<String, String> oc = new KafkaConsumer<>(
```

```
    configs,
    new StringDeserializer(),
    new StringDeserializer()
    );
    Consumer<String, String> consumer = tracing.wrap(oc);
consumer.subscribe(Collections.singleton("mytopic"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

Example decorator instrumentation for OpenTracing

```
//producer instance
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>
(producer, tracer);
TracingKafkaProducer.send(...)

//consumer instance
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);
TracingKafkaConsumer<Integer, String> tracingConsumer = new TracingKafkaConsumer<>
(consumer, tracer);
tracingConsumer.subscribe(Collections.singletonList("mytopic"));
ConsumerRecords<Integer, String> records = tracingConsumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

- To use interceptors, set the interceptor class in the producer or consumer configuration. You use the **KafkaProducer** and **KafkaConsumer** classes in the usual way. The **TracingProducerInterceptor** and **TracingConsumerInterceptor** interceptor classes take care of the tracing capability.

Example producer configuration using interceptors

```
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingProducerInterceptor.class.getName());

KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
producer.send(...);
```

Example consumer configuration using interceptors

```
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingConsumerInterceptor.class.getName());

KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("messages"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

15.3.5. Instrumenting Kafka Streams applications for tracing

Instrument application code to enable tracing in Kafka Streams API applications. Use a decorator pattern or interceptors to instrument your Kafka Streams API applications for tracing. You can then record traces when messages are produced or retrieved from a topic.

Decorator instrumentation

For decorator instrumentation, create a modified Kafka Streams instance for tracing. The OpenTracing instrumentation project provides a **TracingKafkaClientSupplier** class that supports instrumentation of Kafka Streams. You create a wrapped instance of the **TracingKafkaClientSupplier** supplier interface, which provides tracing instrumentation for Kafka Streams. For OpenTelemetry, the process is the same but you need to create a custom **TracingKafkaClientSupplier** class to provide the support.

Interceptor instrumentation

For interceptor instrumentation, add the tracing capability to the Kafka Streams producer and consumer configuration.

Prerequisites

- You have initialized tracing for the client.
  You enable instrumentation in Kafka Streams applications by adding the tracing JARs as dependencies to your project.
- To instrument Kafka Streams with OpenTelemetry, you'll need to write a custom **TracingKafkaClientSupplier**.
- The custom **TracingKafkaClientSupplier** can extend Kafka's **DefaultKafkaClientSupplier**, overriding the producer and consumer creation methods to wrap the instances with the telemetry-related code.

  Example custom TracingKafkaClientSupplier

```
private class TracingKafkaClientSupplier extends DefaultKafkaClientSupplier {
    @Override
    public Producer<byte[], byte[]> getProducer(Map<String, Object> config) {
        KafkaTelemetry telemetry = KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getProducer(config));
    }

    @Override
    public Consumer<byte[], byte[]> getConsumer(Map<String, Object> config) {
        KafkaTelemetry telemetry = KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getConsumer(config));
    }

    @Override
    public Consumer<byte[], byte[]> getRestoreConsumer(Map<String, Object> config) {
        return this.getConsumer(config);
    }

    @Override
    public Consumer<byte[], byte[]> getGlobalConsumer(Map<String, Object> config) {
        return this.getConsumer(config);
    }
}
```

Procedure

Perform these steps for each Kafka Streams API application.

- To use a decorator pattern, create an instance of the **TracingKafkaClientSupplier** supplier interface, then provide the supplier interface to **KafkaStreams**.

  #### Example decorator instrumentation

  ```
  KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
  KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
  supplier);
  streams.start();
  ```

- To use interceptors, set the interceptor class in the Kafka Streams producer and consumer configuration.
  The **TracingProducerInterceptor** and **TracingConsumerInterceptor** interceptor classes take care of the tracing capability.

  #### Example producer and consumer configuration using interceptors

  ```
  props.put(StreamsConfig.PRODUCER_PREFIX +
  ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
  TracingProducerInterceptor.class.getName());
  props.put(StreamsConfig.CONSUMER_PREFIX +
  ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
  TracingConsumerInterceptor.class.getName());
  ```

15.3.6. Introducing a different OpenTelemetry tracing system

Instead of the default OTLP system, you can specify other tracing systems that are supported by OpenTelemetry. You do this by adding the required artifacts to the Kafka image provided with AMQ Streams. Any required implementation specific environment variables must also be set. You then enable the new tracing implementation using the **OTEL_TRACES_EXPORTER** environment variable.

This procedure shows how to implement Zipkin tracing.

Procedure

1. Add the tracing artifacts to the **/opt/kafka/libs/** directory of the AMQ Streams Kafka image. You can use the Kafka container image on the Red Hat Ecosystem Catalog as a base image for creating a new custom image.

   #### OpenTelemetry artifact for Zipkin

   ```
   io.opentelemetry:opentelemetry-exporter-zipkin
   ```

2. Set the tracing exporter and endpoint for the new tracing implementation.

   #### Example Zikpin tracer configuration

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaMirrorMaker2
   metadata:
   ```

```
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-zipkin-service
        - name: OTEL_EXPORTER_ZIPKIN_ENDPOINT
          value: http://zipkin-exporter-host-name:9411/api/v2/spans (1)
        - name: OTEL_TRACES_EXPORTER
          value: zipkin (2)
  tracing:
    type: opentelemetry
#...
```

1. Specifies the Zipkin endpoint to connect to.
2. The Zipkin exporter.

15.3.7. Custom span names

A tracing *span* is a logical unit of work in Jaeger, with an operation name, start time, and duration. Spans have built-in names, but you can specify custom span names in your Kafka client instrumentation where used.

Specifying custom span names is optional and only applies when using a decorator pattern in producer and consumer client instrumentation or Kafka Streams instrumentation.

15.3.7.1. Specifying span names for OpenTelemetry

Custom span names cannot be specified directly with OpenTelemetry. Instead, you retrieve span names by adding code to your client application to extract additional tags and attributes.

Example code to extract attributes

```
//Defines attribute extraction for a producer
private static class ProducerAttribExtractor implements AttributesExtractor < ProducerRecord < ? , ? >
, Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord) {
        set(attributes, AttributeKey.stringKey("prod_start"), "prod1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("prod_end"), "prod2");
    }
}
//Defines attribute extraction for a consumer
private static class ConsumerAttribExtractor implements AttributesExtractor < ConsumerRecord < ? ,
? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord) {
```

```
        set(attributes, AttributeKey.stringKey("con_start"), "con1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("con_end"), "con2");
    }
}
//Extracts the attributes
public static void main(String[] args) throws Exception {
        Map < String, Object > configs = new HashMap < >
(Collections.singletonMap(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"));
        System.setProperty("otel.traces.exporter", "jaeger");
        System.setProperty("otel.service.name", "myapp1");
        KafkaTracing tracing = KafkaTracing.newBuilder(GlobalOpenTelemetry.get())
            .addProducerAttributesExtractors(new ProducerAttribExtractor())
            .addConsumerAttributesExtractors(new ConsumerAttribExtractor())
            .build();
```

15.3.7.2. Specifying span names for OpenTracing

To specify custom span names for OpenTracing, pass a **BiFunction** object as an additional argument when instrumenting producers and consumers.

For more information on built-in names and specifying custom span names to instrument client application code in a decorator pattern, see the OpenTracing Apache Kafka client instrumentation.

CHAPTER 16. UPGRADING AMQ STREAMS

AMQ Streams can be upgraded to version 2.4 to take advantage of new features and enhancements, performance improvements, and security options.

As part of the upgrade, you upgrade Kafka to the latest supported version. Each Kafka release introduces new features, improvements, and bug fixes to your AMQ Streams deployment.

AMQ Streams can be downgraded to the previous version if you encounter issues with the newer version.

Released versions of AMQ Streams are available from the AMQ Streams software downloads page .

Upgrade downtime and availability

If topics are configured for high availability, upgrading AMQ Streams should not cause any downtime for consumers and producers that publish and read data from those topics. Highly available topics have a replication factor of at least 3 and partitions distributed evenly among the brokers.

Upgrading AMQ Streams triggers rolling updates, where all brokers are restarted in turn, at different stages of the process. During rolling updates, not all brokers are online, so overall *cluster availability* is temporarily reduced. A reduction in cluster availability increases the chance that a broker failure will result in lost messages.

16.1. AMQ STREAMS UPGRADE PATHS

Two upgrade paths are possible.

Incremental upgrade

Upgrading AMQ Streams from the previous minor version to version 2.4.

Multi-version upgrade

Upgrading AMQ Streams from an old version to version 2.4 within a single upgrade (skipping one or more intermediate versions).
For example, upgrading from AMQ Streams 2.2 directly to AMQ Streams 2.4.

16.1.1. Supported Kafka versions

Decide which Kafka version to upgrade to before starting the AMQ Streams upgrade process. You can review supported Kafka versions in the AMQ Streams Supported Configurations .

- Kafka 3.4.0 is supported for production use.
- Kafka 3.3.1 is supported only for the purpose of upgrading to AMQ Streams 2.4.

You can only use a Kafka version supported by the version of AMQ Streams you are using. You can upgrade to a higher Kafka version as long as it is supported by your version of AMQ Streams. In some cases, you can also downgrade to a previous supported Kafka version.

16.1.2. Upgrading from an AMQ Streams version earlier than 1.7

If you are upgrading to the latest version of AMQ Streams from a version prior to version 1.7, do the following:

1. Upgrade AMQ Streams to version 1.7 following the standard sequence.
2. Convert AMQ Streams custom resources to **v1beta2** using the *API conversion tool* provided with AMQ Streams.
3. Do one of the following:
   - Upgrade to AMQ Streams 1.8 (where the **ControlPlaneListener** feature gate is disabled by default).
   - Upgrade to AMQ Streams 2.0 or 2.2 (where the **ControlPlaneListener** feature gate is enabled by default) with the **ControlPlaneListener** feature gate disabled.
4. Enable the **ControlPlaneListener** feature gate.
5. Upgrade to AMQ Streams 2.4 following the standard sequence.

AMQ Streams custom resources started using the **v1beta2** API version in release 1.7. CRDs and custom resources must be converted **before** upgrading to AMQ Streams 1.8 or newer. For information on using the API conversion tool, see the AMQ Streams 1.7 upgrade documentation .

NOTE

As an alternative to first upgrading to version 1.7, you can install the custom resources from version 1.7 and then convert the resources.

The **ControlPlaneListener** feature is now permanently enabled in AMQ Streams. You must upgrade to a version of AMQ Streams where it is disabled, then enable it using the **STRIMZI_FEATURE_GATES** environment variable in the Cluster Operator configuration.

Disabling the ControlPlaneListener feature gate

```
env:
  - name: STRIMZI_FEATURE_GATES
    value: -ControlPlaneListener
```

Enabling the ControlPlaneListener feature gate

```
env:
  - name: STRIMZI_FEATURE_GATES
    value: +ControlPlaneListener
```

16.2. REQUIRED UPGRADE SEQUENCE

To upgrade brokers and clients without downtime, you *must* complete the AMQ Streams upgrade procedures in the following order:

1. Make sure your OpenShift cluster version is supported.
   AMQ Streams 2.4 is supported by OpenShift 4.10 to 4.13.

   You can upgrade OpenShift with minimal downtime .
2. Upgrade the Cluster Operator.

3. Upgrade all Kafka brokers and client applications to the latest supported Kafka version.
4. Optional: Upgrade consumers and Kafka Streams applications to use the *incremental cooperative rebalance* protocol for partition rebalances.

16.3. UPGRADING OPENSHIFT WITH MINIMAL DOWNTIME

If you are upgrading OpenShift, refer to the OpenShift upgrade documentation to check the upgrade path and the steps to upgrade your nodes correctly. Before upgrading OpenShift, check the supported versions for your version of AMQ Streams.

When performing your upgrade, you'll want to keep your Kafka clusters available.

You can employ one of the following strategies:

1. Configuring pod disruption budgets
2. Rolling pods by one of these methods:
   a. Using the AMQ Streams Drain Cleaner
   b. Manually by applying an annotation to your pod

You have to configure the pod disruption budget before using one of the methods to roll your pods.

For Kafka to stay operational, topics must also be replicated for high availability. This requires topic configuration that specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

In a highly available environment, the Cluster Operator maintains a minimum number of in-sync replicas for topics during the upgrade process so that there is no downtime.

16.3.1. Rolling pods using the AMQ Streams Drain Cleaner

You can use the AMQ Streams Drain Cleaner tool to evict nodes during an upgrade. The AMQ Streams Drain Cleaner annotates pods with a rolling update pod annotation. This informs the Cluster Operator to perform a rolling update of an evicted pod.

A pod disruption budget allows only a specified number of pods to be unavailable at a given time. During planned maintenance of Kafka broker pods, a pod disruption budget ensures Kafka continues to run in a highly available environment.

You specify a pod disruption budget using a **template** customization for a Kafka component. By default, pod disruption budgets allow only a single pod to be unavailable at a given time.

To do this, you set **maxUnavailable** to **0** (zero). Reducing the maximum pod disruption budget to zero prevents voluntary disruptions, so pods must be evicted manually.

Specifying a pod disruption budget

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    template:
      podDisruptionBudget:
        maxUnavailable: 0
# ...
```

16.3.2. Rolling pods manually while keeping topics available

During an upgrade, you can trigger a manual rolling update of pods through the Cluster Operator. Using **Pod** resources, rolling updates restart the pods of resources with new pods. As with using the AMQ Streams Drain Cleaner, you'll need to set the **maxUnavailable** value to zero for the pod disruption budget.

You need to watch the pods that need to be drained. You then add a pod annotation to make the update.

Here, the annotation updates a Kafka broker.

Performing a manual rolling update on a Kafka broker pod

```
oc annotate pod <cluster_name>-kafka-<index> strimzi.io/manual-rolling-update=true
```

You replace *<cluster_name>* with the name of the cluster. Kafka broker pods are named *<cluster-name>*-kafka-*<index>*, where *<index>* starts at zero and ends at the total number of replicas minus one. For example, **my-cluster-kafka-0**.

Additional resources

- Draining pods using the AMQ Streams Drain Cleaner
- Performing a rolling update using a pod annotation
- **PodDisruptionBudgetTemplate** schema reference
- OpenShift documentation

16.4. UPGRADING THE CLUSTER OPERATOR

Use the same method to upgrade the Cluster Operator as the initial method of deployment.

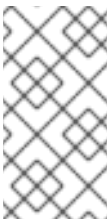
Using installation files

If you deployed the Cluster Operator using the installation YAML files, perform your upgrade by modifying the Operator installation files, as described in Upgrading the Cluster Operator using installation files.

Using the OperatorHub

If you deployed AMQ Streams from the OperatorHub, use the Operator Lifecycle Manager (OLM) to change the update channel for the AMQ Streams operators to a new AMQ Streams version. Updating the channel starts one of the following types of upgrade, depending on your chosen upgrade strategy:

- An automatic upgrade is initiated
- A manual upgrade that requires approval before installation begins



NOTE

If you subscribe to the *stable* channel, you can get automatic updates without changing channels. However, enabling automatic updates is not recommended because of the potential for missing any pre-installation upgrade steps. Use automatic upgrades only on version-specific channels.

For more information on using the OperatorHub to upgrade Operators, see Upgrading installed Operators (OpenShift documentation).

16.4.1. Upgrading the Cluster Operator returns Kafka version error

If you upgrade the Cluster Operator and get an *unsupported Kafka version* error, your Kafka cluster deployment has an older Kafka version that is not supported by the new operator version. This error applies to all installation methods.

If this error occurs, upgrade Kafka to a supported Kafka version. Change the **spec.kafka.version** in the **Kafka** resource to the supported version.

You can use **oc** to check for error messages like this in the **status** of the **Kafka** resource.

Checking the Kafka status for errors

```
oc get kafka <kafka_cluster_name> -n <namespace> -o jsonpath='{.status.conditions}'
```

Replace ***<kafka_cluster_name>*** with the name of your Kafka cluster and ***<namespace>*** with the OpenShift namespace where the pod is running.

16.4.2. Upgrading from AMQ Streams 1.7 or earlier using the OperatorHub

Action required if upgrading from AMQ Streams 1.7 or earlier using the OperatorHub

Before you upgrade the **AMQ Streams Operator** to version 2.4, you need to make the following changes:

- Convert custom resources and CRDs to **v1beta2**
- Upgrade to a version of AMQ Streams where the **ControlPlaneListener** feature gate is disabled

These requirements are described in Section 16.1.2, "Upgrading from an AMQ Streams version earlier than 1.7".

If you are upgrading from AMQ Streams 1.7 or earlier, do the following:

1. Upgrade to AMQ Streams 1.7.
2. Download the **Red Hat AMQ Streams API Conversion Tool** provided with AMQ Streams 1.8 from the AMQ Streams software downloads page .
3. Convert custom resources and CRDs to **v1beta2**.
   For more information, see the AMQ Streams 1.7 upgrade documentation .
4. In the OperatorHub, delete version 1.7 of the **AMQ Streams Operator**.
5. If it also exists, delete version 2.4 of the **AMQ Streams Operator**.
   If it does not exist, go to the next step.

   If the **Approval Strategy** for the AMQ Streams Operator was set to **Automatic**, version 2.4 of the operator might already exist in your cluster. If you did *not* convert custom resources and CRDs to the **v1beta2** API version before release, the operator-managed custom resources and CRDs will be using the old API version. As a result, the 2.4 Operator is stuck in *Pending* status. In this situation, you need to delete version 2.4 of the **AMQ Streams Operator** as well as version 1.7.

   If you delete both operators, reconciliations are paused until the new operator version is installed. Follow the next steps immediately so that any changes to custom resources are not delayed.
6. In the OperatorHub, do one of the following:
   - Upgrade to version 1.8 of the **AMQ Streams Operator** (where the **ControlPlaneListener** feature gate is disabled by default).
   - Upgrade to version 2.0 or 2.2 of the **AMQ Streams Operator** (where the **ControlPlaneListener** feature gate is enabled by default) with the **ControlPlaneListener** feature gate disabled.
7. Upgrade to version 2.4 of the **AMQ Streams Operator** immediately.
   The installed 2.4 operator begins to watch the cluster and performs rolling updates. You might notice a temporary decrease in cluster performance during this process.

16.4.3. Upgrading the Cluster Operator using installation files

This procedure describes how to upgrade a Cluster Operator deployment to use AMQ Streams 2.4.

Follow this procedure if you deployed the Cluster Operator using the installation YAML files.

The availability of Kafka clusters managed by the Cluster Operator is not affected by the upgrade operation.

NOTE

Refer to the documentation supporting a specific version of AMQ Streams for information on how to upgrade to that version.

Prerequisites

- An existing Cluster Operator deployment is available.
- You have downloaded the release artifacts for AMQ Streams 2.4 .

Procedure

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the **/install/cluster-operator** directory). Any changes will be **overwritten** by the new version of the Cluster Operator.
2. Update your custom resources to reflect the supported configuration options available for AMQ Streams version 2.4.
3. Update the Cluster Operator.
   a. Modify the installation files for the new Cluster Operator version according to the namespace the Cluster Operator is running in.
   On Linux, use:

   ```
   sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

   On MacOS, use:

   ```
   sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

   b. If you modified one or more environment variables in your existing Cluster Operator **Deployment**, edit the **install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** file to use those environment variables.
4. When you have an updated configuration, deploy it along with the rest of the installation resources:

   ```
   oc replace -f install/cluster-operator
   ```

   Wait for the rolling updates to complete.
5. If the new Operator version no longer supports the Kafka version you are upgrading from, the Cluster Operator returns an error message to say the version is not supported. Otherwise, no error message is returned.
   - If the error message is returned, upgrade to a Kafka version that is supported by the new Cluster Operator version:
     a. Edit the **Kafka** custom resource.
     b. Change the **spec.kafka.version** property to a supported Kafka version.

- If the error message is *not* returned, go to the next step. You will upgrade the Kafka version later.

6. Get the image for the Kafka pod to ensure the upgrade was successful:

```
oc get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The image tag shows the new Operator version. For example:

```
registry.redhat.io/amq-streams/strimzi-kafka-34-rhel8:2.4.0
```

Your Cluster Operator was upgraded to version 2.4 but the version of Kafka running in the cluster it manages is unchanged.

Following the Cluster Operator upgrade, you must perform a Kafka upgrade.

16.5. SWITCHING TO FIPS MODE WHEN UPGRADING AMQ STREAMS

Upgrade AMQ Streams to run in FIPS mode on FIPS-enabled OpenShift clusters. Until AMQ Streams 2.4, running on FIPS-enabled OpenShift clusters was possible only by disabling FIPS mode using the **FIPS_MODE** environment variable. From release 2.4, AMQ Streams supports FIPS mode. If you run AMQ Streams on a FIPS-enabled OpenShift cluster with the **FIPS_MODE** set to **disabled**, you can enable it by following this procedure.

Prerequisites

- FIPS-enabled OpenShift cluster
- An existing Cluster Operator deployment with the **FIPS_MODE** environment variable set to **disabled**

Procedure

1. Upgrade the Cluster Operator to version 2.4 or newer but keep the **FIPS_MODE** environment variable set to **disabled**.
2. If you initially deployed an AMQ Streams version older than 2.3, it might use old encryption and digest algorithms in its PKCS #12 stores, which are not supported with FIPS enabled. To recreate the certificates with updated algorithms, renew the cluster and clients CA certificates.
   a. To renew the CAs generated by the Cluster Operator, add the **force-renew** annotation to the CA secrets to trigger a renewal.
   b. To renew your own CAs, add the new certificate to the CA secret and update the **ca-cert-generation** annotation with a higher incremental value to capture the update .
3. If you use SCRAM-SHA-512 authentication, check the password length of your users. If they are less than 32 characters long, generate a new password in one of the following ways:
   a. Delete the user secret so that the User Operator generates a new one with a new password of sufficient length.
   b. If you provided your password using the **.spec.authentication.password** properties of the **KafkaUser** custom resource, update the password in the OpenShift secret referenced in the same password configuration. Don't forget to update your clients to use the new

passwords.

4. Ensure that the CA certificates are using the correct algorithms and the SCRAM-SHA-512 passwords are of sufficient length. You can then enable the FIPS mode.

5. Remove the **FIPS_MODE** environment variable from the Cluster Operator deployment. This restarts the Cluster Operator and rolls all the operands to enable the FIPS mode. After the restart is complete, all Kafka clusters now run with FIPS mode enabled.

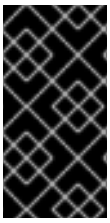
16.6. UPGRADING KAFKA

After you have upgraded your Cluster Operator to 2.4, the next step is to upgrade all Kafka brokers to the latest supported version of Kafka.

Kafka upgrades are performed by the Cluster Operator through rolling updates of the Kafka brokers.

The Cluster Operator initiates rolling updates based on the Kafka cluster configuration.

If **Kafka.spec.kafka.config** contains...	The Cluster Operator initiates...
Both the **inter.broker.protocol.version** and the **log.message.format.version**.	A single rolling update. After the update, the **inter.broker.protocol.version** must be updated manually, followed by **log.message.format.version**. Changing each will trigger a further rolling update.
Either the **inter.broker.protocol.version** or the **log.message.format.version**.	Two rolling updates.
No configuration for the **inter.broker.protocol.version** or the **log.message.format.version**.	Two rolling updates.



IMPORTANT

From Kafka 3.0.0, when the **inter.broker.protocol.version** is set to **3.0** or higher, the **log.message.format.version** option is ignored and doesn't need to be set. The **log.message.format.version** property for brokers and the **message.format.version** property for topics are deprecated and will be removed in a future release of Kafka.

As part of the Kafka upgrade, the Cluster Operator initiates rolling updates for ZooKeeper.

- A single rolling update occurs even if the ZooKeeper version is unchanged.
- Additional rolling updates occur if the new version of Kafka requires a new ZooKeeper version.

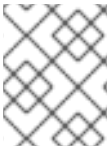
16.6.1. Kafka versions

Kafka's log message format version and inter-broker protocol version specify, respectively, the log format version appended to messages and the version of the Kafka protocol used in a cluster. To ensure the correct versions are used, the upgrade process involves making configuration changes to existing Kafka brokers and code changes to client applications (consumers and producers).

The following table shows the differences between Kafka versions:

Table 16.1. Kafka version differences

AMQ Streams version	Kafka version	Inter-broker protocol version	Log message format version	ZooKeeper version
2.4	3.4.0	3.4	3.4	3.6.3
2.3	3.3.1	3.3	3.3	3.6.3



NOTE

AMQ Streams 2.4 uses Kafka 3.4.0, but Kafka 3.3.1 is also supported for the purpose of upgrading.

Inter-broker protocol version

In Kafka, the network protocol used for inter-broker communication is called the *inter-broker protocol*. Each version of Kafka has a compatible version of the inter-broker protocol. The minor version of the protocol typically increases to match the minor version of Kafka, as shown in the preceding table.

The inter-broker protocol version is set cluster wide in the **Kafka** resource. To change it, you edit the **inter.broker.protocol.version** property in **Kafka.spec.kafka.config**.

Log message format version

When a producer sends a message to a Kafka broker, the message is encoded using a specific format. The format can change between Kafka releases, so messages specify which version of the message format they were encoded with.

The properties used to set a specific message format version are as follows:

- **message.format.version** property for topics
- **log.message.format.version** property for Kafka brokers

From Kafka 3.0.0, the message format version values are assumed to match the **inter.broker.protocol.version** and don't need to be set. The values reflect the Kafka version used.

When upgrading to Kafka 3.0.0 or higher, you can remove these settings when you update the **inter.broker.protocol.version**. Otherwise, set the message format version based on the Kafka version you are upgrading to.

The default value of **message.format.version** for a topic is defined by the **log.message.format.version** that is set on the Kafka broker. You can manually set the **message.format.version** of a topic by modifying its topic configuration.

16.6.2. Strategies for upgrading clients

Upgrading Kafka clients ensures that they benefit from the features, fixes, and improvements that are introduced in new versions of Kafka. Upgraded clients maintain compatibility with other upgraded Kafka components. The performance and stability of the clients might also be improved.

Consider the best approach for upgrading Kafka clients and brokers to ensure a smooth transition. The

chosen upgrade strategy depends on whether you are upgrading brokers or clients first. Since Kafka 3.0, you can upgrade brokers and client independently and in any order. The decision to upgrade clients or brokers first depends on several factors, such as the number of applications that need to be upgraded and how much downtime is tolerable.

If you upgrade clients before brokers, some new features may not work as they are not yet supported by brokers. However, brokers can handle producers and consumers running with different versions and supporting different log message versions.

Upgrading clients when using Kafka versions older than Kafka 3.0

Before Kafka 3.0, you would configure a specific message format for brokers using the **log.message.format.version** property (or the **message.format.version** property at the topic level). This allowed brokers to support older Kafka clients that were using an outdated message format. Otherwise, the brokers would need to convert the messages from the older clients, which came with a significant performance cost.

Apache Kafka Java clients have supported the latest message format version since version 0.11. If all of your clients are using the latest message version, you can remove the **log.message.format.version** or **message.format.version** overrides when upgrading your brokers.

However, if you still have clients that are using an older message format version, we recommend upgrading your clients first. Start with the consumers, then upgrade the producers before removing the **log.message.format.version** or **message.format.version** overrides when upgrading your brokers. This will ensure that all of your clients can support the latest message format version and that the upgrade process goes smoothly.

You can track Kafka client names and versions using this metric:

- **kafka.server:type=socket-server-metrics,clientSoftwareName=<name>,clientSoftwareVersion=<version>,listener=<listener>,networkProcessor=<processor>**

TIP

The following Kafka broker metrics help monitor the performance of message down-conversion:

- **kafka.network:type=RequestMetrics,name=MessageConversionsTimeMs,request={Produce|Fetch}** provides metrics on the time taken to perform message conversion.
- **kafka.server:type=BrokerTopicMetrics,name={Produce|Fetch}MessageConversionsPerSec,topic=([-.\w]+)** provides metrics on the number of messages converted over a period of time.

16.6.3. Kafka version and image mappings

When upgrading Kafka, consider your settings for the **STRIMZI_KAFKA_IMAGES** environment variable and the **Kafka.spec.kafka.version** property.

- Each **Kafka** resource can be configured with a **Kafka.spec.kafka.version**.
- The Cluster Operator's **STRIMZI_KAFKA_IMAGES** environment variable provides a mapping between the Kafka version and the image to be used when that version is requested in a given **Kafka** resource.
    - If **Kafka.spec.kafka.image** is not configured, the default image for the given version is used.

- If **Kafka.spec.kafka.image** is configured, the default image is overridden.

WARNING

The Cluster Operator cannot validate that an image actually contains a Kafka broker of the expected version. Take care to ensure that the given image corresponds to the given Kafka version.

16.6.4. Upgrading Kafka brokers and client applications

Upgrade an AMQ Streams Kafka cluster to the latest supported Kafka version and *inter-broker protocol version*.

You should also choose a strategy for upgrading clients. Kafka clients are upgraded in step 6 of this procedure.

Prerequisites

- The Cluster Operator is up and running.
- Before you upgrade the AMQ Streams Kafka cluster, check that the **Kafka.spec.kafka.config** properties of the **Kafka** resource do *not* contain configuration options that are not supported in the new Kafka version.

Procedure

1. Update the Kafka cluster configuration:

```
oc edit kafka <my_cluster>
```

2. If configured, check that the **inter.broker.protocol.version** and **log.message.format.version** properties are set to the *current* version.
   For example, the current version is 3.3 if upgrading from Kafka version 3.3.1 to 3.4.0:

```
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.3.1
    config:
      log.message.format.version: "3.3"
      inter.broker.protocol.version: "3.3"
      # ...
```

   If **log.message.format.version** and **inter.broker.protocol.version** are not configured, AMQ Streams automatically updates these versions to the current defaults after the update to the Kafka version in the next step.

NOTE

The value of **log.message.format.version** and **inter.broker.protocol.version** must be strings to prevent them from being interpreted as floating point numbers.

3. Change the **Kafka.spec.kafka.version** to specify the new Kafka version; leave the **log.message.format.version** and **inter.broker.protocol.version** at the defaults for the *current* Kafka version.

NOTE

Changing the **kafka.version** ensures that all brokers in the cluster will be upgraded to start using the new broker binaries. During this process, some brokers are using the old binaries while others have already upgraded to the new ones. Leaving the **inter.broker.protocol.version** unchanged at the current setting ensures that the brokers can continue to communicate with each other throughout the upgrade.

For example, if upgrading from Kafka 3.3.1 to 3.4.0:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.4.0 (1)
    config:
      log.message.format.version: "3.3" (2)
      inter.broker.protocol.version: "3.3" (3)
      # ...
```

1. Kafka version is changed to the new version.
2. Message format version is unchanged.
3. Inter-broker protocol version is unchanged.

WARNING

You cannot downgrade Kafka if the **inter.broker.protocol.version** for the new Kafka version changes. The inter-broker protocol version determines the schemas used for persistent metadata stored by the broker, including messages written to **__consumer_offsets**. The downgraded cluster will not understand the messages.

4. If the image for the Kafka cluster is defined in the Kafka custom resource, in **Kafka.spec.kafka.image**, update the **image** to point to a container image with the new Kafka version.
   See Kafka version and image mappings

5. Save and exit the editor, then wait for rolling updates to complete.
   Check the progress of the rolling updates by watching the pod state transitions:

   ```
   oc get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
   ```

   The rolling updates ensure that each pod is using the broker binaries for the new version of Kafka.

6. Depending on your chosen strategy for upgrading clients, upgrade all client applications to use the new version of the client binaries.
   If required, set the **version** property for Kafka Connect and MirrorMaker as the new version of Kafka:

   a. For Kafka Connect, update **KafkaConnect.spec.version**.

   b. For MirrorMaker, update **KafkaMirrorMaker.spec.version**.

   c. For MirrorMaker 2, update **KafkaMirrorMaker2.spec.version**.

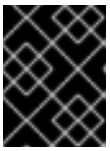
7. If configured, update the Kafka resource to use the new **inter.broker.protocol.version** version.
   Otherwise, go to step 9.
   For example, if upgrading to Kafka 3.4.0:

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   spec:
     # ...
     kafka:
       version: 3.4.0
       config:
         log.message.format.version: "3.3"
         inter.broker.protocol.version: "3.4"
         # ...
   ```

8. Wait for the Cluster Operator to update the cluster.

9. If configured, update the Kafka resource to use the new **log.message.format.version** version.
   Otherwise, go to step 10.
   For example, if upgrading to Kafka 3.4.0:

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   spec:
     # ...
     kafka:
       version: 3.4.0
       config:
         log.message.format.version: "3.4"
         inter.broker.protocol.version: "3.4"
         # ...
   ```



IMPORTANT

From Kafka 3.0.0, when the **inter.broker.protocol.version** is set to **3.0** or higher, the **log.message.format.version** option is ignored and doesn't need to be set.

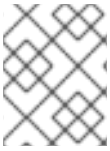
10. Wait for the Cluster Operator to update the cluster.
    - The Kafka cluster and clients are now using the new Kafka version.
    - The brokers are configured to send messages using the inter-broker protocol version and message format version of the new version of Kafka.

Following the Kafka upgrade, if required, you can upgrade consumers to use the incremental cooperative rebalance protocol.

16.7. UPGRADING CONSUMERS TO COOPERATIVE REBALANCING

You can upgrade Kafka consumers and Kafka Streams applications to use the *incremental cooperative rebalance* protocol for partition rebalances instead of the default *eager rebalance* protocol. The new protocol was added in Kafka 2.4.0.

Consumers keep their partition assignments in a cooperative rebalance and only revoke them at the end of the process, if needed to achieve a balanced cluster. This reduces the unavailability of the consumer group or Kafka Streams application.



NOTE

Upgrading to the incremental cooperative rebalance protocol is optional. The eager rebalance protocol is still supported.

Prerequisites

- You have upgraded Kafka brokers and client applications to Kafka 3.4.0.

Procedure

To upgrade a Kafka consumer to use the incremental cooperative rebalance protocol:

1. Replace the Kafka clients **.jar** file with the new version.
2. In the consumer configuration, append **cooperative-sticky** to the **partition.assignment.strategy**. For example, if the **range** strategy is set, change the configuration to **range, cooperative-sticky**.
3. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.
4. Reconfigure each consumer in the group by removing the earlier **partition.assignment.strategy** from the consumer configuration, leaving only the **cooperative-sticky** strategy.
5. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.

To upgrade a Kafka Streams application to use the incremental cooperative rebalance protocol:

1. Replace the Kafka Streams **.jar** file with the new version.
2. In the Kafka Streams configuration, set the **upgrade.from** configuration parameter to the Kafka version you are upgrading from (for example, 2.3).
3. Restart each of the stream processors (nodes) in turn.
4. Remove the **upgrade.from** configuration parameter from the Kafka Streams configuration.
5. Restart each consumer in the group in turn.

CHAPTER 17. DOWNGRADING AMQ STREAMS

If you are encountering issues with the version of AMQ Streams you upgraded to, you can revert your installation to the previous version.

If you used the YAML installation files to install AMQ Streams, you can use the YAML installation files from the previous release to perform the following downgrade procedures:

1. Section 17.1, "Downgrading the Cluster Operator to a previous version"
2. Section 17.2, "Downgrading Kafka"

If the previous version of AMQ Streams does not support the version of Kafka you are using, you can also downgrade Kafka as long as the log message format versions appended to messages match.

WARNING

The following downgrade instructions are only suitable if you installed AMQ Streams using the installation files. If you installed AMQ Streams using another method, like OperatorHub, downgrade may not be supported by that method unless specified in their documentation. To ensure a successful downgrade process, it is essential to use a supported approach.

17.1. DOWNGRADING THE CLUSTER OPERATOR TO A PREVIOUS VERSION

If you are encountering issues with AMQ Streams, you can revert your installation.

This procedure describes how to downgrade a Cluster Operator deployment to a previous version.

Prerequisites

- An existing Cluster Operator deployment is available.
- You have downloaded the installation files for the previous version .

Before you begin

Check the downgrade requirements of the AMQ Streams feature gates. If a feature gate is permanently enabled, you may need to downgrade to a version that allows you to disable it before downgrading to your target version.

Procedure

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the **/install/cluster-operator** directory). Any changes will be **overwritten** by the previous version of the Cluster Operator.
2. Revert your custom resources to reflect the supported configuration options available for the version of AMQ Streams you are downgrading to.

3. Update the Cluster Operator.

   a. Modify the installation files for the previous version according to the namespace the Cluster Operator is running in.
   On Linux, use:

   ```
   sed -i 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

   On MacOS, use:

   ```
   sed -i '' 's/namespace: .*/namespace: my-cluster-operator-namespace/' install/cluster-operator/*RoleBinding*.yaml
   ```

   b. If you modified one or more environment variables in your existing Cluster Operator **Deployment**, edit the **install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** file to use those environment variables.

4. When you have an updated configuration, deploy it along with the rest of the installation resources:

   ```
   oc replace -f install/cluster-operator
   ```

   Wait for the rolling updates to complete.

5. Get the image for the Kafka pod to ensure the downgrade was successful:

   ```
   oc get pod my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
   ```

   The image tag shows the new AMQ Streams version followed by the Kafka version. For example, ***NEW-STRIMZI-VERSION*-kafka-*CURRENT-KAFKA-VERSION***.

Your Cluster Operator was downgraded to the previous version.

17.2. DOWNGRADING KAFKA

Kafka version downgrades are performed by the Cluster Operator.

17.2.1. Kafka version compatibility for downgrades

Kafka downgrades are dependent on compatible current and target Kafka versions, and the state at which messages have been logged.

You cannot revert to the previous Kafka version if that version does not support any of the **inter.broker.protocol.version** settings which have *ever been used* in that cluster, or messages have been added to message logs that use a newer **log.message.format.version**.

The **inter.broker.protocol.version** determines the schemas used for persistent metadata stored by the broker, such as the schema for messages written to **__consumer_offsets**. If you downgrade to a version of Kafka that does not understand an **inter.broker.protocol.version** that has ever been previously used in the cluster the broker will encounter data it cannot understand.

If the target downgrade version of Kafka has:

- The *same* **log.message.format.version** as the current version, the Cluster Operator downgrades by performing a single rolling restart of the brokers.
- A *different* **log.message.format.version**, downgrading is only possible if the running cluster has *always* had **log.message.format.version** set to the version used by the downgraded version. This is typically only the case if the upgrade procedure was aborted before the **log.message.format.version** was changed. In this case, the downgrade requires:
  - Two rolling restarts of the brokers if the interbroker protocol of the two versions is different
  - A single rolling restart if they are the same

Downgrading is *not possible* if the new version has ever used a **log.message.format.version** that is not supported by the previous version, including when the default value for **log.message.format.version** is used. For example, this resource can be downgraded to Kafka version 3.3.1 because the **log.message.format.version** has not been changed:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.4.0
    config:
      log.message.format.version: "3.3"
      # ...
```

The downgrade would not be possible if the **log.message.format.version** was set at **"3.4"** or a value was absent, so that the parameter took the default value for a 3.4.0 broker of 3.4.

IMPORTANT

From Kafka 3.0.0, when the **inter.broker.protocol.version** is set to **3.0** or higher, the **log.message.format.version** option is ignored and doesn't need to be set.

17.2.2. Downgrading Kafka brokers and client applications

Downgrade an AMQ Streams Kafka cluster to a lower (previous) version of Kafka, such as downgrading from 3.4.0 to 3.3.1.

Prerequisites

- The Cluster Operator is up and running.
- Before you downgrade the AMQ Streams Kafka cluster, check the following for the **Kafka** resource:
  - IMPORTANT: Compatibility of Kafka versions.
  - **Kafka.spec.kafka.config** does not contain options that are not supported by the Kafka version being downgraded to.
  - **Kafka.spec.kafka.config** has a **log.message.format.version** and **inter.broker.protocol.version** that is supported by the Kafka version being downgraded to. From Kafka 3.0.0, when the **inter.broker.protocol.version** is set to **3.0** or higher, the **log.message.format.version** option is ignored and doesn't need to be set.

Procedure

1. Update the Kafka cluster configuration.

```
oc edit kafka KAFKA-CONFIGURATION-FILE
```

2. Change the **Kafka.spec.kafka.version** to specify the previous version.
For example, if downgrading from Kafka 3.4.0 to 3.3.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.3.1 (1)
    config:
      log.message.format.version: "3.3" (2)
      inter.broker.protocol.version: "3.3" (3)
      # ...
```

(1) Kafka version is changed to the previous version.

(2) Message format version is unchanged.

(3) Inter-broker protocol version is unchanged.

NOTE

The value of **log.message.format.version** and **inter.broker.protocol.version** must be strings to prevent them from being interpreted as floating point numbers.

3. If the image for the Kafka version is different from the image defined in **STRIMZI_KAFKA_IMAGES** for the Cluster Operator, update **Kafka.spec.kafka.image**.
See Section 16.6.3, "Kafka version and image mappings"

4. Save and exit the editor, then wait for rolling updates to complete.
Check the update in the logs or by watching the pod state transitions:

```
oc logs -f CLUSTER-OPERATOR-POD-NAME | grep -E "Kafka version downgrade from [0-9.]+ to [0-9.]+, phase ([0-9]+) of \1 completed"
```

```
oc get pod -w
```

Check the Cluster Operator logs for an **INFO** level message:

```
Reconciliation #NUM(watch) Kafka(NAMESPACE/NAME): Kafka version downgrade from FROM-VERSION to TO-VERSION, phase 1 of 1 completed
```

5. Downgrade all client applications (consumers) to use the previous version of the client binaries.
The Kafka cluster and clients are now using the previous Kafka version.

6. If you are reverting back to a version of AMQ Streams earlier than 1.7, which uses ZooKeeper for the storage of topic metadata, delete the internal topic store topics from the Kafka cluster.

```
oc run kafka-admin -ti --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 --rm=true
--restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-
topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-
server localhost:9092 --topic __strimzi_store_topic --delete
```

Additional resources

- Section 13.3.2, “Topic Operator topic store”

CHAPTER 18. HANDLING HIGH VOLUMES OF MESSAGES

If your AMQ Streams deployment needs to handle a high volume of messages, you can use configuration options to optimize for throughput and latency.

Producer and consumer configuration can help control the size and frequency of requests to Kafka brokers. For more information on the configuration options, see the following:

- Apache Kafka configuration documentation for producers
- Apache Kafka configuration documentation for consumers

You can also use the same configuration options with the producers and consumers used by the Kafka Connect runtime source connectors (including MirrorMaker 2) and sink connectors.

Source connectors

- Producers from the Kafka Connect runtime send messages to the Kafka cluster.
- For MirrorMaker 2, since the source system is Kafka, consumers retrieve messages from a source Kafka cluster.

Sink connectors

- Consumers from the Kafka Connect runtime retrieve messages from the Kafka cluster.

For consumers, you might increase the amount of data fetched in a single fetch request to reduce latency. You increase the fetch request size using the **fetch.max.bytes** and **max.partition.fetch.bytes** properties. You can also set a maximum limit on the number of messages returned from the consumer buffer using the **max.poll.records** property.

For MirrorMaker 2, configure the **fetch.max.bytes**, **max.partition.fetch.bytes**, and **max.poll.records** values at the source connector level (**consumer.***), as they relate to the specific consumer that fetches messages from the source.

For producers, you might increase the size of the message batches sent in a single produce request. You increase the batch size using the **batch.size** property. A larger batch size reduces the number of outstanding messages ready to be sent and the size of the backlog in the message queue. Messages being sent to the same partition are batched together. A produce request is sent to the target cluster when the batch size is reached. By increasing the batch size, produce requests are delayed and more messages are added to the batch and sent to brokers at the same time. This can improve throughput when you have just a few topic partitions that handle large numbers of messages.

Consider the number and size of the records that the producer handles for a suitable producer batch size.

Use **linger.ms** to add a wait time in milliseconds to delay produce requests when producer load decreases. The delay means that more records can be added to batches if they are under the maximum batch size.

Configure the **batch.size** and **linger.ms** values at the source connector level (**producer.override.***), as they relate to the specific producer that sends messages to the target Kafka cluster.

For Kafka Connect source connectors, the data streaming pipeline to the target Kafka cluster is as follows:

Data streaming pipeline for Kafka Connect source connector

external data source → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

For Kafka Connect sink connectors, the data streaming pipeline to the target external data source is as follows:

Data streaming pipeline for Kafka Connect sink connector

source Kafka topic → (Kafka Connect tasks) sink message queue → consumer buffer → external data source

For MirrorMaker 2, the data mirroring pipeline to the target Kafka cluster is as follows:

Data mirroring pipeline for MirrorMaker 2

source Kafka topic → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

The producer sends messages in its buffer to topics in the target Kafka cluster. While this is happening, Kafka Connect tasks continue to poll the data source to add messages to the source message queue.

The size of the producer buffer for the source connector is set using the **producer.override.buffer.memory** property. Tasks wait for a specified timeout period (**offset.flush.timeout.ms**) before the buffer is flushed. This should be enough time for the sent messages to be acknowledged by the brokers and offset data committed. The source task does not wait for the producer to empty the message queue before committing offsets, except during shutdown.

If the producer is unable to keep up with the throughput of messages in the source message queue, buffering is blocked until there is space available in the buffer within a time period bounded by **max.block.ms**. Any unacknowledged messages still in the buffer are sent during this period. New messages are not added to the buffer until these messages are acknowledged and flushed.

You can try the following configuration changes to keep the underlying source message queue of outstanding messages at a manageable size:

- Increasing the default value in milliseconds of the **offset.flush.timeout.ms**
- Ensuring that there are enough CPU and memory resources
- Increasing the number of tasks that run in parallel by doing the following:
  - Increasing the number of tasks that run in parallel using the **tasksMax** property
  - Increasing the number of worker nodes that run tasks using the **replicas** property

Consider the number of tasks that can run in parallel according to the available CPU and memory resources and number of worker nodes. You might need to keep adjusting the configuration values until they have the desired effect.

18.1. CONFIGURING KAFKA CONNECT FOR HIGH-VOLUME MESSAGES

Kafka Connect fetches data from the source external data system and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows configuration for Kafka Connect using the **KafkaConnect** custom resource.

Example Kafka Connect configuration for handling high volumes of messages

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  replicas: 3
  config:
    offset.flush.timeout.ms: 10000
    # ...
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  # ...
```

Producer configuration is added for the source connector, which is managed using the **KafkaConnector** custom resource.

Example source connector configuration for handling high volumes of messages

```
apiVersion: kafka.strimzi.io/v1beta2
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:
    producer.override.batch.size: 327680
    producer.override.linger.ms: 100
    # ...
```

NOTE

FileStreamSourceConnector and **FileStreamSinkConnector** are provided as example connectors. For information on deploying them as **KafkaConnector** resources, see Section 6.4.3.3, “Deploying KafkaConnector resources”.

Consumer configuration is added for the sink connector.

Example sink connector configuration for handling high volumes of messages

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
```

```
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector
  tasksMax: 2
  config:
    consumer.fetch.max.bytes: 52428800
    consumer.max.partition.fetch.bytes: 1048576
    consumer.max.poll.records: 500
    # ...
```

If you are using the Kafka Connect API instead of the **KafkaConnector** custom resource to manage your connectors, you can add the connector configuration as a JSON object.

Example curl request to add source connector configuration for handling high volumes of messages

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
      "producer.override.batch.size": 327680
      "producer.override.linger.ms": 100
    }
}'
```

18.2. CONFIGURING MIRRORMAKER 2 FOR HIGH-VOLUME MESSAGES

MirrorMaker 2 fetches data from the source cluster and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows the configuration for MirrorMaker 2 using the **KafkaMirrorMaker2** custom resource.

Example MirrorMaker 2 configuration for handling high volumes of messages

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.4.0
  replicas: 1
  connectCluster: "my-cluster-target"
  clusters:
  - alias: "my-cluster-source"
```

```
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
  - alias: "my-cluster-target"
    config:
      offset.flush.timeout.ms: 10000
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 2
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100
        consumer.fetch.max.bytes: 52428800
        consumer.max.partition.fetch.bytes: 1048576
        consumer.max.poll.records: 500
    # ...
  resources:
    requests:
      cpu: "1"
      memory: Gi
    limits:
      cpu: "2"
      memory: 4Gi
```

18.3. CHECKING THE MIRRORMAKER 2 MESSAGE FLOW

If you are using Prometheus and Grafana to monitor your deployment, you can check the MirrorMaker 2 message flow.

The example MirrorMaker 2 Grafana dashboards provided with AMQ Streams show the following metrics related to the flush pipeline.

- The number of messages in Kafka Connect's outstanding messages queue
- The available bytes of the producer buffer
- The offset commit timeout in milliseconds

You can use these metrics to gauge whether or not you need to tune your configuration based on the volume of messages.

Additional resources

- Chapter 14, *Setting up metrics and dashboards for AMQ Streams*
- Section 6.4.3, "Adding connectors"

CHAPTER 19. FINDING INFORMATION ON KAFKA RESTARTS

After the Cluster Operator restarts a Kafka pod in an OpenShift cluster, it emits an OpenShift event into the pod's namespace explaining why the pod restarted. For help in understanding cluster behavior, you can check restart events from the command line.

TIP

You can export and monitor restart events using metrics collection tools like Prometheus. Use the metrics tool with an *event exporter* that can export the output in a suitable format.

19.1. REASONS FOR A RESTART EVENT

The Cluster Operator initiates a restart event for a specific reason. You can check the reason by fetching information on the restart event.

The reason given depends on whether you are using **StrimziPodSet** or **StatefulSet** resources for the creation and management of pods.

Table 19.1. Restart reasons

StrimziPodSet	StatefulSet	Description
CaCertHasOldGeneration	CaCertHasOldGeneration	The pod is still using a server certificate signed with an old CA, so needs to be restarted as part of the certificate update.
CaCertRemoved	CaCertRemoved	Expired CA certificates have been removed, and the pod is restarted to run with the current certificates.
CaCertRenewed	CaCertRenewed	CA certificates have been renewed, and the pod is restarted to run with the updated certificates.
ClientCaCertKeyReplaced	ClientCaCertKeyReplaced	The key used to sign clients CA certificates has been replaced, and the pod is being restarted as part of the CA renewal process.
ClusterCaCertKeyReplaced	ClusterCaCertKeyReplaced	The key used to sign the cluster's CA certificates has been replaced, and the pod is being restarted as part of the CA renewal process.
ConfigChangeRequiresRestart	ConfigChangeRequiresRestart	Some Kafka configuration properties are changed dynamically, but others require that the broker be restarted.
CustomListenerCaCertChanged	CustomListenerCaCertChanged	The CA certificate used to secure the Kafka network listeners has changed, and the pod is restarted to use it.
FileSystemResizeNeeded	FileSystemResizeNeeded	The file system size has been increased, and a restart is needed to apply it.

StrimziPodSet	StatefulSet	Description
KafkaCertificatesChanged	KafkaCertificatesChanged	One or more TLS certificates used by the Kafka broker have been updated, and a restart is needed to use them.
ManualRollingUpdate	ManualRollingUpdate	A user annotated the pod, or the **StatefulSet** or **StrimziPodSet** set it belongs to, to trigger a restart.
PodForceRestartOnError	PodForceRestartOnError	An error occurred that requires a pod restart to rectify.
PodHasOldRevision	JbodVolumesChanged	A disk was added or removed from the Kafka volumes, and a restart is needed to apply the change. When using **StrimziPodSet** resources, the same reason is given if the pod needs to be recreated.
PodHasOldRevision	PodHasOldGeneration	The **StatefulSet** or **StrimziPodSet** that the pod is a member of has been updated, so the pod needs to be recreated. When using **StrimziPodSet** resources, the same reason is given if a disk was added or removed from the Kafka volumes.
PodStuck	PodStuck	The pod is still pending, and is not scheduled or cannot be scheduled, so the operator has restarted the pod in a final attempt to get it running.
PodUnresponsive	PodUnresponsive	AMQ Streams was unable to connect to the pod, which can indicate a broker not starting correctly, so the operator restarted it in an attempt to resolve the issue.

19.2. RESTART EVENT FILTERS

When checking restart events from the command line, you can specify a **field-selector** to filter on OpenShift event fields.

The following fields are available when filtering events with **field-selector**.

regardingObject.kind

The object that was restarted, and for restart events, the kind is always **Pod**.

regarding.namespace

The namespace that the pod belongs to.

regardingObject.name

The pod's name, for example, **strimzi-cluster-kafka-0**.

regardingObject.uid

The unique ID of the pod.

reason

The reason the pod was restarted, for example, **JbodVolumesChanged**.

reportingController

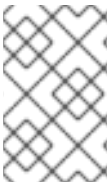
The reporting component is always **strimzi.io/cluster-operator** for AMQ Streams restart events.

source

source is an older version of **reportingController**. The reporting component is always **strimzi.io/cluster-operator** for AMQ Streams restart events.

type

The event type, which is either **Warning** or **Normal**. For AMQ Streams restart events, the type is **Normal**.



NOTE

In older versions of OpenShift, the fields using the **regarding** prefix might use an **involvedObject** prefix instead. **reportingController** was previously called **reportingComponent**.

19.3. CHECKING KAFKA RESTARTS

Use a **oc** command to list restart events initiated by the Cluster Operator. Filter restart events emitted by the Cluster Operator by setting the Cluster Operator as the reporting component using the **reportingController** or **source** event fields.

Prerequisites

- The Cluster Operator is running in the OpenShift cluster.

Procedure

1. Get all restart events emitted by the Cluster Operator:

```
oc -n kafka get events --field-selector reportingController=strimzi.io/cluster-operator
```

Example showing events returned

```
LAST SEEN   TYPE     REASON                  OBJECT                        MESSAGE
2m          Normal   CaCertRenewed           pod/strimzi-cluster-kafka-0   CA certificate
renewed
58m         Normal   PodForceRestartOnError  pod/strimzi-cluster-kafka-1   Pod needs to be
forcibly restarted due to an error
5m47s       Normal   ManualRollingUpdate     pod/strimzi-cluster-kafka-2   Pod was manually
annotated to be rolled
```

You can also specify a **reason** or other **field-selector** options to constrain the events returned.

Here, a specific reason is added:

```
oc -n kafka get events --field-selector reportingController=strimzi.io/cluster-
operator,reason=PodForceRestartOnError
```

2. Use an output format, such as YAML, to return more detailed information about one or more events.

```
oc -n kafka get events --field-selector reportingController=strimzi.io/cluster-
operator,reason=PodForceRestartOnError -o yaml
```

Example showing detailed events output

```
apiVersion: v1
items:
- action: StrimziInitiatedPodRestart
  apiVersion: v1
  eventTime: "2022-05-13T00:22:34.168086Z"
  firstTimestamp: null
  involvedObject:
    kind: Pod
    name: strimzi-cluster-kafka-1
    namespace: kafka
  kind: Event
  lastTimestamp: null
  message: Pod needs to be forcibly restarted due to an error
  metadata:
    creationTimestamp: "2022-05-13T00:22:34Z"
    generateName: strimzi-event
    name: strimzi-eventwppk6
    namespace: kafka
    resourceVersion: "432961"
    uid: 29fcdb9e-f2cf-4c95-a165-a5efcd48edfc
  reason: PodForceRestartOnError
  reportingController: strimzi.io/cluster-operator
  reportingInstance: strimzi-cluster-operator-6458cfb4c6-6bpdp
  source: {}
  type: Normal
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

The following fields are deprecated, so they are not populated for these events:

- **firstTimestamp**
- **lastTimestamp**
- **source**

CHAPTER 20. MANAGING AMQ STREAMS

Managing AMQ Streams requires performing various tasks to keep the Kafka clusters and associated resources running smoothly. Use **oc** commands to check the status of resources, configure maintenance windows for rolling updates, and leverage tools such as the AMQ Streams Drain Cleaner and Kafka Static Quota plugin to manage your deployment effectively.

20.1. WORKING WITH CUSTOM RESOURCES

You can use **oc** commands to retrieve information and perform other operations on AMQ Streams custom resources.

Using **oc** with the **status** subresource of a custom resource allows you to get the information about the resource.

20.1.1. Performing oc operations on custom resources

Use **oc** commands, such as **get**, **describe**, **edit**, or **delete**, to perform operations on resource types. For example, **oc get kafkatopics** retrieves a list of all Kafka topics and **oc get kafkas** retrieves all deployed Kafka clusters.

When referencing resource types, you can use both singular and plural names: **oc get kafkas** gets the same results as **oc get kafka**.

You can also use the *short name* of the resource. Learning short names can save you time when managing AMQ Streams. The short name for **Kafka** is **k**, so you can also run **oc get k** to list all Kafka clusters.

```
oc get k

NAME       DESIRED KAFKA REPLICAS   DESIRED ZK REPLICAS
my-cluster 3                        3
```

Table 20.1. Long and short names for each AMQ Streams resource

AMQ Streams resource	Long name	Short name
Kafka	kafka	k
Kafka Topic	kafkatopic	kt
Kafka User	kafkauser	ku
Kafka Connect	kafkaconnect	kc
Kafka Connector	kafkaconnector	kctr
Kafka Mirror Maker	kafkamirrormaker	kmm
Kafka Mirror Maker 2	kafkamirrormaker2	kmm2

AMQ Streams resource	Long name	Short name
Kafka Bridge	kafkabridge	kb
Kafka Rebalance	kafkarebalance	kr

20.1.1.1. Resource categories

Categories of custom resources can also be used in **oc** commands.

All AMQ Streams custom resources belong to the category **strimzi**, so you can use **strimzi** to get all the AMQ Streams resources with one command.

For example, running **oc get strimzi** lists all AMQ Streams custom resources in a given namespace.

```
oc get strimzi

NAME                                  DESIRED KAFKA REPLICAS DESIRED ZK REPLICAS
kafka.kafka.strimzi.io/my-cluster     3                      3

NAME                                  PARTITIONS REPLICATION FACTOR
kafkatopic.kafka.strimzi.io/kafka-apps 3          3

NAME                                  AUTHENTICATION AUTHORIZATION
kafkauser.kafka.strimzi.io/my-user    tls            simple
```

The **oc get strimzi -o name** command returns all resource types and resource names. The **-o name** option fetches the output in the *type/name* format

```
oc get strimzi -o name

kafka.kafka.strimzi.io/my-cluster
kafkatopic.kafka.strimzi.io/kafka-apps
kafkauser.kafka.strimzi.io/my-user
```

You can combine this **strimzi** command with other commands. For example, you can pass it into a **oc delete** command to delete all resources in a single command.

```
oc delete $(oc get strimzi -o name)

kafka.kafka.strimzi.io "my-cluster" deleted
kafkatopic.kafka.strimzi.io "kafka-apps" deleted
kafkauser.kafka.strimzi.io "my-user" deleted
```

Deleting all resources in a single operation might be useful, for example, when you are testing new AMQ Streams features.

20.1.1.2. Querying the status of sub-resources

There are other values you can pass to the **-o** option. For example, by using **-o yaml** you get the output in YAML format. Using **-o json** will return it as JSON.

You can see all the options in **oc get --help**.

One of the most useful options is the JSONPath support, which allows you to pass JSONPath expressions to query the Kubernetes API. A JSONPath expression can extract or navigate specific parts of any resource.

For example, you can use the JSONPath expression **{.status.listeners[?(@.name=="tls")].bootstrapServers}** to get the bootstrap address from the status of the Kafka custom resource and use it in your Kafka clients.

Here, the command finds the **bootstrapServers** value of the listener named **tls**:

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?(@.name=="tls")].bootstrapServers}{"\n"}'

my-cluster-kafka-bootstrap.myproject.svc:9093
```

By changing the name condition you can also get the address of the other Kafka listeners.

You can use **jsonpath** to extract any other property or group of properties from any custom resource.

20.1.2. AMQ Streams custom resource status information

Status properties provide status information for certain custom resources.

The following table lists the custom resources that provide status information (when deployed) and the schemas that define the status properties.

For more information on the schemas, see the Custom resource API reference.

Table 20.2. Custom resources that provide status information

AMQ Streams resource	Schema reference	Publishes status information on...
Kafka	**KafkaStatus** schema reference	The Kafka cluster
KafkaTopic	**KafkaTopicStatus** schema reference	Kafka topics in the Kafka cluster
KafkaUser	**KafkaUserStatus** schema reference	Kafka users in the Kafka cluster
KafkaConnect	**KafkaConnectStatus** schema reference	The Kafka Connect cluster
KafkaConnector	**KafkaConnectorStatus** schema reference	**KafkaConnector** resources
KafkaMirrorMaker2	**KafkaMirrorMaker2Status** schema reference	The Kafka MirrorMaker 2 cluster
KafkaMirrorMaker	**KafkaMirrorMakerStatus** schema reference	The Kafka MirrorMaker cluster

AMQ Streams resource	Schema reference	Publishes status information on...
KafkaBridge	**KafkaBridgeStatus** schema reference	The AMQ Streams Kafka Bridge
KafkaRebalance	**KafkaRebalance** schema reference	The status and results of a rebalance

The **status** property of a resource provides information on the state of the resource. The **status.conditions** and **status.observedGeneration** properties are common to all resources.

status.conditions

Status conditions describe the *current state* of a resource. Status condition properties are useful for tracking progress related to the resource achieving its *desired state*, as defined by the configuration specified in its **spec**. Status condition properties provide the time and reason the state of the resource changed, and details of events preventing or delaying the operator from realizing the desired state.

status.observedGeneration

Last observed generation denotes the latest reconciliation of the resource by the Cluster Operator. If the value of **observedGeneration** is different from the value of **metadata.generation** ((the current version of the deployment), the operator has not yet processed the latest update to the resource. If these values are the same, the status information reflects the most recent changes to the resource.

The **status** properties also provide resource-specific information. For example, **KafkaStatus** provides information on listener addresses, and the ID of the Kafka cluster.

AMQ Streams creates and maintains the status of custom resources, periodically evaluating the current state of the custom resource and updating its status accordingly. When performing an update on a custom resource using **oc edit**, for example, its **status** is not editable. Moreover, changing the **status** would not affect the configuration of the Kafka cluster.

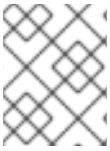
Here we see the **status** properties for a **Kafka** custom resource.

Kafka custom resource status

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
spec:
  # ...
status:
  clusterId: XP9FP2P-RByvEy0W4cOEUA (1)
  conditions: (2)
    - lastTransitionTime: '2023-01-20T17:56:29.396588Z'
      status: 'True'
      type: Ready (3)
  listeners: (4)
```

```
  - addresses:
      - host: my-cluster-kafka-bootstrap.prm-project.svc
        port: 9092
    bootstrapServers: 'my-cluster-kafka-bootstrap.prm-project.svc:9092'
    name: plain
    type: plain
  - addresses:
      - host: my-cluster-kafka-bootstrap.prm-project.svc
        port: 9093
    bootstrapServers: 'my-cluster-kafka-bootstrap.prm-project.svc:9093'
    certificates:
      - |
        -----BEGIN CERTIFICATE-----

        -----END CERTIFICATE-----
    name: tls
    type: tls
  - addresses:
      - host: >-
          2054284155.us-east-2.elb.amazonaws.com
        port: 9095
    bootstrapServers: >-
      2054284155.us-east-2.elb.amazonaws.com:9095
    certificates:
      - |
        -----BEGIN CERTIFICATE-----

        -----END CERTIFICATE-----
    name: external2
    type: external2
  - addresses:
      - host: ip-10-0-172-202.us-east-2.compute.internal
        port: 31644
    bootstrapServers: 'ip-10-0-172-202.us-east-2.compute.internal:31644'
    certificates:
      - |
        -----BEGIN CERTIFICATE-----

        -----END CERTIFICATE-----
    name: external1
    type: external1
observedGeneration: 3 5
```

1. The Kafka cluster ID.
2. Status **conditions** describe the current state of the Kafka cluster.
3. The **Ready** condition indicates that the Cluster Operator considers the Kafka cluster able to handle traffic.
4. The **listeners** describe Kafka bootstrap addresses by type.
5. The **observedGeneration** value indicates the last reconciliation of the **Kafka** custom resource by the Cluster Operator.



NOTE

The Kafka bootstrap addresses listed in the status do not signify that those endpoints or the Kafka cluster is in a **Ready** state.

Accessing status information

You can access status information for a resource from the command line. For more information, see Section 20.1.3, “Finding the status of a custom resource” .

20.1.3. Finding the status of a custom resource

This procedure describes how to find the status of a custom resource.

Prerequisites

- An OpenShift cluster.
- The Cluster Operator is running.

Procedure

- Specify the custom resource and use the **-o jsonpath** option to apply a standard JSONPath expression to select the **status** property:

```
oc get kafka <kafka_resource_name> -o jsonpath='{.status}'
```

  This expression returns all the status information for the specified custom resource. You can use dot notation, such as **status.listeners** or **status.observedGeneration**, to fine-tune the status information you wish to see.

Additional resources

- Section 20.1.2, “AMQ Streams custom resource status information”
- For more information about using JSONPath, see JSONPath support.

20.2. PAUSING RECONCILIATION OF CUSTOM RESOURCES

Sometimes it is useful to pause the reconciliation of custom resources managed by AMQ Streams Operators, so that you can perform fixes or make updates. If reconciliations are paused, any changes made to custom resources are ignored by the Operators until the pause ends.

If you want to pause reconciliation of a custom resource, set the **strimzi.io/pause-reconciliation** annotation to **true** in its configuration. This instructs the appropriate Operator to pause reconciliation of the custom resource. For example, you can apply the annotation to the **KafkaConnect** resource so that reconciliation by the Cluster Operator is paused.

You can also create a custom resource with the pause annotation enabled. The custom resource is created, but it is ignored.

Prerequisites

- The AMQ Streams Operator that manages the custom resource is running.

Procedure

1. Annotate the custom resource in OpenShift, setting **pause-reconciliation** to **true**:

```
oc annotate <kind_of_custom_resource> <name_of_custom_resource> strimzi.io/pause-reconciliation="true"
```

For example, for the **KafkaConnect** custom resource:

```
oc annotate KafkaConnect my-connect strimzi.io/pause-reconciliation="true"
```

2. Check that the status conditions of the custom resource show a change to **ReconciliationPaused**:

```
oc describe <kind_of_custom_resource> <name_of_custom_resource>
```

The **type** condition changes to **ReconciliationPaused** at the **lastTransitionTime**.

Example custom resource with a paused reconciliation condition type

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  annotations:
    strimzi.io/pause-reconciliation: "true"
    strimzi.io/use-connector-resources: "true"
  creationTimestamp: 2021-03-12T10:47:11Z
  #...
spec:
  # ...
status:
  conditions:
  - lastTransitionTime: 2021-03-12T10:47:41.689249Z
    status: "True"
    type: ReconciliationPaused
```

Resuming from pause

- To resume reconciliation, you can set the annotation to **false**, or remove the annotation.

Additional resources

- Finding the status of a custom resource

20.3. MAINTENANCE TIME WINDOWS FOR ROLLING UPDATES

Maintenance time windows allow you to schedule certain rolling updates of your Kafka and ZooKeeper clusters to start at a convenient time.

20.3.1. Maintenance time windows overview

In most cases, the Cluster Operator only updates your Kafka or ZooKeeper clusters in response to changes to the corresponding **Kafka** resource. This enables you to plan when to apply changes to a **Kafka** resource to minimize the impact on Kafka client applications.

However, some updates to your Kafka and ZooKeeper clusters can happen without any corresponding change to the **Kafka** resource. For example, the Cluster Operator will need to perform a rolling restart if a CA (certificate authority) certificate that it manages is close to expiry.

While a rolling restart of the pods should not affect *availability* of the service (assuming correct broker and topic configurations), it could affect *performance* of the Kafka client applications. Maintenance time windows allow you to schedule such spontaneous rolling updates of your Kafka and ZooKeeper clusters to start at a convenient time. If maintenance time windows are not configured for a cluster then it is possible that such spontaneous rolling updates will happen at an inconvenient time, such as during a predictable period of high load.

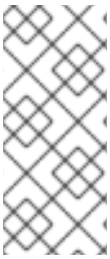
20.3.2. Maintenance time window definition

You configure maintenance time windows by entering an array of strings in the **Kafka.spec.maintenanceTimeWindows** property. Each string is a cron expression interpreted as being in UTC (Coordinated Universal Time, which for practical purposes is the same as Greenwich Mean Time).

The following example configures a single maintenance time window that starts at midnight and ends at 01:59am (UTC), on Sundays, Mondays, Tuesdays, Wednesdays, and Thursdays:

```
# ...
maintenanceTimeWindows:
  - "* * 0-1 ? * SUN,MON,TUE,WED,THU *"
# ...
```

In practice, maintenance windows should be set in conjunction with the **Kafka.spec.clusterCa.renewalDays** and **Kafka.spec.clientsCa.renewalDays** properties of the **Kafka** resource, to ensure that the necessary CA certificate renewal can be completed in the configured maintenance time windows.



NOTE

AMQ Streams does not schedule maintenance operations exactly according to the given windows. Instead, for each reconciliation, it checks whether a maintenance window is currently "open". This means that the start of maintenance operations within a given time window can be delayed by up to the Cluster Operator reconciliation interval. Maintenance time windows must therefore be at least this long.

20.3.3. Configuring a maintenance time window

You can configure a maintenance time window for rolling updates triggered by supported processes.

Prerequisites

- An OpenShift cluster.
- The Cluster Operator is running.

Procedure

1. Add or edit the **maintenanceTimeWindows** property in the **Kafka** resource. For example to allow maintenance between 0800 and 1059 and between 1400 and 1559 you would set the **maintenanceTimeWindows** as shown below:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  maintenanceTimeWindows:
    - "* * 8-10 * * ?"
    - "* * 14-15 * * ?"
```

2. Create or update the resource:

```
oc apply -f <kafka_configuration_file>
```

Additional resources

- Section 20.4.1, “Performing a rolling update using a pod management annotation”
- Section 20.4.2, “Performing a rolling update using a pod annotation”

20.4. MANUALLY STARTING ROLLING UPDATES OF KAFKA AND ZOOKEEPER CLUSTERS

AMQ Streams supports the use of annotations on resources to manually trigger a rolling update of Kafka and ZooKeeper clusters through the Cluster Operator. Rolling updates restart the pods of the resource with new ones.

Manually performing a rolling update on a specific pod or set of pods is usually only required in exceptional circumstances. However, rather than deleting the pods directly, if you perform the rolling update through the Cluster Operator you ensure the following:

- The manual deletion of the pod does not conflict with simultaneous Cluster Operator operations, such as deleting other pods in parallel.
- The Cluster Operator logic handles the Kafka configuration specifications, such as the number of in-sync replicas.

20.4.1. Performing a rolling update using a pod management annotation

This procedure describes how to trigger a rolling update of a Kafka cluster or ZooKeeper cluster.

To trigger the update, you add an annotation to the resource you are using to manage the pods running on the cluster. You annotate the **StatefulSet** or **StrimziPodSet** resource (if you enabled the **UseStrimziPodSets** feature gate).

Prerequisites

To perform a manual rolling update, you need a running Cluster Operator and Kafka cluster.

Procedure

1. Find the name of the resource that controls the Kafka or ZooKeeper pods you want to manually update.
   For example, if your Kafka cluster is named *my-cluster*, the corresponding names are *my-cluster-kafka* and *my-cluster-zookeeper*.

2. Use **oc annotate** to annotate the appropriate resource in OpenShift.

   Annotating a StatefulSet

   ```
   oc annotate statefulset <cluster_name>-kafka strimzi.io/manual-rolling-update=true

   oc annotate statefulset <cluster_name>-zookeeper strimzi.io/manual-rolling-update=true
   ```

   Annotating a StrimziPodSet

   ```
   oc annotate strimzipodset <cluster_name>-kafka strimzi.io/manual-rolling-update=true

   oc annotate strimzipodset <cluster_name>-zookeeper strimzi.io/manual-rolling-update=true
   ```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of all pods within the annotated resource is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of all the pods is complete, the annotation is removed from the resource.

20.4.2. Performing a rolling update using a pod annotation

This procedure describes how to manually trigger a rolling update of an existing Kafka cluster or ZooKeeper cluster using an OpenShift **Pod** annotation. When multiple pods are annotated, consecutive rolling updates are performed within the same reconciliation run.

Prerequisites

To perform a manual rolling update, you need a running Cluster Operator and Kafka cluster.

You can perform a rolling update on a Kafka cluster regardless of the topic replication factor used. But for Kafka to stay operational during the update, you'll need the following:

- A highly available Kafka cluster deployment running with nodes that you wish to update.
- Topics replicated for high availability.
  Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

  Kafka topic replicated for high availability

  ```
  apiVersion: kafka.strimzi.io/v1beta2
  kind: KafkaTopic
  metadata:
    name: my-topic
    labels:
      strimzi.io/cluster: my-cluster
  ```

```
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

Procedure

1. Find the name of the Kafka or ZooKeeper **Pod** you want to manually update.
   For example, if your Kafka cluster is named *my-cluster*, the corresponding **Pod** names are *my-cluster-kafka-index* and *my-cluster-zookeeper-index*. The *index* starts at zero and ends at the total number of replicas minus one.

2. Annotate the **Pod** resource in OpenShift.
   Use **oc annotate**:

   ```
   oc annotate pod cluster-name-kafka-index strimzi.io/manual-rolling-update=true

   oc annotate pod cluster-name-zookeeper-index strimzi.io/manual-rolling-update=true
   ```

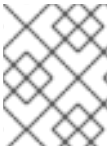
3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of the annotated **Pod** is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of a pod is complete, the annotation is removed from the **Pod**.

20.5. EVICTING PODS WITH THE AMQ STREAMS DRAIN CLEANER

Kafka and ZooKeeper pods might be evicted during OpenShift upgrades, maintenance, or pod rescheduling. If your Kafka broker and ZooKeeper pods were deployed by AMQ Streams, you can use the AMQ Streams Drain Cleaner tool to handle the pod evictions. The AMQ Streams Drain Cleaner handles the eviction instead of OpenShift. You must set the **podDisruptionBudget** for your Kafka deployment to **0** (zero). OpenShift will then no longer be allowed to evict the pod automatically.

By deploying the AMQ Streams Drain Cleaner, you can use the Cluster Operator to move Kafka pods instead of OpenShift. The Cluster Operator ensures that topics are never under-replicated. Kafka can remain operational during the eviction process. The Cluster Operator waits for topics to synchronize, as the OpenShift worker nodes drain consecutively.

An admission webhook notifies the AMQ Streams Drain Cleaner of pod eviction requests to the Kubernetes API. The AMQ Streams Drain Cleaner then adds a rolling update annotation to the pods to be drained. This informs the Cluster Operator to perform a rolling update of an evicted pod.



NOTE

If you are not using the AMQ Streams Drain Cleaner, you can add pod annotations to perform rolling updates manually.

Webhook configuration

The AMQ Streams Drain Cleaner deployment files include a **ValidatingWebhookConfiguration** resource file. The resource provides the configuration for registering the webhook with the Kubernetes API.

The configuration defines the **rules** for the Kubernetes API to follow in the event of a pod eviction request. The rules specify that only **CREATE** operations related to **pods/eviction** sub-resources are intercepted. If these rules are met, the API forwards the notification.

The **clientConfig** points to the AMQ Streams Drain Cleaner service and **/drainer** endpoint that exposes the webhook. The webhook uses a secure TLS connection, which requires authentication. The **caBundle** property specifies the certificate chain to validate HTTPS communication. Certificates are encoded in Base64.

Webhook configuration for pod eviction notifications

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
# ...
webhooks:
  - name: strimzi-drain-cleaner.strimzi.io
    rules:
      - apiGroups:   [""]
        apiVersions: ["v1"]
        operations:  ["CREATE"]
        resources:   ["pods/eviction"]
        scope:       "Namespaced"
    clientConfig:
      service:
        namespace: "strimzi-drain-cleaner"
        name: "strimzi-drain-cleaner"
        path: /drainer
        port: 443
        caBundle: Cg==
    # ...
```

20.5.1. Downloading the AMQ Streams Drain Cleaner deployment files

To deploy and use the AMQ Streams Drain Cleaner, you need to download the deployment files.

The AMQ Streams Drain Cleaner deployment files are available from the AMQ Streams software downloads page.

20.5.2. Deploying the AMQ Streams Drain Cleaner using installation files

Deploy the AMQ Streams Drain Cleaner to the OpenShift cluster where the Cluster Operator and Kafka cluster are running.

Prerequisites

- You have downloaded the AMQ Streams Drain Cleaner deployment files .
- You have a highly available Kafka cluster deployment running with OpenShift worker nodes that you would like to update.
- Topics are replicated for high availability.
  Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

  Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

Excluding Kafka or ZooKeeper

If you don't want to include Kafka or ZooKeeper pods in Drain Cleaner operations, change the default environment variables in the Drain Cleaner **Deployment** configuration file.

- Set **STRIMZI_DRAIN_KAFKA** to **false** to exclude Kafka pods
- Set **STRIMZI_DRAIN_ZOOKEEPER** to **false** to exclude ZooKeeper pods

Example configuration to exclude ZooKeeper pods

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-drain-cleaner
      containers:
        - name: strimzi-drain-cleaner
          # ...
          env:
            - name: STRIMZI_DRAIN_KAFKA
              value: "true"
            - name: STRIMZI_DRAIN_ZOOKEEPER
              value: "false"
          # ...
```

Procedure

1. Configure a pod disruption budget of **0** (zero) for your Kafka deployment using **template** settings in the **Kafka** resource.

   #### Specifying a pod disruption budget

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
     name: my-cluster
     namespace: myproject
   spec:
   ```

```
kafka:
  template:
    podDisruptionBudget:
      maxUnavailable: 0

# ...
zookeeper:
  template:
    podDisruptionBudget:
      maxUnavailable: 0
# ...
```

Reducing the maximum pod disruption budget to zero prevents OpenShift from automatically evicting the pods in case of voluntary disruptions, leaving the AMQ Streams Drain Cleaner and AMQ Streams Cluster Operator to roll the pod which will be scheduled by OpenShift on a different worker node.

Add the same configuration for ZooKeeper if you want to use AMQ Streams Drain Cleaner to drain ZooKeeper nodes.

2. Update the **Kafka** resource:

```
oc apply -f <kafka_configuration_file>
```

3. Deploy the AMQ Streams Drain Cleaner.
   - To run the Drain Cleaner on OpenShift, apply the resources in the **/install/drain-cleaner/openshift** directory.

```
oc apply -f ./install/drain-cleaner/openshift
```

20.5.3. Using the AMQ Streams Drain Cleaner

Use the AMQ Streams Drain Cleaner in combination with the Cluster Operator to move Kafka broker or ZooKeeper pods from nodes that are being drained. When you run the AMQ Streams Drain Cleaner, it annotates pods with a rolling update pod annotation. The Cluster Operator performs rolling updates based on the annotation.

Prerequisites

- You have deployed the AMQ Streams Drain Cleaner .

Procedure

1. Drain a specified OpenShift node hosting the Kafka broker or ZooKeeper pods.

```
oc get nodes
oc drain <name-of-node> --delete-emptydir-data --ignore-daemonsets --timeout=6000s --force
```

2. Check the eviction events in the AMQ Streams Drain Cleaner log to verify that the pods have been annotated for restart.

AMQ Streams Drain Cleaner log show annotations of pods

```
INFO ... Received eviction webhook for Pod my-cluster-zookeeper-2 in namespace my-project
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project will be annotated for restart
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project found and annotated for restart

INFO ... Received eviction webhook for Pod my-cluster-kafka-0 in namespace my-project
INFO ... Pod my-cluster-kafka-0 in namespace my-project will be annotated for restart
INFO ... Pod my-cluster-kafka-0 in namespace my-project found and annotated for restart
```

3. Check the reconciliation events in the Cluster Operator log to verify the rolling updates.

Cluster Operator log shows rolling updates

```
INFO  PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster): Rolling Pod my-cluster-zookeeper-2
INFO  PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster): Rolling Pod my-cluster-kafka-0
INFO  AbstractOperator:500 - Reconciliation #13(timer) Kafka(my-project/my-cluster): reconciled
```

20.5.4. Watching the TLS certificates used by the AMQ Streams Drain Cleaner

By default, the Drain Cleaner deployment watches the secret containing the TLS certificates its uses for authentication. The Drain Cleaner watches for changes, such as certificate renewals. If it detects a change, it restarts to reload the TLS certificates. The Drain Cleaner installation files enable this behavior by default. But you can disable the watching of certificates by setting the **STRIMZI_CERTIFICATE_WATCH_ENABLED** environment variable to **false** in the **Deployment** configuration (**060-Deployment.yaml**) of the Drain Cleaner installation files.

With **STRIMZI_CERTIFICATE_WATCH_ENABLED** enabled, you can also use the following environment variables for watching TLS certificates.

Table 20.3. Drain Cleaner environment variables for watching TLS certificates

Environment Variable	Description	Default
STRIMZI_CERTIFICATE_WATCH_ENABLED	Enables or disables the certificate watch	**false**
STRIMZI_CERTIFICATE_WATCH_NAMESPACE	The namespace where the Drain Cleaner is deployed and where the certificate secret exists	**strimzi-drain-cleaner**
STRIMZI_CERTIFICATE_WATCH_POD_NAME	The Drain Cleaner pod name	-
STRIMZI_CERTIFICATE_WATCH_SECRET_NAME	The name of the secret containing TLS certificates	**strimzi-drain-cleaner**
STRIMZI_CERTIFICATE_WATCH_SECRET_KEYS	The list of fields inside the secret that contain the TLS certificates	**tls.crt, tls.key**

Example environment variable configuration to control watch operations

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-drain-cleaner
  labels:
    app: strimzi-drain-cleaner
  namespace: strimzi-drain-cleaner
spec:
  # ...
    spec:
      serviceAccountName: strimzi-drain-cleaner
      containers:
        - name: strimzi-drain-cleaner
          # ...
          env:
            - name: STRIMZI_DRAIN_KAFKA
              value: "true"
            - name: STRIMZI_DRAIN_ZOOKEEPER
              value: "true"
            - name: STRIMZI_CERTIFICATE_WATCH_ENABLED
              value: "true"
            - name: STRIMZI_CERTIFICATE_WATCH_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_CERTIFICATE_WATCH_POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            # ...
```

TIP

Use the Downward API mechanism to configure **STRIMZI_CERTIFICATE_WATCH_NAMESPACE** and **STRIMZI_CERTIFICATE_WATCH_POD_NAME**.

20.6. DISCOVERING SERVICES USING LABELS AND ANNOTATIONS

Service discovery makes it easier for client applications running in the same OpenShift cluster as AMQ Streams to interact with a Kafka cluster.

A *service discovery* label and annotation is generated for services used to access the Kafka cluster:

- Internal Kafka bootstrap service
- HTTP Bridge service

The label helps to make the service discoverable, and the annotation provides connection details that a client application can use to make the connection.

The service discovery label, **strimzi.io/discovery**, is set as **true** for the **Service** resources. The service discovery annotation has the same key, providing connection details in JSON format for each service.

Example internal Kafka bootstrap service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-
      [ {
        "port" : 9092,
        "tls" : false,
        "protocol" : "kafka",
        "auth" : "scram-sha-512"
      }, {
        "port" : 9093,
        "tls" : true,
        "protocol" : "kafka",
        "auth" : "tls"
      } ]
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/discovery: "true"
    strimzi.io/kind: Kafka
    strimzi.io/name: my-cluster-kafka-bootstrap
  name: my-cluster-kafka-bootstrap
spec:
  #...
```

Example HTTP Bridge service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-
      [ {
        "port" : 8080,
        "tls" : false,
        "auth" : "none",
        "protocol" : "http"
      } ]
  labels:
    strimzi.io/cluster: my-bridge
    strimzi.io/discovery: "true"
    strimzi.io/kind: KafkaBridge
    strimzi.io/name: my-bridge-bridge-service
```

20.6.1. Returning connection details on services

You can find the services by specifying the discovery label when fetching services from the command line or a corresponding API call.

```
oc get service -l strimzi.io/discovery=true
```

The connection details are returned when retrieving the service discovery label.

20.7. RECOVERING A CLUSTER FROM PERSISTENT VOLUMES

You can recover a Kafka cluster from persistent volumes (PVs) if they are still present.

You might want to do this, for example, after:

- A namespace was deleted unintentionally
- A whole OpenShift cluster is lost, but the PVs remain in the infrastructure

20.7.1. Recovery from namespace deletion

Recovery from namespace deletion is possible because of the relationship between persistent volumes and namespaces. A **PersistentVolume** (PV) is a storage resource that lives outside of a namespace. A PV is mounted into a Kafka pod using a **PersistentVolumeClaim** (PVC), which lives inside a namespace.

The reclaim policy for a PV tells a cluster how to act when a namespace is deleted. If the reclaim policy is set as:

- *Delete* (default), PVs are deleted when PVCs are deleted within a namespace
- *Retain*, PVs are not deleted when a namespace is deleted

To ensure that you can recover from a PV if a namespace is deleted unintentionally, the policy must be reset from *Delete* to *Retain* in the PV specification using the **persistentVolumeReclaimPolicy** property:

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
  # ...
  persistentVolumeReclaimPolicy: Retain
```

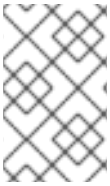
Alternatively, PVs can inherit the reclaim policy of an associated storage class. Storage classes are used for dynamic volume allocation.

By configuring the **reclaimPolicy** property for the storage class, PVs that use the storage class are created with the appropriate reclaim policy. The storage class is configured for the PV using the **storageClassName** property.

```
apiVersion: v1
kind: StorageClass
metadata:
  name: gp2-retain
parameters:
  # ...
# ...
reclaimPolicy: Retain
```

```
apiVersion: v1
kind: PersistentVolume
# ...
```

```
spec:
  # ...
  storageClassName: gp2-retain
```



NOTE

If you are using *Retain* as the reclaim policy, but you want to delete an entire cluster, you need to delete the PVs manually. Otherwise they will not be deleted, and may cause unnecessary expenditure on resources.

20.7.2. Recovery from loss of an OpenShift cluster

When a cluster is lost, you can use the data from disks/volumes to recover the cluster if they were preserved within the infrastructure. The recovery procedure is the same as with namespace deletion, assuming PVs can be recovered and they were created manually.

20.7.3. Recovering a deleted cluster from persistent volumes

This procedure describes how to recover a deleted cluster from persistent volumes (PVs).

In this situation, the Topic Operator identifies that topics exist in Kafka, but the **KafkaTopic** resources do not exist.

When you get to the step to recreate your cluster, you have two options:

1. Use *Option 1* when you can recover all **KafkaTopic** resources.
   The **KafkaTopic** resources must therefore be recovered before the cluster is started so that the corresponding topics are not deleted by the Topic Operator.

2. Use *Option 2* when you are unable to recover all **KafkaTopic** resources.
   In this case, you deploy your cluster without the Topic Operator, delete the Topic Operator topic store metadata, and then redeploy the Kafka cluster with the Topic Operator so it can recreate the **KafkaTopic** resources from the corresponding topics.

NOTE

If the Topic Operator is not deployed, you only need to recover the **PersistentVolumeClaim** (PVC) resources.

Before you begin

In this procedure, it is essential that PVs are mounted into the correct PVC to avoid data corruption. A **volumeName** is specified for the PVC and this must match the name of the PV.

For more information, see Persistent storage.

NOTE

The procedure does not include recovery of **KafkaUser** resources, which must be recreated manually. If passwords and certificates need to be retained, secrets must be recreated before creating the **KafkaUser** resources.

Procedure

1. Check information on the PVs in the cluster:

```
oc get pv
```

Information is presented for PVs with data.

Example output showing columns important to this procedure:

```
NAME                                        RECLAIMPOLICY CLAIM
pvc-5e9c5c7f-3317-11ea-a650-06e1eadd9a4c ... Retain ...    myproject/data-my-cluster-zookeeper-1
pvc-5e9cc72d-3317-11ea-97b0-0aef8816c7ea ... Retain ...    myproject/data-my-cluster-zookeeper-0
pvc-5ead43d1-3317-11ea-97b0-0aef8816c7ea ... Retain ...    myproject/data-my-cluster-zookeeper-2
pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c ... Retain ...    myproject/data-0-my-cluster-kafka-0
pvc-7e21042e-3317-11ea-9786-02deaf9aa87e ... Retain ...    myproject/data-0-my-cluster-kafka-1
pvc-7e226978-3317-11ea-97b0-0aef8816c7ea ... Retain ...    myproject/data-0-my-cluster-kafka-2
```

- *NAME* shows the name of each PV.
- *RECLAIM POLICY* shows that PVs are *retained*.
- *CLAIM* shows the link to the original PVCs.

2. Recreate the original namespace:

```
oc create namespace myproject
```

3. Recreate the original PVC resource specifications, linking the PVCs to the appropriate PV: For example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-0-my-cluster-kafka-0
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
  storageClassName: gp2-retain
  volumeMode: Filesystem
  volumeName: pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c
```

4. Edit the PV specifications to delete the **claimRef** properties that bound the original PVC. For example:

```
apiVersion: v1
kind: PersistentVolume
```

```
metadata:
  annotations:
    kubernetes.io/createdby: aws-ebs-dynamic-provisioner
    pv.kubernetes.io/bound-by-controller: "yes"
    pv.kubernetes.io/provisioned-by: kubernetes.io/aws-ebs
  creationTimestamp: "<date>"
  finalizers:
  - kubernetes.io/pv-protection
  labels:
    failure-domain.beta.kubernetes.io/region: eu-west-1
    failure-domain.beta.kubernetes.io/zone: eu-west-1c
  name: pvc-7e226978-3317-11ea-97b0-0aef8816c7ea
  resourceVersion: "39431"
  selfLink: /api/v1/persistentvolumes/pvc-7e226978-3317-11ea-97b0-0aef8816c7ea
  uid: 7efe6b0d-3317-11ea-a650-06e1eadd9a4c
spec:
  accessModes:
  - ReadWriteOnce
  awsElasticBlockStore:
    fsType: xfs
    volumeID: aws://eu-west-1c/vol-09db3141656d1c258
  capacity:
    storage: 100Gi
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: data-0-my-cluster-kafka-2
    namespace: myproject
    resourceVersion: "39113"
    uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: failure-domain.beta.kubernetes.io/zone
          operator: In
          values:
          - eu-west-1c
        - key: failure-domain.beta.kubernetes.io/region
          operator: In
          values:
          - eu-west-1
  persistentVolumeReclaimPolicy: Retain
  storageClassName: gp2-retain
  volumeMode: Filesystem
```

In the example, the following properties are deleted:

```
claimRef:
  apiVersion: v1
  kind: PersistentVolumeClaim
  name: data-0-my-cluster-kafka-2
  namespace: myproject
  resourceVersion: "39113"
  uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
```

5. Deploy the Cluster Operator.

```
oc create -f install/cluster-operator -n my-project
```

6. Recreate your cluster.
   Follow the steps depending on whether or not you have all the **KafkaTopic** resources needed to recreate your cluster.

   Option 1: If you have **all** the **KafkaTopic** resources that existed before you lost your cluster, including internal topics such as committed offsets from **__consumer_offsets**:

   1. Recreate all **KafkaTopic** resources.
      It is essential that you recreate the resources before deploying the cluster, or the Topic Operator will delete the topics.

   2. Deploy the Kafka cluster.
      For example:

```
oc apply -f kafka.yaml
```

   Option 2: If you do not have all the **KafkaTopic** resources that existed before you lost your cluster:

   1. Deploy the Kafka cluster, as with the first option, but without the Topic Operator by removing the **topicOperator** property from the Kafka resource before deploying.
      If you include the Topic Operator in the deployment, the Topic Operator will delete all the topics.

   2. Delete the internal topic store topics from the Kafka cluster:

```
oc run kafka-admin -ti --image=registry.redhat.io/amq-streams/kafka-34-rhel8:2.4.0 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

      The command must correspond to the type of listener and authentication used to access the Kafka cluster.

   3. Enable the Topic Operator by redeploying the Kafka cluster with the **topicOperator** property to recreate the **KafkaTopic** resources.
      For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {} 1
    #...
```

1 Here we show the default configuration, which has no additional properties. You specify the required configuration using the properties described in the **EntityTopicOperatorSpec** schema reference.

7. Verify the recovery by listing the **KafkaTopic** resources:

```
oc get KafkaTopic
```

20.8. SETTING LIMITS ON BROKERS USING THE KAFKA STATIC QUOTA PLUGIN

Use the *Kafka Static Quota* plugin to set throughput and storage limits on brokers in your Kafka cluster. You enable the plugin and set limits by configuring the **Kafka** resource. You can set a byte-rate threshold and storage quotas to put limits on the clients interacting with your brokers.

You can set byte-rate thresholds for producer and consumer bandwidth. The total limit is distributed across all clients accessing the broker. For example, you can set a byte-rate threshold of 40 MBps for producers. If two producers are running, they are each limited to a throughput of 20 MBps.

Storage quotas throttle Kafka disk storage limits between a soft limit and hard limit. The limits apply to all available disk space. Producers are slowed gradually between the soft and hard limit. The limits prevent disks filling up too quickly and exceeding their capacity. Full disks can lead to issues that are hard to rectify. The hard limit is the maximum storage limit.

NOTE

For JBOD storage, the limit applies across all disks. If a broker is using two 1 TB disks and the quota is 1.1 TB, one disk might fill and the other disk will be almost empty.

Prerequisites

- The Cluster Operator that manages the Kafka cluster is running.

Procedure

1. Add the plugin properties to the **config** of the **Kafka** resource.
   The plugin properties are shown in this example configuration.

   Example Kafka Static Quota plugin configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    config:
      client.quota.callback.class: io.strimzi.kafka.quotas.StaticQuotaCallback 1
      client.quota.callback.static.produce: 1000000 2
      client.quota.callback.static.fetch: 1000000 3
```

```
client.quota.callback.static.storage.soft: 400000000000 4
client.quota.callback.static.storage.hard: 500000000000 5
client.quota.callback.static.storage.check-interval: 5 6
```

1. Loads the Kafka Static Quota plugin.
2. Sets the producer byte-rate threshold. 1 MBps in this example.
3. Sets the consumer byte-rate threshold. 1 MBps in this example.
4. Sets the lower soft limit for storage. 400 GB in this example.
5. Sets the higher hard limit for storage. 500 GB in this example.
6. Sets the interval in seconds between checks on storage. 5 seconds in this example. You can set this to 0 to disable the check.

2. Update the resource.

```
oc apply -f <kafka_configuration_file>
```

Additional resources

- **KafkaUserQuotas** schema reference

20.9. UNINSTALLING AMQ STREAMS

You can uninstall AMQ Streams on OpenShift 4.10 to 4.13 from the OperatorHub using the OpenShift Container Platform web console or CLI.

Use the same approach you used to install AMQ Streams.

When you uninstall AMQ Streams, you will need to identify resources created specifically for a deployment and referenced from the AMQ Streams resource.

Such resources include:

- Secrets (Custom CAs and certificates, Kafka Connect secrets, and other Kafka secrets)
- Logging **ConfigMaps** (of type **external**)

These are resources referenced by **Kafka**, **KafkaConnect**, **KafkaMirrorMaker**, or **KafkaBridge** configuration.

WARNING

Deleting **CustomResourceDefinitions** results in the garbage collection of the corresponding custom resources (**Kafka**, **KafkaConnect**, **KafkaMirrorMaker**, or **KafkaBridge**) and the resources dependent on them (Deployments, StatefulSets, and other dependent resources).

20.9.1. Uninstalling AMQ Streams from the OperatorHub using the web console

This procedure describes how to uninstall AMQ Streams from the OperatorHub and remove resources related to the deployment.

You can perform the steps from the console or use alternative CLI commands.

Prerequisites

- Access to an OpenShift Container Platform web console using an account with **cluster-admin** or **strimzi-admin** permissions.
- You have identified the resources to be deleted.
  You can use the following **oc** CLI command to find resources and also verify that they have been removed when you have uninstalled AMQ Streams.

  Command to find resources related to an AMQ Streams deployment

  ```
  oc get <resource_type> --all-namespaces | grep <kafka_cluster_name>
  ```

  Replace *<resource_type>* with the type of the resource you are checking, such as **secret** or **configmap**.

Procedure

1. Navigate in the OpenShift web console to **Operators > Installed Operators**.
2. For the installed **AMQ Streams** operator, select the options icon (three vertical dots) and click **Uninstall Operator**.
   The operator is removed from **Installed Operators**.
3. Navigate to **Home** > **Projects** and select the project where you installed AMQ Streams and the Kafka components.
4. Click the options under **Inventory** to delete related resources.
   Resources include the following:
   - Deployments
   - StatefulSets
   - Pods
   - Services

- ConfigMaps
- Secrets

TIP

Use the search to find related resources that begin with the name of the Kafka cluster. You can also find the resources under **Workloads**.

Alternative CLI commands

You can use CLI commands to uninstall AMQ Streams from the OperatorHub.

1. Delete the AMQ Streams subscription.

```
oc delete subscription amq-streams -n openshift-operators
```

2. Delete the cluster service version (CSV).

```
oc delete csv amqstreams.<version>  -n openshift-operators
```

3. Remove related CRDs.

```
oc get crd -l app=strimzi -o name | xargs oc delete
```

20.9.2. Uninstalling AMQ Streams using the CLI

This procedure describes how to use the **oc** command-line tool to uninstall AMQ Streams and remove resources related to the deployment.

Prerequisites

- Access to an OpenShift cluster using an account with **cluster-admin** or **strimzi-admin** permissions.
- You have identified the resources to be deleted.
  You can use the following **oc** CLI command to find resources and also verify that they have been removed when you have uninstalled AMQ Streams.

  Command to find resources related to an AMQ Streams deployment

  ```
  oc get <resource_type> --all-namespaces | grep <kafka_cluster_name>
  ```

  Replace *<resource_type>* with the type of the resource you are checking, such as **secret** or **configmap**.

Procedure

1. Delete the Cluster Operator **Deployment**, related **CustomResourceDefinitions**, and **RBAC** resources.
   Specify the installation files used to deploy the Cluster Operator.

   ```
   oc delete -f install/cluster-operator
   ```

2. Delete the resources you identified in the prerequisites.

```
oc delete <resource_type> <resource_name> -n <namespace>
```

Replace *<resource_type>* with the type of resource you are deleting and *<resource_name>* with the name of the resource.

Example to delete a secret

```
oc delete secret my-cluster-clients-ca -n my-project
```

20.10. FREQUENTLY ASKED QUESTIONS

20.10.1. Questions related to the Cluster Operator

20.10.1.1. Why do I need cluster administrator privileges to install AMQ Streams?

To install AMQ Streams, you need to be able to create the following cluster-scoped resources:

- Custom Resource Definitions (CRDs) to instruct OpenShift about resources that are specific to AMQ Streams, such as **Kafka** and **KafkaConnect**
- **ClusterRoles** and **ClusterRoleBindings**

Cluster-scoped resources, which are not scoped to a particular OpenShift namespace, typically require *cluster administrator* privileges to install.

As a cluster administrator, you can inspect all the resources being installed (in the /**install**/ directory) to ensure that the **ClusterRoles** do not grant unnecessary privileges.

After installation, the Cluster Operator runs as a regular **Deployment**, so any standard (non-admin) OpenShift user with privileges to access the **Deployment** can configure it. The cluster administrator can grant standard users the privileges necessary to manage **Kafka** custom resources.

See also:

- Why does the Cluster Operator need to create **ClusterRoleBindings**?
- Can standard OpenShift users create Kafka custom resources?

20.10.1.2. Why does the Cluster Operator need to create ClusterRoleBindings?

OpenShift has built-in privilege escalation prevention, which means that the Cluster Operator cannot grant privileges it does not have itself, specifically, it cannot grant such privileges in a namespace it cannot access. Therefore, the Cluster Operator must have the privileges necessary for *all* the components it orchestrates.

The Cluster Operator needs to be able to grant access so that:

- The Topic Operator can manage **KafkaTopics**, by creating **Roles** and **RoleBindings** in the namespace that the operator runs in

- The User Operator can manage **KafkaUsers**, by creating **Roles** and **RoleBindings** in the namespace that the operator runs in
- The failure domain of a **Node** is discovered by AMQ Streams, by creating a **ClusterRoleBinding**

When using rack-aware partition assignment, the broker pod needs to be able to get information about the **Node** it is running on, for example, the Availability Zone in Amazon AWS. A **Node** is a cluster-scoped resource, so access to it can only be granted through a **ClusterRoleBinding**, not a namespace-scoped **RoleBinding**.

20.10.1.3. Can standard OpenShift users create Kafka custom resources?

By default, standard OpenShift users will not have the privileges necessary to manage the custom resources handled by the Cluster Operator. The cluster administrator can grant a user the necessary privileges using OpenShift RBAC resources.

For more information, see Section 4.6, “Designating AMQ Streams administrators” .

20.10.1.4. What do the *failed to acquire lock* warnings in the log mean?

For each cluster, the Cluster Operator executes only one operation at a time. The Cluster Operator uses locks to make sure that there are never two parallel operations running for the same cluster. Other operations must wait until the current operation completes before the lock is released.

INFO

Examples of cluster operations include *cluster creation* , *rolling update*, *scale down* , and *scale up*.

If the waiting time for the lock takes too long, the operation times out and the following warning message is printed to the log:

```
2018-03-04 17:09:24 WARNING AbstractClusterOperations:290 - Failed to acquire lock for kafka cluster lock::kafka::myproject::my-cluster
```

Depending on the exact configuration of **STRIMZI_FULL_RECONCILIATION_INTERVAL_MS** and **STRIMZI_OPERATION_TIMEOUT_MS**, this warning message might appear occasionally without indicating any underlying issues. Operations that time out are picked up in the next periodic reconciliation, so that the operation can acquire the lock and execute again.

Should this message appear periodically, even in situations when there should be no other operations running for a given cluster, it might indicate that the lock was not properly released due to an error. If this is the case, try restarting the Cluster Operator.

20.10.1.5. Why is hostname verification failing when connecting to NodePorts using TLS?

Currently, off-cluster access using NodePorts with TLS encryption enabled does not support TLS hostname verification. As a result, the clients that verify the hostname will fail to connect. For example, the Java client will fail with the following exception:

```
Caused by: java.security.cert.CertificateException: No subject alternative names matching IP address 168.72.15.231 found
 at sun.security.util.HostnameChecker.matchIP(HostnameChecker.java:168)
 at sun.security.util.HostnameChecker.match(HostnameChecker.java:94)
 at sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:455)
 at sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:436)
 at sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:252)
```

```
at sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:136)
at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1501)
... 17 more
```

To connect, you must disable hostname verification. In the Java client, you can do this by setting the configuration option **ssl.endpoint.identification.algorithm** to an empty string.

When configuring the client using a properties file, you can do it this way:

```
ssl.endpoint.identification.algorithm=
```

When configuring the client directly in Java, set the configuration option to an empty string:

```
props.put("ssl.endpoint.identification.algorithm", "");
```

CHAPTER 21. USING METERING ON AMQ STREAMS

You can use the Metering tool that is available on OpenShift to generate metering reports from different data sources. As a cluster administrator, you can use metering to analyze what is happening in your cluster. You can either write your own, or use predefined SQL queries to define how you want to process data from the different data sources you have available. Using Prometheus as a default data source, you can generate reports on pods, namespaces, and most other OpenShift resources.

You can also use the OpenShift Metering operator to analyze your installed AMQ Streams components to determine whether you are in compliance with your Red Hat subscription.

To use metering with AMQ Streams, you must first install and configure the Metering operator on OpenShift Container Platform.

21.1. METERING RESOURCES

Metering has many resources which can be used to manage the deployment and installation of metering, as well as the reporting functionality metering provides. Metering is managed using the following CRDs:

Table 21.1. Metering resources

Name	Description
MeteringConfig	Configures the metering stack for deployment. Contains customizations and configuration options to control each component that makes up the metering stack.
Reports	Controls what query to use, when, and how often the query should be run, and where to store the results.
ReportQueries	Contains the SQL queries used to perform analysis on the data contained within **ReportDataSources**.
ReportDataSources	Controls the data available to ReportQueries and Reports. Allows configuring access to different databases for use within metering.

21.2. METERING LABELS FOR AMQ STREAMS

The following table lists the metering labels for AMQ Streams infrastructure components and integrations.

Table 21.2. Metering Labels

Label	Possible values
com.company	**Red_Hat**
rht.prod_name	**Red_Hat_Application_Foundations**
rht.prod_ver	**2023.Q2**

Label	Possible values
rht.comp	**AMQ_Streams**
rht.comp_ver	**2.4**
rht.subcomp	Infrastructure **cluster-operator** **entity-operator** **topic-operator** **user-operator** **zookeeper**
	Application **kafka-broker** **kafka-connect** **kafka-connect-build** **kafka-mirror-maker2** **kafka-mirror-maker** **cruise-control** **kafka-bridge** **kafka-exporter** **drain-cleaner**
rht.subcomp_t	**infrastructure** **application**

Examples

- Infrastructure example (where the infrastructure component is **entity-operator**)

```
com.company=Red_Hat
rht.prod_name=Red_Hat_Application_Foundations
rht.prod_ver=2023.Q2
rht.comp=AMQ_Streams
rht.comp_ver=2.4
rht.subcomp=entity-operator
rht.subcomp_t=infrastructure
```

- Application example (where the integration deployment name is **kafka-bridge**)

```
com.company=Red_Hat
rht.prod_name=Red_Hat_Application_Foundations
rht.prod_ver=2023.Q2
rht.comp=AMQ_Streams
rht.comp_ver=2.4
rht.subcomp=kafka-bridge
rht.subcomp_t=application
```

APPENDIX A. USING YOUR SUBSCRIPTION

AMQ Streams is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing Your Account

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

Activating a Subscription

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading Zip and Tar Files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **AMQ Streams for Apache Kafka** entries in the **INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ Streams product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Installing packages with DNF

To install a package and all the package dependencies, use:

```
dnf install <package_name>
```

To install a previously-downloaded package from a local directory, use:

```
dnf install <path_to_download_package>
```

Revised on 2023-07-28 16:31:49 UTC