Red Hat AMQ 7.7

Using AMQ Streams on OpenShift

For use with AMQ Streams 1.5 on OpenShift Container Platform

Last Updated: 2020-07-09

Red Hat AMQ 7.7 Using AMQ Streams on OpenShift

For use with AMQ Streams 1.5 on OpenShift Container Platform

Legal Notice

Abstract

This guide describes how to install, configure, and manage Red Hat AMQ Streams to build a large-scale messaging network.

Table of Contents

CHAPTER 1. OVERVIEW OF AMQ STREAMS

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

This guide describes configuration of Kafka components, as well as instructions for using AMQ Streams Operators. Procedures relate to how you might want to modify your deployment and introduce additional features, such as Cruise Control or distributed tracing.

1.1. KAFKA CAPABILITIES

The underlying data stream-processing capabilities and component architecture of Kafka can deliver:

- Microservices and other applications to share data with extremely high throughput and low latency
- Message ordering guarantees
- Message rewind/replay from data storage to reconstruct an application state
- Message compaction to remove old records when using a key-value log
- Horizontal scalability in a cluster configuration
- Replication of data to control fault tolerance
- Retention of high volumes of data for immediate access

1.2. KAFKA USE CASES

Kafka's capabilities make it suitable for:

- Event-driven architectures
- Event sourcing to capture changes to the state of an application as a log of events
- Message brokering
- Website activity tracking
- Operational monitoring through metrics
- Log collection and aggregation
- Commit logs for distributed systems
- Stream processing so that applications can respond to data in real time

1.3. HOW AMQ STREAMS SUPPORTS KAFKA

AMQ Streams provides container images and Operators for running Kafka on OpenShift. AMQ Streams Operators are fundamental to the running of AMQ Streams. The Operators provided with AMQ Streams are purpose-built with specialist operational knowledge to effectively manage Kafka.

Operators simplify the process of:

- Deploying and running Kafka clusters
- Deploying and running Kafka components
- Configuring access to Kafka
- Securing access to Kafka
- Upgrading Kafka
- Managing brokers
- Creating and managing topics
- Creating and managing users

1.4. AMQ STREAMS OPERATORS

AMQ Streams supports Kafka using *Operators* to deploy and manage the components and dependencies of Kafka to OpenShift.

Operators are a method of packaging, deploying, and managing an OpenShift application. AMQ Streams Operators extend OpenShift functionality, automating common and complex tasks related to a Kafka deployment. By implementing knowledge of Kafka operations in code, Kafka administration tasks are simplified and require less manual intervention.

Operators

AMQ Streams provides Operators for managing a Kafka cluster running within an OpenShift cluster.

Cluster Operator

Deploys and manages Apache Kafka clusters, Kafka Connect, Kafka MirrorMaker, Kafka Bridge, Kafka Exporter, and the Entity Operator

Entity Operator

Comprises the Topic Operator and User Operator

Topic Operator

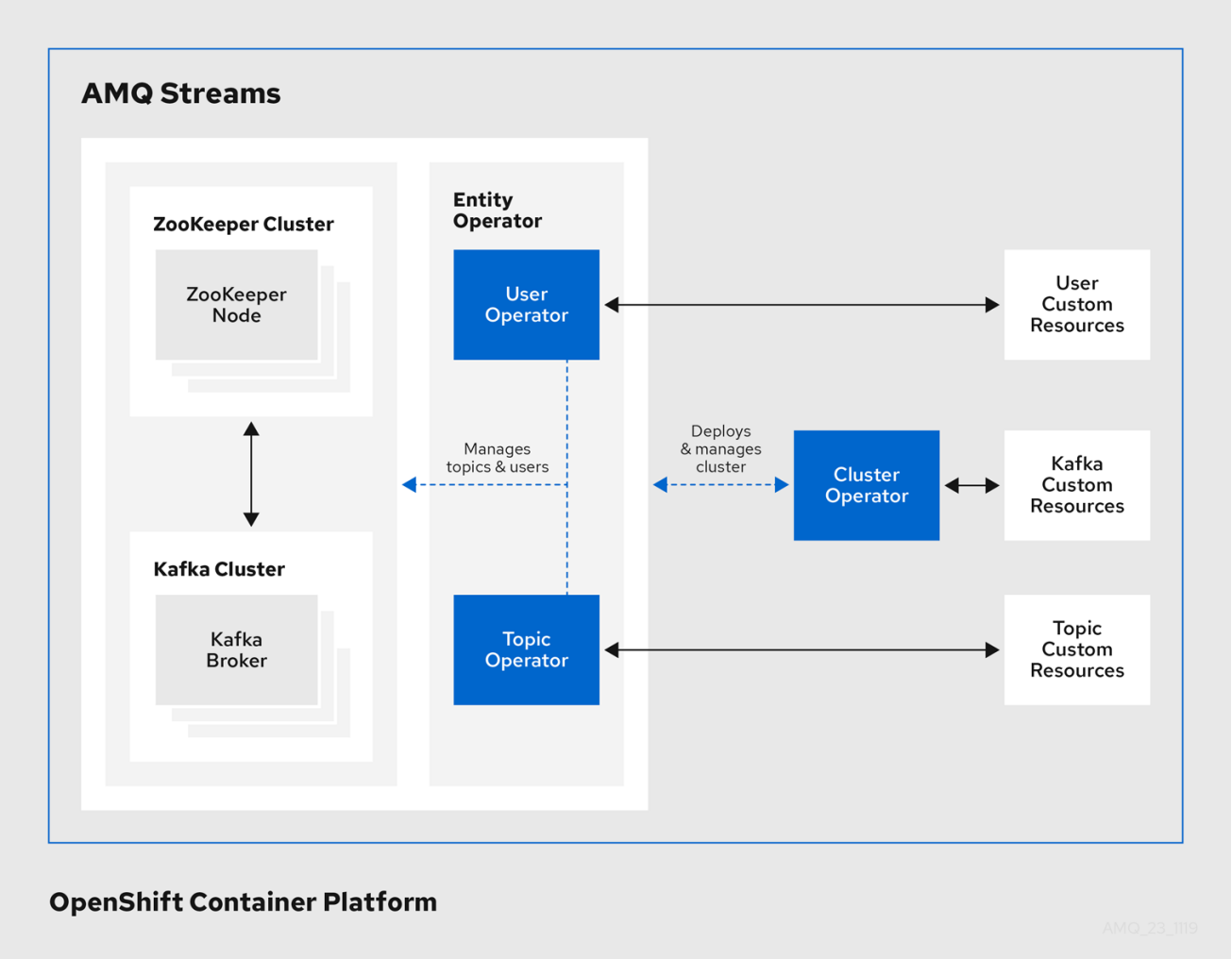
Manages Kafka topics

User Operator

Manages Kafka users

The Cluster Operator can deploy the Topic Operator and User Operator as part of an **Entity Operator** configuration at the same time as a Kafka cluster.

Operators within the AMQ Streams architecture



1.4.1. Cluster Operator

AMQ Streams uses the Cluster Operator to deploy and manage clusters for:

- Kafka (including ZooKeeper, Entity Operator, Kafka Exporter, and Cruise Control)
- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

Custom resources are used to deploy the clusters.

For example, to deploy a Kafka cluster:

- A **Kafka** resource with the cluster configuration is created within the OpenShift cluster.
- The Cluster Operator deploys a corresponding Kafka cluster, based on what is declared in the **Kafka** resource.

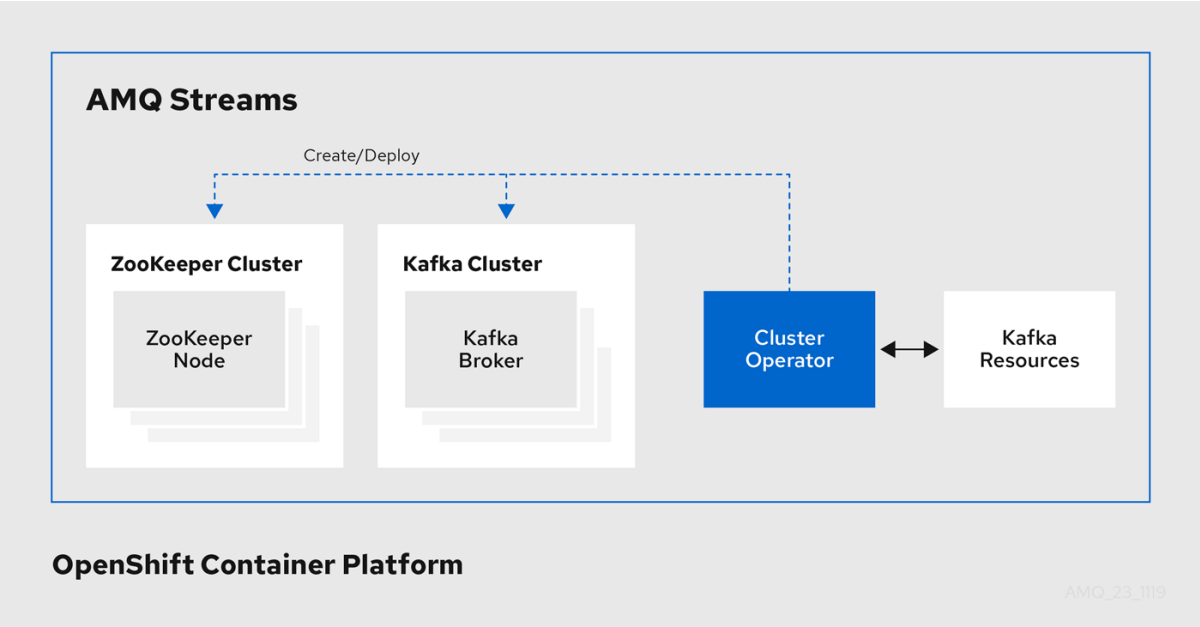
The Cluster Operator can also deploy (through configuration of the **Kafka** resource):

- A Topic Operator to provide operator-style topic management through **KafkaTopic** custom resources

- A User Operator to provide operator-style user management through **KafkaUser** custom resources

The Topic Operator and User Operator function within the Entity Operator on deployment.

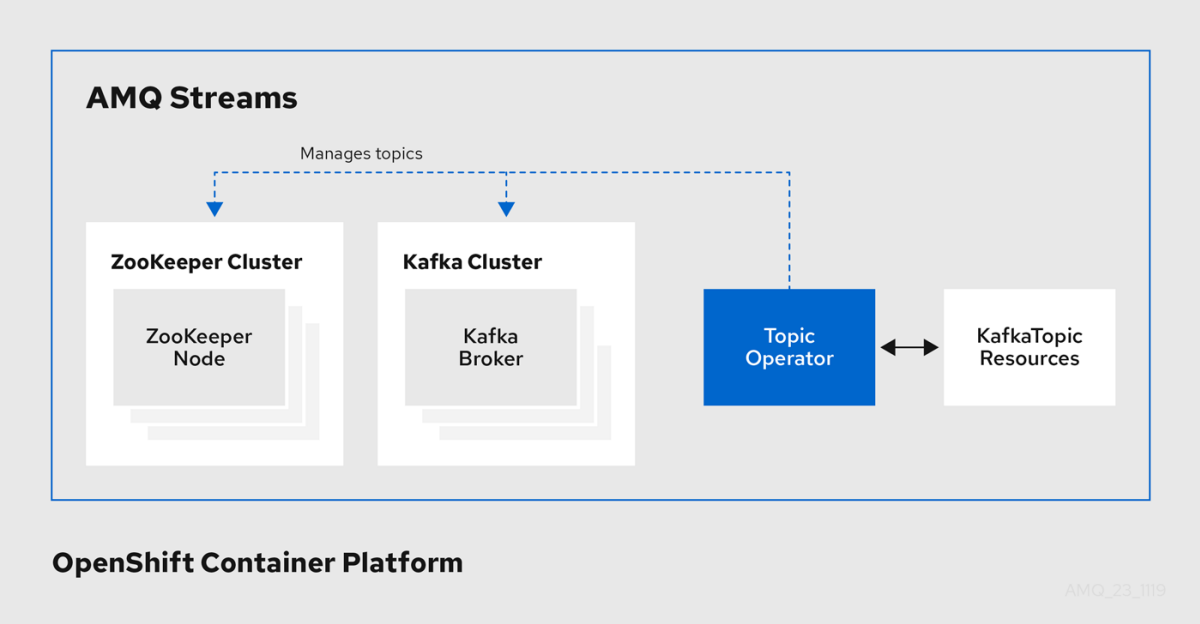
Example architecture for the Cluster Operator



1.4.2. Topic Operator

The Topic Operator provides a way of managing topics in a Kafka cluster through OpenShift resources.

Example architecture for the Topic Operator



The role of the Topic Operator is to keep a set of **KafkaTopic** OpenShift resources describing Kafka topics in-sync with corresponding Kafka topics.

Specifically, if a **KafkaTopic** is:

- Created, the Topic Operator creates the topic

- Deleted, the Topic Operator deletes the topic
- Changed, the Topic Operator updates the topic

Working in the other direction, if a topic is:

- Created within the Kafka cluster, the Operator creates a **KafkaTopic**
- Deleted from the Kafka cluster, the Operator deletes the **KafkaTopic**
- Changed in the Kafka cluster, the Operator updates the **KafkaTopic**

This allows you to declare a **KafkaTopic** as part of your application's deployment and the Topic Operator will take care of creating the topic for you. Your application just needs to deal with producing or consuming from the necessary topics.

If the topic is reconfigured or reassigned to different Kafka nodes, the **KafkaTopic** will always be up to date.

1.4.3. User Operator

The User Operator manages Kafka users for a Kafka cluster by watching for **KafkaUser** resources that describe Kafka users, and ensuring that they are configured properly in the Kafka cluster.

For example, if a **KafkaUser** is:

- Created, the User Operator creates the user it describes
- Deleted, the User Operator deletes the user it describes
- Changed, the User Operator updates the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the OpenShift resources. Kafka topics can be created by applications directly in Kafka, but it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator.

The User Operator allows you to declare a **KafkaUser** resource as part of your application's deployment. You can specify the authentication and authorization mechanism for the user. You can also configure *user quotas* that control usage of Kafka resources to ensure, for example, that a user does not monopolize access to a broker.

When the user is created, the user credentials are created in a **Secret**. Your application needs to use the user and its credentials for authentication and to produce or consume messages.

In addition to managing credentials for authentication, the User Operator also manages authorization rules by including a description of the user's access rights in the **KafkaUser** declaration.

1.5. 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.

Additional resources

- Extend the Kubernetes API with CustomResourceDefinitions

1.5.1. 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.

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/v1beta1
kind: CustomResourceDefinition
metadata: 1
  name: kafkatopics.kafka.strimzi.io
  labels:
    app: strimzi
```

```
spec: 2
  group: kafka.strimzi.io
  versions:
    v1beta1
  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/v1beta1
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.

1.6. DOCUMENT CONVENTIONS

Replaceables

In this document, replaceable text is styled in **monospace**, with italics, uppercase, and hyphens.

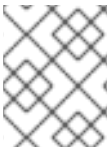
For example, in the following code, you will want to replace ***MY-NAMESPACE*** with the name of your namespace:

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

CHAPTER 2. GETTING STARTED WITH AMQ STREAMS

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.

AMQ Streams is based on Strimzi 0.18.x. This section describes the procedures to deploy AMQ Streams on OpenShift 3.11 and later.



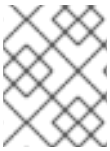
NOTE

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

2.1. PREPARING FOR YOUR AMQ STREAMS DEPLOYMENT

This section shows how you prepare for a 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 you 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.

2.1.1. Deployment prerequisites

To deploy AMQ Streams, make sure:

- An OpenShift 3.11 and later cluster is available
  AMQ Streams is based on AMQ Streams Strimzi 0.18.x.
- The **oc** command-line tool is installed and configured to connect to the running cluster.

NOTE

AMQ Streams supports some features that are specific to OpenShift, where such integration benefits OpenShift users and there is no equivalent implementation using standard OpenShift.

2.1.2. Downloading AMQ Streams release artifacts

To install AMQ Streams, download and extract the release artifacts from the **amq-streams-*<version>*-ocp-install-examples.zip** file from the AMQ Streams download site .

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.

You deploy AMQ Streams to an OpenShift cluster using the **oc** command-line tool.

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.

2.1.3. 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:

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/amq7/amq-streams-kafka-25-rhel7:1.5.0 • registry.redhat.io/amq7/amq-streams-kafka-24-rhel7:1.5.0	AMQ Streams image for running Kafka, including: • Kafka Broker • Kafka Connect / S2I • Kafka Mirror Maker • ZooKeeper • TLS Sidecars
Operator	• registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0	AMQ Streams image for running the operators: • Cluster Operator • Topic Operator • User Operator • Kafka Initializer
Kafka Bridge	• registry.redhat.io/amq7/amq-streams-bridge-rhel7:1.5.0	AMQ Streams image for running the AMQ Streams Kafka Bridge

2.1.4. 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 apply -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
```

2.1.5. AMQ Streams installation methods

There are two ways to install AMQ Streams on OpenShift.

Installation method	Description	Supported versions

Installation method	Description	Supported versions
Installation artifacts (YAML files)	Download the **amq-streams-x.y.z-ocp-install-examples.zip** file from the AMQ Streams download site. Next, 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.	OpenShift 3.11 and later
OperatorHub	Use the **AMQ Streams Operator** in the OperatorHub to deploy the Cluster Operator to a single namespace or all namespaces.	OpenShift 4.x only

For the greatest flexibility, choose the installation artifacts method. Choose the OperatorHub method if you want to install AMQ Streams to OpenShift 4 in a standard configuration using the OpenShift 4 web console. The OperatorHub also allows you to take advantage of automatic updates.

In the case of both methods, the Cluster Operator is deployed to your OpenShift cluster, ready for you to deploy the other components of AMQ Streams, starting with a Kafka cluster, using the YAML example files provided.

AMQ Streams installation artifacts
The AMQ Streams installation artifacts contain various YAML files that can be deployed to OpenShift, using **oc**, to create custom resources, including:

- Deployments
- Custom resource definitions (CRDs)
- Roles and role bindings
- Service accounts

YAML installation files are provided for the Cluster Operator, Topic Operator, User Operator, and the Strimzi Admin role.

OperatorHub
In OpenShift 4, the *Operator Lifecycle Manager (OLM)* helps cluster administrators to install, update, and manage the lifecycle of all Operators and their associated services running across their clusters. The OLM is part of the *Operator Framework*, an open source toolkit designed to manage Kubernetes-native applications (Operators) in an effective, automated, and scalable way.

The *OperatorHub* is part of the OpenShift 4 web console. Cluster administrators can use it to discover, install, and upgrade Operators. Operators can be pulled from the OperatorHub, installed on the OpenShift cluster to a single (project) namespace or all (projects) namespaces, and managed by the OLM. Engineering teams can then independently manage the software in development, test, and production environments using the OLM.

NOTE

The OperatorHub is not available in versions of OpenShift earlier than version 4.

AMQ Streams Operator

The *AMQ Streams Operator* is available to install from the OperatorHub. Once installed, the AMQ Streams Operator deploys the Cluster Operator to your OpenShift cluster, along with the necessary CRDs and role-based access control (RBAC) resources.

Additional resources

Installing AMQ Streams using the installation artifacts:

- Section 2.2.1.2, "Deploying the Cluster Operator to watch a single namespace"
- Section 2.2.1.3, "Deploying the Cluster Operator to watch multiple namespaces"
- Section 2.2.1.4, "Deploying the Cluster Operator to watch all namespaces"

Installing AMQ Streams from the OperatorHub:

- Section 2.2.1.5, "Deploying the Cluster Operator from the OperatorHub"
- *Operators* guide in the OpenShift documentation.

2.2. CREATE THE KAFKA CLUSTER

In order to create your Kafka cluster, you deploy the Cluster Operator to manage the Kafka cluster, then deploy the Kafka cluster.

When deploying the Kafka cluster using the **Kafka** resource, you can deploy the Topic Operator and User Operator at the same time. Alternatively, if you are using a non-AMQ Streams Kafka cluster, you can deploy the Topic Operator and User Operator as standalone components.

Deploying a Kafka cluster with the Topic Operator and User Operator

Perform these deployment steps if you want to use the Topic Operator and User Operator with a Kafka cluster managed by AMQ Streams.

1. Deploy the Cluster Operator
2. Use the Cluster Operator to deploy the:
   a. Kafka cluster
   b. Topic Operator
   c. User Operator

Deploying a standalone Topic Operator and User Operator

Perform these deployment steps if you want to use the Topic Operator and User Operator with a Kafka cluster that is **not managed** by AMQ Streams.

1. Deploy the standalone Topic Operator
2. Deploy the standalone User Operator

2.2.1. Deploying the Cluster Operator

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

The procedures in this section show:

- How to deploy the Cluster Operator to *watch*:
  - A single namespace
  - Multiple namespaces
  - All namespaces
- Alternative deployment options:
  - How to deploy the Cluster Operator deployment from the OperatorHub .

2.2.1.1. Watch options for a Cluster Operator deployment

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

You can choose to deploy the Cluster Operator to watch Kafka resources from:

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

NOTE

AMQ Streams provides example YAML files to make the deployment process easier.

The Cluster Operator watches for changes to the following resources:

- **Kafka** for the Kafka cluster.
- **KafkaConnect** for the Kafka Connect cluster.
- **KafkaConnectS2I** for the Kafka Connect cluster with Source2Image support.
- **KafkaConnector** for creating and managing connectors in a Kafka Connect cluster.
- **KafkaMirrorMaker** for the Kafka MirrorMaker instance.
- **KafkaBridge** for the Kafka Bridge instance

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.

2.2.1.2. 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

- This procedure requires use of an OpenShift user account which is able to create **CustomResourceDefinitions**, **ClusterRoles** and **ClusterRoleBindings**. Use of Role Base Access Control (RBAC) in the OpenShift cluster usually means that permission to create, edit, and delete these resources is limited to OpenShift cluster administrators, such as **system:admin**.

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 apply -f install/cluster-operator -n my-cluster-operator-namespace
   ```

3. Verify that the Cluster Operator was successfully deployed:

   ```
   oc get deployments
   ```

2.2.1.3. 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

- This procedure requires use of an OpenShift user account which is able to create

CustomResourceDefinitions, **ClusterRoles** and **ClusterRoleBindings**. Use of Role Base Access Control (RBAC) in the OpenShift cluster usually means that permission to create, edit, and delete these resources is limited to OpenShift cluster administrators, such as **system:admin**.

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/050-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/amq7/amq-streams-rhel7-operator:1.5.0
           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 apply -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -n watched-namespace
   oc apply -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-operator-delegation.yaml -n watched-namespace
   oc apply -f install/cluster-operator/032-RoleBinding-strimzi-cluster-operator-topic-operator-delegation.yaml -n watched-namespace
   ```

4. Deploy the Cluster Operator:

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

5. Verify that the Cluster Operator was successfully deployed:

```
oc get deployments
```

2.2.1.4. 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

- This procedure requires use of an OpenShift user account which is able to create **CustomResourceDefinitions**, **ClusterRoles** and **ClusterRoleBindings**. Use of Role Base Access Control (RBAC) in the OpenShift cluster usually means that permission to create, edit, and delete these resources is limited to OpenShift cluster administrators, such as **system:admin**.

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/050-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/amq7/amq-streams-rhel7-operator:1.5.0
        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-entity-operator-delegation --clusterrole=strimzi-entity-operator --serviceaccount my-cluster-operator-namespace:strimzi-cluster-operator
oc create clusterrolebinding strimzi-cluster-operator-topic-operator-delegation --clusterrole=strimzi-topic-operator --serviceaccount my-cluster-operator-namespace:strimzi-cluster-operator
```

   Replace ***my-cluster-operator-namespace*** with the namespace you want to install the Cluster Operator into.

4. Deploy the Cluster Operator to your OpenShift cluster.

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

5. Verify that the Cluster Operator was successfully deployed:

```
oc get deployments
```

2.2.1.5. Deploying the Cluster Operator from the OperatorHub

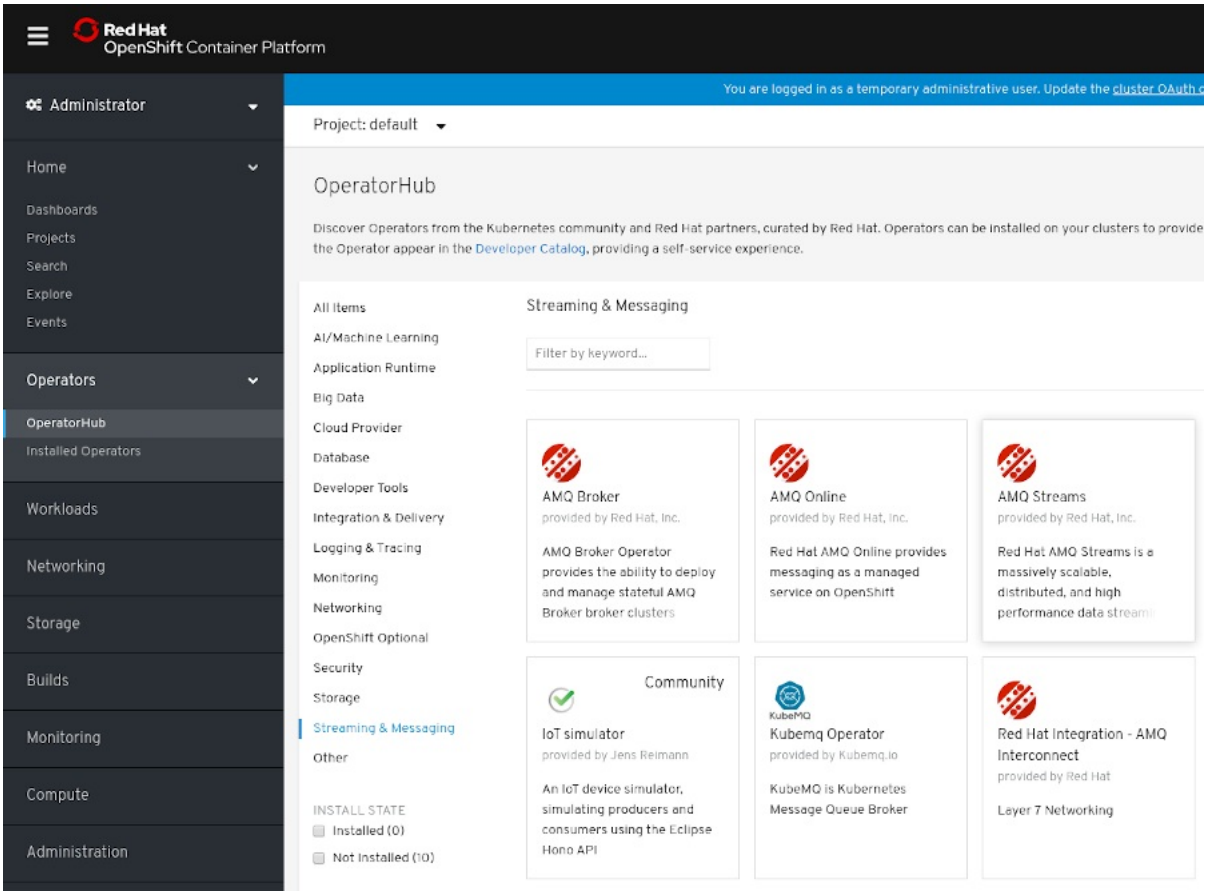
You can deploy the Cluster Operator to your OpenShift cluster by installing the AMQ Streams Operator from the OperatorHub. The OperatorHub is available in OpenShift 4 only.

Prerequisites

- The **Red Hat Operators OperatorSource** is enabled in your OpenShift cluster. If you can see Red Hat Operators in the OperatorHub, the correct **OperatorSource** is enabled. For more information, see the *Operators* guide.
- Installation requires a user with sufficient privileges to install Operators from the OperatorHub.

Procedure

1. In the OpenShift 4 web console, click **Operators > OperatorHub**.
2. Search or browse for the **AMQ Streams** Operator, in the **Streaming & Messaging** category.



3. Click the **AMQ Streams** tile and then, in the sidebar on the right, click **Install**.

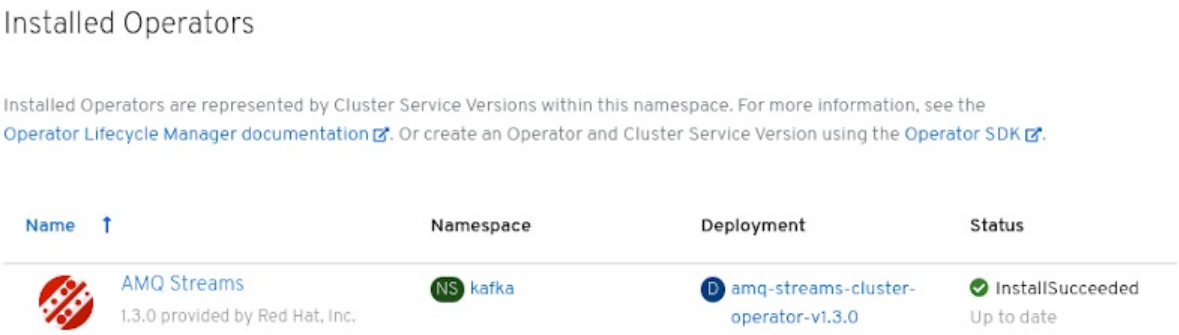
4. On the Create Operator Subscription screen, choose from the following installation and update options:

   - **Installation Mode**: Choose to install the AMQ Streams Operator to all (projects) namespaces in the cluster (the default option) or a specific (project) namespace. It is good practice to use namespaces to separate functions. We recommend that you dedicate a specific namespace to the Kafka cluster and other AMQ Streams components.

   - **Approval Strategy**: 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.

5. Click **Subscribe**; the AMQ Streams Operator is installed to your OpenShift cluster.
   The AMQ Streams Operator deploys the Cluster Operator, CRDs, and role-based access control (RBAC) resources to the selected namespace, or to all namespaces.

6. On the Installed Operators screen, check the progress of the installation. The AMQ Streams Operator is ready to use when its status changes to **InstallSucceeded**.



Next, you can deploy the other components of AMQ Streams, starting with a Kafka cluster, using the YAML example files.

Additional resources

- Section 2.1.5, "AMQ Streams installation methods"
- Section 2.2.2.1, "Deploying the Kafka cluster"

2.2.2. Deploying Kafka

Apache Kafka is an open-source distributed publish-subscribe messaging system for fault-tolerant real-time data feeds.

The procedures in this section show:

- How to use the Cluster Operator to deploy:
  - An *ephemeral* or *persistent* Kafka cluster
  - The Topic Operator and User Operator by configuring the **Kafka** custom resource:
    - Topic Operator
    - User Operator
- Alternative standalone deployment procedures for the Topic Operator and User Operator:
  - Deploy the standalone Topic Operator
  - Deploy the standalone User Operator

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

2.2.2.1. Deploying the Kafka cluster

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

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

AMQ Streams provides example YAMLs files for deployment in **examples/kafka/**:

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 **PersistentVolumes** to store ZooKeeper and Kafka data. The **PersistentVolume** is acquired using a **PersistentVolumeClaim** to make it independent of the actual type of the **PersistentVolume**. For example, it can use Amazon EBS volumes in Amazon AWS deployments without any changes in the YAML files. The **PersistentVolumeClaim** can use a **StorageClass** to trigger automatic volume provisioning.

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.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
# ...
```

For more information about configuring the **Kafka** resource, see Kafka cluster configuration

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create and deploy an *ephemeral* or *persistent* cluster.
   For development or testing, you might prefer to use an ephemeral cluster. You can use a persistent cluster in any situation.

   - 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. Verify that the Kafka cluster was successfully deployed:

```
oc get deployments
```

2.2.2.2. 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**.

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 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

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

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

```
oc apply -f <your-file>
```

2.2.2.3. 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**.

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 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/v1beta1
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 <your-file>
```

2.2.3. Alternative standalone deployment options for AMQ Streams Operators

When deploying a Kafka cluster using the Cluster Operator, you can also deploy the Topic Operator and User Operator. Alternatively, you can perform a standalone deployment.

A standalone deployment means the Topic Operator and User Operator can operate with a Kafka cluster that is not managed by AMQ Streams.

2.2.3.1. Deploying the standalone Topic Operator

This procedure shows how to deploy the Topic Operator as a standalone component.

A standalone deployment requires configuration of environment variables, and is more complicated than deploying the Topic Operator using the Cluster Operator . However, a standalone deployment is more flexible as the Topic Operator can operate with *any* Kafka cluster, not necessarily one deployed by the Cluster Operator.

Prerequisites

- You need an existing Kafka cluster for the Topic Operator to connect to.

Procedure

1. Edit the **Deployment.spec.template.spec.containers[0].env** properties in the **install/topic-operator/05-Deployment-strimzi-topic-operator.yaml** file by setting:

   a. **STRIMZI_KAFKA_BOOTSTRAP_SERVERS** to list the bootstrap brokers in your Kafka cluster, given as a comma-separated list of ***hostname*:*port*** pairs.

   b. **STRIMZI_ZOOKEEPER_CONNECT** to list the ZooKeeper nodes, given as a comma-separated list of ***hostname*:*port*** pairs. This should be the same ZooKeeper cluster that your Kafka cluster is using.

   c. **STRIMZI_NAMESPACE** to the OpenShift namespace in which you want the operator to watch for **KafkaTopic** resources.

   d. **STRIMZI_RESOURCE_LABELS** to the label selector used to identify the **KafkaTopic** resources managed by the operator.

   e. **STRIMZI_FULL_RECONCILIATION_INTERVAL_MS** to specify the interval between periodic reconciliations, in milliseconds.

   f. **STRIMZI_TOPIC_METADATA_MAX_ATTEMPTS** to specify the number of attempts at getting topic metadata from Kafka. The time between each attempt is defined as an exponential back-off. Consider increasing this value when topic creation could take more time due to the number of partitions or replicas. Default **6**.

   g. **STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS** to the ZooKeeper session timeout, in milliseconds. For example, **10000**. Default **20000** (20 seconds).

   h. **STRIMZI_TOPICS_PATH** to the Zookeeper node path where the Topic Operator stores its metadata. Default **/strimzi/topics**.

   i. **STRIMZI_TLS_ENABLED** to enable TLS support for encrypting the communication with Kafka brokers. Default **true**.

   j. **STRIMZI_TRUSTSTORE_LOCATION** to the path to the truststore containing certificates for enabling TLS based communication. Mandatory only if TLS is enabled through **STRIMZI_TLS_ENABLED**.

   k. **STRIMZI_TRUSTSTORE_PASSWORD** to the password for accessing the truststore defined by **STRIMZI_TRUSTSTORE_LOCATION**. Mandatory only if TLS is enabled through **STRIMZI_TLS_ENABLED**.

   l. **STRIMZI_KEYSTORE_LOCATION** to the path to the keystore containing private keys for enabling TLS based communication. Mandatory only if TLS is enabled through **STRIMZI_TLS_ENABLED**.

   m. **STRIMZI_KEYSTORE_PASSWORD** to the password for accessing the keystore defined by **STRIMZI_KEYSTORE_LOCATION**. Mandatory only if TLS is enabled through **STRIMZI_TLS_ENABLED**.

   n. **STRIMZI_LOG_LEVEL** to the level for printing logging messages. The value can be set to: **ERROR**, **WARNING**, **INFO**, **DEBUG**, and **TRACE**. Default **INFO**.

   o. **STRIMZI_JAVA_OPTS** *(optional)* to the Java options used for the JVM running the Topic Operator. An example is **-Xmx=512M -Xms=256M**.

   p. **STRIMZI_JAVA_SYSTEM_PROPERTIES** *(optional)* to list the **-D** options which are set to the Topic Operator. An example is **-Djavax.net.debug=verbose -DpropertyName=value**.

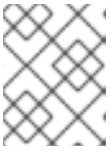
2. Deploy the Topic Operator:

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

3. Verify that the Topic Operator has been deployed successfully:

```
oc describe deployment strimzi-topic-operator
```

The Topic Operator is deployed when the **Replicas:** entry shows **1 available**.



NOTE

You may experience a delay with the deployment if you have a slow connection to the OpenShift cluster and the images have not been downloaded before.

2.2.3.2. Deploying the standalone User Operator

This procedure shows how to deploy the User Operator as a standalone component.

A standalone deployment requires configuration of environment variables, and is more complicated than deploying the User Operator using the Cluster Operator . However, a standalone deployment is more flexible as the User Operator can operate with *any* Kafka cluster, not necessarily one deployed by the Cluster Operator.

Prerequisites

- You need an existing Kafka cluster for the User Operator to connect to.

Procedure

1. Edit the following **Deployment.spec.template.spec.containers[0].env** properties in the **install/user-operator/05-Deployment-strimzi-user-operator.yaml** file by setting:

   a. **STRIMZI_KAFKA_BOOTSTRAP_SERVERS** to list the Kafka brokers, given as a comma-separated list of ***hostname*:*port*** pairs.

   b. **STRIMZI_ZOOKEEPER_CONNECT** to list the ZooKeeper nodes, given as a comma-separated list of ***hostname*:*port*** pairs. This must be the same ZooKeeper cluster that your Kafka cluster is using. Connecting to ZooKeeper nodes with TLS encryption is not supported.

   c. **STRIMZI_NAMESPACE** to the OpenShift namespace in which you want the operator to watch for **KafkaUser** resources.

   d. **STRIMZI_LABELS** to the label selector used to identify the **KafkaUser** resources managed by the operator.

   e. **STRIMZI_FULL_RECONCILIATION_INTERVAL_MS** to specify the interval between periodic reconciliations, in milliseconds.

   f. **STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS** to the ZooKeeper session timeout, in milliseconds. For example, **10000**. Default **20000** (20 seconds).

g. **STRIMZI_CA_CERT_NAME** to point to an OpenShift **Secret** that contains the public key of the Certificate Authority for signing new user certificates for TLS client authentication. The **Secret** must contain the public key of the Certificate Authority under the key **ca.crt**.

h. **STRIMZI_CA_KEY_NAME** to point to an OpenShift **Secret** that contains the private key of the Certificate Authority for signing new user certificates for TLS client authentication. The **Secret** must contain the private key of the Certificate Authority under the key **ca.key**.

i. **STRIMZI_CLUSTER_CA_CERT_SECRET_NAME** to point to an OpenShift **Secret** containing the public key of the Certificate Authority used for signing Kafka brokers certificates for enabling TLS-based communication. The **Secret** must contain the public key of the Certificate Authority under the key **ca.crt**. This environment variable is optional and should be set only if the communication with the Kafka cluster is TLS based.

j. **STRIMZI_EO_KEY_SECRET_NAME** to point to an OpenShift **Secret** containing the private key and related certificate for TLS client authentication against the Kafka cluster. The **Secret** must contain the keystore with the private key and certificate under the key **entity-operator.p12**, and the related password under the key **entity-operator.password**. This environment variable is optional and should be set only if TLS client authentication is needed when the communication with the Kafka cluster is TLS based.

k. **STRIMZI_CA_VALIDITY** the validity period for the Certificate Authority. Default is **365** days.

l. **STRIMZI_CA_RENEWAL** the renewal period for the Certificate Authority.

m. **STRIMZI_LOG_LEVEL** to the level for printing logging messages. The value can be set to: **ERROR**, **WARNING**, **INFO**, **DEBUG**, and **TRACE**. Default **INFO**.

n. **STRIMZI_GC_LOG_ENABLED** to enable garbage collection (GC) logging. Default **true**. Default is **30** days to initiate certificate renewal before the old certificates expire.

o. **STRIMZI_JAVA_OPTS** *(optional)* to the Java options used for the JVM running User Operator. An example is **-Xmx=512M -Xms=256M**.

p. **STRIMZI_JAVA_SYSTEM_PROPERTIES** *(optional)* to list the **-D** options which are set to the User Operator. An example is **-Djavax.net.debug=verbose -DpropertyName=value**.

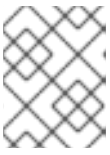
2. Deploy the User Operator:

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

3. Verify that the User Operator has been deployed successfully:

```
oc describe deployment strimzi-user-operator
```

The User Operator is deployed when the **Replicas:** entry shows **1 available**.



NOTE

You may experience a delay with the deployment if you have a slow connection to the OpenShift cluster and the images have not been downloaded before.

2.3. DEPLOY KAFKA CONNECT

Kafka Connect is a tool for streaming data between Apache Kafka and external systems.

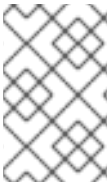
In AMQ Streams, Kafka Connect is deployed in distributed mode. Kafka Connect can also work in standalone mode, but this is not supported by AMQ Streams.

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.

Kafka Connect is typically used to integrate Kafka with external databases and storage and messaging systems.

The procedures in this section show how to:

- Deploy a Kafka Connect cluster using a **KafkaConnect** resource
- Create a Kafka Connect image containing the connectors you need to make your connection
- Create and manage connectors using a **KafkaConnector** resource or the Kafka Connect REST API
- Deploy a **KafkaConnector** resource to Kafka Connect



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.

2.3.1. 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 is implemented as a **Deployment** 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.

In this procedure, we use the example file provided with AMQ Streams:

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

For more information about configuring the **KafkaConnect** resource, see:

- Kafka Connect cluster configuration
- Kafka Connect cluster configuration with Source2Image support

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Deploy Kafka Connect to your OpenShift cluster.

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

2. Verify that Kafka Connect was successfully deployed:

```
oc get deployments
```

2.3.2. Extending Kafka Connect with connector plug-ins

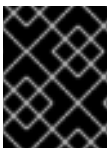
The AMQ Streams container images for Kafka Connect include two built-in file connectors for moving file-based data into and out of your Kafka cluster.

File Connector	Description
FileStreamSourceConnector	Transfers data to your Kafka cluster from a file (the source).
FileStreamSinkConnector	Transfers data from your Kafka cluster to a file (the sink).

The Cluster Operator can also use images that you have created to deploy a Kafka Connect cluster to your OpenShift cluster.

The procedures in this section show how to add your own connector classes to connector images by:

- Creating a container image from the Kafka Connect base image (manually or using continuous integration)
- Creating a container image using OpenShift builds and Source-to-Image (S2I) (available only on OpenShift)



IMPORTANT

You create the configuration for connectors directly using the Kafka Connect REST API or KafkaConnector custom resources.

2.3.2.1. Creating a Docker image from the Kafka Connect base image

This procedure shows how to create a custom image and add it 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 plug-ins.

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

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create a new **Dockerfile** using **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** as the base image:

```
FROM registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Example plug-in file

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-3.4.2.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mongodb-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mongodb-driver-3.4.2.jar
│   ├── mongodb-driver-core-3.4.2.jar
│   └── README.md
├── debezium-connector-mysql
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mysql-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mysql-binlog-connector-java-0.13.0.jar
│   ├── mysql-connector-java-5.1.40.jar
│   ├── README.md
│   └── wkb-1.0.2.jar
└── debezium-connector-postgres
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-postgres-0.7.1.jar
    ├── debezium-core-0.7.1.jar
    ├── LICENSE.txt
    ├── postgresql-42.0.0.jar
    ├── protobuf-java-2.6.1.jar
    └── README.md
```

2. Build the container image.

3. Push your custom image to your container registry.

4. Point to the new container image.
   You can either:

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

```
apiVersion: kafka.strimzi.io/v1beta1
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).

or

- In the **install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml** file, edit the **STRIMZI_KAFKA_CONNECT_IMAGES** variable to point to the new container image, and then reinstall the Cluster Operator.

Additional resources

- For more information on the **KafkaConnect.spec.image property**, see Container images.
- For more information on the **STRIMZI_KAFKA_CONNECT_IMAGES** variable, see Cluster Operator Configuration.

2.3.2.2. Creating a container image using OpenShift builds and Source-to-Image

This procedure shows how to use OpenShift builds and the Source-to-Image (S2I) framework to create a new container image.

An OpenShift build takes a builder image with S2I support, together with source code and binaries provided by the user, and uses them to build a new container image. Once built, container images are stored in OpenShift's local container image repository and are available for use in deployments.

A Kafka Connect builder image with S2I support is provided on the Red Hat Ecosystem Catalog as part of the **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** image. This S2I image takes your binaries (with plug-ins and connectors) and stores them in the **/tmp/kafka-plugins/s2i** directory. It creates a new Kafka Connect image from this directory, which can then be used with the Kafka Connect deployment. When started using the enhanced image, Kafka Connect loads any third-party plug-ins from the **/tmp/kafka-plugins/s2i** directory.

Procedure

1. On the command line, use the **oc apply** command to create and deploy a Kafka Connect S2I cluster:

```
oc apply -f examples/connect/kafka-connect-s2i.yaml
```

2. Create a directory with Kafka Connect plug-ins:

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-3.4.2.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mongodb-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mongodb-driver-3.4.2.jar
│   ├── mongodb-driver-core-3.4.2.jar
│   └── README.md
├── debezium-connector-mysql
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mysql-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mysql-binlog-connector-java-0.13.0.jar
│   ├── mysql-connector-java-5.1.40.jar
│   ├── README.md
│   └── wkb-1.0.2.jar
└── debezium-connector-postgres
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-postgres-0.7.1.jar
    ├── debezium-core-0.7.1.jar
    ├── LICENSE.txt
    ├── postgresql-42.0.0.jar
    ├── protobuf-java-2.6.1.jar
    └── README.md
```

3. Use the **oc start-build** command to start a new build of the image using the prepared directory:

```
oc start-build my-connect-cluster-connect --from-dir ./my-plugins/
```

NOTE

The name of the build is the same as the name of the deployed Kafka Connect cluster.

4. When the build has finished, the new image is used automatically by the Kafka Connect deployment.

2.3.3. Creating and managing connectors

When you have created a container image for your connector plug-in, you need to create a connector instance in your Kafka Connect cluster. You can then configure, monitor, and manage a running connector instance.

A connector is an instance of a particular *connector class* that knows how to communicate with the relevant external system in terms of messages. Connectors are available for many external systems, or you can create your own.

You can create *source* and *sink* types of connector.

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.

AMQ Streams provides two APIs for creating and managing connectors:

- **KafkaConnector** resources (referred to as **KafkaConnectors**)
- Kafka Connect REST API

Using the APIs, you can:

- Check the status of a connector instance
- Reconfigure a running connector
- Increase or decrease the number of tasks for a connector instance
- Restart failed tasks (not supported by **KafkaConnector** resource)
- Pause a connector instance
- Resume a previously paused connector instance
- Delete a connector instance

2.3.3.1. KafkaConnector resources

KafkaConnectors allow you to create and manage connector instances for Kafka Connect in an OpenShift-native way, so an HTTP client such as cURL is not required. Like other Kafka resources, you declare a connector's desired state in a **KafkaConnector** YAML file that is deployed to your OpenShift cluster to create the connector instance.

You manage a running connector instance by updating its corresponding **KafkaConnector**, and then applying the updates. You remove a connector by deleting its corresponding **KafkaConnector**.

To ensure compatibility with earlier versions of AMQ Streams, **KafkaConnectors** are disabled by default. To enable them for a Kafka Connect cluster, you must use annotations on the **KafkaConnect** resource. For instructions, see Enabling **KafkaConnector** resources.

When **KafkaConnectors** are enabled, the Cluster Operator begins to watch for them. It updates the configurations of running connector instances to match the configurations defined in their **KafkaConnectors**.

AMQ Streams includes an example **KafkaConnector**, named **examples/connect/source-connector.yaml**. You can use this example to create and manage a **FileStreamSourceConnector**.

2.3.3.2. Availability of the Kafka Connect REST API

The Kafka Connect REST API is available on port 8083 as the **<connect-cluster-name>-connect-api** service.

If **KafkaConnectors** are enabled, manual changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator.

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

2.3.4. Deploying a KafkaConnector resource to Kafka Connect

This procedure describes how to deploy the example **KafkaConnector** to a Kafka Connect cluster.

The example YAML will create a **FileStreamSourceConnector** to send each line of the license file to Kafka as a message in a topic named **my-topic**.

Prerequisites

- A Kafka Connect deployment in which **KafkaConnectors** are enabled
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
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
  config: 5
    file: "/opt/kafka/LICENSE"
    topic: my-topic
    # ...
```

1. Enter a name for the **KafkaConnector** resource. This will be used as the name of the connector within Kafka Connect. You can choose any name that is valid for an OpenShift resource.
2. Enter the name of the Kafka Connect cluster in which to create the connector.
3. The name or alias of the connector class. This should be present in the image being used by the Kafka Connect cluster.
4. The maximum number of tasks that the connector can create.
5. Configuration settings for the connector. Available configuration options depend on the connector class.

2. Create the **KafkaConnector** in your OpenShift cluster:

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

3. Check that the resource was created:

```
oc get kctr --selector strimzi.io/cluster=my-connect-cluster -o name
```

2.4. DEPLOY 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.

2.4.1. 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.

In this procedure, we use the example files provided with AMQ Streams:

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

For information about configuring **KafkaMirrorMaker** or **KafkaMirrorMaker2** resources, see Kafka MirrorMaker configuration.

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 2.0:

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

2. Verify that MirrorMaker was successfully deployed:

```
oc get deployments
```

2.5. DEPLOY KAFKA BRIDGE

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

2.5.1. 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.

In this procedure, we use the example file provided with AMQ Streams:

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

For information about configuring the **KafkaBridge** resource, see Kafka Bridge configuration .

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. Verify that Kafka Bridge was successfully deployed:

```
oc get deployments
```

2.6. DEPLOYING EXAMPLE CLIENTS

This procedure shows how to deploy example producer and consumer clients that use the Kafka cluster you created to send and receive messages.

Prerequisites

- The Kafka cluster is available for the clients.

Procedure

1. Deploy a Kafka producer.

```
oc run kafka-producer -ti --image=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0
--rm=true --restart=Never -- bin/kafka-console-producer.sh --broker-list 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/amq7/amq-streams-kafka-25-rhel7:1.5.0
--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.

CHAPTER 3. DEPLOYMENT CONFIGURATION

This chapter describes how to configure different aspects of the supported deployments:

- Kafka clusters
- Kafka Connect clusters
- Kafka Connect clusters with *Source2Image* support
- Kafka Mirror Maker
- Kafka Bridge
- OAuth 2.0 token-based authentication
- OAuth 2.0 token-based authorization

3.1. KAFKA CLUSTER CONFIGURATION

The full schema of the **Kafka** resource is described in the Section B.1, "**Kafka** schema reference". All labels that are applied to the desired **Kafka** resource will also be applied to the OpenShift resources making up the Kafka cluster. This provides a convenient mechanism for resources to be labeled as required.

3.1.1. Sample Kafka YAML configuration

For help in understanding the configuration options available for your Kafka deployment, refer to sample YAML file provided here.

The sample shows only some of the possible configuration options, but those that are particularly important include:

- Resource requests (CPU / Memory)
- JVM options for maximum and minimum memory allocation
- Listeners (and authentication)
- Authentication
- Storage
- Rack awareness
- Metrics

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3 (1)
    version: 1.5 (2)
```

```
    resources: 3
      requests:
        memory: 64Gi
        cpu: "8"
      limits: 4
        memory: 64Gi
        cpu: "12"
    jvmOptions: 5
      -Xms: 8192m
      -Xmx: 8192m
    listeners: 6
      tls:
        authentication:7
          type: tls
      external: 8
        type: route
        authentication:
          type: tls
        configuration:
          brokerCertChainAndKey: 9
            secretName: my-secret
            certificate: my-certificate.crt
            key: my-key.key
    authorization: 10
      type: simple
    config: 11
      auto.create.topics.enable: "false"
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 12
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
    storage: 13
      type: persistent-claim 14
      size: 10000Gi 15
    rack: 16
      topologyKey: failure-domain.beta.kubernetes.io/zone
    metrics: 17
      lowercaseOutputName: true
      rules: 18
      # Special cases and very specific rules
      - pattern : kafka.server<type=(.+), name=(.+), clientId=(.+), topic=(.+), partition=(.*)><>Value
        name: kafka_server_$1_$2
        type: GAUGE
        labels:
          clientId: "$3"
          topic: "$4"
          partition: "$5"
        # ...
  zookeeper: 19
    replicas: 3
    resources:
      requests:
```

```
      memory: 8Gi
      cpu: "2"
    limits:
      memory: 8Gi
      cpu: "2"
  jvmOptions:
    -Xms: 4096m
    -Xmx: 4096m
  storage:
    type: persistent-claim
    size: 1000Gi
  metrics:
    # ...
entityOperator: 20
  topicOperator:
    resources:
      requests:
        memory: 512Mi
        cpu: "1"
      limits:
        memory: 512Mi
        cpu: "1"
  userOperator:
    resources:
      requests:
        memory: 512Mi
        cpu: "1"
      limits:
        memory: 512Mi
        cpu: "1"
kafkaExporter: 21
  # ...
cruiseControl: 22
  # ...
```

1. Replicas specifies the number of broker nodes.
2. Kafka version, which can be changed by following the upgrade procedure .
3. Resource requests specify the resources to reserve for a given container .
4. Resource limits specify the maximum resources that can be consumed by a container.
5. JVM options can specify the minimum (**-Xms**) and maximum (**-Xmx**) memory allocation for JVM.
6. Listeners configure how clients connect to the Kafka cluster via bootstrap addresses. Listeners are configured as **plain** (without encryption), **tls** or **external**.
7. Listener authentication mechanisms may be configured for each listener, and specified as mutual TLS or SCRAM-SHA.
8. External listener configuration specifies how the Kafka cluster is exposed outside OpenShift, such as through a **route**, **loadbalancer** or **nodeport**.
9.

Optional configuration for a Kafka listener certificate managed by an external Certificate Authority. The **brokerCertChainAndKey** property specifies a **Secret** that holds a server certificate and a

10 Authorization enables **simple** authorization on the Kafka broker using the **SimpleAclAuthorizer** Kafka plugin.

11 Config specifies the broker configuration. Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

12 SSL properties for external listeners to run with a specific *cipher suite* for a TLS version.

13 Storage is configured as **ephemeral**, **persistent-claim** or **jbod**.

14 Storage size for persistent volumes may be increased and additional volumes may be added to JBOD storage.

15 Persistent storage has additional configuration options, such as a storage **id** and **class** for dynamic volume provisioning.

16 Rack awareness is configured to spread replicas across different racks . A **topology** key must match the label of a cluster node.

17 Kafka metrics configuration for use with Prometheus.

18 Kafka rules for exporting metrics to a Grafana dashboard through the JMX Exporter. A set of rules provided with AMQ Streams may be copied to your Kafka resource configuration.

19 ZooKeeper-specific configuration, which contains properties similar to the Kafka configuration.

20 Entity Operator configuration, which specifies the configuration for the Topic Operator and User Operator.

21 Kafka Exporter configuration, which is used to expose data as Prometheus metrics.

22 Cruise Control configuration, which is used to rebalance the Kafka cluster.

3.1.2. Data storage considerations

An efficient data storage infrastructure is essential to the optimal performance of AMQ Streams.

Block storage is required. File storage, such as NFS, does not work with Kafka.

For your block storage, you can choose, for example:

- Cloud-based block storage solutions, such as Amazon Elastic Block Store (EBS)
- Local persistent volumes
- Storage Area Network (SAN) volumes accessed by a protocol such as *Fibre Channel* or *iSCSI*

NOTE

AMQ Streams does not require OpenShift raw block volumes.

3.1.2.1. File systems

It is recommended that you configure your storage system to use the *XFS* file system. AMQ Streams is also compatible with the *ext4* file system, but this might require additional configuration for best results.

3.1.2.2. Apache Kafka and ZooKeeper storage

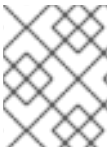
Use separate disks for Apache Kafka and ZooKeeper.

Three types of data storage are supported:

- Ephemeral (Recommended for development only)
- Persistent
- JBOD (Just a Bunch of Disks, suitable for Kafka only)

For more information, see Kafka and ZooKeeper storage .

Solid-state drives (SSDs), though not essential, can improve the performance of Kafka in large clusters where data is sent to and received from multiple topics asynchronously. SSDs are particularly effective with ZooKeeper, which requires fast, low latency data access.



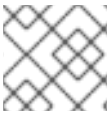
NOTE

You do not need to provision replicated storage because Kafka and ZooKeeper both have built-in data replication.

3.1.3. Kafka and ZooKeeper storage types

As stateful applications, Kafka and ZooKeeper need to store data on disk. AMQ Streams supports three storage types for this data:

- Ephemeral
- Persistent
- JBOD storage



NOTE

JBOD storage is supported only for Kafka, not for ZooKeeper.

When configuring a **Kafka** resource, you can specify the type of storage used by the Kafka broker and its corresponding ZooKeeper node. You configure the storage type using the **storage** property in the following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**

The storage type is configured in the **type** field.

WARNING

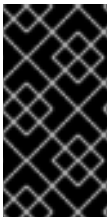
The storage type cannot be changed after a Kafka cluster is deployed.

Additional resources

- For more information about ephemeral storage, see ephemeral storage schema reference.
- For more information about persistent storage, see persistent storage schema reference.
- For more information about JBOD storage, see JBOD schema reference.
- For more information about the schema for **Kafka**, see **Kafka** schema reference.

3.1.3.1. Ephemeral storage

Ephemeral storage uses the **`emptyDir` volumes** volumes to store data. To use ephemeral storage, the **type** field should be set to **ephemeral**.



IMPORTANT

emptyDir volumes are not persistent and the data stored in them will be lost when the Pod is restarted. After the new pod is started, it has to recover all data from other nodes of the cluster. Ephemeral storage is not suitable for use with single node ZooKeeper clusters and for Kafka topics with replication factor 1, because it will lead to data loss.

An example of Ephemeral storage

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: ephemeral
    # ...
  zookeeper:
    # ...
    storage:
      type: ephemeral
    # ...
```

3.1.3.1.1. Log directories

The ephemeral volume will be used by the Kafka brokers as log directories mounted into the following path:

/var/lib/kafka/data/kafka-log_idx_

Where ***idx*** is the Kafka broker pod index. For example **/var/lib/kafka/data/kafka-log0**.

3.1.3.2. Persistent storage

Persistent storage uses Persistent Volume Claims to provision persistent volumes for storing data. Persistent Volume Claims can be used to provision volumes of many different types, depending on the Storage Class which will provision the volume. The data types which can be used with persistent volume claims include many types of SAN storage as well as Local persistent volumes.

To use persistent storage, the **type** has to be set to **persistent-claim**. Persistent storage supports additional configuration options:

id (optional)

Storage identification number. This option is mandatory for storage volumes defined in a JBOD storage declaration. Default is **0**.

size (required)

Defines the size of the persistent volume claim, for example, "1000Gi".

class (optional)

The OpenShift Storage Class to use for dynamic volume provisioning.

selector (optional)

Allows selecting a specific persistent volume to use. It contains key:value pairs representing labels for selecting such a volume.

deleteClaim (optional)

Boolean value which specifies if the Persistent Volume Claim has to be deleted when the cluster is undeployed. Default is **false**.

WARNING

Increasing the size of persistent volumes in an existing AMQ Streams cluster is only supported in OpenShift versions that support persistent volume resizing. The persistent volume to be resized must use a storage class that supports volume expansion. For other versions of OpenShift and storage classes which do not support volume expansion, you must decide the necessary storage size before deploying the cluster. Decreasing the size of existing persistent volumes is not possible.

Example fragment of persistent storage configuration with 1000Gi size

```
# ...
storage:
  type: persistent-claim
  size: 1000Gi
# ...
```

The following example demonstrates the use of a storage class.

Example fragment of persistent storage configuration with specific Storage Class

```
# ...
storage:
  type: persistent-claim
  size: 1Gi
  class: my-storage-class
# ...
```

Finally, a **selector** can be used to select a specific labeled persistent volume to provide needed features such as an SSD.

Example fragment of persistent storage configuration with selector

```
# ...
storage:
  type: persistent-claim
  size: 1Gi
  selector:
    hdd-type: ssd
  deleteClaim: true
# ...
```

3.1.3.2.1. Storage class overrides

You can specify a different storage class for one or more Kafka brokers, instead of using the default storage class. This is useful if, for example, storage classes are restricted to different availability zones or data centers. You can use the **overrides** field for this purpose.

In this example, the default storage class is named **my-storage-class**:

Example AMQ Streams cluster using storage class overrides

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  # ...
  kafka:
    replicas: 3
    storage:
      deleteClaim: true
      size: 100Gi
      type: persistent-claim
      class: my-storage-class
      overrides:
        - broker: 0
          class: my-storage-class-zone-1a
        - broker: 1
          class: my-storage-class-zone-1b
```

```
      - broker: 2
        class: my-storage-class-zone-1c
# ...
```

As a result of the configured **overrides** property, the broker volumes use the following storage classes:

- The persistent volumes of broker 0 will use **my-storage-class-zone-1a**.
- The persistent volumes of broker 1 will use **my-storage-class-zone-1b**.
- The persistent volumes of broker 2 will use **my-storage-class-zone-1c**.

The **overrides** property is currently used only to override storage class configurations. Overriding other storage configuration fields is not currently supported. Other fields from the storage configuration are currently not supported.

3.1.3.2.2. Persistent Volume Claim naming

When persistent storage is used, it creates Persistent Volume Claims with the following names:

data-*cluster-name*-kafka-*idx*

Persistent Volume Claim for the volume used for storing data for the Kafka broker pod ***idx***.

data-*cluster-name*-zookeeper-*idx*

Persistent Volume Claim for the volume used for storing data for the ZooKeeper node pod ***idx***.

3.1.3.2.3. Log directories

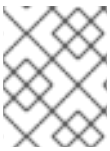
The persistent volume will be used by the Kafka brokers as log directories mounted into the following path:

/var/lib/kafka/data/kafka-log_idx_

Where ***idx*** is the Kafka broker pod index. For example **/var/lib/kafka/data/kafka-log0**.

3.1.3.3. Resizing persistent volumes

You can provision increased storage capacity by increasing the size of the persistent volumes used by an existing AMQ Streams cluster. Resizing persistent volumes is supported in clusters that use either a single persistent volume or multiple persistent volumes in a JBOD storage configuration.



NOTE

You can increase but not decrease the size of persistent volumes. Decreasing the size of persistent volumes is not currently supported in OpenShift.

Prerequisites

- An OpenShift cluster with support for volume resizing.
- The Cluster Operator is running.
- A Kafka cluster using persistent volumes created using a storage class that supports volume expansion.

Procedure

1. In a **Kafka** resource, increase the size of the persistent volume allocated to the Kafka cluster, the ZooKeeper cluster, or both.

   - To increase the volume size allocated to the Kafka cluster, edit the **spec.kafka.storage** property.
   - To increase the volume size allocated to the ZooKeeper cluster, edit the **spec.zookeeper.storage** property.
     For example, to increase the volume size from **1000Gi** to **2000Gi**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: persistent-claim
      size: 2000Gi
      class: my-storage-class
    # ...
  zookeeper:
    # ...
```

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

```
oc apply -f your-file
```

   OpenShift increases the capacity of the selected persistent volumes in response to a request from the Cluster Operator. When the resizing is complete, the Cluster Operator restarts all pods that use the resized persistent volumes. This happens automatically.

Additional resources

For more information about resizing persistent volumes in OpenShift, see Resizing Persistent Volumes using Kubernetes.

3.1.3.4. JBOD storage overview

You can configure AMQ Streams to use JBOD, a data storage configuration of multiple disks or volumes. JBOD is one approach to providing increased data storage for Kafka brokers. It can also improve performance.

A JBOD configuration is described by one or more volumes, each of which can be either ephemeral or persistent. The rules and constraints for JBOD volume declarations are the same as those for ephemeral and persistent storage. For example, you cannot change the size of a persistent storage volume after it has been provisioned.

3.1.3.4.1. JBOD configuration

To use JBOD with AMQ Streams, the storage **type** must be set to **jbod**. The **volumes** property allows you to describe the disks that make up your JBOD storage array or configuration. The following fragment shows an example JBOD configuration:

```
# ...
storage:
  type: jbod
  volumes:
  - id: 0
    type: persistent-claim
    size: 100Gi
    deleteClaim: false
  - id: 1
    type: persistent-claim
    size: 100Gi
    deleteClaim: false
# ...
```

The ids cannot be changed once the JBOD volumes are created.

Users can add or remove volumes from the JBOD configuration.

3.1.3.4.2. JBOD and Persistent Volume Claims

When persistent storage is used to declare JBOD volumes, the naming scheme of the resulting Persistent Volume Claims is as follows:

data-*id*-*cluster-name*-kafka-*idx*

Where ***id*** is the ID of the volume used for storing data for Kafka broker pod ***idx***.

3.1.3.4.3. Log directories

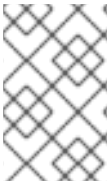
The JBOD volumes will be used by the Kafka brokers as log directories mounted into the following path:

/var/lib/kafka/data-*id*/kafka-log_idx_

Where ***id*** is the ID of the volume used for storing data for Kafka broker pod ***idx***. For example **/var/lib/kafka/data-0/kafka-log0**.

3.1.3.5. Adding volumes to JBOD storage

This procedure describes how to add volumes to a Kafka cluster configured to use JBOD storage. It cannot be applied to Kafka clusters configured to use any other storage type.



NOTE

When adding a new volume under an **id** which was already used in the past and removed, you have to make sure that the previously used **PersistentVolumeClaims** have been deleted.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

- A Kafka cluster with JBOD storage

Procedure

1. Edit the **spec.kafka.storage.volumes** property in the **Kafka** resource. Add the new volumes to the **volumes** array. For example, add the new volume with id **2**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 2
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3. Create new topics or reassign existing partitions to the new disks.

Additional resources

For more information about reassigning topics, see Section 3.1.25.2, “Partition reassignment”.

3.1.3.6. Removing volumes from JBOD storage

This procedure describes how to remove volumes from Kafka cluster configured to use JBOD storage. It cannot be applied to Kafka clusters configured to use any other storage type. The JBOD storage always has to contain at least one volume.

IMPORTANT

To avoid data loss, you have to move all partitions before removing the volumes.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- A Kafka cluster with JBOD storage with two or more volumes

Procedure

1. Reassign all partitions from the disks which are you going to remove. Any data in partitions still assigned to the disks which are going to be removed might be lost.
2. Edit the **spec.kafka.storage.volumes** property in the **Kafka** resource. Remove one or more volumes from the **volumes** array. For example, remove the volumes with ids **1** and **2**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

For more information about reassigning topics, see Section 3.1.25.2, “Partition reassignment”.

3.1.4. Kafka broker replicas

A Kafka cluster can run with many brokers. You can configure the number of brokers used for the Kafka cluster in **Kafka.spec.kafka.replicas**. The best number of brokers for your cluster has to be determined based on your specific use case.

3.1.4.1. Configuring the number of broker nodes

This procedure describes how to configure the number of Kafka broker nodes in a new cluster. It only applies to new clusters with no partitions. If your cluster already has topics defined, see Section 3.1.25, “Scaling clusters”.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- A Kafka cluster with no topics defined yet

Procedure

1. Edit the **replicas** property in the **Kafka** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    replicas: 3
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

If your cluster already has topics defined, see Section 3.1.25, "Scaling clusters".

3.1.5. Kafka broker configuration

AMQ Streams allows you to customize the configuration of the Kafka brokers in your Kafka cluster. You can specify and configure most of the options listed in the "Broker Configs" section of the Apache Kafka documentation. You cannot configure options that are related to the following areas:

- Security (Encryption, Authentication, and Authorization)
- Listener configuration
- Broker ID configuration
- Configuration of log data directories
- Inter-broker communication
- ZooKeeper connectivity

These options are automatically configured by AMQ Streams.

3.1.5.1. Kafka broker configuration

The **config** property in **Kafka.spec.kafka** contains Kafka broker configuration options as keys with values in one of the following JSON types:

- String
- Number
- Boolean

You can specify and configure all of the options in the "Broker Configs" section of the Apache Kafka documentation apart from those managed directly by AMQ Streams. Specifically, you are prevented from modifying all configuration options with keys equal to or starting with one of the following strings:

- **listeners**
- **advertised.**
- **broker.**
- **listener.**
- **host.name**
- **port**
- **inter.broker.listener.name**
- **sasl.**
- **ssl.**
- **security.**
- **password.**
- **principal.builder.class**
- **log.dir**
- **zookeeper.connect**
- **zookeeper.set.acl**
- **authorizer.**
- **super.user**

If the **config** property specifies a restricted option, it is ignored and a warning message is printed to the Cluster Operator log file. All other supported options are passed to Kafka.

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example Kafka broker configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
```

```
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    config:
      num.partitions: 1
      num.recovery.threads.per.data.dir: 1
      default.replication.factor: 3
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 1
      log.retention.hours: 168
      log.segment.bytes: 1073741824
      log.retention.check.interval.ms: 300000
      num.network.threads: 3
      num.io.threads: 8
      socket.send.buffer.bytes: 102400
      socket.receive.buffer.bytes: 102400
      socket.request.max.bytes: 104857600
      group.initial.rebalance.delay.ms: 0
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (1)
      ssl.enabled.protocols: "TLSv1.2" (2)
      ssl.protocol: "TLSv1.2" (3)
    # ...
```

1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

3.1.5.2. Configuring Kafka brokers

You can configure an existing Kafka broker, or create a new Kafka broker with a specified configuration.

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.kafka.config** property in the **Kafka** resource, enter one or more Kafka configuration settings. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
```

```
spec:
  kafka:
    # ...
    config:
      default.replication.factor: 3
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 1
    # ...
  zookeeper:
    # ...
```

3. Apply the new configuration to create or update the resource.
   Use **oc apply**:

   ```
   oc apply -f kafka.yaml
   ```

   where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

3.1.6. Kafka broker listeners

You can configure the listeners enabled in Kafka brokers. The following types of listeners are supported:

- Plain listener on port 9092 (without TLS encryption)
- TLS listener on port 9093 (with TLS encryption)
- External listener on port 9094 for access from outside of OpenShift

OAuth 2.0

If you are using OAuth 2.0 token-based authentication, you can configure the listeners to connect to your authorization server. For more information, see Using OAuth 2.0 token-based authentication .

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 12.8, "Kafka listener certificates".

3.1.6.1. Kafka listeners

You can configure Kafka broker listeners using the **listeners** property in the **Kafka.spec.kafka** resource. The **listeners** property contains three sub-properties:

- **plain**
- **tls**
- **external**

Each listener will only be defined when the **listeners** object has the given property.

An example of listeners property with all listeners enabled

```
# ...
listeners:
  plain: {}
  tls: {}
  external:
    type: loadbalancer
# ...
```

An example of listeners property with only the plain listener enabled

```
# ...
listeners:
  plain: {}
# ...
```

3.1.6.2. Configuring Kafka listeners

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **listeners** property in the **Kafka.spec.kafka** resource.
   An example configuration of the plain (unencrypted) listener without authentication:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       listeners:
         plain: {}
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.
   This can be done using **oc apply**:

   ```
   oc apply -f your-file
   ```

Additional resources

- For more information about the schema, see **KafkaListeners** schema reference.

3.1.6.3. Listener authentication

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

- Mutual TLS authentication (only on the listeners with TLS encryption)
- SCRAM-SHA authentication

If no **authentication** property is specified then the listener does not authenticate clients which connect through that listener.

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

3.1.6.3.1. Authentication configuration for a listener

The following example shows:

- A **plain** listener configured for SCRAM-SHA authentication
- A **tls** listener with mutual TLS authentication
- An **external** listener with mutual TLS authentication

An example showing listener authentication configuration

```
# ...
listeners:
  plain:
    authentication:
      type: scram-sha-512
  tls:
    authentication:
      type: tls
  external:
    type: loadbalancer
    tls: true
    authentication:
      type: tls
# ...
```

3.1.6.3.2. Mutual TLS authentication

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

Mutual authentication or two-way authentication is when both the server and the client present certificates. 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. When you configure mutual authentication, the broker authenticates the client and the client authenticates the broker.

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 server obtains proof of the identity of the browser.

3.1.6.3.2.1. When to use mutual TLS authentication for clients

Mutual TLS authentication is recommended for authenticating Kafka clients when:

- The client supports authentication using mutual TLS authentication
- It is necessary to use the TLS certificates rather than passwords
- You can reconfigure and restart client applications periodically so that they do not use expired certificates.

3.1.6.3.3. SCRAM-SHA 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 TLS-encrypted client connections. TLS authentication is always used internally between Kafka brokers and ZooKeeper nodes. When used with a TLS client connection, the TLS protocol provides encryption, but is not used for authentication.

The following properties of SCRAM make it safe to use SCRAM-SHA 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.

3.1.6.3.3.1. Supported SCRAM credentials

AMQ Streams supports SCRAM-SHA-512 only. When a **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.

3.1.6.3.3.2. When to use SCRAM-SHA authentication for clients

SCRAM-SHA is recommended for authenticating Kafka clients when:

- The client supports authentication using SCRAM-SHA-512
- It is necessary to use passwords rather than the TLS certificates
- Authentication for unencrypted communication is required

3.1.6.4. External listeners

Use an external listener to expose your AMQ Streams Kafka cluster to a client outside an OpenShift environment.

Additional resources

- Accessing Apache Kafka in Strimzi

3.1.6.4.1. Customizing advertised addresses on external listeners

By default, AMQ Streams tries to automatically determine the hostnames and ports that your Kafka

cluster advertises to its clients. This is not sufficient in all situations, because the infrastructure on which AMQ Streams is running might not provide the right hostname or port through which Kafka can be accessed. You can customize the advertised hostname and port in the **overrides** property of the external listener. AMQ Streams will then automatically configure the advertised address in the Kafka brokers and add it to the broker certificates so it can be used for TLS hostname verification. Overriding the advertised host and ports is available for all types of external listeners.

Example of an external listener configured with overrides for advertised addresses

```
# ...
listeners:
  external:
    type: route
    authentication:
      type: tls
    overrides:
      brokers:
      - broker: 0
        advertisedHost: example.hostname.0
        advertisedPort: 12340
      - broker: 1
        advertisedHost: example.hostname.1
        advertisedPort: 12341
      - broker: 2
        advertisedHost: example.hostname.2
        advertisedPort: 12342
# ...
```

Additionally, you can specify the name of the bootstrap service. This name will be added to the broker certificates and can be used for TLS hostname verification. Adding the additional bootstrap address is available for all types of external listeners.

Example of an external listener configured with an additional bootstrap address

```
# ...
listeners:
  external:
    type: route
    authentication:
      type: tls
    overrides:
      bootstrap:
        address: example.hostname
# ...
```

3.1.6.4.2. Route external listeners

An external listener of type **route** exposes Kafka using OpenShift **Routes** and the HAProxy router.

NOTE

route is only supported on OpenShift

3.1.6.4.2.1. Exposing Kafka using OpenShift Routes

When exposing Kafka using OpenShift **Routes** and the HAProxy router, a dedicated **Route** is created for every Kafka broker pod. An additional **Route** is created to serve as a Kafka bootstrap address. Kafka clients can use these **Routes** to connect to Kafka on port 443.

TLS encryption is always used with **Routes**.

By default, the route hosts are automatically assigned by OpenShift. However, you can override the assigned route hosts by specifying the requested hosts in the **overrides** property. AMQ Streams will not perform any validation that the requested hosts are available; you must ensure that they are free and can be used.

Example of an external listener of type routes configured with overrides for OpenShift route hosts

```
# ...
listeners:
  external:
    type: route
    authentication:
      type: tls
    overrides:
      bootstrap:
        host: bootstrap.myrouter.com
      brokers:
      - broker: 0
        host: broker-0.myrouter.com
      - broker: 1
        host: broker-1.myrouter.com
      - broker: 2
        host: broker-2.myrouter.com
# ...
```

For more information on using **Routes** to access Kafka, see Section 3.1.6.4.2.2, “Accessing Kafka using OpenShift routes”.

3.1.6.4.2.2. Accessing Kafka using OpenShift routes

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type **route**.
   An example configuration with an external listener configured to use **Routes**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
```

```
    external:
      type: route
      # ...
  # ...
zookeeper:
  # ...
```

2. Create or update the resource.

```
oc apply -f your-file
```

3. Find the address of the bootstrap **Route**.

```
oc get routes CLUSTER-NAME-kafka-bootstrap -o=jsonpath='{.status.ingress[0].host}{"\n"}'
```

Use the address together with port 443 in your Kafka client as the *bootstrap* address.

4. Extract the public certificate of the broker certification authority

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

Use the extracted certificate in your Kafka client to configure TLS connection. If you enabled any authentication, you will also need to configure SASL or TLS authentication.

Additional resources

- For more information about the schema, see **KafkaListeners** schema reference.

3.1.6.4.3. Loadbalancer external listeners

External listeners of type **loadbalancer** expose Kafka by using **Loadbalancer** type **Services**.

3.1.6.4.3.1. Exposing Kafka using loadbalancers

When exposing Kafka using **Loadbalancer** type **Services**, a new loadbalancer service is created for every Kafka broker pod. An additional loadbalancer is created to serve as a Kafka *bootstrap* address. Loadbalancers listen to connections on port 9094.

By default, TLS encryption is enabled. To disable it, set the **tls** field to **false**.

Example of an external listener of type loadbalancer

```
# ...
listeners:
  external:
    type: loadbalancer
    authentication:
      type: tls
# ...
```

For more information on using loadbalancers to access Kafka, see Section 3.1.6.4.3.4, “Accessing Kafka using loadbalancers”.

3.1.6.4.3.2. Customizing the DNS names of external loadbalancer listeners

On **loadbalancer** listeners, you can use the **dnsAnnotations** property to add additional annotations to the loadbalancer services. You can use these annotations to instrument DNS tooling such as External DNS, which automatically assigns DNS names to the loadbalancer services.

Example of an external listener of type loadbalancer using dnsAnnotations

```
# ...
listeners:
  external:
    type: loadbalancer
    authentication:
      type: tls
    overrides:
      bootstrap:
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-bootstrap.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      brokers:
      - broker: 0
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-0.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 1
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-1.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 2
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-2.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
# ...
```

3.1.6.4.3.3. Customizing the loadbalancer IP addresses

On **loadbalancer** listeners, you can use the **loadBalancerIP** property to request a specific IP address when creating a loadbalancer. Use this property when you need to use a loadbalancer with a specific IP address. The **loadBalancerIP** field is ignored if the cloud provider does not support the feature.

Example of an external listener of type loadbalancer with specific loadbalancer IP address requests

```
# ...
listeners:
  external:
    type: loadbalancer
    authentication:
      type: tls
    overrides:
      bootstrap:
        loadBalancerIP: 172.29.3.10
      brokers:
      - broker: 0
        loadBalancerIP: 172.29.3.1
```

```
    - broker: 1
      loadBalancerIP: 172.29.3.2
    - broker: 2
      loadBalancerIP: 172.29.3.3
# ...
```

3.1.6.4.3.4. Accessing Kafka using loadbalancers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type **loadbalancer**.
   An example configuration with an external listener configured to use loadbalancers:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      external:
        type: loadbalancer
        authentication:
          type: tls
        # ...
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3. Find the hostname of the bootstrap loadbalancer.
   This can be done using **oc get**:

```
oc get service cluster-name-kafka-external-bootstrap -
o=jsonpath='{.status.loadBalancer.ingress[0].hostname}{"\n"}'
```

   If no hostname was found (nothing was returned by the command), use the loadbalancer IP address.

   This can be done using **oc get**:

```
oc get service cluster-name-kafka-external-bootstrap -
o=jsonpath='{.status.loadBalancer.ingress[0].ip}{"\n"}'
```

Use the hostname or IP address together with port 9094 in your Kafka client as the *bootstrap* address.

4. Unless TLS encryption was disabled, extract the public certificate of the broker certification authority.
   This can be done using **oc get**:

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

   Use the extracted certificate in your Kafka client to configure TLS connection. If you enabled any authentication, you will also need to configure SASL or TLS authentication.

Additional resources

- For more information about the schema, see **KafkaListeners** schema reference.

3.1.6.4.4. Node Port external listeners

External listeners of type **nodeport** expose Kafka by using **NodePort** type **Services**.

3.1.6.4.4.1. Exposing Kafka using node ports

When exposing Kafka using **NodePort** type **Services**, Kafka clients connect directly to the nodes of OpenShift. You must enable access to the ports on the OpenShift nodes for each client (for example, in firewalls or security groups). Each Kafka broker pod is then accessible on a separate port.

An additional **NodePort** type of service is created to serve as a Kafka bootstrap address.

When configuring the advertised addresses for the Kafka broker pods, AMQ Streams uses the address of the node on which the given pod is running. Nodes often have multiple addresses. The address type used is based on the first type found in the following order of priority:

1. ExternalDNS
2. ExternalIP
3. Hostname
4. InternalDNS
5. InternalIP

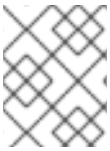
You can use the **preferredAddressType** property in your listener configuration to specify the first address type checked as the node address. This property is useful, for example, if your deployment does not have DNS support, or you only want to expose a broker internally through an internal DNS or IP address. If an address of this type is found, it is used. If the preferred address type is not found, AMQ Streams proceeds through the types in the standard order of priority.

Example of an external listener configured with a preferred address type

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
```

```
    listeners:
      external:
        type: nodeport
        tls: true
        authentication:
          type: tls
        configuration:
          preferredAddressType: InternalDNS
    # ...
  zookeeper:
    # ...
```

By default, TLS encryption is enabled. To disable it, set the **tls** field to **false**.



NOTE

TLS hostname verification is not currently supported when exposing Kafka clusters using node ports.

By default, the port numbers used for the bootstrap and broker services are automatically assigned by OpenShift. However, you can override the assigned node ports by specifying the requested port numbers in the **overrides** property. AMQ Streams does not perform any validation on the requested ports; you must ensure that they are free and available for use.

Example of an external listener configured with overrides for node ports

```
# ...
listeners:
  external:
    type: nodeport
    tls: true
    authentication:
      type: tls
    overrides:
      bootstrap:
        nodePort: 32100
      brokers:
      - broker: 0
        nodePort: 32000
      - broker: 1
        nodePort: 32001
      - broker: 2
        nodePort: 32002
# ...
```

For more information on using node ports to access Kafka, see Section 3.1.6.4.4.3, "Accessing Kafka using node ports".

3.1.6.4.4.2. Customizing the DNS names of external node port listeners

On **nodeport** listeners, you can use the **dnsAnnotations** property to add additional annotations to the nodeport services. You can use these annotations to instrument DNS tooling such as External DNS, which automatically assigns DNS names to the cluster nodes.

Example of an external listener of type nodeport using dnsAnnotations

```
# ...
listeners:
  external:
    type: nodeport
    tls: true
    authentication:
      type: tls
    overrides:
      bootstrap:
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-bootstrap.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      brokers:
      - broker: 0
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-0.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 1
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-1.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 2
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-2.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
# ...
```

3.1.6.4.4.3. Accessing Kafka using node ports

This procedure describes how to access a AMQ Streams Kafka cluster from an external client using node ports.

To connect to a broker, you need the hostname (advertised address) and port number for the Kafka *bootstrap* address, as well as the certificate used for authentication.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Deploy the Kafka cluster with an external listener enabled and configured to the type **nodeport**. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
```

```
    external:
      type: nodeport
      tls: true
      authentication:
        type: tls
      configuration:
        brokerCertChainAndKey: 1
          secretName: my-secret
          certificate: my-certificate.crt
          key: my-key.key
        preferredAddressType: InternalDNS 2
  # ...
zookeeper:
  # ...
```

1 Optional configuration for a Kafka listener certificate managed by an external Certificate Authority. The **brokerCertChainAndKey** property specifies a **Secret** that holds a server certificate and a private key. Kafka listener certificates can also be configured for TLS listeners.

2 Optional configuration to specify a preference for the first address type used by AMQ Streams as the node address.

2. Create or update the resource.

```
oc apply -f your-file
```

3. Find the port number of the bootstrap service.

```
oc get service cluster-name-kafka-external-bootstrap -o=jsonpath='{.spec.ports[0].nodePort}{"\n"}'
```

The port is used in the Kafka *bootstrap* address.

4. Find the address of the OpenShift node.

```
oc get node node-name -o=jsonpath='{range .status.addresses[*]}{.type}{"\t"}{.address}{"\n"}'
```

If several different addresses are returned, select the address type you want based on the following order:

1. ExternalDNS
2. ExternalIP
3. Hostname
4. InternalDNS
5. InternalIP

Use the address with the port found in the previous step in the Kafka *bootstrap* address.

5. Unless TLS encryption was disabled, extract the public certificate of the broker certification authority.

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

Use the extracted certificate in your Kafka client to configure TLS connection. If you enabled any authentication, you will also need to configure SASL or TLS authentication.

Additional resources

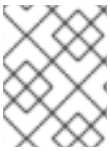
- For more information about the schema, see **KafkaListeners** schema reference.

3.1.6.4.5. OpenShift Ingress external listeners

External listeners of type **ingress** exposes Kafka by using Kubernetes **Ingress** and the NGINX Ingress Controller for Kubernetes.

3.1.6.4.5.1. Exposing Kafka using Kubernetes Ingress

When exposing Kafka using using Kubernetes **Ingress** and the NGINX Ingress Controller for Kubernetes, a dedicated **Ingress** resource is created for every Kafka broker pod. An additional **Ingress** resource is created to serve as a Kafka bootstrap address. Kafka clients can use these **Ingress** resources to connect to Kafka on port 443.



NOTE

External listeners using **Ingress** have been currently tested only with the NGINX Ingress Controller for Kubernetes.

Kafka uses a binary protocol over TCP, but the NGINX Ingress Controller for Kubernetes is designed to work with the HTTP protocol. To be able to pass the Kafka connections through the Ingress, AMQ Streams uses the TLS passthrough feature of the NGINX Ingress Controller for Kubernetes. Make sure TLS passthrough is enabled in your NGINX Ingress Controller for Kubernetes deployment. For more information about enabling TLS passthrough see TLS passthrough documentation. Because it is using the TLS passthrough functionality, TLS encryption cannot be disabled when exposing Kafka using **Ingress**.

The Ingress controller does not assign any hostnames automatically. You have to specify the hostnames which should be used by the bootstrap and per-broker services in the **spec.kafka.listeners.external.configuration** section. You also have to make sure that the hostnames resolve to the Ingress endpoints. AMQ Streams will not perform any validation that the requested hosts are available and properly routed to the Ingress endpoints.

Example of an external listener of type ingress

```
# ...
listeners:
  external:
    type: ingress
    authentication:
      type: tls
    configuration:
      bootstrap:
        host: bootstrap.myingress.com
      brokers:
      - broker: 0
        host: broker-0.myingress.com
```

```
      - broker: 1
        host: broker-1.myingress.com
      - broker: 2
        host: broker-2.myingress.com
# ...
```

For more information on using **Ingress** to access Kafka, see Section 3.1.6.4.5.4, "Accessing Kafka using ingress".

3.1.6.4.5.2. Configuring the Ingress class

By default, the **Ingress** class is set to **nginx**. You can change the **Ingress** class using the **class** property.

Example of an external listener of type ingress using Ingress class nginx-internal

```
# ...
listeners:
  external:
    type: ingress
    class: nginx-internal
    # ...
# ...
```

3.1.6.4.5.3. Customizing the DNS names of external ingress listeners

On **ingress** listeners, you can use the **dnsAnnotations** property to add additional annotations to the ingress resources. You can use these annotations to instrument DNS tooling such as External DNS, which automatically assigns DNS names to the ingress resources.

Example of an external listener of type ingress using dnsAnnotations

```
# ...
listeners:
  external:
    type: ingress
    authentication:
      type: tls
    configuration:
      bootstrap:
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: bootstrap.myingress.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
        host: bootstrap.myingress.com
      brokers:
      - broker: 0
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: broker-0.myingress.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
        host: broker-0.myingress.com
      - broker: 1
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: broker-1.myingress.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
        host: broker-1.myingress.com
```

```
    - broker: 2
      dnsAnnotations:
        external-dns.alpha.kubernetes.io/hostname: broker-2.myingress.com.
        external-dns.alpha.kubernetes.io/ttl: "60"
      host: broker-2.myingress.com
# ...
```

3.1.6.4.5.4. Accessing Kafka using ingress

This procedure shows how to access AMQ Streams Kafka clusters from outside of OpenShift using Ingress.

Prerequisites

- An OpenShift cluster
- Deployed NGINX Ingress Controller for Kubernetes with TLS passthrough enabled
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type **ingress**. An example configuration with an external listener configured to use **Ingress**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      external:
        type: ingress
        authentication:
          type: tls
        configuration:
          bootstrap:
            host: bootstrap.myingress.com
          brokers:
          - broker: 0
            host: broker-0.myingress.com
          - broker: 1
            host: broker-1.myingress.com
          - broker: 2
            host: broker-2.myingress.com
    # ...
  zookeeper:
    # ...
```

2. Make sure the hosts in the **configuration** section properly resolve to the Ingress endpoints.
3. Create or update the resource.

```
oc apply -f your-file
```

4. Extract the public certificate of the broker certificate authority

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

5. Use the extracted certificate in your Kafka client to configure the TLS connection. If you enabled any authentication, you will also need to configure SASL or TLS authentication. Connect with your client to the host you specified in the configuration on port 443.

Additional resources

- For more information about the schema, see **KafkaListeners** schema reference.

3.1.6.5. Network policies

AMQ Streams automatically creates a **NetworkPolicy** resource for every listener that is enabled on a Kafka broker. By default, a **NetworkPolicy** grants access to a listener to all applications and namespaces.

If you want to restrict access to a listener at the network level to only selected applications or namespaces, use the **networkPolicyPeers** field.

Use network policies in conjunction with authentication and authorization.

Each listener can have a different **networkPolicyPeers** configuration.

3.1.6.5.1. Network policy configuration for a listener

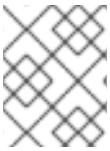
The following example shows a **networkPolicyPeers** configuration for a **plain** and a **tls** listener:

```
# ...
listeners:
  plain:
    authentication:
      type: scram-sha-512
    networkPolicyPeers:
      - podSelector:
          matchLabels:
            app: kafka-sasl-consumer
      - podSelector:
          matchLabels:
            app: kafka-sasl-producer
  tls:
    authentication:
      type: tls
    networkPolicyPeers:
      - namespaceSelector:
          matchLabels:
            project: myproject
      - namespaceSelector:
          matchLabels:
            project: myproject2
# ...
```

In the example:

- Only application pods matching the labels **app: kafka-sasl-consumer** and **app: kafka-sasl-producer** can connect to the **plain** listener. The application pods must be running in the same namespace as the Kafka broker.
- Only application pods running in namespaces matching the labels **project: myproject** and **project: myproject2** can connect to the **tls** listener.

The syntax of the **networkPolicyPeers** field is the same as the **from** field in **NetworkPolicy** resources. For more information about the schema, see NetworkPolicyPeer API reference and the **KafkaListeners** schema reference.



NOTE

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

3.1.6.5.2. Restricting access to Kafka listeners using networkPolicyPeers

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

Prerequisites

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

Procedure

1. Open the **Kafka** resource.
2. In the **networkPolicyPeers** field, 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      tls:
        networkPolicyPeers:
          - podSelector:
              matchLabels:
                app: kafka-client
    # ...
  zookeeper:
    # ...
```

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

```
oc apply -f your-file
```

Additional resources

- For more information about the schema, see NetworkPolicyPeer API reference and the **KafkaListeners** schema reference.

3.1.7. Authentication and Authorization

AMQ Streams supports authentication and authorization. Authentication can be configured independently for each listener. Authorization is always configured for the whole Kafka cluster.

3.1.7.1. Authentication

Authentication is configured as part of the listener configuration in the **authentication** property. The authentication mechanism is defined by the **type** field.

When the **authentication** property is missing, no authentication is enabled on a given listener. The listener will accept all connections without authentication.

Supported authentication mechanisms:

- TLS client authentication
- SASL SCRAM-SHA-512
- OAuth 2.0 token based authentication

3.1.7.1.1. TLS client authentication

TLS Client authentication is enabled by specifying the **type** as **tls**. The TLS client authentication is supported only on the **tls** listener.

An example of authentication with type tls

```
# ...
authentication:
  type: tls
# ...
```

3.1.7.2. Configuring authentication in Kafka brokers

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.kafka.listeners** property in the **Kafka** resource, add the **authentication** field to the listeners for which you want to enable authentication. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      tls:
        authentication:
          type: tls
    # ...
  zookeeper:
    # ...
```

3. Apply the new configuration to create or update the resource.
   Use **oc apply**:

   ```
   oc apply -f kafka.yaml
   ```

   where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

Additional resources

- For more information about the supported authentication mechanisms, see authentication reference.
- For more information about the schema for **Kafka**, see **Kafka** schema reference.

3.1.7.3. Authorization

You can configure authorization for Kafka brokers using the **authorization** property in the **Kafka.spec.kafka** resource. If the **authorization** property is missing, no authorization is enabled. When enabled, authorization is applied to all enabled listeners. The authorization method is defined in the **type** field.

You can configure:

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

3.1.7.3.1. Simple authorization

Simple authorization in AMQ Streams uses the **SimpleAclAuthorizer** plugin, the default Access Control Lists (ACLs) authorization plugin provided with Apache Kafka. ACLs allow you to define which users have access to which resources at a granular level. To enable simple authorization, set the **type** field to **simple**.

An example of Simple authorization

```
# ...
authorization:
```

```
  type: simple
# ...
```

Access rules for users are defined using Access Control Lists (ACLs) . You can optionally designate a list of super users in the **superUsers** field.

3.1.7.3.2. Super users

Super users can access all resources in your Kafka cluster regardless of any access restrictions defined in ACLs. To designate super users for a Kafka cluster, enter a list of user principals in the **superUsers** field. If a user uses TLS Client Authentication, the username will be the common name from their certificate subject prefixed with **CN=**.

An example of designating super users

```
# ...
authorization:
  type: simple
  superUsers:
    - CN=fred
    - sam
    - CN=edward
# ...
```

NOTE

The **super.user** configuration option in the **config** property in **Kafka.spec.kafka** is ignored. Designate super users in the **authorization** property instead. For more information, see Kafka broker configuration .

3.1.7.4. Configuring authorization in Kafka brokers

Configure authorization and designate super users for a particular Kafka broker.

Prerequisites

- An OpenShift cluster
- The Cluster Operator is running

Procedure

1. Add or edit the **authorization** property in the **Kafka.spec.kafka** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    authorization:
      type: simple
      superUsers:
        - CN=fred
        - sam
```

```
      - CN=edward
  # ...
zookeeper:
  # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

- For more information about the supported authorization methods, see authorization reference.
- For more information about the schema for **Kafka**, see **Kafka** schema reference.
- For more information about configuring user authentication, see Kafka User resource.

3.1.8. ZooKeeper replicas

ZooKeeper clusters or ensembles usually run with an odd number of nodes, typically three, five, or seven.

The majority of nodes must be available in order to maintain an effective quorum. If the ZooKeeper cluster loses its quorum, it will stop responding to clients and the Kafka brokers will stop working. Having a stable and highly available ZooKeeper cluster is crucial for AMQ Streams.

Three-node cluster

A three-node ZooKeeper cluster requires at least two nodes to be up and running in order to maintain the quorum. It can tolerate only one node being unavailable.

Five-node cluster

A five-node ZooKeeper cluster requires at least three nodes to be up and running in order to maintain the quorum. It can tolerate two nodes being unavailable.

Seven-node cluster

A seven-node ZooKeeper cluster requires at least four nodes to be up and running in order to maintain the quorum. It can tolerate three nodes being unavailable.

NOTE

For development purposes, it is also possible to run ZooKeeper with a single node.

Having more nodes does not necessarily mean better performance, as the costs to maintain the quorum will rise with the number of nodes in the cluster. Depending on your availability requirements, you can decide for the number of nodes to use.

3.1.8.1. Number of ZooKeeper nodes

The number of ZooKeeper nodes can be configured using the **replicas** property in **Kafka.spec.zookeeper**.

An example showing replicas configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    replicas: 3
    # ...
```

3.1.8.2. Changing the number of ZooKeeper replicas

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.zookeeper.replicas** property in the **Kafka** resource, enter the number of replicated ZooKeeper servers. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    replicas: 3
    # ...
```

3. Apply the new configuration to create or update the resource.
   Use **oc apply**:

```
oc apply -f kafka.yaml
```

   where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

3.1.9. ZooKeeper configuration

AMQ Streams allows you to customize the configuration of Apache ZooKeeper nodes. You can specify and configure most of the options listed in the ZooKeeper documentation.

Options which cannot be configured are those related to the following areas:

- Security (Encryption, Authentication, and Authorization)
- Listener configuration
- Configuration of data directories
- ZooKeeper cluster composition

These options are automatically configured by AMQ Streams.

3.1.9.1. ZooKeeper configuration

ZooKeeper nodes are configured using the **config** property in **Kafka.spec.zookeeper**. This property contains the ZooKeeper configuration options as keys. The values can be described using one of the following JSON types:

- String
- Number
- Boolean

Users can specify and configure the options listed in ZooKeeper documentation with the exception of those options which are managed directly by AMQ Streams. Specifically, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **server.**
- **dataDir**
- **dataLogDir**
- **clientPort**
- **authProvider**
- **quorum.auth**
- **requireClientAuthScheme**

When one of the forbidden options is present in the **config** property, it is ignored and a warning message is printed to the Cluster Operator log file. All other options are passed to ZooKeeper.

IMPORTANT

The Cluster Operator does not validate keys or values in the provided **config** object. When invalid configuration is provided, the ZooKeeper cluster might not start or might become unstable. In such cases, the configuration in the **Kafka.spec.zookeeper.config** object should be fixed and the Cluster Operator will roll out the new configuration to all ZooKeeper nodes.

Selected options have default values:

- **timeTick** with default value **2000**

- **initLimit** with default value **5**
- **syncLimit** with default value **2**
- **autopurge.purgeInterval** with default value **1**

These options will be automatically configured when they are not present in the **Kafka.spec.zookeeper.config** property.

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example ZooKeeper configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (1)
      ssl.enabled.protocols: "TLSv1.2" (2)
      ssl.protocol: "TLSv1.2" (3)
    # ...
```

1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

3.1.9.2. Configuring ZooKeeper

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.zookeeper.config** property in the **Kafka** resource, enter one or more ZooKeeper configuration settings. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
```

```
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
    # ...
```

3. Apply the new configuration to create or update the resource.
   Use **oc apply**:

   ```
   oc apply -f kafka.yaml
   ```

   where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

3.1.10. ZooKeeper connection

ZooKeeper services are secured with encryption and authentication and are not intended to be used by external applications that are not part of AMQ Streams.

However, if you want to use Kafka CLI tools that require a connection to ZooKeeper, such as the **kafka-topics** tool, you can use a terminal inside a Kafka container and connect to the local end of the TLS tunnel to ZooKeeper by using **localhost:2181** as the ZooKeeper address.

3.1.10.1. Connecting to ZooKeeper from a terminal

Open a terminal inside a Kafka container to use Kafka CLI tools that require a ZooKeeper connection.

Prerequisites

- An OpenShift cluster is available.
- A kafka cluster is running.
- The Cluster Operator is running.

Procedure

1. Open the terminal using the OpenShift console or run the **exec** command from your CLI.
   For example:

   ```
   oc exec -it my-cluster-kafka-0 -- bin/kafka-topics.sh --list --zookeeper localhost:2181
   ```

   Be sure to use **localhost:2181**.

   You can now run Kafka commands to ZooKeeper.

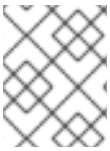
3.1.11. Entity Operator

The Entity Operator is responsible for managing Kafka-related entities in a running Kafka cluster.

The Entity Operator comprises the:

- Topic Operator to manage Kafka topics
- User Operator to manage Kafka users

Through **Kafka** resource configuration, the Cluster Operator can deploy the Entity Operator, including one or both operators, when deploying a Kafka cluster.



NOTE

When deployed, the Entity Operator contains the operators according to the deployment configuration.

The operators are automatically configured to manage the topics and users of the Kafka cluster.

3.1.11.1. Entity Operator configuration properties

Use the **entityOperator** property in **Kafka.spec** to configure the Entity Operator.

The **entityOperator** property supports several sub-properties:

- **tlsSidecar**
- **topicOperator**
- **userOperator**
- **template**

The **tlsSidecar** property contains the configuration of the TLS sidecar container, which is used to communicate with ZooKeeper. For more information on configuring the TLS sidecar, see Section 3.1.20, "TLS sidecar".

The **template** property contains the configuration of the Entity Operator pod, such as labels, annotations, affinity, and tolerations. For more information on configuring templates, see Section 3.9.1, "Template properties".

The **topicOperator** property contains the configuration of the Topic Operator. When this option is missing, the Entity Operator is deployed without the Topic Operator.

The **userOperator** property contains the configuration of the User Operator. When this option is missing, the Entity Operator is deployed without the User Operator.

For more information on the properties to configure the Entity Operator, see the **EntityUserOperatorSpec** schema reference.

Example of basic configuration enabling both operators

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
```

```
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

If an empty object (**{}**) is used for the **topicOperator** and **userOperator**, all properties use their default values.

When both **topicOperator** and **userOperator** properties are missing, the Entity Operator is not deployed.

3.1.11.2. Topic Operator configuration properties

Topic Operator deployment can be configured using additional options inside the **topicOperator** object. The following properties are supported:

watchedNamespace

The OpenShift namespace in which the topic operator watches for **KafkaTopics**. Default is the namespace where the Kafka cluster is deployed.

reconciliationIntervalSeconds

The interval between periodic reconciliations in seconds. Default **90**.

zookeeperSessionTimeoutSeconds

The ZooKeeper session timeout in seconds. Default **20**.

topicMetadataMaxAttempts

The number of attempts at getting topic metadata from Kafka. The time between each attempt is defined as an exponential back-off. Consider increasing this value when topic creation could take more time due to the number of partitions or replicas. Default **6**.

image

The **image** property can be used to configure the container image which will be used. For more details about configuring custom container images, see Section 3.1.19, “Container images”.

resources

The **resources** property configures the amount of resources allocated to the Topic Operator. For more details about resource request and limit configuration, see Section 3.1.12, “CPU and memory resources”.

logging

The **logging** property configures the logging of the Topic Operator. For more details, see Section 3.1.11.4, “Operator loggers”.

Example of Topic Operator configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
```

```
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

3.1.11.3. User Operator configuration properties

User Operator deployment can be configured using additional options inside the **userOperator** object. The following properties are supported:

watchedNamespace

The OpenShift namespace in which the user operator watches for **KafkaUsers**. Default is the namespace where the Kafka cluster is deployed.

reconciliationIntervalSeconds

The interval between periodic reconciliations in seconds. Default **120**.

zookeeperSessionTimeoutSeconds

The ZooKeeper session timeout in seconds. Default **6**.

image

The **image** property can be used to configure the container image which will be used. For more details about configuring custom container images, see Section 3.1.19, “Container images”.

resources

The **resources** property configures the amount of resources allocated to the User Operator. For more details about resource request and limit configuration, see Section 3.1.12, “CPU and memory resources”.

logging

The **logging** property configures the logging of the User Operator. For more details, see Section 3.1.11.4, “Operator loggers”.

Example of User Operator configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-user-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

3.1.11.4. Operator loggers

The Topic Operator and User Operator have a configurable logger:

- **rootLogger.level**

The operators use the Apache **log4j2** logger implementation.

Use the **logging** property in the **Kafka** resource to configure loggers and logger levels.

You can set the log levels by specifying the logger and level directly (inline) or use a custom (external) ConfigMap. If a ConfigMap is used, you set **logging.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j2.properties**.

Here we see examples of **inline** and **external** logging.

Inline logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
    # ...
    userOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
# ...
```

External logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
```

```
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: external
        name: customConfigMap
# ...
```

Additional resources

- Garbage collector (GC) logging can also be enabled (or disabled). For more information about GC logging, see Section 3.1.18.1, "JVM configuration"
- For more information about log levels, see Apache logging services.

3.1.11.5. Configuring the Entity Operator

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **entityOperator** property in the **Kafka** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
    userOperator:
      watchedNamespace: my-user-namespace
      reconciliationIntervalSeconds: 60
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.12. CPU and memory resources

For every deployed container, AMQ Streams allows you to request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

AMQ Streams uses the OpenShift syntax for specifying CPU and memory resources.

3.1.12.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the following resources:

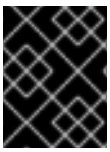
- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

Additional resources

- For more information about managing computing resources on OpenShift, see Managing Compute Resources for Containers.

3.1.12.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.



IMPORTANT

If the resource request is for more than the available free resources in the OpenShift cluster, the pod is not scheduled.

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**
- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

```
# ...
resources:
  requests:
    cpu: 12
    memory: 64Gi
# ...
```

3.1.12.1.2. Resource limits

Limits specify the maximum resources that can be consumed by a given container. The limit is not reserved and might not always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

```
# ...
resources:
  limits:
    cpu: 12
    memory: 64Gi
# ...
```

3.1.12.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

- Number of CPU cores as integer (**5** CPU core) or decimal (**2.5** CPU core).
- Number or *millicpus* / *millicores* (**100m**) where 1000 *millicores* is the same **1** CPU core.

Example CPU units

```
# ...
resources:
  requests:
    cpu: 500m
  limits:
    cpu: 2.5
# ...
```

NOTE

The computing power of 1 CPU core may differ depending on the platform where OpenShift is deployed.

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

3.1.12.1.4. Supported memory formats

Memory requests and limits are specified in megabytes, gigabytes, mebibytes, and gibibytes.

- To specify memory in megabytes, use the **M** suffix. For example **1000M**.
- To specify memory in gigabytes, use the **G** suffix. For example **1G**.
- To specify memory in mebibytes, use the **Mi** suffix. For example **1000Mi**.
- To specify memory in gibibytes, use the **Gi** suffix. For example **1Gi**.

An example of using different memory units

```
# ...
resources:
  requests:
    memory: 512Mi
  limits:
    memory: 2Gi
# ...
```

Additional resources

- For more details about memory specification and additional supported units, see Meaning of memory.

3.1.12.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **resources** property in the resource specifying the cluster deployment. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
```

```
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema, see {K8sResourceRequirementsAPI}.

3.1.13. Kafka loggers

Kafka has its own configurable loggers:

- **kafka.root.logger.level**
- **log4j.logger.org.I0Itec.zkclient.ZkClient**
- **log4j.logger.org.apache.zookeeper**
- **log4j.logger.kafka**
- **log4j.logger.org.apache.kafka**
- **log4j.logger.kafka.request.logger**
- **log4j.logger.kafka.network.Processor**
- **log4j.logger.kafka.server.KafkaApis**
- **log4j.logger.kafka.network.RequestChannel$**
- **log4j.logger.kafka.controller**
- **log4j.logger.kafka.log.LogCleaner**
- **log4j.logger.state.change.logger**
- **log4j.logger.kafka.authorizer.logger**

ZooKeeper also has a configurable logger:

- **zookeeper.root.logger**

Kafka and ZooKeeper use the Apache **log4j** logger implementation.

Use the **logging** property to configure loggers and logger levels.

You can set the log levels by specifying the logger and level directly (inline) or use a custom (external) ConfigMap. If a ConfigMap is used, you set **logging.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**.

Here we see examples of **inline** and **external** logging.

Inline logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # ...
  logging:
    type: inline
    loggers:
      kafka.root.logger.level: "INFO"
  # ...
  zookeeper:
    # ...
    logging:
      type: inline
      loggers:
        zookeeper.root.logger: "INFO"
  # ...
  entityOperator:
    # ...
    topicOperator:
      # ...
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
    # ...
    userOperator:
      # ...
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
    # ...
```

External logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # ...
  logging:
    type: external
    name: customConfigMap
  # ...
```

Operators use the Apache **log4j2** logger implementation, so the logging configuration is described inside the ConfigMap using **log4j2.properties**. For more information, see Section 3.1.11.4, "Operator loggers".

Additional resources

- Garbage collector (GC) logging can also be enabled (or disabled). For more information on garbage collection, see Section 3.1.18.1, "JVM configuration"
- For more information about log levels, see Apache logging services.

3.1.14. Kafka rack awareness

The rack awareness feature in AMQ Streams helps to spread the Kafka broker pods and Kafka topic replicas across different racks. Enabling rack awareness helps to improve availability of Kafka brokers and the topics they are hosting.

NOTE

"Rack" might represent an availability zone, data center, or an actual rack in your data center.

3.1.14.1. Configuring rack awareness in Kafka brokers

Kafka rack awareness can be configured in the **rack** property of **Kafka.spec.kafka**. The **rack** object has one mandatory field named **topologyKey**. This key needs to match one of the labels assigned to the OpenShift cluster nodes. The label is used by OpenShift when scheduling the Kafka broker pods to nodes. If the OpenShift cluster is running on a cloud provider platform, that label should represent the availability zone where the node is running. Usually, the nodes are labeled with **failure-domain.beta.kubernetes.io/zone** that can be easily used as the **topologyKey** value. This has the effect of spreading the broker pods across zones, and also setting the brokers' **broker.rack** configuration parameter inside Kafka broker.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Consult your OpenShift administrator regarding the node label that represents the zone / rack into which the node is deployed.
2. Edit the **rack** property in the **Kafka** resource using the label as the topology key.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
```

```
    rack:
      topologyKey: failure-domain.beta.kubernetes.io/zone
    # ...
```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

- For information about Configuring init container image for Kafka rack awareness, see Section 3.1.19, “Container images”.

3.1.15. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts to fix it.

OpenShift supports two types of Healthcheck probes:

- Liveness probes
- Readiness probes

For more details about the probes, see Configure Liveness and Readiness Probes. Both types of probes are used in AMQ Streams components.

Users can configure selected options for liveness and readiness probes.

3.1.15.1. Healthcheck configurations

Liveness and readiness probes can be configured using the **livenessProbe** and **readinessProbe** properties in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**

- **KafkaBridge.spec**

Both **livenessProbe** and **readinessProbe** support the following options:

- **initialDelaySeconds**
- **timeoutSeconds**
- **periodSeconds**
- **successThreshold**
- **failureThreshold**

For more information about the **livenessProbe** and **readinessProbe** options, see Section B.40, "**Probe** schema reference".

An example of liveness and readiness probe configuration

```
# ...
readinessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
# ...
```

3.1.15.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **livenessProbe** or **readinessProbe** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    readinessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
```

```
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.16. Prometheus metrics

AMQ Streams supports Prometheus metrics using Prometheus JMX exporter to convert the JMX metrics supported by Apache Kafka and ZooKeeper to Prometheus metrics. When metrics are enabled, they are exposed on port 9404.

For more information about setting up and deploying Prometheus and Grafana, see Introducing Metrics to Kafka.

3.1.16.1. Metrics configuration

Prometheus metrics are enabled by configuring the **metrics** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**

When the **metrics** property is not defined in the resource, the Prometheus metrics will be disabled. To enable Prometheus metrics export without any further configuration, you can set it to an empty object (**{}**).

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

The **metrics** property might contain additional configuration for the Prometheus JMX exporter.

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
```

```
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

3.1.16.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **metrics** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.17. JMX Options

AMQ Streams supports obtaining JMX metrics from the Kafka brokers by opening a JMX port on 9999. You can obtain various metrics about each Kafka broker, for example, usage data such as the

BytesPerSecond value or the request rate of the network of the broker. AMQ Streams supports opening a password and username protected JMX port or a non-protected JMX port.

3.1.17.1. Configuring JMX options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

You can configure JMX options by using the **jmxOptions** property in the following resources:

- **Kafka.spec.kafka**

You can configure username and password protection for the JMX port that is opened on the Kafka brokers.

Securing the JMX Port

You can secure the JMX port to prevent unauthorized pods from accessing the port. Currently the JMX port can only be secured using a username and password. To enable security for the JMX port, set the **type** parameter in the **authentication** field to **password**.:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jmxOptions:
      authentication:
        type: "password"
    # ...
  zookeeper:
    # ...
```

This allows you to deploy a pod internally into a cluster and obtain JMX metrics by using the headless service and specifying which broker you want to address. To get JMX metrics from broker *0* we address the headless service appending broker *0* in front of the headless service:

```
"<cluster-name>-kafka-0-<cluster-name>-<headless-service-name>"
```

If the JMX port is secured, you can get the username and password by referencing them from the JMX secret in the deployment of your pod.

Using an open JMX port

To disable security for the JMX port, do not fill in the **authentication** field

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
```

```
kafka:
  # ...
  jmxOptions: {}
  # ...
zookeeper:
  # ...
```

This will just open the JMX Port on the headless service and you can follow a similar approach as described above to deploy a pod into the cluster. The only difference is that any pod will be able to read from the JMX port.

3.1.18. JVM Options

The following components of AMQ Streams run inside a Virtual Machine (VM):

- Apache Kafka
- Apache ZooKeeper
- Apache Kafka Connect
- Apache Kafka MirrorMaker
- AMQ Streams Kafka Bridge

JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure some of these options.

3.1.18.1. JVM configuration

JVM options can be configured using the **jvmOptions** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaBridge.spec**

Only a selected subset of available JVM options can be configured. The following options are supported:

-Xms and -Xmx

-Xms configures the minimum initial allocation heap size when the JVM starts. **-Xmx** configures the maximum heap size.

NOTE

The units accepted by JVM settings such as **-Xmx** and **-Xms** are those accepted by the JDK **java** binary in the corresponding image. Accordingly, **1g** or **1G** means 1,073,741,824 bytes, and **Gi** is not a valid unit suffix. This is in contrast to the units used for memory requests and limits, which follow the OpenShift convention where **1G** means 1,000,000,000 bytes, and **1Gi** means 1,073,741,824 bytes

The default values used for **-Xms** and **-Xmx** depends on whether there is a memory request limit configured for the container:

- If there is a memory limit then the JVM's minimum and maximum memory will be set to a value corresponding to the limit.
- If there is no memory limit then the JVM's minimum memory will be set to **128M** and the JVM's maximum memory will not be defined. This allows for the JVM's memory to grow as-needed, which is ideal for single node environments in test and development.

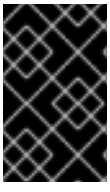
IMPORTANT

Setting **-Xmx** explicitly requires some care:

- The JVM's overall memory usage will be approximately 4 × the maximum heap, as configured by **-Xmx**.
- If **-Xmx** is set without also setting an appropriate OpenShift memory limit, it is possible that the container will be killed should the OpenShift node experience memory pressure (from other Pods running on it).
- If **-Xmx** is set without also setting an appropriate OpenShift memory request, it is possible that the container will be scheduled to a node with insufficient memory. In this case, the container will not start but crash (immediately if **-Xms** is set to **-Xmx**, or some later time if not).

When setting **-Xmx** explicitly, it is recommended to:

- set the memory request and the memory limit to the same value,
- use a memory request that is at least 4.5 × the **-Xmx**,
- consider setting **-Xms** to the same value as **-Xmx**.



IMPORTANT

Containers doing lots of disk I/O (such as Kafka broker containers) will need to leave some memory available for use as operating system page cache. On such containers, the requested memory should be significantly higher than the memory used by the JVM.

Example fragment configuring -Xmx and -Xms

```
# ...
jvmOptions:
  "-Xmx": "2g"
```

```
  "-Xms": "2g"
# ...
```

In the above example, the JVM will use 2 GiB (=2,147,483,648 bytes) for its heap. Its total memory usage will be approximately 8GiB.

Setting the same value for initial (**-Xms**) and maximum (**-Xmx**) heap sizes avoids the JVM having to allocate memory after startup, at the cost of possibly allocating more heap than is really needed. For Kafka and ZooKeeper pods such allocation could cause unwanted latency. For Kafka Connect avoiding over allocation may be the most important concern, especially in distributed mode where the effects of over-allocation will be multiplied by the number of consumers.

-server

-server enables the server JVM. This option can be set to true or false.

Example fragment configuring -server

```
# ...
jvmOptions:
  "-server": true
# ...
```

NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

-XX

-XX object can be used for configuring advanced runtime options of a JVM. The **-server** and **-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

Example showing the use of the -XX object

```
jvmOptions:
  "-XX":
    "UseG1GC": true
    "MaxGCPauseMillis": 20
    "InitiatingHeapOccupancyPercent": 35
    "ExplicitGCInvokesConcurrent": true
    "UseParNewGC": false
```

The example configuration above will result in the following JVM options:

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -
XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```

NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

3.1.18.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is disabled by default. To enable it, set the **gcLoggingEnabled** property as follows:

Example of enabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: true
# ...
```

3.1.18.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **jvmOptions** property in the **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, **KafkaMirrorMaker**, or **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jvmOptions:
      "-Xmx": "8g"
      "-Xms": "8g"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.19. Container images

AMQ Streams allows you to configure container images which will be used for its components. Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such a case, you should either copy the AMQ Streams images or build them from the source. If the configured image is not compatible with AMQ Streams images, it might not work properly.

3.1.19.1. Container image configurations

You can specify which container image to use for each component using the **image** property in the following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

3.1.19.1.1. Configuring the image property for Kafka, Kafka Connect, and Kafka MirrorMaker

Kafka, Kafka Connect (including Kafka Connect with S2I support), and Kafka MirrorMaker support multiple versions of Kafka. Each component requires its own image. The default images for the different Kafka versions are configured in the following environment variables:

- **STRIMZI_KAFKA_IMAGES**
- **STRIMZI_KAFKA_CONNECT_IMAGES**
- **STRIMZI_KAFKA_CONNECT_S2I_IMAGES**
- **STRIMZI_KAFKA_MIRROR_MAKER_IMAGES**

These environment variables contain mappings between the Kafka versions and their corresponding images. The mappings are used together with the **image** and **version** properties:

- If neither **image** nor **version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the environment variable.
- If **image** is given but **version** is not, then the given image is used and the **version** is assumed to be the Cluster Operator's default Kafka version.
- If **version** is given but **image** is not, then the image that corresponds to the given version in the environment variable is used.
- If both **version** and **image** are given, then the given image is used. The image is assumed to contain a Kafka image with the given version.

The **image** and **version** for the different components can be configured in the following properties:

- For Kafka in **spec.kafka.image** and **spec.kafka.version**.

- For Kafka Connect, Kafka Connect S2I, and Kafka MirrorMaker in **spec.image** and **spec.version**.

WARNING

It is recommended to provide only the **version** and leave the **image** property unspecified. This reduces the chance of making a mistake when configuring the custom resource. If you need to change the images used for different versions of Kafka, it is preferable to configure the Cluster Operator's environment variables.

3.1.19.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property is missing, the **image** specified in the Cluster Operator configuration will be used. If the **image** name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Exporter:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Bridge:

1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE** environment variable from the Cluster Operator configuration.
2. **registry.redhat.io/amq7/amq-streams-bridge-rhel7:1.5.0** container image.

- For Kafka broker initializer:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.

WARNING

Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such case, you should either copy the AMQ Streams images or build them from source. In case the configured image is not compatible with AMQ Streams images, it might not work properly.

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.1.19.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **image** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
```

```
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.20. TLS sidecar

A sidecar is a container that runs in a pod but serves a supporting purpose. In AMQ Streams, the TLS sidecar uses TLS to encrypt and decrypt all communication between the various components and ZooKeeper. ZooKeeper does not have native TLS support.

The TLS sidecar is used in:

- Kafka brokers
- ZooKeeper nodes
- Entity Operator

3.1.20.1. TLS sidecar configuration

The TLS sidecar can be configured using the **tlsSidecar** property in:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator**

The TLS sidecar supports the following additional options:

- **image**
- **resources**
- **logLevel**
- **readinessProbe**
- **livenessProbe**

The **resources** property can be used to specify the memory and CPU resources allocated for the TLS sidecar.

The **image** property can be used to configure the container image which will be used. For more details about configuring custom container images, see Section 3.1.19, “Container images”.

The **logLevel** property is used to specify the logging level. Following logging levels are supported:

- emerg
- alert
- crit
- err
- warning
- notice
- info
- debug

The default value is *notice*.

For more information about configuring the **readinessProbe** and **livenessProbe** properties for the healthchecks, see Section 3.1.15.1, “Healthcheck configurations”.

Example of TLS sidecar configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    tlsSidecar:
      image: my-org/my-image:latest
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
      logLevel: debug
      readinessProbe:
        initialDelaySeconds: 15
        timeoutSeconds: 5
      livenessProbe:
        initialDelaySeconds: 15
        timeoutSeconds: 5
    # ...
  zookeeper:
    # ...
```

3.1.20.2. Configuring TLS sidecar

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **tlsSidecar** property in the **Kafka** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    tlsSidecar:
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.21. Configuring pod scheduling

IMPORTANT

When two applications are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.1.21.1. Scheduling pods based on other applications

3.1.21.1.1. Avoid critical applications to share the node

Pod anti-affinity can be used to ensure that critical applications are never scheduled on the same disk. When running Kafka cluster, it is recommended to use pod anti-affinity to ensure that the Kafka brokers do not share the nodes with other workloads like databases.

3.1.21.1.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.1.21.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **affinity** property in the resource specifying the cluster deployment. Use labels to specify the pods which should not be scheduled on the same nodes. The **topologyKey** should be set to **kubernetes.io/hostname** to specify that the selected pods should not be scheduled on nodes with the same hostname. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
```

```
    # ...
zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.21.2. Scheduling pods to specific nodes

3.1.21.2.1. Node scheduling

The OpenShift cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while other might be optimized for storage (fast local SSDs) or network. Using different nodes helps to optimize both costs and performance. To achieve the best possible performance, it is important to allow scheduling of AMQ Streams components to use the right nodes.

OpenShift uses node affinity to schedule workloads onto specific nodes. Node affinity allows you to create a scheduling constraint for the node on which the pod will be scheduled. The constraint is specified as a label selector. You can specify the label using either the built-in node label like **beta.kubernetes.io/instance-type** or custom labels to select the right node.

3.1.21.2.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.1.21.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster

- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   This can be done using **oc label**:

   ```
   oc label node your-node node-type=fast-network
   ```

   Alternatively, some of the existing labels might be reused.

2. Edit the **affinity** property in the resource specifying the cluster deployment. For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
         pod:
           affinity:
             nodeAffinity:
               requiredDuringSchedulingIgnoredDuringExecution:
                 nodeSelectorTerms:
                   - matchExpressions:
                     - key: node-type
                       operator: In
                       values:
                       - fast-network
       # ...
     zookeeper:
       # ...
   ```

3. Create or update the resource.
   This can be done using **oc apply**:

   ```
   oc apply -f your-file
   ```

3.1.21.3. Using dedicated nodes

3.1.21.3.1. Dedicated nodes

Cluster administrators can mark selected OpenShift nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

3.1.21.3.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.1.21.3.3. Tolerations

Tolerations can be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The format of the **tolerations** property follows the OpenShift specification. For more details, see the Kubernetes taints and tolerations .

3.1.21.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated.
2. Make sure there are no workloads scheduled on these nodes.

3. Set the taints on the selected nodes:
   This can be done using **oc adm taint**:

```
oc adm taint node your-node dedicated=Kafka:NoSchedule
```

4. Additionally, add a label to the selected nodes as well.
   This can be done using **oc label**:

```
oc label node your-node dedicated=Kafka
```

5. Edit the **affinity** and **tolerations** properties in the resource specifying the cluster deployment.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
              - matchExpressions:
                - key: dedicated
                  operator: In
                  values:
                  - Kafka
    # ...
  zookeeper:
    # ...
```

6. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.1.22. Kafka Exporter

You can configure the **Kafka** resource to automatically deploy Kafka Exporter in your cluster.

Kafka Exporter extracts data for analysis as Prometheus metrics, primarily data relating to offsets, consumer groups, consumer lag and topics.

For information on Kafka Exporter and why it is important to monitor consumer lag for performance, see Kafka Exporter.

3.1.22.1. Configuring Kafka Exporter

This procedure shows how to configure Kafka Exporter in the **Kafka** resource through **KafkaExporter** properties.

For more information about configuring the **Kafka** resource, see the sample Kafka YAML configuration.

The properties relevant to the Kafka Exporter configuration are shown in this procedure.

You can configure these properties as part of a deployment or redeployment of the Kafka cluster.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **KafkaExporter** properties for the **Kafka** resource.
   The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    image: my-org/my-image: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.

2. Create or update the resource:

```
oc apply -f kafka.yaml
```

What to do next

After configuring and deploying Kafka Exporter, you can enable Grafana to present the Kafka Exporter dashboards.

Additional resources

KafkaExporterTemplate schema reference.

3.1.23. Performing a rolling update of a Kafka cluster

This procedure describes how to manually trigger a rolling update of an existing Kafka cluster by using an OpenShift annotation.

Prerequisites

- A running Kafka cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the **StatefulSet** that controls the Kafka pods you want to manually update. For example, if your Kafka cluster is named *my-cluster*, the corresponding **StatefulSet** is named *my-cluster-kafka*.

2. Annotate the **StatefulSet** resource in OpenShift. For example, using **oc annotate**:

```
oc annotate statefulset cluster-name-kafka 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 **StatefulSet** 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 **StatefulSet**.

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2.1, "Deploying the Cluster Operator".
- For more information about deploying the Kafka cluster, see Section 2.2.2.1, "Deploying the Kafka cluster".

3.1.24. Performing a rolling update of a ZooKeeper cluster

This procedure describes how to manually trigger a rolling update of an existing ZooKeeper cluster by using an OpenShift annotation.

Prerequisites

- A running ZooKeeper cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the **StatefulSet** that controls the ZooKeeper pods you want to manually update.
   For example, if your Kafka cluster is named *my-cluster*, the corresponding **StatefulSet** is named *my-cluster-zookeeper*.
2. Annotate the **StatefulSet** resource in OpenShift. For example, using **oc annotate**:

```
oc annotate statefulset 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 **StatefulSet** 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 **StatefulSet**.

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2.1, "Deploying the Cluster Operator".
- For more information about deploying the ZooKeeper cluster, see Section 2.2.2.1, "Deploying the Kafka cluster".

3.1.25. Scaling clusters

3.1.25.1. Scaling Kafka clusters

3.1.25.1.1. Adding brokers to a cluster

The primary way of increasing throughput for a topic is to increase the number of partitions for that topic. That works because the extra partitions allow the load of the topic to be shared between the different brokers in the cluster. However, in situations where every broker is constrained by a particular resource (typically I/O) using more partitions will not result in increased throughput. Instead, you need to add brokers to the cluster.

When you add an extra broker to the cluster, Kafka does not assign any partitions to it automatically. You must decide which partitions to move from the existing brokers to the new broker.

Once the partitions have been redistributed between all the brokers, the resource utilization of each broker should be reduced.

3.1.25.1.2. Removing brokers from a cluster

Because AMQ Streams uses **StatefulSets** to manage broker pods, you cannot remove *any* pod from the cluster. You can only remove one or more of the highest numbered pods from the cluster. For example, in a cluster of 12 brokers the pods are named ***cluster-name*-kafka-0** up to ***cluster-name*-kafka-11**. If you decide to scale down by one broker, the ***cluster-name*-kafka-11** will be removed.

Before you remove a broker from a cluster, ensure that it is not assigned to any partitions. You should also decide which of the remaining brokers will be responsible for each of the partitions on the broker being decommissioned. Once the broker has no assigned partitions, you can scale the cluster down safely.

3.1.25.2. Partition reassignment

The Topic Operator does not currently support reassigning replicas to different brokers, so it is necessary to connect directly to broker pods to reassign replicas to brokers.

Within a broker pod, the **kafka-reassign-partitions.sh** utility allows you to reassign partitions to different brokers.

It 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 just need to reassign some of the 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 of the partitions in the file have been moved to their intended brokers. If the reassignment is complete, --verify also removes any throttles 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 need to 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.

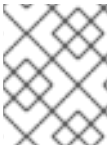
3.1.25.2.1. Reassignment JSON file

The *reassignment JSON file* has a specific structure:

```
{
  "version": 1,
  "partitions": [
    <PartitionObjects>
  ]
}
```

Where *<PartitionObjects>* is a comma-separated list of objects like:

```
{
  "topic": <TopicName>,
  "partition": <Partition>,
  "replicas": [ <AssignedBrokerIds> ]
}
```



NOTE

Although Kafka also supports a **"log_dirs"** property this should not be used in AMQ Streams.

The following is an example reassignment JSON file that assigns topic **topic-a**, partition **4** to brokers **2**, **4** and **7**, and topic **topic-b** partition **2** to brokers **1**, **5** and **7**:

```
{
  "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.

3.1.25.2.2. 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.

For example:

```
{
      "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" ]
}
```

3.1.25.3. Generating reassignment JSON files

This procedure describes how to generate a reassignment JSON file that reassigns all the partitions for a given set of topics using the **kafka-reassign-partitions.sh** tool.

Prerequisites

- A running Cluster Operator
- A **Kafka** resource
- A set of topics to reassign the partitions of

Procedure

1. Prepare a JSON file named ***topics.json*** that lists the topics to move. It must have the following structure:

   ```
   {
     "version": 1,
     "topics": [
       <TopicObjects>
     ]
   }
   ```

   where *<TopicObjects>* is a comma-separated list of objects like:

```
{
  "topic": <TopicName>
}
```

For example if you want to reassign all the partitions of **topic-a** and **topic-b**, you would need to prepare a ***topics.json*** file like this:

```
{
  "version": 1,
  "topics": [
    { "topic": "topic-a"},
    { "topic": "topic-b"}
  ]
}
```

2. Copy the ***topics.json*** file to one of the broker pods:

```
cat topics.json | oc exec -c kafka <BrokerPod> -i -- \
  /bin/bash -c \
  'cat > /tmp/topics.json'
```

3. Use the **kafka-reassign-partitions.sh** command to generate the reassignment JSON.

```
oc exec <BrokerPod> -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --topics-to-move-json-file /tmp/topics.json \
  --broker-list <BrokerList> \
  --generate
```

For example, to move all the partitions of **topic-a** and **topic-b** to brokers **4** and **7**

```
oc exec <BrokerPod> -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --topics-to-move-json-file /tmp/topics.json \
  --broker-list 4,7 \
  --generate
```

3.1.25.4. Creating reassignment JSON files manually

You can manually create the reassignment JSON file if you want to move specific partitions.

3.1.25.5. Reassignment throttles

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. This might cause the reassignment to take longer to complete.

- If the throttle is too low then 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 then clients will be impacted.

For example, for producers, this could manifest as higher than normal latency waiting for acknowledgement. For consumers, this could manifest as a drop in throughput caused by higher latency between polls.

3.1.25.6. Scaling up a Kafka cluster

This procedure describes how to increase the number of brokers in a Kafka cluster.

Prerequisites

- An existing Kafka cluster.
- A *reassignment JSON file* named ***reassignment.json*** that describes how partitions should be reassigned to brokers in the enlarged cluster.

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. Copy the ***reassignment.json*** file to the broker pod on which you will later execute the commands:

   ```
   cat reassignment.json | \
     oc exec broker-pod -c kafka -i -- /bin/bash -c \
     'cat > /tmp/reassignment.json'
   ```

   For example:

   ```
   cat reassignment.json | \
     oc exec my-cluster-kafka-0 -c kafka -i -- /bin/bash -c \
     'cat > /tmp/reassignment.json'
   ```

4. Execute the partition reassignment using the **kafka-reassign-partitions.sh** command line tool from the same broker pod.

   ```
   oc exec broker-pod -c kafka -it -- \
     bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
     --reassignment-json-file /tmp/reassignment.json \
     --execute
   ```

   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:

   ```
   oc exec my-cluster-kafka-0 -c kafka -it -- \
     bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
     --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.

5. If you need to change the throttle during reassignment you can use the same command line with a different throttled rate. For example:

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

6. Periodically verify whether 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.

```
oc exec broker-pod -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

For example,

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

7. The reassignment has finished when the **--verify** command reports each of the partitions being moved as completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

3.1.25.7. Scaling down a Kafka cluster

Additional resources

This procedure describes how to decrease the number of brokers in a Kafka cluster.

Prerequisites

- An existing Kafka cluster.
- A *reassignment JSON file* named ***reassignment.json*** describing how partitions should be reassigned to brokers in the cluster once the broker(s) in the highest numbered **Pod(s)** have been removed.

Procedure

1. Copy the ***reassignment.json*** file to the broker pod on which you will later execute the commands:

```
cat reassignment.json | \
  oc exec broker-pod -c kafka -i -- /bin/bash -c \
  'cat > /tmp/reassignment.json'
```

For example:

```
cat reassignment.json | \
  oc exec my-cluster-kafka-0 -c kafka -i -- /bin/bash -c \
  'cat > /tmp/reassignment.json'
```

2. Execute the partition reassignment using the **kafka-reassign-partitions.sh** command line tool from the same broker pod.

```
oc exec broker-pod -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --execute
```

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:

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --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.

3. If you need to change the throttle during reassignment you can use the same command line with a different throttled rate. For example:

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

4. Periodically verify whether 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.

```
oc exec broker-pod -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

For example,

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
```

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
--reassignment-json-file /tmp/reassignment.json \
--verify
```

5. The reassignment has finished when the **--verify** command reports each of the partitions being moved as completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

6. Once all the partition reassignments have finished, the broker(s) 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$** then the broker still has live partitions and it 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 **Pod(s)** 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.

7. Once you have confirmed that the broker has no live partitions you can edit the **Kafka.spec.kafka.replicas** of your **Kafka** resource, which will scale down the **StatefulSet**, deleting the highest numbered broker **Pod(s)**.

3.1.26. Deleting Kafka nodes manually

Additional resources

This procedure describes how to delete an existing Kafka node by using an OpenShift annotation. Deleting a Kafka node consists of deleting both the **Pod** on which the Kafka broker is running and the related **PersistentVolumeClaim** (if the cluster was deployed with persistent storage). After deletion, the **Pod** and its related **PersistentVolumeClaim** are recreated automatically.

WARNING

Deleting a **PersistentVolumeClaim** can cause permanent data loss. The following procedure should only be performed if you have encountered storage issues.

Prerequisites

- A running Kafka cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the **Pod** that you want to delete.
   For example, if the cluster is named *cluster-name*, the pods are named *cluster-name*-kafka-*index*, where *index* starts at zero and ends at the total number of replicas.

2. Annotate the **Pod** resource in OpenShift.
   Use **oc annotate**:

   ```
   oc annotate pod cluster-name-kafka-index strimzi.io/delete-pod-and-pvc=true
   ```

3. Wait for the next reconciliation, when the annotated pod with the underlying persistent volume claim will be deleted and then recreated.

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2.1, "Deploying the Cluster Operator".
- For more information about deploying the Kafka cluster, see Section 2.2.2.1, "Deploying the Kafka cluster".

3.1.27. Deleting ZooKeeper nodes manually

This procedure describes how to delete an existing ZooKeeper node by using an OpenShift annotation. Deleting a ZooKeeper node consists of deleting both the **Pod** on which ZooKeeper is running and the related **PersistentVolumeClaim** (if the cluster was deployed with persistent storage). After deletion, the **Pod** and its related **PersistentVolumeClaim** are recreated automatically.

WARNING

Deleting a **PersistentVolumeClaim** can cause permanent data loss. The following procedure should only be performed if you have encountered storage issues.

Prerequisites

- A running ZooKeeper cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the **Pod** that you want to delete.
   For example, if the cluster is named *cluster-name*, the pods are named *cluster-name*-zookeeper-*index*, where *index* starts at zero and ends at the total number of replicas.

2. Annotate the **Pod** resource in OpenShift.
   Use **oc annotate**:

   ```
   oc annotate pod cluster-name-zookeeper-index strimzi.io/delete-pod-and-pvc=true
   ```

3. Wait for the next reconciliation, when the annotated pod with the underlying persistent volume claim will be deleted and then recreated.

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2.1, "Deploying the Cluster Operator".
- For more information about deploying the ZooKeeper cluster, see Section 2.2.2.1, "Deploying the Kafka cluster".

3.1.28. 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.

3.1.28.1. 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.

3.1.28.2. 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.

Additional resources

- For more information about the Cluster Operator configuration, see Section 4.1.3, "Cluster Operator Configuration".

3.1.28.3. 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  maintenanceTimeWindows:
    - "* * 8-10 * * ?"
    - "* * 14-15 * * ?"
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

- Performing a rolling update of a Kafka cluster, see Section 3.1.23, "Performing a rolling update of a Kafka cluster"
- Performing a rolling update of a ZooKeeper cluster, see Section 3.1.24, "Performing a rolling update of a ZooKeeper cluster"

3.1.29. Renewing CA certificates manually

Unless the **Kafka.spec.clusterCa.generateCertificateAuthority** and **Kafka.spec.clientsCa.generateCertificateAuthority** objects are set to **false**, the cluster and clients CA certificates will auto-renew at the start of their respective certificate renewal periods. You can manually renew one or both of these certificates before the certificate renewal period starts, if required for security reasons. A renewed certificate uses the same private key as the old certificate.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which CA certificates and private keys are installed.

Procedure

- Apply the **strimzi.io/force-renew** annotation to the **Secret** that contains the CA certificate that you want to renew.

Certificate	Secret	Annotate command
Cluster CA	*<cluster-name>*-cluster-ca-cert	**oc annotate secret *<cluster-name>*-cluster-ca-cert strimzi.io/force-renew=true**
Clients CA	*<cluster-name>*-clients-ca-cert	**oc annotate secret *<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.

Additional resources

- Section 12.2, "Secrets"
- Section 3.1.28, "Maintenance time windows for rolling updates"
- Section B.64, "**CertificateAuthority** schema reference"

3.1.30. Replacing private keys

You can replace the private keys used by the cluster CA and clients CA certificates. When a private key is replaced, the Cluster Operator generates a new CA certificate for the new private key.

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.

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 12.2, "Secrets"
- Section 3.1.28, "Maintenance time windows for rolling updates"

3.1.31. List of resources created as part of Kafka cluster

The following resources will created by the Cluster Operator in the OpenShift cluster:

***cluster-name*-kafka**
: StatefulSet which is in charge of managing the Kafka broker pods.

***cluster-name*-kafka-brokers**
: Service needed to have DNS resolve the Kafka broker pods IP addresses directly.

***cluster-name*-kafka-bootstrap**
: Service can be used as bootstrap servers for Kafka clients.

***cluster-name*-kafka-external-bootstrap**
: Bootstrap service for clients connecting from outside of the OpenShift cluster. This resource will be created only when external listener is enabled.

cluster-name*-kafka-*pod-id

Service used to route traffic from outside of the OpenShift cluster to individual pods. This resource will be created only when external listener is enabled.

***cluster-name*-kafka-external-bootstrap**

Bootstrap route for clients connecting from outside of the OpenShift cluster. This resource will be created only when external listener is enabled and set to type **route**.

cluster-name*-kafka-*pod-id

Route for traffic from outside of the OpenShift cluster to individual pods. This resource will be created only when external listener is enabled and set to type **route**.

***cluster-name*-kafka-config**

ConfigMap which contains the Kafka ancillary configuration and is mounted as a volume by the Kafka broker pods.

***cluster-name*-kafka-brokers**

Secret with Kafka broker keys.

***cluster-name*-kafka**

Service account used by the Kafka brokers.

***cluster-name*-kafka**

Pod Disruption Budget configured for the Kafka brokers.

strimzi-*namespace-name*-*cluster-name*-kafka-init

Cluster role binding used by the Kafka brokers.

***cluster-name*-zookeeper**

StatefulSet which is in charge of managing the ZooKeeper node pods.

***cluster-name*-zookeeper-nodes**

Service needed to have DNS resolve the ZooKeeper pods IP addresses directly.

***cluster-name*-zookeeper-client**

Service used by Kafka brokers to connect to ZooKeeper nodes as clients.

***cluster-name*-zookeeper-config**

ConfigMap which contains the ZooKeeper ancillary configuration and is mounted as a volume by the ZooKeeper node pods.

***cluster-name*-zookeeper-nodes**

Secret with ZooKeeper node keys.

***cluster-name*-zookeeper**

Pod Disruption Budget configured for the ZooKeeper nodes.

***cluster-name*-entity-operator**

Deployment with Topic and User Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-topic-operator-config**

Configmap with ancillary configuration for Topic Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-user-operator-config**

Configmap with ancillary configuration for User Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-operator-certs**

Secret with Entity operators keys for communication with Kafka and ZooKeeper. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-operator**

Service account used by the Entity Operator.

strimzi-*cluster-name*-topic-operator

Role binding used by the Entity Operator.

strimzi-*cluster-name*-user-operator

Role binding used by the Entity Operator.

***cluster-name*-cluster-ca**

Secret with the Cluster CA used to encrypt the cluster communication.

***cluster-name*-cluster-ca-cert**

Secret with the Cluster CA public key. This key can be used to verify the identity of the Kafka brokers.

***cluster-name*-clients-ca**

Secret with the Clients CA used to encrypt the communication between Kafka brokers and Kafka clients.

***cluster-name*-clients-ca-cert**

Secret with the Clients CA public key. This key can be used to verify the identity of the Kafka brokers.

***cluster-name*-cluster-operator-certs**

Secret with Cluster operators keys for communication with Kafka and ZooKeeper.

data-*cluster-name*-kafka-*idx*

Persistent Volume Claim for the volume used for storing data for the Kafka broker pod ***idx***. This resource will be created only if persistent storage is selected for provisioning persistent volumes to store data.

data-*id*-*cluster-name*-kafka-*idx*

Persistent Volume Claim for the volume ***id*** used for storing data for the Kafka broker pod ***idx***. This resource is only created if persistent storage is selected for JBOD volumes when provisioning persistent volumes to store data.

data-*cluster-name*-zookeeper-*idx*

Persistent Volume Claim for the volume used for storing data for the ZooKeeper node pod ***idx***. This resource will be created only if persistent storage is selected for provisioning persistent volumes to store data.

***cluster-name*-jmx**

Secret with JMX username and password used to secure the Kafka broker port.

3.2. KAFKA CONNECT CLUSTER CONFIGURATION

The full schema of the **KafkaConnect** resource is described in the Section B.74, "**KafkaConnect** schema reference". All labels that are applied to the desired **KafkaConnect** resource will also be applied to the OpenShift resources making up the Kafka Connect cluster. This provides a convenient mechanism for resources to be labeled as required.

3.2.1. Replicas

Kafka Connect clusters can consist of one or more nodes. The number of nodes is defined in the **KafkaConnect** and **KafkaConnectS2I** resources. Running a Kafka Connect cluster with multiple nodes

can provide better availability and scalability. However, when running Kafka Connect on OpenShift it is not necessary to run multiple nodes of Kafka Connect for high availability. If a node where Kafka Connect is deployed to crashes, OpenShift will automatically reschedule the Kafka Connect pod to a different node. However, running Kafka Connect with multiple nodes can provide faster failover times, because the other nodes will be up and running already.

3.2.1.1. Configuring the number of nodes

The number of Kafka Connect nodes is configured using the **replicas** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **replicas** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  replicas: 3
  # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.2. Bootstrap servers

A Kafka Connect cluster always works in combination with a Kafka cluster. A Kafka cluster is specified as a list of bootstrap servers. On OpenShift, the list must ideally contain the Kafka cluster bootstrap service named ***cluster-name*-kafka-bootstrap**, and a port of 9092 for plain traffic or 9093 for encrypted traffic.

The list of bootstrap servers is configured in the **bootstrapServers** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The servers must be defined as a comma-separated list specifying one or more Kafka brokers, or a service pointing to Kafka brokers specified as a ***hostname*:_port_** pairs.

When using Kafka Connect with a Kafka cluster not managed by AMQ Streams, you can specify the bootstrap servers list according to the configuration of the cluster.

3.2.2.1. Configuring bootstrap servers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **bootstrapServers** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.3. Connecting to Kafka brokers using TLS

By default, Kafka Connect tries to connect to Kafka brokers using a plain text connection. If you prefer to use TLS, additional configuration is required.

3.2.3.1. TLS support in Kafka Connect

TLS support is configured in the **tls** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The **tls** property contains a list of secrets with key names under which the certificates are stored. The certificates must be stored in X509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-secret
        certificate: ca.crt
      - secretName: my-other-secret
        certificate: certificate.crt
  # ...
```

When multiple certificates are stored in the same secret, it can be listed multiple times.

An example showing TLS configuration with multiple certificates from the same secret

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-secret
        certificate: ca.crt
      - secretName: my-secret
        certificate: ca2.crt
  # ...
```

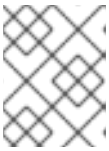
3.2.3.2. Configuring TLS in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** for the certificate used for TLS Server Authentication, and the key under which the certificate is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the TLS certificate used in authentication in a file and create a **Secret**.

   NOTE

   The secrets created by the Cluster Operator for Kafka cluster may be used directly.

   This can be done using **oc create**:

   ```
   oc create secret generic my-secret --from-file=my-file.crt
   ```

2. Edit the **tls** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: KafkaConnect
   metadata:
     name: my-connect
   spec:
     # ...
     tls:
       trustedCertificates:
         - secretName: my-cluster-cluster-cert
           certificate: ca.crt
     # ...
   ```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.4. Connecting to Kafka brokers with Authentication

By default, Kafka Connect will try to connect to Kafka brokers without authentication. Authentication is enabled through the **KafkaConnect** and **KafkaConnectS2I** resources.

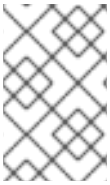
3.2.4.1. Authentication support in Kafka Connect

Authentication is configured through the **authentication** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The **authentication** property specifies the type of the authentication mechanisms which should be used and additional configuration details depending on the mechanism. The supported authentication types are:

- TLS client authentication
- SASL-based authentication using the SCRAM-SHA-512 mechanism
- SASL-based authentication using the PLAIN mechanism
- OAuth 2.0 token based authentication

3.2.4.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.



NOTE

TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Connect see Section 3.2.3, “Connecting to Kafka brokers using TLS”.

An example TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  # ...
```

3.2.4.1.2. SASL based SCRAM-SHA-512 authentication

To configure Kafka Connect to use SASL-based SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. This authentication mechanism requires a username and password.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

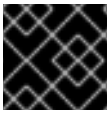
3.2.4.1.3. SASL based PLAIN authentication

To configure Kafka Connect to use SASL-based PLAIN authentication, set the **type** property to **plain**. This authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of such a **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.



IMPORTANT

Do not specify the actual password in the **password** field.

An example showing SASL based PLAIN client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: plain
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

3.2.4.2. Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** with the public and private keys used for TLS Client Authentication, and the keys under which they are stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the keys used for authentication in a file and create the **Secret**.

NOTE

Secrets created by the User Operator may be used.

This can be done using **oc create**:

```
oc create secret generic my-secret --from-file=my-public.crt --from-file=my-private.key
```

2. Edit the **authentication** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
```

```
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: my-public.crt
      key: my-private.key
  # ...
```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

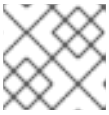
3.2.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- Username of the user which should be used for authentication
- If they exist, the name of the **Secret** with the password used for authentication and the key under which the password is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare a file with the password used in authentication and create the **Secret**.

   NOTE

   Secrets created by the User Operator may be used.

   This can be done using **oc create**:

```
echo -n '<password>' > <my-password.txt>
oc create secret generic <my-secret> --from-file=<my-password.txt>
```

2. Edit the **authentication** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: <my-username>
    passwordSecret:
```

```
      secretName: <my-secret>
      password: <my-password.txt>
  # ...
```

3. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.5. Kafka Connect configuration

AMQ Streams allows you to customize the configuration of Apache Kafka Connect nodes by editing certain options listed in Apache Kafka documentation.

Configuration options that cannot be configured relate to:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Listener / REST interface configuration
- Plugin path configuration

These options are automatically configured by AMQ Streams.

3.2.5.1. Kafka Connect configuration

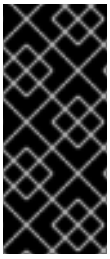
Kafka Connect is configured using the **config** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. This property contains the Kafka Connect configuration options as keys. The values can be one of the following JSON types:

- String
- Number
- Boolean

You can specify and configure the options listed in the Apache Kafka documentation with the exception of those options that are managed directly by AMQ Streams. Specifically, configuration options with keys equal to or starting with one of the following strings are forbidden:

- **ssl.**
- **sasl.**
- **security.**
- **listeners**
- **plugin.path**
- **rest.**
- **bootstrap.servers**

When a forbidden option is present in the **config** property, it is ignored and a warning message is printed to the Cluster Operator log file. All other options are passed to Kafka Connect.



IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Connect cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaConnect.spec.config** or **KafkaConnectS2I.spec.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Connect nodes.

Certain options have default values:

- **group.id** with default value **connect-cluster**
- **offset.storage.topic** with default value **connect-cluster-offsets**
- **config.storage.topic** with default value **connect-cluster-configs**
- **status.storage.topic** with default value **connect-cluster-status**
- **key.converter** with default value **org.apache.kafka.connect.json.JsonConverter**
- **value.converter** with default value **org.apache.kafka.connect.json.JsonConverter**

These options are automatically configured in case they are not present in the **KafkaConnect.spec.config** or **KafkaConnectS2I.spec.config** properties.

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example Kafka Connect configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 1
    ssl.enabled.protocols: "TLSv1.2" 2
    ssl.protocol: "TLSv1.2" 3
  # ...
```

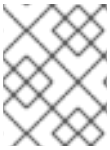
1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

3.2.5.2. Kafka Connect configuration 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/v1beta1
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. Kafka Connect cluster group that the instance belongs to.
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.

3.2.5.3. Configuring Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **config** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3. If authorization is enabled for Kafka Connect, configure the Kafka Connect user to enable access to the Kafka Connect consumer group and topics.

3.2.6. Kafka Connect user authorization

If authorization is enabled for Kafka Connect, the Kafka Connect user must be configured to provide read/write access rights to the Kafka Connect consumer group and internal topics.

The properties for the consumer group and internal topics are automatically configured by AMQ Streams, or they can be specified explicitly in the **spec** for the **KafkaConnect** or **KafkaConnectS2I** configuration.

The following example shows the configuration of the properties in the **KafkaConnect** resource, which need to be represented in the Kafka Connect user configuration.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
```

```
# ...
config:
  group.id: my-connect-cluster 1
  offset.storage.topic: my-connect-cluster-offsets 2
  config.storage.topic: my-connect-cluster-configs 3
  status.storage.topic: my-connect-cluster-status 4
  # ...
# ...
```

1. Kafka Connect cluster group that the instance belongs to.
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.

3.2.6.1. Configuring Kafka Connect user authorization

This procedure describes how to authorize user access to Kafka Connect.

When any type of authorization is being used in Kafka, a Kafka Connect user requires access rights to the consumer group and the internal topics of Kafka Connect.

This procedure shows how access is provided when **simple** authorization is being used.

Simple authorization uses ACL rules, handled by the Kafka **SimpleAclAuthorizer** plugin, to provide the right level of access. For more information on configuring a **KafkaUser** resource to use simple authorization, see Kafka User resource.

NOTE

The default values for the consumer group and topics will differ when running multiple instances.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **authorization** property in the **KafkaUser** resource to provide access rights to the user. In the following example, access rights are configured for the Kafka Connect topics and consumer group using **literal** name values:

Property	Name
offset.storage.topic	**connect-cluster-offsets**
status.storage.topic	**connect-cluster-status**

Property	Name
config.storage.topic	**connect-cluster-configs**
group	**connect-cluster**

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      # access to offset.storage.topic
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Write
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Create
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Describe
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Read
        host: "*"
      # access to status.storage.topic
      - resource:
          type: topic
          name: connect-cluster-status
          patternType: literal
        operation: Write
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-status
          patternType: literal
```

```
    operation: Create
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-status
      patternType: literal
    operation: Describe
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-status
      patternType: literal
    operation: Read
    host: "*"
  # access to config.storage.topic
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Write
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Create
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Describe
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Read
    host: "*"
  # consumer group
  - resource:
      type: group
      name: connect-cluster
      patternType: literal
    operation: Read
    host: "*"
```

2. Create or update the resource.

```
oc apply -f your-file
```

3.2.7. CPU and memory resources

For every deployed container, AMQ Streams allows you to request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

AMQ Streams uses the OpenShift syntax for specifying CPU and memory resources.

3.2.7.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the following resources:

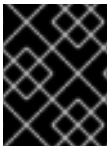
- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

Additional resources

- For more information about managing computing resources on OpenShift, see Managing Compute Resources for Containers.

3.2.7.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.



IMPORTANT

If the resource request is for more than the available free resources in the OpenShift cluster, the pod is not scheduled.

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**
- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

```
# ...
resources:
  requests:
    cpu: 12
    memory: 64Gi
# ...
```

3.2.7.1.2. Resource limits

Limits specify the maximum resources that can be consumed by a given container. The limit is not reserved and might not always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

```
# ...
resources:
  limits:
    cpu: 12
    memory: 64Gi
# ...
```

3.2.7.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

- Number of CPU cores as integer (**5** CPU core) or decimal (**2.5** CPU core).
- Number or *millicpus* / *millicores* (**100m**) where 1000 *millicores* is the same **1** CPU core.

Example CPU units

```
# ...
resources:
  requests:
    cpu: 500m
  limits:
    cpu: 2.5
# ...
```

NOTE

The computing power of 1 CPU core may differ depending on the platform where OpenShift is deployed.

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

3.2.7.1.4. Supported memory formats

Memory requests and limits are specified in megabytes, gigabytes, mebibytes, and gibibytes.

- To specify memory in megabytes, use the **M** suffix. For example **1000M**.
- To specify memory in gigabytes, use the **G** suffix. For example **1G**.
- To specify memory in mebibytes, use the **Mi** suffix. For example **1000Mi**.
- To specify memory in gibibytes, use the **Gi** suffix. For example **1Gi**.

An example of using different memory units

```
# ...
resources:
  requests:
    memory: 512Mi
  limits:
    memory: 2Gi
# ...
```

Additional resources

- For more details about memory specification and additional supported units, see Meaning of memory.

3.2.7.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **resources** property in the resource specifying the cluster deployment. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
```

```
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema, see {K8sResourceRequirementsAPI}.

3.2.8. Kafka Connect loggers

Kafka Connect has its own configurable loggers:

- **connect.root.logger.level**
- **log4j.logger.org.reflections**

Kafka Connect uses the Apache **log4j** logger implementation.

Use the **logging** property to configure loggers and logger levels.

You can set the log levels by specifying the logger and level directly (inline) or use a custom (external) ConfigMap. If a ConfigMap is used, you set **logging.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**.

Here we see examples of **inline** and **external** logging.

Inline logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
spec:
  # ...
  logging:
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  # ...
```

External logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
```

```
spec:
  # ...
  logging:
    type: external
    name: customConfigMap
  # ...
```

Additional resources

- Garbage collector (GC) logging can also be enabled (or disabled). For more information about GC logging, see Section 3.1.18.1, "JVM configuration"
- For more information about log levels, see Apache logging services.

3.2.9. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts to fix it.

OpenShift supports two types of Healthcheck probes:

- Liveness probes
- Readiness probes

For more details about the probes, see Configure Liveness and Readiness Probes. Both types of probes are used in AMQ Streams components.

Users can configure selected options for liveness and readiness probes.

3.2.9.1. Healthcheck configurations

Liveness and readiness probes can be configured using the **livenessProbe** and **readinessProbe** properties in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaBridge.spec**

Both **livenessProbe** and **readinessProbe** support the following options:

- **initialDelaySeconds**
- **timeoutSeconds**
- **periodSeconds**
- **successThreshold**
- **failureThreshold**

For more information about the **livenessProbe** and **readinessProbe** options, see Section B.40, "**Probe** schema reference".

An example of liveness and readiness probe configuration

```
# ...
readinessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
# ...
```

3.2.9.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **livenessProbe** or **readinessProbe** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    readinessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.10. Prometheus metrics

AMQ Streams supports Prometheus metrics using Prometheus JMX exporter to convert the JMX metrics supported by Apache Kafka and ZooKeeper to Prometheus metrics. When metrics are enabled, they are exposed on port 9404.

For more information about setting up and deploying Prometheus and Grafana, see Introducing Metrics to Kafka.

3.2.10.1. Metrics configuration

Prometheus metrics are enabled by configuring the **metrics** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**

When the **metrics** property is not defined in the resource, the Prometheus metrics will be disabled. To enable Prometheus metrics export without any further configuration, you can set it to an empty object (**{}**).

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

The **metrics** property might contain additional configuration for the Prometheus JMX exporter.

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
```

```
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

3.2.10.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **metrics** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.11. JVM Options

The following components of AMQ Streams run inside a Virtual Machine (VM):

- Apache Kafka
- Apache ZooKeeper
- Apache Kafka Connect

- Apache Kafka MirrorMaker
- AMQ Streams Kafka Bridge

JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure some of these options.

3.2.11.1. JVM configuration

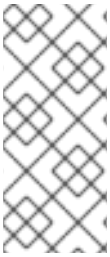
JVM options can be configured using the **jvmOptions** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaBridge.spec**

Only a selected subset of available JVM options can be configured. The following options are supported:

-Xms and -Xmx

-Xms configures the minimum initial allocation heap size when the JVM starts. **-Xmx** configures the maximum heap size.



NOTE

The units accepted by JVM settings such as **-Xmx** and **-Xms** are those accepted by the JDK **java** binary in the corresponding image. Accordingly, **1g** or **1G** means 1,073,741,824 bytes, and **Gi** is not a valid unit suffix. This is in contrast to the units used for memory requests and limits, which follow the OpenShift convention where **1G** means 1,000,000,000 bytes, and **1Gi** means 1,073,741,824 bytes

The default values used for **-Xms** and **-Xmx** depends on whether there is a memory request limit configured for the container:

- If there is a memory limit then the JVM's minimum and maximum memory will be set to a value corresponding to the limit.
- If there is no memory limit then the JVM's minimum memory will be set to **128M** and the JVM's maximum memory will not be defined. This allows for the JVM's memory to grow as-needed, which is ideal for single node environments in test and development.

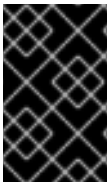
IMPORTANT

Setting **-Xmx** explicitly requires some care:

- The JVM's overall memory usage will be approximately 4 × the maximum heap, as configured by **-Xmx**.
- If **-Xmx** is set without also setting an appropriate OpenShift memory limit, it is possible that the container will be killed should the OpenShift node experience memory pressure (from other Pods running on it).
- If **-Xmx** is set without also setting an appropriate OpenShift memory request, it is possible that the container will be scheduled to a node with insufficient memory. In this case, the container will not start but crash (immediately if **-Xms** is set to **-Xmx**, or some later time if not).

When setting **-Xmx** explicitly, it is recommended to:

- set the memory request and the memory limit to the same value,
- use a memory request that is at least 4.5 × the **-Xmx**,
- consider setting **-Xms** to the same value as **-Xmx**.



IMPORTANT

Containers doing lots of disk I/O (such as Kafka broker containers) will need to leave some memory available for use as operating system page cache. On such containers, the requested memory should be significantly higher than the memory used by the JVM.

Example fragment configuring -Xmx and -Xms

```
# ...
jvmOptions:
  "-Xmx": "2g"
  "-Xms": "2g"
# ...
```

In the above example, the JVM will use 2 GiB (=2,147,483,648 bytes) for its heap. Its total memory usage will be approximately 8GiB.

Setting the same value for initial (**-Xms**) and maximum (**-Xmx**) heap sizes avoids the JVM having to allocate memory after startup, at the cost of possibly allocating more heap than is really needed. For Kafka and ZooKeeper pods such allocation could cause unwanted latency. For Kafka Connect avoiding over allocation may be the most important concern, especially in distributed mode where the effects of over-allocation will be multiplied by the number of consumers.

-server

-server enables the server JVM. This option can be set to true or false.

Example fragment configuring -server

```
# ...
jvmOptions:
```

```
  "-server": true
# ...
```

NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

-XX

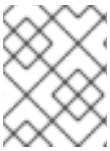
-XX object can be used for configuring advanced runtime options of a JVM. The **-server** and **-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

Example showing the use of the **-XX** object

```
jvmOptions:
  "-XX":
    "UseG1GC": true
    "MaxGCPauseMillis": 20
    "InitiatingHeapOccupancyPercent": 35
    "ExplicitGCInvokesConcurrent": true
    "UseParNewGC": false
```

The example configuration above will result in the following JVM options:

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -
XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```



NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

3.2.11.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is disabled by default. To enable it, set the **gcLoggingEnabled** property as follows:

Example of enabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: true
# ...
```

3.2.11.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **jvmOptions** property in the **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, **KafkaMirrorMaker**, or **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jvmOptions:
      "-Xmx": "8g"
      "-Xms": "8g"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.12. Container images

AMQ Streams allows you to configure container images which will be used for its components. Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such a case, you should either copy the AMQ Streams images or build them from the source. If the configured image is not compatible with AMQ Streams images, it might not work properly.

3.2.12.1. Container image configurations

You can specify which container image to use for each component using the **image** property in the following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

3.2.12.1.1. Configuring the image property for Kafka, Kafka Connect, and Kafka MirrorMaker

Kafka, Kafka Connect (including Kafka Connect with S2I support), and Kafka MirrorMaker support multiple versions of Kafka. Each component requires its own image. The default images for the different Kafka versions are configured in the following environment variables:

- **STRIMZI_KAFKA_IMAGES**
- **STRIMZI_KAFKA_CONNECT_IMAGES**
- **STRIMZI_KAFKA_CONNECT_S2I_IMAGES**
- **STRIMZI_KAFKA_MIRROR_MAKER_IMAGES**

These environment variables contain mappings between the Kafka versions and their corresponding images. The mappings are used together with the **image** and **version** properties:

- If neither **image** nor **version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the environment variable.
- If **image** is given but **version** is not, then the given image is used and the **version** is assumed to be the Cluster Operator's default Kafka version.
- If **version** is given but **image** is not, then the image that corresponds to the given version in the environment variable is used.
- If both **version** and **image** are given, then the given image is used. The image is assumed to contain a Kafka image with the given version.

The **image** and **version** for the different components can be configured in the following properties:

- For Kafka in **spec.kafka.image** and **spec.kafka.version**.
- For Kafka Connect, Kafka Connect S2I, and Kafka MirrorMaker in **spec.image** and **spec.version**.

WARNING

It is recommended to provide only the **version** and leave the **image** property unspecified. This reduces the chance of making a mistake when configuring the custom resource. If you need to change the images used for different versions of Kafka, it is preferable to configure the Cluster Operator's environment variables.

3.2.12.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property is missing, the **image** specified in the Cluster Operator configuration will be used. If the **image** name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka broker TLS sidecar:

1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.

- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Exporter:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Bridge:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-bridge-rhel7:1.5.0** container image.
- For Kafka broker initializer:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.

WARNING

Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such case, you should either copy the AMQ Streams images or build them from source. In case the configured image is not compatible with AMQ Streams images, it might not work properly.

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.2.12.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **image** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

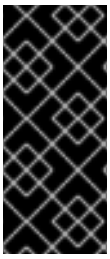
```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.13. Configuring pod scheduling



IMPORTANT

When two applications are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.2.13.1. Scheduling pods based on other applications

3.2.13.1.1. Avoid critical applications to share the node

Pod anti-affinity can be used to ensure that critical applications are never scheduled on the same disk. When running Kafka cluster, it is recommended to use pod anti-affinity to ensure that the Kafka brokers do not share the nodes with other workloads like databases.

3.2.13.1.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.2.13.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **affinity** property in the resource specifying the cluster deployment. Use labels to specify the pods which should not be scheduled on the same nodes. The **topologyKey** should be set to **kubernetes.io/hostname** to specify that the selected pods should not be scheduled on nodes with the same hostname. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.13.2. Scheduling pods to specific nodes

3.2.13.2.1. Node scheduling

The OpenShift cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while other might be optimized for storage (fast local SSDs) or network. Using different nodes helps to optimize both costs and performance. To achieve the best possible performance, it is important to allow scheduling of AMQ Streams components to use the right nodes.

OpenShift uses node affinity to schedule workloads onto specific nodes. Node affinity allows you to create a scheduling constraint for the node on which the pod will be scheduled. The constraint is specified as a label selector. You can specify the label using either the built-in node label like **beta.kubernetes.io/instance-type** or custom labels to select the right node.

3.2.13.2.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**

- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.2.13.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   This can be done using **oc label**:

   ```
   oc label node your-node node-type=fast-network
   ```

   Alternatively, some of the existing labels might be reused.

2. Edit the **affinity** property in the resource specifying the cluster deployment. For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
         pod:
           affinity:
             nodeAffinity:
               requiredDuringSchedulingIgnoredDuringExecution:
                 nodeSelectorTerms:
                   - matchExpressions:
                     - key: node-type
                       operator: In
                       values:
                       - fast-network
   ```

```
    # ...
zookeeper:
    # ...
```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.13.3. Using dedicated nodes

3.2.13.3.1. Dedicated nodes

Cluster administrators can mark selected OpenShift nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

3.2.13.3.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.2.13.3.3. Tolerations

Tolerations can be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka.template.pod**

- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The format of the **tolerations** property follows the OpenShift specification. For more details, see the Kubernetes taints and tolerations .

3.2.13.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated.
2. Make sure there are no workloads scheduled on these nodes.
3. Set the taints on the selected nodes:
   This can be done using **oc adm taint**:

```
oc adm taint node your-node dedicated=Kafka:NoSchedule
```

4. Additionally, add a label to the selected nodes as well.
   This can be done using **oc label**:

```
oc label node your-node dedicated=Kafka
```

5. Edit the **affinity** and **tolerations** properties in the resource specifying the cluster deployment.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
```

```
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: dedicated
                operator: In
                values:
                - Kafka
  # ...
zookeeper:
  # ...
```

6. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.2.14. Using external configuration and secrets

Connectors are created, reconfigured, and deleted using the Kafka Connect HTTP REST interface, or by using **KafkaConnectors**. For more information on these methods, see Section 2.3.3, "Creating and managing connectors". The connector configuration is passed to Kafka Connect as part of an HTTP request and stored within Kafka itself.

ConfigMaps and Secrets are standard OpenShift resources used for storing configurations and confidential data. Whichever method you use to manage connectors, you can use ConfigMaps and Secrets to configure certain elements of a connector. You can then reference the configuration values in HTTP REST commands (this keeps the configuration separate and more secure, if needed). This method applies especially to confidential data, such as usernames, passwords, or certificates.

3.2.14.1. Storing connector configurations externally

You can mount ConfigMaps or Secrets into a Kafka Connect pod as volumes or environment variables. Volumes and environment variables are configured in the **externalConfiguration** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**.

3.2.14.1.1. External configuration as environment variables

The **env** property is used to specify one or more environment variables. These variables can contain a value from either a ConfigMap or a Secret.

NOTE

The names of user-defined environment variables cannot start with **KAFKA_** or **STRIMZI_**.

To mount a value from a Secret to an environment variable, use the **valueFrom** property and the **secretKeyRef** as shown in the following example.

Example of an environment variable set to a value from a Secret

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
```

```
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: my-key
```

A common use case for mounting Secrets to environment variables is when your connector needs to communicate with Amazon AWS and needs to read the **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY** environment variables with credentials.

To mount a value from a ConfigMap to an environment variable, use **configMapKeyRef** in the **valueFrom** property as shown in the following example.

Example of an environment variable set to a value from a ConfigMap

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

3.2.14.1.2. External configuration as volumes

You can also mount ConfigMaps or Secrets to a Kafka Connect pod as volumes. Using volumes instead of environment variables is useful in the following scenarios:

- Mounting truststores or keystores with TLS certificates
- Mounting a properties file that is used to configure Kafka Connect connectors

In the **volumes** property of the **externalConfiguration** resource, list the ConfigMaps or Secrets that will be mounted as volumes. Each volume must specify a name in the **name** property and a reference to ConfigMap or Secret.

Example of volumes with external configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
```

```
externalConfiguration:
  volumes:
    - name: connector1
      configMap:
        name: connector1-configuration
    - name: connector1-certificates
      secret:
        secretName: connector1-certificates
```

The volumes will be mounted inside the Kafka Connect containers in the path **/opt/kafka/external-configuration/*<volume-name>***. For example, the files from a volume named **connector1** would appear in the directory **/opt/kafka/external-configuration/connector1**.

The **FileConfigProvider** has to be used to read the values from the mounted properties files in connector configurations.

3.2.14.2. Mounting Secrets as environment variables

You can create an OpenShift Secret and mount it to Kafka Connect as an environment variable.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing the information that will be mounted as an environment variable. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-creds
type: Opaque
data:
  awsAccessKey: QUtJQVhYWFhYWFhYWFhYWFg=
  awsSecretAccessKey: Ylhsd1IYTnpkMjI5WkE=
```

2. Create or edit the Kafka Connect resource. Configure the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: AWS_ACCESS_KEY_ID
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsAccessKey
      - name: AWS_SECRET_ACCESS_KEY
```

```
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: awsSecretAccessKey
```

3. Apply the changes to your Kafka Connect deployment.
   Use **oc apply**:

```
oc apply -f your-file
```

The environment variables are now available for use when developing your connectors.

Additional resources

- For more information about external configuration in Kafka Connect, see Section B.84, "**ExternalConfiguration** schema reference".

3.2.14.3. Mounting Secrets as volumes

You can create an OpenShift Secret, mount it as a volume to Kafka Connect, and then use it to configure a Kafka Connect connector.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing a properties file that defines the configuration options for your connector configuration. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |-
    dbUsername: my-user
    dbPassword: my-password
```

2. Create or edit the Kafka Connect resource. Configure the **FileConfigProvider** in the **config** section and the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: file
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
```

```
#...
externalConfiguration:
  volumes:
    - name: connector-config
      secret:
        secretName: mysecret
```

3. Apply the changes to your Kafka Connect deployment.
Use **oc apply**:

```
oc apply -f your-file
```

4. Configure your connector

- If you are using the Kafka Connect HTTP REST interface, use the values from the mounted properties file in your JSON payload with connector configuration. For example:

```
{
  "name":"my-connector",
  "config":{
    "connector.class":"MyDbConnector",
    "tasks.max":"3",
    "database": "my-postgresql:5432",
    "username":"${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbUsername}",
    "password":"${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbPassword}",
    # ...
  }
}
```

- If you are using a **KafkaConnector** resource, use the values from the mounted properties file in the **spec.config** section of your custom resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: my-connector
  # ...
spec:
  class: "MyDbConnector"
  tasksMax: 3
  config:
    database: "my-postgresql:5432"
    username: "${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbUsername}"
    password: "${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbPassword}"
```

Additional resources

- For more information about external configuration in Kafka Connect, see Section B.84, "**ExternalConfiguration** schema reference".

3.2.15. Enabling KafkaConnector resources

To enable **KafkaConnectors** for a Kafka Connect cluster, add the **strimzi.io/use-connector-resources** annotation to the **KafkaConnect** or **KafkaConnectS2I** custom resource.

Prerequisites

- A running Cluster Operator

Procedure

1. Edit the **KafkaConnect** or **KafkaConnectS2I** resource. Add the **strimzi.io/use-connector-resources** annotation. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
```

2. Create or update the resource using **oc apply**:

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

Additional resources

- Section 2.3.3, "Creating and managing connectors"
- Section 2.3.4, "Deploying a **KafkaConnector** resource to Kafka Connect"
- Section B.74, "**KafkaConnect** schema reference"
- Section B.90, "**KafkaConnectS2I** schema reference"

3.2.16. List of resources created as part of Kafka Connect cluster

The following resources will created by the Cluster Operator in the OpenShift cluster:

***connect-cluster-name*-connect**

Deployment which is in charge to create the Kafka Connect worker node pods.

***connect-cluster-name*-connect-api**

Service which exposes the REST interface for managing the Kafka Connect cluster.

***connect-cluster-name*-config**

ConfigMap which contains the Kafka Connect ancillary configuration and is mounted as a volume by the Kafka broker pods.

***connect-cluster-name*-connect**

Pod Disruption Budget configured for the Kafka Connect worker nodes.

3.3. KAFKA CONNECT CLUSTER CONFIGURATION WITH SOURCE2IMAGE SUPPORT

The full schema of the **KafkaConnectS2I** resource is described in the Section B.90, "**KafkaConnectS2I** schema reference". All labels that are applied to the desired **KafkaConnectS2I** resource will also be applied to the OpenShift resources making up the Kafka Connect cluster with Source2Image support. This provides a convenient mechanism for resources to be labeled as required.

3.3.1. Replicas

Kafka Connect clusters can consist of one or more nodes. The number of nodes is defined in the **KafkaConnect** and **KafkaConnectS2I** resources. Running a Kafka Connect cluster with multiple nodes can provide better availability and scalability. However, when running Kafka Connect on OpenShift it is not necessary to run multiple nodes of Kafka Connect for high availability. If a node where Kafka Connect is deployed to crashes, OpenShift will automatically reschedule the Kafka Connect pod to a different node. However, running Kafka Connect with multiple nodes can provide faster failover times, because the other nodes will be up and running already.

3.3.1.1. Configuring the number of nodes

The number of Kafka Connect nodes is configured using the **replicas** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **replicas** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  replicas: 3
  # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.2. Bootstrap servers

A Kafka Connect cluster always works in combination with a Kafka cluster. A Kafka cluster is specified as a list of bootstrap servers. On OpenShift, the list must ideally contain the Kafka cluster bootstrap service named ***cluster-name*-kafka-bootstrap**, and a port of 9092 for plain traffic or 9093 for encrypted traffic.

The list of bootstrap servers is configured in the **bootstrapServers** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The servers must be defined as a comma-separated list specifying one or more Kafka brokers, or a service pointing to Kafka brokers specified as a ***hostname*:_port_** pairs.

When using Kafka Connect with a Kafka cluster not managed by AMQ Streams, you can specify the bootstrap servers list according to the configuration of the cluster.

3.3.2.1. Configuring bootstrap servers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **bootstrapServers** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.3. Connecting to Kafka brokers using TLS

By default, Kafka Connect tries to connect to Kafka brokers using a plain text connection. If you prefer to use TLS, additional configuration is required.

3.3.3.1. TLS support in Kafka Connect

TLS support is configured in the **tls** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The **tls** property contains a list of secrets with key names under which the certificates are stored. The certificates must be stored in X509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
```

```
tls:
  trustedCertificates:
    - secretName: my-secret
      certificate: ca.crt
    - secretName: my-other-secret
      certificate: certificate.crt
# ...
```

When multiple certificates are stored in the same secret, it can be listed multiple times.

An example showing TLS configuration with multiple certificates from the same secret

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-secret
        certificate: ca.crt
      - secretName: my-secret
        certificate: ca2.crt
  # ...
```

3.3.3.2. Configuring TLS in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** for the certificate used for TLS Server Authentication, and the key under which the certificate is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the TLS certificate used in authentication in a file and create a **Secret**.

   NOTE

   The secrets created by the Cluster Operator for Kafka cluster may be used directly.

   This can be done using **oc create**:

   ```
   oc create secret generic my-secret --from-file=my-file.crt
   ```

2. Edit the **tls** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        certificate: ca.crt
  # ...
```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.4. Connecting to Kafka brokers with Authentication

By default, Kafka Connect will try to connect to Kafka brokers without authentication. Authentication is enabled through the **KafkaConnect** and **KafkaConnectS2I** resources.

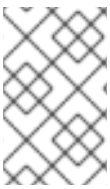
3.3.4.1. Authentication support in Kafka Connect

Authentication is configured through the **authentication** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The **authentication** property specifies the type of the authentication mechanisms which should be used and additional configuration details depending on the mechanism. The supported authentication types are:

- TLS client authentication
- SASL-based authentication using the SCRAM-SHA-512 mechanism
- SASL-based authentication using the PLAIN mechanism
- OAuth 2.0 token based authentication

3.3.4.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.



NOTE

TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Connect see Section 3.3.3, “Connecting to Kafka brokers using TLS”.

An example TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
```

```
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  # ...
```

3.3.4.1.2. SASL based SCRAM-SHA-512 authentication

To configure Kafka Connect to use SASL-based SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. This authentication mechanism requires a username and password.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

3.3.4.1.3. SASL based PLAIN authentication

To configure Kafka Connect to use SASL-based PLAIN authentication, set the **type** property to **plain**. This authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of such a **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example showing SASL based PLAIN client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: plain
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

3.3.4.2. Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** with the public and private keys used for TLS Client Authentication, and the keys under which they are stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the keys used for authentication in a file and create the **Secret**.

NOTE

Secrets created by the User Operator may be used.

This can be done using **oc create**:

```
oc create secret generic my-secret --from-file=my-public.crt --from-file=my-private.key
```

2. Edit the **authentication** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: my-public.crt
      key: my-private.key
  # ...
```

3. Create or update the resource.
This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- Username of the user which should be used for authentication
- If they exist, the name of the **Secret** with the password used for authentication and the key under which the password is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare a file with the password used in authentication and create the **Secret**.

NOTE

Secrets created by the User Operator may be used.

This can be done using **oc create**:

```
echo -n '<password>' > <my-password.txt>
oc create secret generic <my-secret> --from-file=<my-password.txt>
```

2. Edit the **authentication** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: <my-username>
    passwordSecret:
      secretName: <my-secret>
      password: <my-password.txt>
  # ...
```

3. Create or update the resource.
On OpenShift this can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.5. Kafka Connect configuration

AMQ Streams allows you to customize the configuration of Apache Kafka Connect nodes by editing certain options listed in Apache Kafka documentation.

Configuration options that cannot be configured relate to:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Listener / REST interface configuration
- Plugin path configuration

These options are automatically configured by AMQ Streams.

3.3.5.1. Kafka Connect configuration

Kafka Connect is configured using the **config** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. This property contains the Kafka Connect configuration options as keys. The values can be one of the following JSON types:

- String
- Number
- Boolean

You can specify and configure the options listed in the Apache Kafka documentation with the exception of those options that are managed directly by AMQ Streams. Specifically, configuration options with keys equal to or starting with one of the following strings are forbidden:

- **ssl.**
- **sasl.**
- **security.**
- **listeners**
- **plugin.path**
- **rest.**
- **bootstrap.servers**

When a forbidden option is present in the **config** property, it is ignored and a warning message is printed to the Cluster Operator log file. All other options are passed to Kafka Connect.

IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Connect cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaConnect.spec.config** or **KafkaConnectS2I.spec.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Connect nodes.

Certain options have default values:

- **group.id** with default value **connect-cluster**
- **offset.storage.topic** with default value **connect-cluster-offsets**
- **config.storage.topic** with default value **connect-cluster-configs**
- **status.storage.topic** with default value **connect-cluster-status**
- **key.converter** with default value **org.apache.kafka.connect.json.JsonConverter**
- **value.converter** with default value **org.apache.kafka.connect.json.JsonConverter**

These options are automatically configured in case they are not present in the **KafkaConnect.spec.config** or **KafkaConnectS2I.spec.config** properties.

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example Kafka Connect configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
```

```
# ...
config:
  group.id: my-connect-cluster
  offset.storage.topic: my-connect-cluster-offsets
  config.storage.topic: my-connect-cluster-configs
  status.storage.topic: my-connect-cluster-status
  key.converter: org.apache.kafka.connect.json.JsonConverter
  value.converter: org.apache.kafka.connect.json.JsonConverter
  key.converter.schemas.enable: true
  value.converter.schemas.enable: true
  config.storage.replication.factor: 3
  offset.storage.replication.factor: 3
  status.storage.replication.factor: 3
  ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 1
  ssl.enabled.protocols: "TLSv1.2" 2
  ssl.protocol: "TLSv1.2" 3
# ...
```

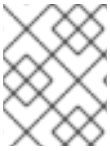
1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

3.3.5.2. Kafka Connect configuration 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/v1beta1
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. Kafka Connect cluster group that the instance belongs to.
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.

3.3.5.3. Configuring Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **config** property in the **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3. If authorization is enabled for Kafka Connect, configure the Kafka Connect user to enable access to the Kafka Connect consumer group and topics.

3.3.6. Kafka Connect user authorization

If authorization is enabled for Kafka Connect, the Kafka Connect user must be configured to provide read/write access rights to the Kafka Connect consumer group and internal topics.

The properties for the consumer group and internal topics are automatically configured by AMQ Streams, or they can be specified explicitly in the **spec** for the **KafkaConnect** or **KafkaConnectS2I** configuration.

The following example shows the configuration of the properties in the **KafkaConnect** resource, which need to be represented in the Kafka Connect user configuration.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster 1
    offset.storage.topic: my-connect-cluster-offsets 2
    config.storage.topic: my-connect-cluster-configs 3
    status.storage.topic: my-connect-cluster-status 4
    # ...
  # ...
```

1. Kafka Connect cluster group that the instance belongs to.
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.

3.3.6.1. Configuring Kafka Connect user authorization

This procedure describes how to authorize user access to Kafka Connect.

When any type of authorization is being used in Kafka, a Kafka Connect user requires access rights to the consumer group and the internal topics of Kafka Connect.

This procedure shows how access is provided when **simple** authorization is being used.

Simple authorization uses ACL rules, handled by the Kafka **SimpleAclAuthorizer** plugin, to provide the right level of access. For more information on configuring a **KafkaUser** resource to use simple authorization, see Kafka User resource.

NOTE

The default values for the consumer group and topics will differ when running multiple instances.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **authorization** property in the **KafkaUser** resource to provide access rights to the user. In the following example, access rights are configured for the Kafka Connect topics and consumer group using **literal** name values:

Property	Name
offset.storage.topic	**connect-cluster-offsets**
status.storage.topic	**connect-cluster-status**
config.storage.topic	**connect-cluster-configs**
group	**connect-cluster**

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      # access to offset.storage.topic
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Write
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Create
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Describe
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
```

```
      patternType: literal
    operation: Read
    host: "*"
  # access to status.storage.topic
  - resource:
      type: topic
      name: connect-cluster-status
      patternType: literal
    operation: Write
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-status
      patternType: literal
    operation: Create
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-status
      patternType: literal
    operation: Describe
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-status
      patternType: literal
    operation: Read
    host: "*"
  # access to config.storage.topic
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Write
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Create
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Describe
    host: "*"
  - resource:
      type: topic
      name: connect-cluster-configs
      patternType: literal
    operation: Read
    host: "*"
  # consumer group
  - resource:
      type: group
```

```
    name: connect-cluster
    patternType: literal
  operation: Read
  host: "*"
```

2. Create or update the resource.

```
oc apply -f your-file
```

3.3.7. CPU and memory resources

For every deployed container, AMQ Streams allows you to request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

AMQ Streams uses the OpenShift syntax for specifying CPU and memory resources.

3.3.7.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

Additional resources

- For more information about managing computing resources on OpenShift, see Managing Compute Resources for Containers.

3.3.7.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.

IMPORTANT

If the resource request is for more than the available free resources in the OpenShift cluster, the pod is not scheduled.

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**
- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

```
# ...
resources:
  requests:
    cpu: 12
    memory: 64Gi
# ...
```

3.3.7.1.2. Resource limits

Limits specify the maximum resources that can be consumed by a given container. The limit is not reserved and might not always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

```
# ...
resources:
  limits:
    cpu: 12
    memory: 64Gi
# ...
```

3.3.7.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

- Number of CPU cores as integer (**5** CPU core) or decimal (**2.5** CPU core).
- Number or *millicpus* / *millicores* (**100m**) where 1000 *millicores* is the same **1** CPU core.

Example CPU units

```
# ...
resources:
  requests:
    cpu: 500m
  limits:
    cpu: 2.5
# ...
```

NOTE

The computing power of 1 CPU core may differ depending on the platform where OpenShift is deployed.

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

3.3.7.1.4. Supported memory formats

Memory requests and limits are specified in megabytes, gigabytes, mebibytes, and gibibytes.

- To specify memory in megabytes, use the **M** suffix. For example **1000M**.
- To specify memory in gigabytes, use the **G** suffix. For example **1G**.
- To specify memory in mebibytes, use the **Mi** suffix. For example **1000Mi**.
- To specify memory in gibibytes, use the **Gi** suffix. For example **1Gi**.

An example of using different memory units

```
# ...
resources:
  requests:
    memory: 512Mi
  limits:
    memory: 2Gi
# ...
```

Additional resources

- For more details about memory specification and additional supported units, see Meaning of memory.

3.3.7.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **resources** property in the resource specifying the cluster deployment. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema, see {K8sResourceRequirementsAPI}.

3.3.8. Kafka Connect with S2I loggers

Kafka Connect with Source2Image support has its own configurable loggers:

- **connect.root.logger.level**
- **log4j.logger.org.reflections**

Kafka Connect uses the Apache **log4j** logger implementation.

Use the **logging** property to configure loggers and logger levels.

You can set the log levels by specifying the logger and level directly (inline) or use a custom (external) ConfigMap. If a ConfigMap is used, you set **logging.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**.

Here we see examples of **inline** and **external** logging.

Inline logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
spec:
  # ...
  logging:
```

```
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  # ...
```

External logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
spec:
  # ...
  logging:
    type: external
    name: customConfigMap
  # ...
```

Additional resources

- Garbage collector (GC) logging can also be enabled (or disabled). For more information about GC logging, see Section 3.1.18.1, “JVM configuration”
- For more information about log levels, see Apache logging services.

3.3.9. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts to fix it.

OpenShift supports two types of Healthcheck probes:

- Liveness probes
- Readiness probes

For more details about the probes, see Configure Liveness and Readiness Probes. Both types of probes are used in AMQ Streams components.

Users can configure selected options for liveness and readiness probes.

3.3.9.1. Healthcheck configurations

Liveness and readiness probes can be configured using the **livenessProbe** and **readinessProbe** properties in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**

- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaBridge.spec**

Both **livenessProbe** and **readinessProbe** support the following options:

- **initialDelaySeconds**
- **timeoutSeconds**
- **periodSeconds**
- **successThreshold**
- **failureThreshold**

For more information about the **livenessProbe** and **readinessProbe** options, see Section B.40, "**Probe** schema reference".

An example of liveness and readiness probe configuration

```
# ...
readinessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
# ...
```

3.3.9.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **livenessProbe** or **readinessProbe** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
```

```
    # ...
    readinessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.10. Prometheus metrics

AMQ Streams supports Prometheus metrics using Prometheus JMX exporter to convert the JMX metrics supported by Apache Kafka and ZooKeeper to Prometheus metrics. When metrics are enabled, they are exposed on port 9404.

For more information about setting up and deploying Prometheus and Grafana, see Introducing Metrics to Kafka.

3.3.10.1. Metrics configuration

Prometheus metrics are enabled by configuring the **metrics** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**

When the **metrics** property is not defined in the resource, the Prometheus metrics will be disabled. To enable Prometheus metrics export without any further configuration, you can set it to an empty object (**{}**).

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

The **metrics** property might contain additional configuration for the Prometheus JMX exporter.

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

3.3.10.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **metrics** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.11. JVM Options

The following components of AMQ Streams run inside a Virtual Machine (VM):

- Apache Kafka
- Apache ZooKeeper
- Apache Kafka Connect
- Apache Kafka MirrorMaker
- AMQ Streams Kafka Bridge

JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure some of these options.

3.3.11.1. JVM configuration

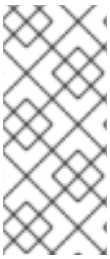
JVM options can be configured using the **jvmOptions** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaBridge.spec**

Only a selected subset of available JVM options can be configured. The following options are supported:

-Xms and -Xmx

-Xms configures the minimum initial allocation heap size when the JVM starts. **-Xmx** configures the maximum heap size.

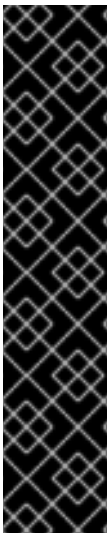


NOTE

The units accepted by JVM settings such as **-Xmx** and **-Xms** are those accepted by the JDK **java** binary in the corresponding image. Accordingly, **1g** or **1G** means 1,073,741,824 bytes, and **Gi** is not a valid unit suffix. This is in contrast to the units used for memory requests and limits, which follow the OpenShift convention where **1G** means 1,000,000,000 bytes, and **1Gi** means 1,073,741,824 bytes

The default values used for **-Xms** and **-Xmx** depends on whether there is a memory request limit configured for the container:

- If there is a memory limit then the JVM's minimum and maximum memory will be set to a value corresponding to the limit.
- If there is no memory limit then the JVM's minimum memory will be set to **128M** and the JVM's maximum memory will not be defined. This allows for the JVM's memory to grow as-needed, which is ideal for single node environments in test and development.



IMPORTANT

Setting **-Xmx** explicitly requires some care:

- The JVM's overall memory usage will be approximately 4 × the maximum heap, as configured by **-Xmx**.
- If **-Xmx** is set without also setting an appropriate OpenShift memory limit, it is possible that the container will be killed should the OpenShift node experience memory pressure (from other Pods running on it).
- If **-Xmx** is set without also setting an appropriate OpenShift memory request, it is possible that the container will be scheduled to a node with insufficient memory. In this case, the container will not start but crash (immediately if **-Xms** is set to **-Xmx**, or some later time if not).

When setting **-Xmx** explicitly, it is recommended to:

- set the memory request and the memory limit to the same value,
- use a memory request that is at least 4.5 × the **-Xmx**,
- consider setting **-Xms** to the same value as **-Xmx**.

IMPORTANT

Containers doing lots of disk I/O (such as Kafka broker containers) will need to leave some memory available for use as operating system page cache. On such containers, the requested memory should be significantly higher than the memory used by the JVM.

Example fragment configuring -Xmx and -Xms

```
# ...
jvmOptions:
  "-Xmx": "2g"
  "-Xms": "2g"
# ...
```

In the above example, the JVM will use 2 GiB (=2,147,483,648 bytes) for its heap. Its total memory usage will be approximately 8GiB.

Setting the same value for initial (**-Xms**) and maximum (**-Xmx**) heap sizes avoids the JVM having to allocate memory after startup, at the cost of possibly allocating more heap than is really needed. For Kafka and ZooKeeper pods such allocation could cause unwanted latency. For Kafka Connect avoiding over allocation may be the most important concern, especially in distributed mode where the effects of over-allocation will be multiplied by the number of consumers.

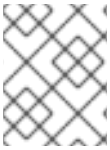
-server

-server enables the server JVM. This option can be set to true or false.

Example fragment configuring -server

```
# ...
jvmOptions:
```

```
  "-server": true
# ...
```



NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

-XX

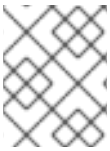
-XX object can be used for configuring advanced runtime options of a JVM. The **-server** and **-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

Example showing the use of the **-XX** object

```
jvmOptions:
  "-XX":
    "UseG1GC": true
    "MaxGCPauseMillis": 20
    "InitiatingHeapOccupancyPercent": 35
    "ExplicitGCInvokesConcurrent": true
    "UseParNewGC": false
```

The example configuration above will result in the following JVM options:

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -
XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```



NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

3.3.11.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is disabled by default. To enable it, set the **gcLoggingEnabled** property as follows:

Example of enabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: true
# ...
```

3.3.11.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **jvmOptions** property in the **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, **KafkaMirrorMaker**, or **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jvmOptions:
      "-Xmx": "8g"
      "-Xms": "8g"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.12. Container images

AMQ Streams allows you to configure container images which will be used for its components. Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such a case, you should either copy the AMQ Streams images or build them from the source. If the configured image is not compatible with AMQ Streams images, it might not work properly.

3.3.12.1. Container image configurations

You can specify which container image to use for each component using the **image** property in the following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

3.3.12.1.1. Configuring the image property for Kafka, Kafka Connect, and Kafka MirrorMaker

Kafka, Kafka Connect (including Kafka Connect with S2I support), and Kafka MirrorMaker support multiple versions of Kafka. Each component requires its own image. The default images for the different Kafka versions are configured in the following environment variables:

- **STRIMZI_KAFKA_IMAGES**
- **STRIMZI_KAFKA_CONNECT_IMAGES**
- **STRIMZI_KAFKA_CONNECT_S2I_IMAGES**
- **STRIMZI_KAFKA_MIRROR_MAKER_IMAGES**

These environment variables contain mappings between the Kafka versions and their corresponding images. The mappings are used together with the **image** and **version** properties:

- If neither **image** nor **version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the environment variable.
- If **image** is given but **version** is not, then the given image is used and the **version** is assumed to be the Cluster Operator's default Kafka version.
- If **version** is given but **image** is not, then the image that corresponds to the given version in the environment variable is used.
- If both **version** and **image** are given, then the given image is used. The image is assumed to contain a Kafka image with the given version.

The **image** and **version** for the different components can be configured in the following properties:

- For Kafka in **spec.kafka.image** and **spec.kafka.version**.
- For Kafka Connect, Kafka Connect S2I, and Kafka MirrorMaker in **spec.image** and **spec.version**.

WARNING

It is recommended to provide only the **version** and leave the **image** property unspecified. This reduces the chance of making a mistake when configuring the custom resource. If you need to change the images used for different versions of Kafka, it is preferable to configure the Cluster Operator's environment variables.

3.3.12.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property is missing, the **image** specified in the Cluster Operator configuration will be used. If the **image** name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka broker TLS sidecar:

1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.

- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Exporter:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Bridge:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-bridge-rhel7:1.5.0** container image.
- For Kafka broker initializer:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.

WARNING

Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such case, you should either copy the AMQ Streams images or build them from source. In case the configured image is not compatible with AMQ Streams images, it might not work properly.

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.3.12.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **image** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.13. Configuring pod scheduling

IMPORTANT

When two applications are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.3.13.1. Scheduling pods based on other applications

3.3.13.1.1. Avoid critical applications to share the node

Pod anti-affinity can be used to ensure that critical applications are never scheduled on the same disk. When running Kafka cluster, it is recommended to use pod anti-affinity to ensure that the Kafka brokers do not share the nodes with other workloads like databases.

3.3.13.1.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.3.13.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **affinity** property in the resource specifying the cluster deployment. Use labels to specify the pods which should not be scheduled on the same nodes. The **topologyKey** should be set to **kubernetes.io/hostname** to specify that the selected pods should not be scheduled on nodes with the same hostname. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.13.2. Scheduling pods to specific nodes

3.3.13.2.1. Node scheduling

The OpenShift cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while other might be optimized for storage (fast local SSDs) or network. Using different nodes helps to optimize both costs and performance. To achieve the best possible performance, it is important to allow scheduling of AMQ Streams components to use the right nodes.

OpenShift uses node affinity to schedule workloads onto specific nodes. Node affinity allows you to create a scheduling constraint for the node on which the pod will be scheduled. The constraint is specified as a label selector. You can specify the label using either the built-in node label like **beta.kubernetes.io/instance-type** or custom labels to select the right node.

3.3.13.2.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**

- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.3.13.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   This can be done using **oc label**:

   ```
   oc label node your-node node-type=fast-network
   ```

   Alternatively, some of the existing labels might be reused.

2. Edit the **affinity** property in the resource specifying the cluster deployment. For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
         pod:
           affinity:
             nodeAffinity:
               requiredDuringSchedulingIgnoredDuringExecution:
                 nodeSelectorTerms:
                   - matchExpressions:
                     - key: node-type
                       operator: In
                       values:
                       - fast-network
   ```

```
    # ...
zookeeper:
    # ...
```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.13.3. Using dedicated nodes

3.3.13.3.1. Dedicated nodes

Cluster administrators can mark selected OpenShift nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

3.3.13.3.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.3.13.3.3. Tolerations

Tolerations can be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka.template.pod**

- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The format of the **tolerations** property follows the OpenShift specification. For more details, see the Kubernetes taints and tolerations .

3.3.13.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated.
2. Make sure there are no workloads scheduled on these nodes.
3. Set the taints on the selected nodes:
   This can be done using **oc adm taint**:

```
oc adm taint node your-node dedicated=Kafka:NoSchedule
```

4. Additionally, add a label to the selected nodes as well.
   This can be done using **oc label**:

```
oc label node your-node dedicated=Kafka
```

5. Edit the **affinity** and **tolerations** properties in the resource specifying the cluster deployment.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
```

```
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: dedicated
                operator: In
                values:
                - Kafka
  # ...
zookeeper:
  # ...
```

6. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.3.14. Using external configuration and secrets

Connectors are created, reconfigured, and deleted using the Kafka Connect HTTP REST interface, or by using **KafkaConnectors**. For more information on these methods, see Section 2.3.3, "Creating and managing connectors". The connector configuration is passed to Kafka Connect as part of an HTTP request and stored within Kafka itself.

ConfigMaps and Secrets are standard OpenShift resources used for storing configurations and confidential data. Whichever method you use to manage connectors, you can use ConfigMaps and Secrets to configure certain elements of a connector. You can then reference the configuration values in HTTP REST commands (this keeps the configuration separate and more secure, if needed). This method applies especially to confidential data, such as usernames, passwords, or certificates.

3.3.14.1. Storing connector configurations externally

You can mount ConfigMaps or Secrets into a Kafka Connect pod as volumes or environment variables. Volumes and environment variables are configured in the **externalConfiguration** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**.

3.3.14.1.1. External configuration as environment variables

The **env** property is used to specify one or more environment variables. These variables can contain a value from either a ConfigMap or a Secret.

NOTE

The names of user-defined environment variables cannot start with **KAFKA_** or **STRIMZI_**.

To mount a value from a Secret to an environment variable, use the **valueFrom** property and the **secretKeyRef** as shown in the following example.

Example of an environment variable set to a value from a Secret

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
```

```
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: my-key
```

A common use case for mounting Secrets to environment variables is when your connector needs to communicate with Amazon AWS and needs to read the **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY** environment variables with credentials.

To mount a value from a ConfigMap to an environment variable, use **configMapKeyRef** in the **valueFrom** property as shown in the following example.

Example of an environment variable set to a value from a ConfigMap

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

3.3.14.1.2. External configuration as volumes

You can also mount ConfigMaps or Secrets to a Kafka Connect pod as volumes. Using volumes instead of environment variables is useful in the following scenarios:

- Mounting truststores or keystores with TLS certificates
- Mounting a properties file that is used to configure Kafka Connect connectors

In the **volumes** property of the **externalConfiguration** resource, list the ConfigMaps or Secrets that will be mounted as volumes. Each volume must specify a name in the **name** property and a reference to ConfigMap or Secret.

Example of volumes with external configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
```

```
externalConfiguration:
  volumes:
    - name: connector1
      configMap:
        name: connector1-configuration
    - name: connector1-certificates
      secret:
        secretName: connector1-certificates
```

The volumes will be mounted inside the Kafka Connect containers in the path **/opt/kafka/external-configuration/*<volume-name>***. For example, the files from a volume named **connector1** would appear in the directory **/opt/kafka/external-configuration/connector1**.

The **FileConfigProvider** has to be used to read the values from the mounted properties files in connector configurations.

3.3.14.2. Mounting Secrets as environment variables

You can create an OpenShift Secret and mount it to Kafka Connect as an environment variable.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing the information that will be mounted as an environment variable. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-creds
type: Opaque
data:
  awsAccessKey: QUtJQVhYWFhYWFhYWFhYWFg=
  awsSecretAccessKey: Ylhsd1IYTnpkMjI5WkE=
```

2. Create or edit the Kafka Connect resource. Configure the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: AWS_ACCESS_KEY_ID
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsAccessKey
      - name: AWS_SECRET_ACCESS_KEY
```

```
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: awsSecretAccessKey
```

3. Apply the changes to your Kafka Connect deployment.
   Use **oc apply**:

```
oc apply -f your-file
```

The environment variables are now available for use when developing your connectors.

Additional resources

- For more information about external configuration in Kafka Connect, see Section B.84, "**ExternalConfiguration** schema reference".

3.3.14.3. Mounting Secrets as volumes

You can create an OpenShift Secret, mount it as a volume to Kafka Connect, and then use it to configure a Kafka Connect connector.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing a properties file that defines the configuration options for your connector configuration. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |-
    dbUsername: my-user
    dbPassword: my-password
```

2. Create or edit the Kafka Connect resource. Configure the **FileConfigProvider** in the **config** section and the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: file
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
```

```
#...
externalConfiguration:
  volumes:
    - name: connector-config
      secret:
        secretName: mysecret
```

3. Apply the changes to your Kafka Connect deployment.
Use **oc apply**:

```
oc apply -f your-file
```

4. Configure your connector

- If you are using the Kafka Connect HTTP REST interface, use the values from the mounted properties file in your JSON payload with connector configuration. For example:

```
{
  "name":"my-connector",
  "config":{
    "connector.class":"MyDbConnector",
    "tasks.max":"3",
    "database": "my-postgresql:5432",
    "username":"${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbUsername}",
    "password":"${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbPassword}",
    # ...
  }
}
```

- If you are using a **KafkaConnector** resource, use the values from the mounted properties file in the **spec.config** section of your custom resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnector
metadata:
  name: my-connector
  # ...
spec:
  class: "MyDbConnector"
  tasksMax: 3
  config:
    database: "my-postgresql:5432"
    username: "${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbUsername}"
    password: "${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbPassword}"
```

Additional resources

- For more information about external configuration in Kafka Connect, see Section B.84, "**ExternalConfiguration** schema reference".

3.3.15. Enabling KafkaConnector resources

To enable **KafkaConnectors** for a Kafka Connect cluster, add the **strimzi.io/use-connector-resources** annotation to the **KafkaConnect** or **KafkaConnectS2I** custom resource.

Prerequisites

- A running Cluster Operator

Procedure

1. Edit the **KafkaConnect** or **KafkaConnectS2I** resource. Add the **strimzi.io/use-connector-resources** annotation. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
```

2. Create or update the resource using **oc apply**:

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

Additional resources

- Section 2.3.3, "Creating and managing connectors"
- Section 2.3.4, "Deploying a **KafkaConnector** resource to Kafka Connect"
- Section B.74, "**KafkaConnect** schema reference"
- Section B.90, "**KafkaConnectS2I** schema reference"

3.3.16. List of resources created as part of Kafka Connect cluster with Source2Image support

The following resources will created by the Cluster Operator in the OpenShift cluster:

***connect-cluster-name*-connect-source**

ImageStream which is used as the base image for the newly-built Docker images.

***connect-cluster-name*-connect**

BuildConfig which is responsible for building the new Kafka Connect Docker images.

***connect-cluster-name*-connect**

ImageStream where the newly built Docker images will be pushed.

***connect-cluster-name*-connect**

DeploymentConfig which is in charge of creating the Kafka Connect worker node pods.

***connect-cluster-name*-connect-api**

Service which exposes the REST interface for managing the Kafka Connect cluster.

***connect-cluster-name*-config**

ConfigMap which contains the Kafka Connect ancillary configuration and is mounted as a volume by the Kafka broker pods.

***connect-cluster-name*-connect**

Pod Disruption Budget configured for the Kafka Connect worker nodes.

3.3.17. Integrating with Debezium for change data capture

Red Hat Debezium is a distributed change data capture platform. It captures row-level changes in databases, creates change event records, and streams the records to Kafka topics. Debezium is built on Apache Kafka. You can deploy and integrate Debezium with AMQ Streams. Following a deployment of AMQ Streams, you deploy Debezium as a connector configuration through Kafka Connect. Debezium passes change event records to AMQ Streams on OpenShift. Applications can read these *change event streams* and access the change events in the order in which they occurred.

Debezium has multiple uses, including:

- Data replication
- Updating caches and search indexes
- Simplifying monolithic applications
- Data integration
- Enabling streaming queries

To capture database changes, deploy Kafka Connect with a Debezium database connector . You configure a **KafkaConnector** resource to define the connector instance.

For more information on deploying Debezium with AMQ Streams, refer to the product documentation. The Debezium documentation includes a *Getting Started with Debezium* guide that guides you through the process of setting up the services and connector required to view change event records for database updates.

3.4. KAFKA MIRRORMAKER CONFIGURATION

This chapter describes how to configure a Kafka MirrorMaker deployment in your AMQ Streams cluster to replicate data between Kafka clusters.

You can use AMQ Streams with MirrorMaker or MirrorMaker 2.0. MirrorMaker 2.0 is the latest version, and offers a more efficient way to mirror data between Kafka clusters.

If you are using MirrorMaker, you configure the **KafkaMirrorMaker** resource.

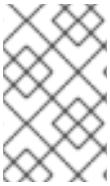
The following procedure shows how the resource is configured:

- Configuring Kafka MirrorMaker

Supported properties are also described in more detail for your reference:

- Kafka MirrorMaker configuration properties

The full schema of the **KafkaMirrorMaker** resource is described in the KafkaMirrorMaker schema reference.



NOTE

Labels applied to a **KafkaMirrorMaker** resource are also applied to the OpenShift resources comprising Kafka MirrorMaker. This provides a convenient mechanism for resources to be labeled as required.

3.4.1. Configuring Kafka MirrorMaker

Use the properties of the **KafkaMirrorMaker** resource to configure your Kafka MirrorMaker deployment.

You can configure access control for producers and consumers using TLS or SASL authentication. This procedure shows a configuration that uses TLS encryption and authentication on the consumer and producer side.

Prerequisites

- AMQ Streams and Kafka is deployed
- Source and target Kafka clusters are available

Procedure

1. Edit the **spec** properties for the **KafkaMirrorMaker** resource.
   The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  replicas: 3 (1)
  consumer:
    bootstrapServers: my-source-cluster-kafka-bootstrap:9092 (2)
    groupId: "my-group" (3)
    numStreams: 2 (4)
    offsetCommitInterval: 120000 (5)
    tls: (6)
      trustedCertificates:
      - secretName: my-source-cluster-ca-cert
        certificate: ca.crt
    authentication: (7)
      type: tls
      certificateAndKey:
        secretName: my-source-secret
        certificate: public.crt
        key: private.key
    config: (8)
      max.poll.records: 100
      receive.buffer.bytes: 32768
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (9)
      ssl.enabled.protocols: "TLSv1.2"
```

```
    ssl.protocol: "TLSv1.2"
producer:
  bootstrapServers: my-target-cluster-kafka-bootstrap:9092
  abortOnSendFailure: false (10)
  tls:
    trustedCertificates:
    - secretName: my-target-cluster-ca-cert
      certificate: ca.crt
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-target-secret
      certificate: public.crt
      key: private.key
  config:
    compression.type: gzip
    batch.size: 8192
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (11)
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
whitelist: "my-topic|other-topic" (12)
resources: (13)
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
logging: (14)
  type: inline
  loggers:
    mirrormaker.root.logger: "INFO"
readinessProbe: (15)
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
metrics: (16)
  lowercaseOutputName: true
  rules:
    - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
      name: "kafka_server_$1_$2_total"
    - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*,
      topic=(.+)><>Count"
      name: "kafka_server_$1_$2_total"
      labels:
        topic: "$3"
jvmOptions: (17)
  "-Xmx": "1g"
  "-Xms": "1g"
image: my-org/my-image:latest (18)
template: (19)
  pod:
    affinity:
```

```
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
  connectContainer: 20
    env:
      - name: JAEGER_SERVICE_NAME
        value: my-jaeger-service
      - name: JAEGER_AGENT_HOST
        value: jaeger-agent-name
      - name: JAEGER_AGENT_PORT
        value: "6831"
tracing: 21
  type: jaeger
```

1. The number of replica nodes.
2. Bootstrap servers for consumer and producer.
3. Group ID for the consumer.
4. The number of consumer streams.
5. The offset auto-commit interval in milliseconds.
6. TLS encryption with key names under which TLS certificates are stored in X.509 format for consumer or producer. For more details see **KafkaMirrorMakerTls** schema reference.
7. Authentication for consumer or producer, using the TLS mechanism, as shown here, using OAuth bearer tokens, or a SASL-based SCRAM-SHA-512 or PLAIN mechanism.
8. Kafka configuration options for consumer and producer.
9. SSL properties for external listeners to run with a specific *cipher suite* for a TLS version.
10. If set to **true**, Kafka MirrorMaker will exit and the container will restart following a send failure for a message.
11. SSL properties for external listeners to run with a specific *cipher suite* for a TLS version.
12. Topics mirrored from source to target Kafka cluster.
13. Requests for reservation of supported resources, currently **cpu** and **memory**, and limits to specify the maximum resources that can be consumed.
14. Specified loggers and log levels added directly (**inline**) or indirectly (**external**) through a ConfigMap. A custom ConfigMap must be placed under the **log4j.properties** or **log4j2.properties** key. MirrorMaker has a single logger called **mirrormaker.root.logger**. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
15. Healthchecks to know when to restart a container (liveness) and when a container can

accept traffic (readiness).

16 Prometheus metrics, which are enabled with configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using **metrics: {}**.

17 JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka MirrorMaker.

18 ADVANCED OPTION: Container image configuration, which is recommended only in special situations.

19 Template customization. Here a pod is scheduled based with anti-affinity, so the pod is not scheduled on nodes with the same hostname.

20 Environment variables are also set for distributed tracing using Jaeger.

21 Distributed tracing is enabled for Jaeger.

WARNING

With the **abortOnSendFailure** property set to **false**, the producer attempts to send the next message in a topic. The original message might be lost, as there is no attempt to resend a failed message.

2. Create or update the resource:

```
oc apply -f <your-file>
```

3.4.2. Kafka MirrorMaker configuration properties

Use the **spec** configuration properties of the **KafkaMirrorMaker** resource to set up your MirrorMaker deployment.

Supported properties are described here for your reference.

3.4.2.1. Replicas

Use the **replicas** property to configure replicas.

You can run multiple MirrorMaker replicas to provide better availability and scalability. When running Kafka MirrorMaker on OpenShift it is not absolutely necessary to run multiple replicas of the Kafka MirrorMaker for high availability. When the node where the Kafka MirrorMaker has deployed crashes, OpenShift will automatically reschedule the Kafka MirrorMaker pod to a different node. However, running Kafka MirrorMaker with multiple replicas can provide faster failover times as the other nodes will be up and running.

3.4.2.2. Bootstrap servers

Use the **consumer.bootstrapServers** and **producer.bootstrapServers** properties to configure lists of bootstrap servers for the consumer and producer.

Kafka MirrorMaker always works together with two Kafka clusters (source and target). The source and the target Kafka clusters are specified in the form of two lists of comma-separated list of ***<hostname>:<port>*** pairs. Each comma-separated list contains one or more Kafka brokers or a **Service** pointing to Kafka brokers specified as a **<hostname>:<port>** pairs.

The bootstrap server lists can refer to Kafka clusters that do not need to be deployed in the same OpenShift cluster. They can even refer to a Kafka cluster not deployed by AMQ Streams, or deployed by AMQ Streams but on a different OpenShift cluster accessible outside.

If on the same OpenShift cluster, each list must ideally contain the Kafka cluster bootstrap service which is named ***<cluster-name>*-kafka-bootstrap** and a port of 9092 for plain traffic or 9093 for encrypted traffic. If deployed by AMQ Streams but on different OpenShift clusters, the list content depends on the approach used for exposing the clusters (routes, nodeports or loadbalancers).

When using Kafka MirrorMaker with a Kafka cluster not managed by AMQ Streams, you can specify the bootstrap servers list according to the configuration of the given cluster.

3.4.2.3. Whitelist

Use the **whitelist** property to configure a list of topics that Kafka MirrorMaker mirrors from the source to the target Kafka cluster.

The property allows any regular expression from the simplest case with a single topic name to complex patterns. For example, you can mirror topics A and B using "A|B" or all topics using "*". You can also pass multiple regular expressions separated by commas to the Kafka MirrorMaker.

3.4.2.4. Consumer group identifier

Use the **consumer.groupId** property to configure a consumer group identifier for the consumer.

Kafka MirrorMaker uses a Kafka consumer to consume messages, behaving like any other Kafka consumer client. Messages consumed from the source Kafka cluster are mirrored to a target Kafka cluster. A group identifier is required, as the consumer needs to be part of a consumer group for the assignment of partitions.

3.4.2.5. Consumer streams

Use the **consumer.numStreams** property to configure the number of streams for the consumer.

You can increase the throughput in mirroring topics by increasing the number of consumer threads. Consumer threads belong to the consumer group specified for Kafka MirrorMaker. Topic partitions are assigned across the consumer threads, which consume messages in parallel.

3.4.2.6. Offset auto-commit interval

Use the **consumer.offsetCommitInterval** property to configure an offset auto-commit interval for the consumer.

You can specify the regular time interval at which an offset is committed after Kafka MirrorMaker has consumed data from the source Kafka cluster. The time interval is set in milliseconds, with a default value of 60,000.

3.4.2.7. Abort on message send failure

Use the **producer.abortOnSendFailure** property to configure how to handle message send failure from the producer.

By default, if an error occurs when sending a message from Kafka MirrorMaker to a Kafka cluster:

- The Kafka MirrorMaker container is terminated in OpenShift.
- The container is then recreated.

If the **abortOnSendFailure** option is set to **false**, message sending errors are ignored.

3.4.2.8. Kafka producer and consumer

Use the **consumer.config** and **producer.config** properties to configure Kafka options for the consumer and producer.

The **config** property contains the Kafka MirrorMaker consumer and producer configuration options as keys, with values set in one of the following JSON types:

- String
- Number
- Boolean

Exceptions

You can specify and configure standard Kafka consumer and producer options:

- Apache Kafka configuration documentation for producers
- Apache Kafka configuration documentation for consumers

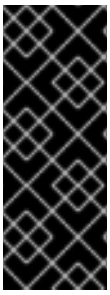
However, there are exceptions for options automatically configured and managed directly by AMQ Streams related to:

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

Specifically, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **ssl.**
- **sasl.**
- **security.**
- **bootstrap.servers**
- **group.id**

When a forbidden option is present in the **config** property, it is ignored and a warning message is printed to the Cluster Operator log file. All other options are passed to Kafka MirrorMaker.



IMPORTANT

The Cluster Operator does not validate keys or values in the provided **config** object. When an invalid configuration is provided, the Kafka MirrorMaker might not start or might become unstable. In such cases, the configuration in the **KafkaMirrorMaker.spec.consumer.config** or **KafkaMirrorMaker.spec.producer.config** object should be fixed and the Cluster Operator will roll out the new configuration for Kafka MirrorMaker.

3.4.2.9. CPU and memory resources

Use the **reources.requests** and **resources.limits** properties to configure resource requests and limits.

For every deployed container, AMQ Streams allows you to request specific resources and define the maximum consumption of those resources.

AMQ Streams supports requests and limits for the following types of resources:

- **cpu**
- **memory**

AMQ Streams uses the OpenShift syntax for specifying these resources.

For more information about managing computing resources on OpenShift, see Managing Compute Resources for Containers.

Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.

IMPORTANT

If the resource request is for more than the available free resources in the OpenShift cluster, the pod is not scheduled.

A request may be configured for one or more supported resources.

Resource limits

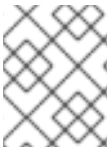
Limits specify the maximum resources that can be consumed by a given container. The limit is not reserved and might not always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

A resource may be configured for one or more supported limits.

Supported CPU formats

CPU requests and limits are supported in the following formats:

- Number of CPU cores as integer (**5** CPU core) or decimal (**2.5** CPU core).
- Number or *millicpus* / *millicores* (**100m**) where 1000 *millicores* is the same **1** CPU core.



NOTE

The computing power of 1 CPU core may differ depending on the platform where OpenShift is deployed.

For more information on CPU specification, see the Meaning of CPU.

Supported memory formats

Memory requests and limits are specified in megabytes, gigabytes, mebibytes, and gibibytes.

- To specify memory in megabytes, use the **M** suffix. For example **1000M**.
- To specify memory in gigabytes, use the **G** suffix. For example **1G**.
- To specify memory in mebibytes, use the **Mi** suffix. For example **1000Mi**.
- To specify memory in gibibytes, use the **Gi** suffix. For example **1Gi**.

For more details about memory specification and additional supported units, see Meaning of memory.

3.4.2.10. Kafka MirrorMaker loggers

Kafka MirrorMaker has its own configurable logger:

- **mirrormaker.root.logger**

MirrorMaker uses the Apache **log4j** logger implementation.

Use the **logging** property to configure loggers and logger levels.

You can set the log levels by specifying the logger and level directly (inline) or use a custom (external) ConfigMap. If a ConfigMap is used, you set **logging.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**.

Here we see examples of **inline** and **external** logging:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: inline
    loggers:
      mirrormaker.root.logger: "INFO"
  # ...
```

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: external
    name: customConfigMap
  # ...
```

Additional resources

- Garbage collector (GC) logging can also be enabled (or disabled). For more information about GC logging, see Section 3.1.18.1, "JVM configuration"
- For more information about log levels, see Apache logging services.

3.4.2.11. Healthchecks

Use the **livenessProbe** and **readinessProbe** properties to configure healthcheck probes supported in AMQ Streams.

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts to fix it.

For more details about the probes, see Configure Liveness and Readiness Probes.

Both **livenessProbe** and **readinessProbe** support the following options:

- **initialDelaySeconds**
- **timeoutSeconds**
- **periodSeconds**
- **successThreshold**
- **failureThreshold**

An example of liveness and readiness probe configuration

```
# ...
readinessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
# ...
```

For more information about the **livenessProbe** and **readinessProbe** options, see Probe schema reference.

3.4.2.12. Prometheus metrics

Use the **metrics** property to enable and configure Prometheus metrics.

The **metrics** property can also contain additional configuration for the Prometheus JMX exporter. AMQ Streams supports Prometheus metrics using Prometheus JMX exporter to convert the JMX metrics supported by Apache Kafka and ZooKeeper to Prometheus metrics.

To enable Prometheus metrics export without any further configuration, you can set it to an empty object (**{}**).

When metrics are enabled, they are exposed on port 9404.

When the **metrics** property is not defined in the resource, the Prometheus metrics are disabled.

For more information about setting up and deploying Prometheus and Grafana, see Introducing Metrics to Kafka.

3.4.2.13. JVM Options

Use the **jvmOptions** property to configure supported options for the JVM on which the component is running.

Supported JVM options help to optimize performance for different platforms and architectures.

For more information on the supported options, see JVM configuration.

3.4.2.14. Container images

Use the **image** property to configure the container image used by the component.

Overriding container images is recommended only in special situations where you need to use a different container registry or a customized image.

For example, if your network does not allow access to the container repository used by AMQ Streams, you can copy the AMQ Streams images or build them from the source. However, if the configured image is not compatible with AMQ Streams images, it might not work properly.

A copy of the container image might also be customized and used for debugging.

For more information see Container image configurations.

3.4.3. List of resources created as part of Kafka MirrorMaker

The following resources are created by the Cluster Operator in the OpenShift cluster:

<*mirror-maker-name*>-mirror-maker

Deployment which is responsible for creating the Kafka MirrorMaker pods.

<*mirror-maker-name*>-config

ConfigMap which contains ancillary configuration for the the Kafka MirrorMaker, and is mounted as a volume by the Kafka broker pods.

<*mirror-maker-name*>-mirror-maker

Pod Disruption Budget configured for the Kafka MirrorMaker worker nodes.

3.5. KAFKA MIRRORMAKER 2.0 CONFIGURATION

This section describes how to configure a Kafka MirrorMaker 2.0 deployment in your AMQ Streams cluster.

MirrorMaker 2.0 is used to replicate data between two or more active Kafka clusters, within or across data centers.

Data replication across clusters supports scenarios that require:

- Recovery of data in the event of a system failure
- Aggregation of data for analysis

- Restriction of data access to a specific cluster
- Provision of data at a specific location to improve latency

If you are using MirrorMaker 2.0, you configure the **KafkaMirrorMaker2** resource.

MirrorMaker 2.0 introduces an entirely new way of replicating data between clusters.

As a result, the resource configuration differs from the previous version of MirrorMaker. If you choose to use MirrorMaker 2.0, there is currently no legacy support, so any resources must be manually converted into the new format.

How MirrorMaker 2.0 replicates data is described here:

- MirrorMaker 2.0 data replication

The following procedure shows how the resource is configured for MirrorMaker 2.0:

- Synchronizing data between Kafka clusters

The full schema of the **KafkaMirrorMaker2** resource is described in the KafkaMirrorMaker2 schema reference.

3.5.1. MirrorMaker 2.0 data replication

MirrorMaker 2.0 consumes messages from a source Kafka cluster and writes them to a target Kafka cluster.

MirrorMaker 2.0 uses:

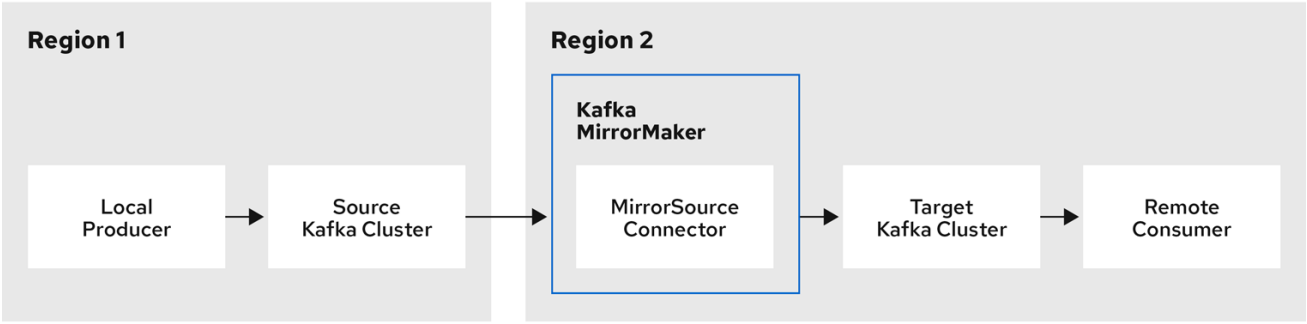
- Source cluster configuration to consume data from the source cluster
- Target cluster configuration to output data to the target cluster

MirrorMaker 2.0 is based on the Kafka Connect framework, *connectors* managing the transfer of data between clusters. A MirrorMaker 2.0 **MirrorSourceConnector** replicates topics from a source cluster to a target cluster.

The process of *mirroring* data from one cluster to another cluster is asynchronous. The recommended pattern is for messages to be produced locally alongside the source Kafka cluster, then consumed remotely close to the target Kafka cluster.

MirrorMaker 2.0 can be used with more than one source cluster.

Figure 3.1. Replication across two clusters



3.5.2. Cluster configuration

You can use MirrorMaker 2.0 in *active/passive* or *active/active* cluster configurations.

- In an *active/passive* configuration, the data from an active cluster is replicated in a passive cluster, which remains on standby, for example, for data recovery in the event of system failure.
- In an *active/active* configuration, both clusters are active and provide the same data simultaneously, which is useful if you want to make the same data available locally in different geographical locations.

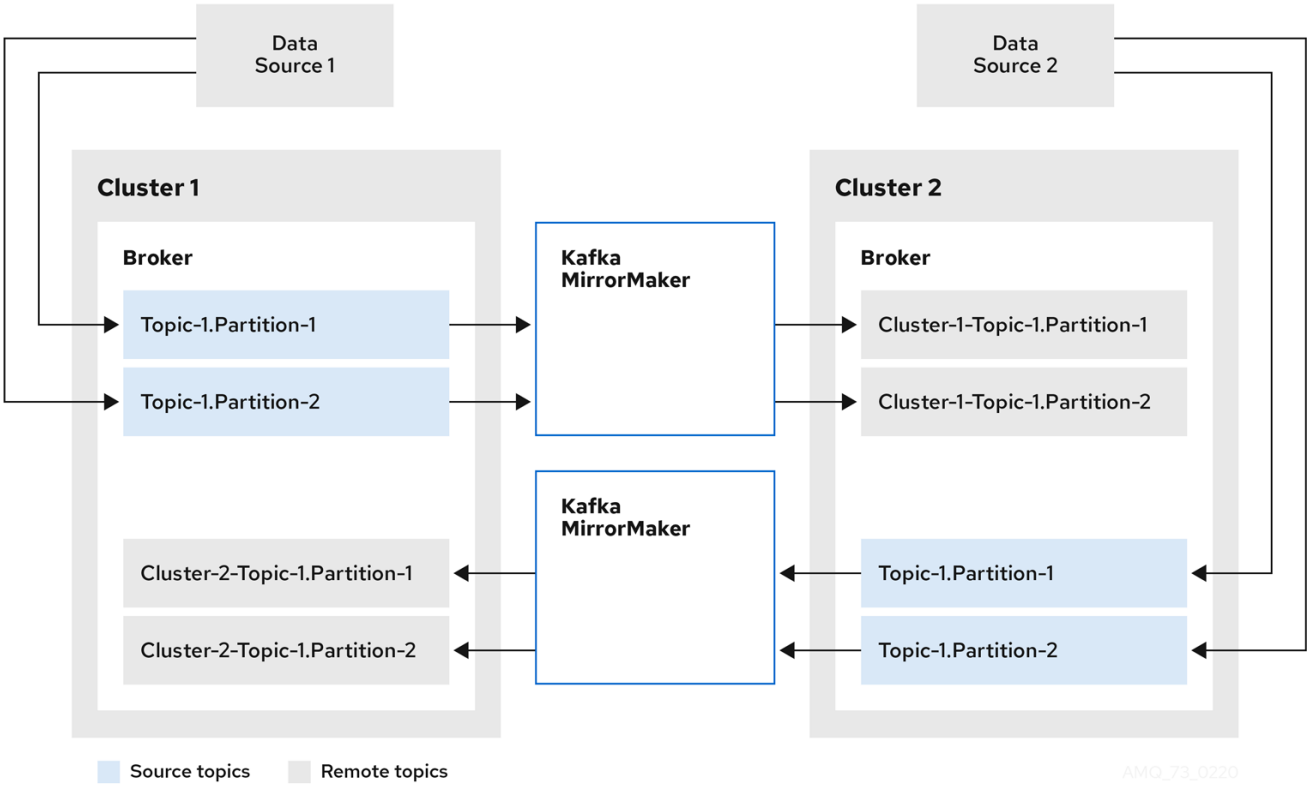
The expectation is that producers and consumers connect to active clusters only.

3.5.2.1. Bidirectional replication

The MirrorMaker 2.0 architecture supports bidirectional replication in an *active/active* cluster configuration. A MirrorMaker 2.0 cluster is required at each target destination.

Each cluster replicates the data of the other cluster using the concept of *source* and *remote* topics. As the same topics are stored in each cluster, remote topics are automatically renamed by MirrorMaker 2.0 to represent the source cluster.

Figure 3.2. Topic renaming



By flagging the originating cluster, topics are not replicated back to that cluster.

The concept of replication through *remote* topics is useful when configuring an architecture that requires data aggregation. Consumers can subscribe to source and remote topics within the same cluster, without the need for a separate aggregation cluster.

3.5.2.2. Topic configuration synchronization

Topic configuration is automatically synchronized between source and target clusters. By synchronizing configuration properties, the need for rebalancing is reduced.

3.5.2.3. Data integrity

MirrorMaker 2.0 monitors source topics and propagates any configuration changes to remote topics, checking for and creating missing partitions. Only MirrorMaker 2.0 can write to remote topics.

3.5.2.4. Offset tracking

MirrorMaker 2.0 tracks offsets for consumer groups using *internal topics*.

- The *offset sync* topic maps the source and target offsets for replicated topic partitions from record metadata
- The *checkpoint* topic maps the last committed offset in the source and target cluster for replicated topic partitions in each consumer group

Offsets for the *checkpoint* topic are tracked at predetermined intervals through configuration. Both topics enable replication to be fully restored from the correct offset position on failover.

MirrorMaker 2.0 uses its **MirrorCheckpointConnector** to emit *checkpoints* for offset tracking.

3.5.2.5. Connectivity checks

A *heartbeat* internal topic checks connectivity between clusters.

The *heartbeat* topic is replicated from the source cluster.

Target clusters use the topic to check:

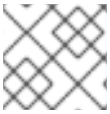
- The connector managing connectivity between clusters is running
- The source cluster is available

MirrorMaker 2.0 uses its **MirrorHeartbeatConnector** to emit *heartbeats* that perform these checks.

3.5.3. ACL rules synchronization

ACL access to remote topics is possible if you are **not** using the User Operator.

If **SimpleAclAuthorizer** is being used, without the User Operator, ACL rules that manage access to brokers also apply to remote topics. Users that can read a source topic can read its remote equivalent.



NOTE

OAuth 2.0 authorization does not support access to remote topics in this way.

3.5.4. Synchronizing data between Kafka clusters using MirrorMaker 2.0

Use MirrorMaker 2.0 to synchronize data between Kafka clusters through configuration.

The configuration must specify:

- Each Kafka cluster
- Connection information for each cluster, including TLS authentication
- The replication flow and direction
  - Cluster to cluster
  - Topic to topic

Use the properties of the **KafkaMirrorMaker2** resource to configure your Kafka MirrorMaker 2.0 deployment.

NOTE

The previous version of MirrorMaker continues to be supported. If you wish to use the resources configured for the previous version, they must be updated to the format supported by MirrorMaker 2.0.

MirrorMaker 2.0 provides default configuration values for properties such as replication factors. A minimal configuration, with defaults left unchanged, would be something like this example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker2
```

```
metadata:
  name: my-mirror-maker2
spec:
  version: 2.5.0
  connectCluster: "my-cluster-target"
  clusters:
  - alias: "my-cluster-source"
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
  - alias: "my-cluster-target"
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector: {}
```

You can configure access control for source and target clusters using TLS or SASL authentication. This procedure shows a configuration that uses TLS encryption and authentication for the source and target cluster.

Prerequisites

- AMQ Streams and Kafka is deployed
- Source and target Kafka clusters are available

Procedure

1. Edit the **spec** properties for the **KafkaMirrorMaker2** resource.
   The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 2.5.0 (1)
  replicas: 3 (2)
  connectCluster: "my-cluster-target" (3)
  clusters: (4)
  - alias: "my-cluster-source" (5)
    authentication: (6)
      certificateAndKey:
        certificate: source.crt
        key: source.key
        secretName: my-user-source
      type: tls
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092 (7)
    tls: (8)
      trustedCertificates:
      - certificate: ca.crt
        secretName: my-cluster-source-cluster-ca-cert
  - alias: "my-cluster-target" (9)
    authentication: (10)
      certificateAndKey:
```

```
      certificate: target.crt
      key: target.key
      secretName: my-user-target
    type: tls
  bootstrapServers: my-cluster-target-kafka-bootstrap:9092 (11)
  config: (12)
    config.storage.replication.factor: 1
    offset.storage.replication.factor: 1
    status.storage.replication.factor: 1
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (13)
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
  tls: (14)
    trustedCertificates:
    - certificate: ca.crt
      secretName: my-cluster-target-cluster-ca-cert
mirrors: (15)
- sourceCluster: "my-cluster-source" (16)
  targetCluster: "my-cluster-target" (17)
  sourceConnector: (18)
    config:
      replication.factor: 1 (19)
      offset-syncs.topic.replication.factor: 1 (20)
      sync.topic.acls.enabled: "false" (21)
  heartbeatConnector: (22)
    config:
      heartbeats.topic.replication.factor: 1 (23)
  checkpointConnector: (24)
    config:
      checkpoints.topic.replication.factor: 1 (25)
  topicsPattern: ".*" (26)
  groupsPattern: "group1|group2|group3" (27)
resources: (28)
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
logging: (29)
  type: inline
  loggers:
    connect.root.logger.level: "INFO"
readinessProbe: (30)
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
jvmOptions: (31)
  "-Xmx": "1g"
  "-Xms": "1g"
image: my-org/my-image:latest (32)
```

```
template: 33
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
  connectContainer: 34
    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 35
externalConfiguration: 36
  env:
    - name: AWS_ACCESS_KEY_ID
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: awsAccessKey
    - name: AWS_SECRET_ACCESS_KEY
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: awsSecretAccessKey
```

1. The Kafka Connect version.
2. The number of replica nodes.
3. The cluster alias for Kafka Connect.
4. Specification for the Kafka clusters being synchronized.
5. The cluster alias for the source Kafka cluster.
6. Authentication for the source cluster, using the TLS mechanism, as shown here, using OAuth bearer tokens, or a SASL-based SCRAM-SHA-512 or PLAIN mechanism.
7. Bootstrap server for connection to the source Kafka cluster.
8. TLS encryption with key names under which TLS certificates are stored in X.509 format for the source Kafka cluster. For more details see **KafkaMirrorMaker2Tls** schema reference.
9. The cluster alias for the target Kafka cluster.

10 Authentication for the target Kafka cluster is configured in the same way as for the source Kafka cluster.

11 Bootstrap server for connection to the target Kafka cluster.

12 Kafka Connect configuration. Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

13 SSL properties for external listeners to run with a specific *cipher suite* for a TLS version.

14 TLS encryption for the target Kafka cluster is configured in the same way as for the source Kafka cluster.

15 MirrorMaker 2.0 connectors.

16 The alias of the source cluster used by the MirrorMaker 2.0 connectors.

17 The alias of the target cluster used by the MirrorMaker 2.0 connectors.

18 The configuration for the **MirrorSourceConnector** that creates remote topics. The **config** overrides the default configuration options.

19 The replication factor for mirrored topics created at the target cluster.

20 The replication factor for the **MirrorSourceConnector offset-syncs** internal topic that maps the offsets of the source and target clusters.

21 When enabled, ACLs are applied to synchronized topics. The default is **true**.

22 The configuration for the **MirrorHeartbeatConnector** that performs connectivity checks. The **config** overrides the default configuration options.

23 The replication factor for the heartbeat topic created at the target cluster.

24 The configuration for the **MirrorCheckpointConnector** that tracks offsets. The **config** overrides the default configuration options.

25 The replication factor for the checkpoints topic created at the target cluster.

26 Topic replication from the source cluster defined as regular expression patterns. Here we request all topics.

27 Consumer group replication from the source cluster defined as regular expression patterns. Here we request three consumer groups by name. You can use comma-separated lists.

28 Requests for reservation of supported resources, currently **cpu** and **memory**, and limits to specify the maximum resources that can be consumed.

29 Specified loggers and log levels added directly (**inline**) or indirectly (**external**) through a ConfigMap. A custom ConfigMap must be placed under the **log4j.properties** or **log4j2.properties** key. Kafka Connect has a single logger called **connect.root.logger.level**. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

30 Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).

31 JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka MirrorMaker.

32 ADVANCED OPTION: Container image configuration, which is recommended only in special situations.

33 Template customization. Here a pod is scheduled based with anti-affinity, so the pod is not scheduled on nodes with the same hostname.

34 Environment variables are also set for distributed tracing using Jaeger.

35 Distributed tracing is enabled for Jaeger.

36 OpenShift Secret mounted to Kafka MirrorMaker as an environment variable.

2. Create or update the resource:

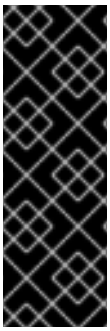
```
oc apply -f <your-file>
```

3.6. KAFKA BRIDGE CONFIGURATION

The full schema of the **KafkaBridge** resource is described in the Section B.115, "**KafkaBridge** schema reference". All labels that are applied to the desired **KafkaBridge** resource will also be applied to the OpenShift resources making up the Kafka Bridge cluster. This provides a convenient mechanism for resources to be labeled as required.

3.6.1. Replicas

Kafka Bridge can run multiple nodes. The number of nodes is defined in the **KafkaBridge** resource. Running a Kafka Bridge with multiple nodes can provide better availability and scalability. However, when running Kafka Bridge on OpenShift it is not absolutely necessary to run multiple nodes of Kafka Bridge for high availability.



IMPORTANT

If a node where Kafka Bridge is deployed to crashes, OpenShift will automatically reschedule the Kafka Bridge pod to a different node. In order to prevent issues arising when client consumer requests are processed by different Kafka Bridge instances, addressed-based routing must be employed to ensure that requests are routed to the right Kafka Bridge instance. Additionally, each independent Kafka Bridge instance must have a replica. A Kafka Bridge instance has its own state which is not shared with another instances.

3.6.1.1. Configuring the number of nodes

The number of Kafka Bridge nodes is configured using the **replicas** property in **KafkaBridge.spec**.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **replicas** property in the **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  replicas: 3
  # ...
```

2. Create or update the resource.

```
oc apply -f your-file
```

3.6.2. Bootstrap servers

A Kafka Bridge always works in combination with a Kafka cluster. A Kafka cluster is specified as a list of bootstrap servers. On OpenShift, the list must ideally contain the Kafka cluster bootstrap service named ***cluster-name*-kafka-bootstrap**, and a port of 9092 for plain traffic or 9093 for encrypted traffic.

The list of bootstrap servers is configured in the **bootstrapServers** property in **KafkaBridge.kafka.spec**. The servers must be defined as a comma-separated list specifying one or more Kafka brokers, or a service pointing to Kafka brokers specified as a ***hostname*:_port_** pairs.

When using Kafka Bridge with a Kafka cluster not managed by AMQ Streams, you can specify the bootstrap servers list according to the configuration of the cluster.

3.6.2.1. Configuring bootstrap servers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **bootstrapServers** property in the **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  # ...
```

2. Create or update the resource.

```
oc apply -f your-file
```

-

3.6.3. Connecting to Kafka brokers using TLS

By default, Kafka Bridge tries to connect to Kafka brokers using a plain text connection. If you prefer to use TLS, additional configuration is required.

3.6.3.1. TLS support for Kafka connection to the Kafka Bridge

TLS support for Kafka connection is configured in the **tls** property in **KafkaBridge.spec**. The **tls** property contains a list of secrets with key names under which the certificates are stored. The certificates must be stored in X509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  tls:
    trustedCertificates:
    - secretName: my-secret
      certificate: ca.crt
    - secretName: my-other-secret
      certificate: certificate.crt
  # ...
```

When multiple certificates are stored in the same secret, it can be listed multiple times.

An example showing TLS configuration with multiple certificates from the same secret

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  tls:
    trustedCertificates:
    - secretName: my-secret
      certificate: ca.crt
    - secretName: my-secret
      certificate: ca2.crt
  # ...
```

3.6.3.2. Configuring TLS in Kafka Bridge

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

- If they exist, the name of the **Secret** for the certificate used for TLS Server Authentication, and the key under which the certificate is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the TLS certificate used in authentication in a file and create a **Secret**.

NOTE

The secrets created by the Cluster Operator for Kafka cluster may be used directly.

```
oc create secret generic my-secret --from-file=my-file.crt
```

2. Edit the **tls** property in the **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  tls:
    trustedCertificates:
    - secretName: my-cluster-cluster-cert
      certificate: ca.crt
  # ...
```

3. Create or update the resource.

```
oc apply -f your-file
```

3.6.4. Connecting to Kafka brokers with Authentication

By default, Kafka Bridge will try to connect to Kafka brokers without authentication. Authentication is enabled through the **KafkaBridge** resources.

3.6.4.1. Authentication support in Kafka Bridge

Authentication is configured through the **authentication** property in **KafkaBridge.spec**. The **authentication** property specifies the type of the authentication mechanisms which should be used and additional configuration details depending on the mechanism. The currently supported authentication types are:

- TLS client authentication
- SASL-based authentication using the SCRAM-SHA-512 mechanism
- SASL-based authentication using the PLAIN mechanism
- OAuth 2.0 token based authentication

3.6.4.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.

NOTE

TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Bridge see Section 3.6.3, "Connecting to Kafka brokers using TLS".

An example TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  # ...
```

3.6.4.1.2. SCRAM-SHA-512 authentication

To configure Kafka Bridge to use SASL-based SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. This authentication mechanism requires a username and password.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: scram-sha-512
```

```
    username: my-bridge-user
    passwordSecret:
      secretName: my-bridge-user
      password: my-bridge-password-key
  # ...
```

3.6.4.1.3. SASL-based PLAIN authentication

To configure Kafka Bridge to use SASL-based PLAIN authentication, set the **type** property to **plain**. This authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example showing SASL based PLAIN client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: plain
    username: my-bridge-user
    passwordSecret:
      secretName: my-bridge-user
      password: my-bridge-password-key
  # ...
```

3.6.4.2. Configuring TLS client authentication in Kafka Bridge

Prerequisites

- An OpenShift cluster

- A running Cluster Operator
- If they exist, the name of the **Secret** with the public and private keys used for TLS Client Authentication, and the keys under which they are stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the keys used for authentication in a file and create the **Secret**.

NOTE

Secrets created by the User Operator may be used.

```
oc create secret generic my-secret --from-file=my-public.crt --from-file=my-private.key
```

2. Edit the **authentication** property in the **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: my-public.crt
    key: my-private.key
  # ...
```

3. Create or update the resource.

```
oc apply -f your-file
```

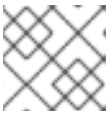
3.6.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Bridge

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- Username of the user which should be used for authentication
- If they exist, the name of the **Secret** with the password used for authentication and the key under which the password is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare a file with the password used in authentication and create the **Secret**.



NOTE

Secrets created by the User Operator may be used.

```
echo -n '<password>' > <my-password.txt>
oc create secret generic <my-secret> --from-file=<my-password.txt>
```

2. Edit the **authentication** property in the **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: <my-username>
    passwordSecret:
      secretName: <my-secret>
      password: <my-password.txt>
  # ...
```

3. Create or update the resource.

```
oc apply -f your-file
```

3.6.5. Kafka Bridge configuration

AMQ Streams allows you to customize the configuration of Apache Kafka Bridge nodes by editing certain options listed in Apache Kafka configuration documentation for consumers and Apache Kafka configuration documentation for producers.

Configuration options that can be configured relate to:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Consumer configuration
- Producer configuration
- HTTP configuration

3.6.5.1. Kafka Bridge Consumer configuration

Kafka Bridge consumer is configured using the properties in **KafkaBridge.spec.consumer**. This property contains the Kafka Bridge consumer configuration options as keys. The values can be one of the following JSON types:

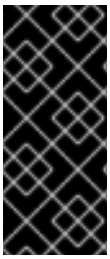
- String
- Number

- Boolean

Users can specify and configure the options listed in the Apache Kafka configuration documentation for consumers with the exception of those options which are managed directly by AMQ Streams. Specifically, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **ssl.**
- **sasl.**
- **security.**
- **bootstrap.servers**
- **group.id**

When one of the forbidden options is present in the **config** property, it will be ignored and a warning message will be printed to the Cluster Operator log file. All other options will be passed to Kafka



IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Bridge cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaBridge.spec.consumer.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Bridge nodes.

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example Kafka Bridge consumer configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  consumer:
    config:
      auto.offset.reset: earliest
      enable.auto.commit: true
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 1
      ssl.enabled.protocols: "TLSv1.2" 2
      ssl.protocol: "TLSv1.2" 3
    # ...
```

1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

3.6.5.2. Kafka Bridge Producer configuration

Kafka Bridge producer is configured using the properties in **KafkaBridge.spec.producer**. This property contains the Kafka Bridge producer configuration options as keys. The values can be one of the following JSON types:

- String
- Number
- Boolean

Users can specify and configure the options listed in the Apache Kafka configuration documentation for producers with the exception of those options which are managed directly by AMQ Streams. Specifically, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **ssl.**
- **sasl.**
- **security.**
- **bootstrap.servers**

IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Bridge cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaBridge.spec.producer.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Bridge nodes.

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example Kafka Bridge producer configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  producer:
    config:
      acks: 1
      delivery.timeout.ms: 300000
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 1
      ssl.enabled.protocols: "TLSv1.2" 2
      ssl.protocol: "TLSv1.2" 3
    # ...
```

1 The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.

2 The SSl protocol **TLSv1.2** is enabled.

3 Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

3.6.5.3. Kafka Bridge HTTP configuration

Kafka Bridge HTTP configuration is set using the properties in **KafkaBridge.spec.http**.

As well as enabling HTTP access to a Kafka cluster, HTTP properties provide the capability to enable and define access control for the Kafka Bridge through Cross-Origin Resource Sharing (CORS). CORS is a HTTP mechanism that allows browser access to selected resources from more than one origin. To configure CORS, you define a list of allowed resource origins and HTTP methods to access them. Additional HTTP headers in requests describe the origins that are permitted access to the Kafka cluster .

Example Kafka Bridge HTTP configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  http:
    port: 8080 1
    cors:
      allowedOrigins: "https://strimzi.io" 2
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH" 3
  # ...
```

1 The default HTTP configuration for the Kafka Bridge to listen on port 8080.

2 Comma-separated list of allowed CORS origins. You can use a URL or a Java regular expression.

3 Comma-separated list of allowed HTTP methods for CORS.

3.6.5.4. Configuring Kafka Bridge

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **kafka**, **http**, **consumer** or **producer** property in the **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
```

```
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    port: 8080
  consumer:
    config:
      auto.offset.reset: earliest
  producer:
    config:
      delivery.timeout.ms: 300000
  # ...
```

2. Create or update the resource.

```
oc apply -f your-file
```

3.6.6. CPU and memory resources

For every deployed container, AMQ Streams allows you to request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

AMQ Streams uses the OpenShift syntax for specifying CPU and memory resources.

3.6.6.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

Additional resources

- For more information about managing computing resources on OpenShift, see Managing Compute Resources for Containers.

3.6.6.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.

IMPORTANT

If the resource request is for more than the available free resources in the OpenShift cluster, the pod is not scheduled.

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**
- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

```
# ...
resources:
  requests:
    cpu: 12
    memory: 64Gi
# ...
```

3.6.6.1.2. Resource limits

Limits specify the maximum resources that can be consumed by a given container. The limit is not reserved and might not always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

```
# ...
resources:
  limits:
```

```
    cpu: 12
    memory: 64Gi
# ...
```

3.6.6.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

- Number of CPU cores as integer (**5** CPU core) or decimal (**2.5** CPU core).
- Number or *millicpus* / *millicores* (**100m**) where 1000 *millicores* is the same **1** CPU core.

Example CPU units

```
# ...
resources:
  requests:
    cpu: 500m
  limits:
    cpu: 2.5
# ...
```

NOTE

The computing power of 1 CPU core may differ depending on the platform where OpenShift is deployed.

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

3.6.6.1.4. Supported memory formats

Memory requests and limits are specified in megabytes, gigabytes, mebibytes, and gibibytes.

- To specify memory in megabytes, use the **M** suffix. For example **1000M**.
- To specify memory in gigabytes, use the **G** suffix. For example **1G**.
- To specify memory in mebibytes, use the **Mi** suffix. For example **1000Mi**.
- To specify memory in gibibytes, use the **Gi** suffix. For example **1Gi**.

An example of using different memory units

```
# ...
resources:
  requests:
    memory: 512Mi
  limits:
    memory: 2Gi
# ...
```

Additional resources

- For more details about memory specification and additional supported units, see Meaning of memory.

3.6.6.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **resources** property in the resource specifying the cluster deployment. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema, see {K8sResourceRequirementsAPI}.

3.6.7. Kafka Bridge loggers

Kafka Bridge has its own configurable loggers:

- **log4j.logger.io.strimzi.kafka.bridge**
- **log4j.logger.http.openapi.operation.*<operation-id>***

You can replace ***<operation-id>*** in the **log4j.logger.http.openapi.operation.*<operation-id>*** logger to set log levels for specific operations:

- **createConsumer**
- **deleteConsumer**

- **subscribe**
- **unsubscribe**
- **poll**
- **assign**
- **commit**
- **send**
- **sendToPartition**
- **seekToBeginning**
- **seekToEnd**
- **seek**
- **healthy**
- **ready**
- **openapi**

Each operation is defined according OpenAPI specification, and has a corresponding API endpoint through which the bridge receives requests from HTTP clients. You can change the log level on each endpoint to create fine-grained logging information about the incoming and outgoing HTTP requests.

Kafka Bridge uses the Apache **log4j** logger implementation. Loggers are defined in the **log4j.properties** file, which has the following default configuration for **healthy** and **ready** endpoints:

```
log4j.logger.http.openapi.operation.healthy=WARN, out
log4j.additivity.http.openapi.operation.healthy=false
log4j.logger.http.openapi.operation.ready=WARN, out
log4j.additivity.http.openapi.operation.ready=false
```

The log level of all other operations is set to **INFO** by default.

Use the **logging** property to configure loggers and logger levels.

You can set the log levels by specifying the logger and level directly (inline) or use a custom (external) ConfigMap. If a ConfigMap is used, you set **logging.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**.

Here we see examples of **inline** and **external** logging.

Inline logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaBridge
spec:
  # ...
  logging:
    type: inline
```

```
    loggers:
      log4j.logger.io.strimzi.kafka.bridge: "INFO"
  # ...
```

External logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaBridge
spec:
  # ...
  logging:
    type: external
    name: customConfigMap
  # ...
```

Additional resources

- Garbage collector (GC) logging can also be enabled (or disabled). For more information about GC logging, see Section 3.1.18.1, “JVM configuration”
- For more information about log levels, see Apache logging services.

3.6.8. JVM Options

The following components of AMQ Streams run inside a Virtual Machine (VM):

- Apache Kafka
- Apache ZooKeeper
- Apache Kafka Connect
- Apache Kafka MirrorMaker
- AMQ Streams Kafka Bridge

JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure some of these options.

3.6.8.1. JVM configuration

JVM options can be configured using the **jvmOptions** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaBridge.spec**

Only a selected subset of available JVM options can be configured. The following options are supported:

-Xms and -Xmx

-Xms configures the minimum initial allocation heap size when the JVM starts. **-Xmx** configures the maximum heap size.

NOTE

The units accepted by JVM settings such as **-Xmx** and **-Xms** are those accepted by the JDK **java** binary in the corresponding image. Accordingly, **1g** or **1G** means 1,073,741,824 bytes, and **Gi** is not a valid unit suffix. This is in contrast to the units used for memory requests and limits, which follow the OpenShift convention where **1G** means 1,000,000,000 bytes, and **1Gi** means 1,073,741,824 bytes

The default values used for **-Xms** and **-Xmx** depends on whether there is a memory request limit configured for the container:

- If there is a memory limit then the JVM's minimum and maximum memory will be set to a value corresponding to the limit.
- If there is no memory limit then the JVM's minimum memory will be set to **128M** and the JVM's maximum memory will not be defined. This allows for the JVM's memory to grow as-needed, which is ideal for single node environments in test and development.

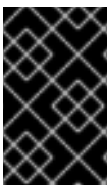
IMPORTANT

Setting **-Xmx** explicitly requires some care:

- The JVM's overall memory usage will be approximately 4 × the maximum heap, as configured by **-Xmx**.
- If **-Xmx** is set without also setting an appropriate OpenShift memory limit, it is possible that the container will be killed should the OpenShift node experience memory pressure (from other Pods running on it).
- If **-Xmx** is set without also setting an appropriate OpenShift memory request, it is possible that the container will be scheduled to a node with insufficient memory. In this case, the container will not start but crash (immediately if **-Xms** is set to **-Xmx**, or some later time if not).

When setting **-Xmx** explicitly, it is recommended to:

- set the memory request and the memory limit to the same value,
- use a memory request that is at least 4.5 × the **-Xmx**,
- consider setting **-Xms** to the same value as **-Xmx**.



IMPORTANT

Containers doing lots of disk I/O (such as Kafka broker containers) will need to leave some memory available for use as operating system page cache. On such containers, the requested memory should be significantly higher than the memory used by the JVM.

Example fragment configuring -Xmx and -Xms

```
# ...
jvmOptions:
  "-Xmx": "2g"
  "-Xms": "2g"
# ...
```

In the above example, the JVM will use 2 GiB (=2,147,483,648 bytes) for its heap. Its total memory usage will be approximately 8GiB.

Setting the same value for initial (**-Xms**) and maximum (**-Xmx**) heap sizes avoids the JVM having to allocate memory after startup, at the cost of possibly allocating more heap than is really needed. For Kafka and ZooKeeper pods such allocation could cause unwanted latency. For Kafka Connect avoiding over allocation may be the most important concern, especially in distributed mode where the effects of over-allocation will be multiplied by the number of consumers.

-server

-server enables the server JVM. This option can be set to true or false.

Example fragment configuring -server

```
# ...
jvmOptions:
  "-server": true
# ...
```

NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

-XX

-XX object can be used for configuring advanced runtime options of a JVM. The **-server** and **-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

Example showing the use of the -XX object

```
jvmOptions:
  "-XX":
    "UseG1GC": true
    "MaxGCPauseMillis": 20
    "InitiatingHeapOccupancyPercent": 35
    "ExplicitGCInvokesConcurrent": true
    "UseParNewGC": false
```

The example configuration above will result in the following JVM options:

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -
XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```

NOTE

When neither of the two options (**-server** and **-XX**) is specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** will be used.

3.6.8.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is disabled by default. To enable it, set the **gcLoggingEnabled** property as follows:

Example of enabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: true
# ...
```

3.6.8.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **jvmOptions** property in the **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, **KafkaMirrorMaker**, or **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jvmOptions:
      "-Xmx": "8g"
      "-Xms": "8g"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.6.9. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts to fix it.

OpenShift supports two types of Healthcheck probes:

- Liveness probes
- Readiness probes

For more details about the probes, see Configure Liveness and Readiness Probes. Both types of probes are used in AMQ Streams components.

Users can configure selected options for liveness and readiness probes.

3.6.9.1. Healthcheck configurations

Liveness and readiness probes can be configured using the **livenessProbe** and **readinessProbe** properties in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.KafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaBridge.spec**

Both **livenessProbe** and **readinessProbe** support the following options:

- **initialDelaySeconds**
- **timeoutSeconds**
- **periodSeconds**
- **successThreshold**
- **failureThreshold**

For more information about the **livenessProbe** and **readinessProbe** options, see Section B.40, "**Probe** schema reference".

An example of liveness and readiness probe configuration

```
# ...
readinessProbe:
```

```
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  # ...
```

3.6.9.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **livenessProbe** or **readinessProbe** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    readinessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.6.10. Container images

AMQ Streams allows you to configure container images which will be used for its components. Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such a case, you should either copy the AMQ Streams images or build them from the source. If the configured image is not compatible with AMQ Streams images, it might not work properly.

3.6.10.1. Container image configurations

You can specify which container image to use for each component using the **image** property in the following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

3.6.10.1.1. Configuring the **image** property for Kafka, Kafka Connect, and Kafka MirrorMaker

Kafka, Kafka Connect (including Kafka Connect with S2I support), and Kafka MirrorMaker support multiple versions of Kafka. Each component requires its own image. The default images for the different Kafka versions are configured in the following environment variables:

- **STRIMZI_KAFKA_IMAGES**
- **STRIMZI_KAFKA_CONNECT_IMAGES**
- **STRIMZI_KAFKA_CONNECT_S2I_IMAGES**
- **STRIMZI_KAFKA_MIRROR_MAKER_IMAGES**

These environment variables contain mappings between the Kafka versions and their corresponding images. The mappings are used together with the **image** and **version** properties:

- If neither **image** nor **version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the environment variable.
- If **image** is given but **version** is not, then the given image is used and the **version** is assumed to be the Cluster Operator's default Kafka version.
- If **version** is given but **image** is not, then the image that corresponds to the given version in the environment variable is used.
- If both **version** and **image** are given, then the given image is used. The image is assumed to contain a Kafka image with the given version.

The **image** and **version** for the different components can be configured in the following properties:

- For Kafka in **spec.kafka.image** and **spec.kafka.version**.
- For Kafka Connect, Kafka Connect S2I, and Kafka MirrorMaker in **spec.image** and **spec.version**.

WARNING

It is recommended to provide only the **version** and leave the **image** property unspecified. This reduces the chance of making a mistake when configuring the custom resource. If you need to change the images used for different versions of Kafka, it is preferable to configure the Cluster Operator's environment variables.

3.6.10.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property is missing, the **image** specified in the Cluster Operator configuration will be used. If the **image** name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Exporter:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0** container image.
- For Kafka Bridge:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE** environment variable from the Cluster Operator configuration.

2. **registry.redhat.io/amq7/amq-streams-bridge-rhel7:1.5.0** container image.

- For Kafka broker initializer:

1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** environment variable from the Cluster Operator configuration.

2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0** container image.

WARNING

Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such case, you should either copy the AMQ Streams images or build them from source. In case the configured image is not compatible with AMQ Streams images, it might not work properly.

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.6.10.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **image** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
```

```
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.6.11. Configuring pod scheduling

IMPORTANT

When two applications are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.6.11.1. Scheduling pods based on other applications

3.6.11.1.1. Avoid critical applications to share the node

Pod anti-affinity can be used to ensure that critical applications are never scheduled on the same disk. When running Kafka cluster, it is recommended to use pod anti-affinity to ensure that the Kafka brokers do not share the nodes with other workloads like databases.

3.6.11.1.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.6.11.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **affinity** property in the resource specifying the cluster deployment. Use labels to specify the pods which should not be scheduled on the same nodes. The **topologyKey** should be set to **kubernetes.io/hostname** to specify that the selected pods should not be scheduled on nodes with the same hostname. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.6.11.2. Scheduling pods to specific nodes

3.6.11.2.1. Node scheduling

The OpenShift cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while other might be optimized for storage (fast local SSDs) or network. Using different nodes helps to optimize both costs and performance. To achieve the best possible performance, it is important to allow scheduling of AMQ Streams components to use the right nodes.

OpenShift uses node affinity to schedule workloads onto specific nodes. Node affinity allows you to create a scheduling constraint for the node on which the pod will be scheduled. The constraint is specified as a label selector. You can specify the label using either the built-in node label like **beta.kubernetes.io/instance-type** or custom labels to select the right node.

3.6.11.2.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.6.11.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   This can be done using **oc label**:

   ```
   oc label node your-node node-type=fast-network
   ```

   Alternatively, some of the existing labels might be reused.

2. Edit the **affinity** property in the resource specifying the cluster deployment. For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
   ```

```
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: node-type
                  operator: In
                  values:
                  - fast-network
  # ...
zookeeper:
  # ...
```

3. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.6.11.3. Using dedicated nodes

3.6.11.3.1. Dedicated nodes

Cluster administrators can mark selected OpenShift nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

3.6.11.3.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity

- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation .

3.6.11.3.3. Tolerations

Tolerations can be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

The format of the **tolerations** property follows the OpenShift specification. For more details, see the Kubernetes taints and tolerations .

3.6.11.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated.
2. Make sure there are no workloads scheduled on these nodes.
3. Set the taints on the selected nodes:
   This can be done using **oc adm taint**:

   ```
   oc adm taint node your-node dedicated=Kafka:NoSchedule
   ```

4. Additionally, add a label to the selected nodes as well.
   This can be done using **oc label**:

   ```
   oc label node your-node dedicated=Kafka
   ```

5. Edit the **affinity** and **tolerations** properties in the resource specifying the cluster deployment.
   For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: Kafka
   spec:
   ```

```
kafka:
  # ...
  template:
    pod:
      tolerations:
        - key: "dedicated"
          operator: "Equal"
          value: "Kafka"
          effect: "NoSchedule"
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: dedicated
                operator: In
                values:
                - Kafka
  # ...
zookeeper:
  # ...
```

6. Create or update the resource.
   This can be done using **oc apply**:

```
oc apply -f your-file
```

3.6.12. List of resources created as part of Kafka Bridge cluster

The following resources are created by the Cluster Operator in the OpenShift cluster:

***bridge-cluster-name*-bridge**

Deployment which is in charge to create the Kafka Bridge worker node pods.

***bridge-cluster-name*-bridge-service**

Service which exposes the REST interface of the Kafka Bridge cluster.

***bridge-cluster-name*-bridge-config**

ConfigMap which contains the Kafka Bridge ancillary configuration and is mounted as a volume by the Kafka broker pods.

***bridge-cluster-name*-bridge**

Pod Disruption Budget configured for the Kafka Bridge worker nodes.

3.7. USING OAUTH 2.0 TOKEN-BASED AUTHENTICATION

AMQ Streams supports the use of OAuth 2.0 authentication using the *SASL OAUTHBEARER* mechanism.

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. OAuth 2.0 authentication can also be used in conjunction with ACL-based Kafka authorization regardless of the authorization server used.

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 Mirror Maker, Kafka Connect and the Kafka Bridge

Additional resources

- OAuth 2.0 site

3.7.1. OAuth 2.0 authentication mechanism

The Kafka *SASL OAUTHBEARER* mechanism is used to establish authenticated sessions with a Kafka broker.

A Kafka client initiates a session with the Kafka broker using the *SASL OAUTHBEARER* mechanism for credentials exchange, where credentials take the form of an access token.

Kafka brokers and clients need to be configured to use OAuth 2.0.

3.7.2. 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

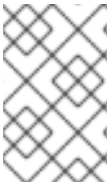
NOTE

In relation to the authorization server, Kafka brokers and Kafka clients are both regarded as OAuth 2.0 clients.

3.7.2.1. 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.

3.7.2.2. 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/v1beta1
kind: Kafka
spec:
  kafka:
    listeners:
      tls:
        authentication:
          type: oauth
          #...
```

You can configure **plain**, **tls** and **external** listeners, as described in Kafka broker listeners, but it is recommended not to use **plain** listeners or **external** listeners with disabled TLS encryption with OAuth 2.0 as this creates a vulnerability 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:
  tls:
    authentication:
      type: oauth
  external:
    type: loadbalancer
    tls: true
    authentication:
      type: oauth
    #...
```

The **tls** property is *true* by default, so it can be left out.

When you've 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 .

3.7.2.3. 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 **validIssuerUrl** 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/v1beta1
kind: Kafka
spec:
  kafka:
    listeners:
      tls:
        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
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
```

3.7.2.4. 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/v1beta1
kind: Kafka
spec:
  kafka:
    listeners:
      tls:
        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
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
```

3.7.3. 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

3.7.4. OAuth 2.0 client authentication flow

In this section, we explain and visualize the communication flow between Kafka client, Kafka broker, and authorization server during Kafka session initiation. The flow depends on the client and server configuration.

When a Kafka client sends an access token as credentials to a Kafka broker, the token needs to be validated.

Depending on the authorization server used, and the configuration options available, you may prefer to use:

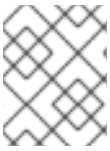
- Fast local token validation based on JWT signature checking and local token introspection, without contacting the authorization server
- An OAuth 2.0 introspection endpoint provided by the authorization server

Using fast local token validation requires the authorization server to provide a JWKS endpoint with public certificates that are used to validate signatures on the tokens.

Another option is to use an OAuth 2.0 introspection endpoint on the 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, and checks the response to confirm whether or not the token is valid.

Kafka client credentials can also be configured for:

- Direct local access using a previously generated long-lived access token
- Contact with the authorization server for a new access token to be issued



NOTE

An authorization server might only allow the use of opaque access tokens, which means that local token validation is not possible.

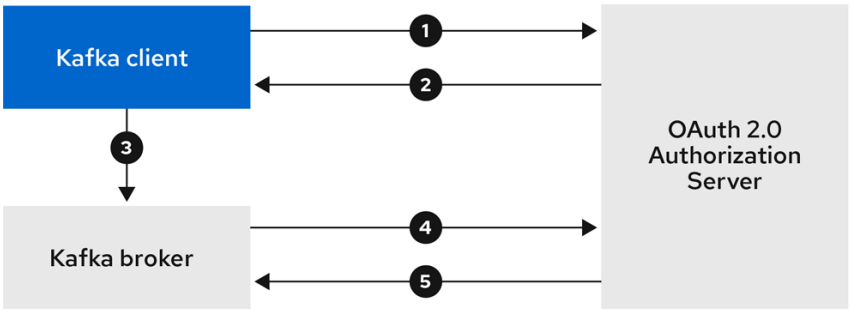
3.7.4.1. Example client authentication flows

Here you can see the communication flows, for different configurations of Kafka clients and brokers, during Kafka session authentication.

- 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

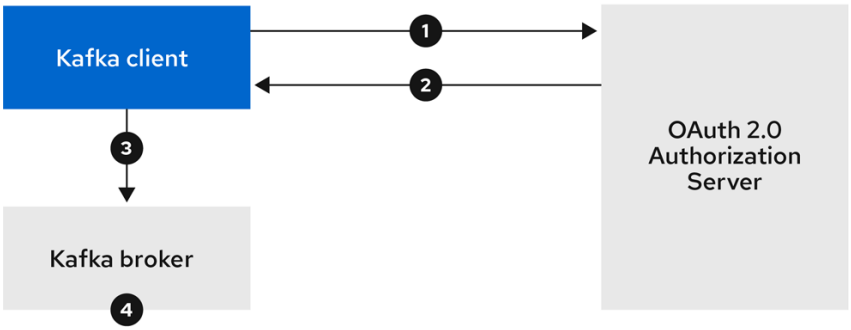
- Client using long-lived access token, with broker delegating validation to authorization server
- Client using long-lived access token, with broker performing fast local validation

Client using client ID and secret, with broker delegating validation to authorization server



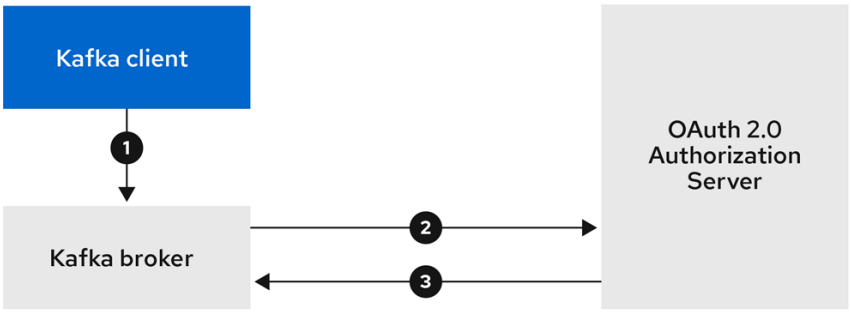
1. Kafka client requests access token from authorization server, using client ID and secret, and optionally a refresh token.
2. Authorization server generates a new access token.
3. Kafka client authenticates with the Kafka broker using the *SASL OAUTHBEARER* mechanism to pass the access token.
4. Kafka broker validates the access token by calling a token introspection endpoint on authorization server, using its own client ID and secret.
5. Kafka client session is established if the token is valid.

Client using client ID and secret, with broker performing fast local token validation



1. Kafka client authenticates with authorization server from the token endpoint, using a client ID and secret, and optionally a refresh token.
2. Authorization server generates a new access token.
3. Kafka client authenticates with the Kafka broker using the *SASL OAUTHBEARER* mechanism to pass the access token.
4. 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. Kafka client authenticates with the Kafka broker using the *SASL OAUTHBEARER* mechanism to pass the long-lived access token.
2. Kafka broker validates the access token by calling a token introspection endpoint on authorization server, using its own client ID and secret.
3. Kafka client session is established if the token is valid.

Client using long-lived access token, with broker performing fast local validation

1. Kafka client authenticates with the Kafka broker using the *SASL OAUTHBEARER* mechanism to pass the long-lived access token.
2. Kafka broker validates the access token locally using 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.

3.7.5. 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

3.7.5.1. Configuring Red Hat Single Sign-On as an OAuth 2.0 authorization server

This procedure describes how to deploy Red Hat Single Sign-On as an authorization server and configure it for integration with AMQ Streams.

The authorization server provides a central point for authentication and authorization, and management of users, clients, and permissions. Red Hat Single Sign-On has a concept of realms where a *realm* represents a separate set of users, clients, permissions, and other configuration. You can use a default *master realm*, or create a new one. Each realm exposes its own OAuth 2.0 endpoints, which means that application clients and application servers all need to use the same realm.

To use OAuth 2.0 with AMQ Streams, you use a deployment of Red Hat Single Sign-On to create and manage authentication realms.

NOTE

If you already have Red Hat Single Sign-On deployed, you can skip the deployment step and use your current deployment.

Before you begin

You will need to be familiar with using Red Hat Single Sign-On.

For deployment and administration instructions, see:

- Red Hat Single Sign-On for OpenShift
- Server Administration Guide

Prerequisites

- AMQ Streams and Kafka is running

For the Red Hat Single Sign-On deployment:

- Check the Red Hat Single Sign-On Supported Configurations
- Installation requires a user with a cluster-admin role, such as system:admin

Procedure

1. Deploy Red Hat Single Sign-On to your OpenShift cluster.
   Check the progress of the deployment in your OpenShift web console.
2. Log in to the Red Hat Single Sign-On Admin Console to create the OAuth 2.0 policies for AMQ Streams.
   Login details are provided when you deploy Red Hat Single Sign-On.

3. Create and enable a realm.
   You can use an existing master realm.
4. Adjust the session and token timeouts for the realm, if required.
5. Create a client called **kafka-broker**.
6. From the **Settings** tab, set:
   - **Access Type** to **Confidential**
   - **Standard Flow Enabled** to **OFF** to disable web login for this client
   - **Service Accounts Enabled** to **ON** to allow this client to authenticate in its own name
7. Click **Save** before continuing.
8. From the **Credentials** tab, take a note of the secret for using in your AMQ Streams Kafka cluster configuration.
9. Repeat the client creation steps for any application client that will connect to your Kafka brokers.
   Create a definition for each new client.

   You will use the names as client IDs in your configuration.

What to do next

After deploying and configuring the authorization server, configure the Kafka brokers to use OAuth 2.0 .

3.7.5.2. 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 configuration of TLS listeners. 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 have two connection options for your listener configuration when delegating token validation to the authorization server:

- 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
- Kafka broker listeners
- Authentication and Authorization

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

```
external:
  type: loadbalancer
  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)
    tlsTrustedCertificates: (5)
    - secretName: oauth-server-cert
      certificate: ca.crt
    disableTlsHostnameVerification: true (6)
    jwksExpirySeconds: 360 (7)
    jwksRefreshSeconds: 300 (8)
    enableECDSA: "true" (9)
```

(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) Trusted certificates for TLS connection to the authorization server.

(6) (Optional) Disable TLS hostname verification. Default is **false**.

(7) 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.

(8) The period between refreshes of JWKs certificates. The interval must be at least 60 seconds shorter than the expiry interval. Default is **300** seconds.

9 (Optional) If ECDSA is used for signing JWT tokens on authorization server, then this needs to be enabled. It installs additional crypto providers using BouncyCastle crypto

Example 2: Configuring token validation using an introspection endpoint

```
external:
  type: loadbalancer
  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
```

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.

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
  fallbackUserNameClaim: client_id 2
  fallbackUserNamePrefix: client-account- 3
  validTokenType: bearer 4
  userInfoEndpointUri: https://OAUTH-SERVER-
ADDRESS/auth/realms/external/protocol/openid-connect/userinfo 5
```

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 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.

3 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

4 (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.

5 (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.

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 po -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

3.7.5.3. Configuring Kafka Java clients to use OAuth 2.0

This procedure describes how to configure Kafka producer and consumer APIs to use OAuth 2.0 for interaction with Kafka brokers.

Add a client callback plugin to your *pom.xml* file, and configure the system properties.

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>0.5.0.redhat-00001</version>
</dependency>
```

2. Configure the system properties for the callback:

For example:

```
System.setProperty(ClientConfig.OAUTH_TOKEN_ENDPOINT_URI, “https://<auth-server-address>/auth/realms/master/protocol/openid-connect/token”); (1)
System.setProperty(ClientConfig.OAUTH_CLIENT_ID, "<client-name>"); (2)
System.setProperty(ClientConfig.OAUTH_CLIENT_SECRET, "<client-secret>"); (3)
```

1. URI of the authorization server token endpoint.
2. Client ID, which is the name used when creating the *client* in the authorization server.
3. Client secret created when creating the *client* in the authorization server.

3. Enable the *SASL OAUTHBEARER* mechanism on a TLS encrypted connection in the Kafka client configuration:
For example:

```
props.put("sasl.jaas.config",
"org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required;");
props.put("security.protocol", "SASL_SSL"); (1)
props.put("sasl.mechanism", "OAUTHBEARER");
props.put("sasl.login.callback.handler.class",
"io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler");
```

1. Here we use **SASL_SSL** for use over TLS connections. Use **SASL_PLAINTEXT** over unencrypted connections.

4. Verify that the Kafka client can access the Kafka brokers.

What to do next

- Configure Kafka components to use OAuth 2.0

3.7.5.4. 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:

- KafkaClientAuthenticationOAuth schema reference

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/v1beta1
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:

   - Client ID and secret
   - Client ID and refresh token
   - Access token
   - TLS

   KafkaClientAuthenticationOAuth schema reference provides examples of each .

   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/v1beta1
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
```

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**.

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.

3.8. USING OAUTH 2.0 TOKEN-BASED AUTHORIZATION

IMPORTANT

OAuth 2.0 authorization is a Technology Preview only. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope.

Authorizing access to Kafka brokers

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 **SimpleACLAuthorizer** 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
- ACL authorization
- Red Hat Single Sign-On documentation

3.8.1. 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.

3.8.1.1. 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.

3.8.2. 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.
- You need to understand how to manage policies and permissions for Red Hat Single Sign-On Authorization Services, as described in the Red Hat Single Sign-On documentation .

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/v1beta1
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
  #...
```

(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.

(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 **SimpleACLAuthorizer** if access is denied by Red Hat Single Sign-On Authorization Services policies. The 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.

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 po -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.

3.9. CUSTOMIZING DEPLOYMENTS

AMQ Streams creates several OpenShift resources, such as **Deployments**, **StatefulSets**, **Pods**, and **Services**, which are managed by OpenShift operators. Only the operator that is responsible for managing a particular OpenShift resource can change that resource. If you try to manually change an operator-managed OpenShift resource, the operator will revert your changes back.

However, changing an operator-managed OpenShift resource can be useful if you want to perform certain tasks, such as:

- Adding custom labels or annotations that control how **Pods** are treated by Istio or other services;
- Managing how **Loadbalancer**-type Services are created by the cluster.

You can make these types of changes using the **template** property in the AMQ Streams custom resources.

3.9.1. Template properties

You can use the **template** property to configure aspects of the resource creation process. You can include it in the following resources and properties:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator**
- **Kafka.spec.kafkaExporter**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMakerSpec**
- **KafkaBridge.spec**

In the following example, the **template** property is used to modify the labels in a Kafka broker's **StatefulSet**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  labels:
    app: my-cluster
spec:
  kafka:
    # ...
    template:
      statefulset:
        metadata:
          labels:
            mylabel: myvalue
    # ...
```

3.9.1.1. Supported template properties for a Kafka cluster

statefulset

Configures the **StatefulSet** used by the Kafka broker.

pod

Configures the Kafka broker **Pods** created by the **StatefulSet**.

bootstrapService

Configures the bootstrap service used by clients running within OpenShift to connect to the Kafka broker.

brokersService

Configures the headless service.

externalBootstrapService

Configures the bootstrap service used by clients connecting to Kafka brokers from outside of OpenShift.

perPodService

Configures the per-Pod services used by clients connecting to the Kafka broker from outside OpenShift to access individual brokers.

externalBootstrapRoute

Configures the bootstrap route used by clients connecting to the Kafka brokers from outside of OpenShift using OpenShift **Routes**.

perPodRoute

Configures the per-Pod routes used by clients connecting to the Kafka broker from outside OpenShift to access individual brokers using OpenShift **Routes**.

podDisruptionBudget

Configures the Pod Disruption Budget for Kafka broker **StatefulSet**.

kafkaContainer

Configures the container used to run the Kafka broker, including custom environment variables.

tlsSidecarContainer

Configures the TLS sidecar container, including custom environment variables.

initContainer

Configures the container used to initialize the brokers.

persistentVolumeClaim

Configures the metadata of the Kafka **PersistentVolumeClaims**.

Additional resources

Section B.48, "**KafkaClusterTemplate** schema reference".

3.9.1.2. Supported template properties for a ZooKeeper cluster

statefulset

Configures the ZooKeeper **StatefulSet**.

pod

Configures the ZooKeeper **Pods** created by the **StatefulSet**.

clientsService

Configures the service used by clients to access ZooKeeper.

nodesService

Configures the headless service.

podDisruptionBudget

Configures the Pod Disruption Budget for ZooKeeper **StatefulSet**.

zookeeperContainer

Configures the container used to run the ZooKeeper Node, including custom environment variables.

tlsSidecarContainer

Configures the TLS sidecar container, including custom environment variables.

persistentVolumeClaim

Configures the metadata of the ZooKeeper **PersistentVolumeClaims**.

Additional resources

Section B.58, "**ZookeeperClusterTemplate** schema reference".

3.9.1.3. Supported template properties for Entity Operator

deployment

Configures the Deployment used by the Entity Operator.

pod

Configures the Entity Operator **Pod** created by the **Deployment**.

topicOperatorContainer

Configures the container used to run the Topic Operator, including custom environment variables.

userOperatorContainer

Configures the container used to run the User Operator, including custom environment variables.

tlsSidecarContainer

Configures the TLS sidecar container, including custom environment variables.

Additional resources

Section B.63, "**EntityOperatorTemplate** schema reference".

3.9.1.4. Supported template properties for Kafka Exporter

deployment

Configures the Deployment used by Kafka Exporter.

pod

Configures the Kafka Exporter **Pod** created by the **Deployment**.

services

Configures the Kafka Exporter services.

container

Configures the container used to run Kafka Exporter, including custom environment variables.

Additional resources

Section B.69, "**KafkaExporterTemplate** schema reference".

3.9.1.5. Supported template properties for Kafka Connect and Kafka Connect with Source2Image support

deployment

Configures the Kafka Connect **Deployment**.

pod

Configures the Kafka Connect **Pods** created by the **Deployment**.

apiService

Configures the service used by the Kafka Connect REST API.

podDisruptionBudget

Configures the Pod Disruption Budget for Kafka Connect **Deployment**.

connectContainer

Configures the container used to run Kafka Connect, including custom environment variables.

Additional resources

Section B.83, "**KafkaConnectTemplate** schema reference".

3.9.1.6. Supported template properties for Kafka MirrorMaker

deployment

Configures the Kafka MirrorMaker **Deployment**.

pod

Configures the Kafka MirrorMaker **Pods** created by the **Deployment**.

podDisruptionBudget

Configures the Pod Disruption Budget for Kafka MirrorMaker **Deployment**.

mirrorMakerContainer

Configures the container used to run Kafka MirrorMaker, including custom environment variables.

Additional resources

Section B.113, "**KafkaMirrorMakerTemplate** schema reference".

3.9.2. Labels and Annotations

For every resource, you can configure additional **Labels** and **Annotations**. **Labels** and **Annotations** are used to identify and organize resources, and are configured in the **metadata** property.

For example:

```
# ...
template:
    statefulset:
        metadata:
            labels:
                label1: value1
                label2: value2
            annotations:
                annotation1: value1
                annotation2: value2
# ...
```

The **labels** and **annotations** fields can contain any labels or annotations that do not contain the reserved string **strimzi.io**. Labels and annotations containing **strimzi.io** are used internally by AMQ Streams and cannot be configured.

For Kafka Connect, annotations on the **KafkaConnect** resource are used to enable the creation and management of connectors using **KafkaConnector** resources. For more information, see Section 3.2.15, "Enabling **KafkaConnector** resources".

NOTE

The **metadata** property is not applicable to container templates, such as the **kafkaContainer**.

3.9.3. Customizing Pods

In addition to Labels and Annotations, you can customize some other fields on Pods. These fields are described in the following table and affect how the Pod is created.

Field	Description
terminationGracePeriodSeconds	Defines the period of time, in seconds, by which the Pod must have terminated gracefully. After the grace period, the Pod and its containers are forcefully terminated (killed). The default value is **30** seconds. NOTE: You might need to increase the grace period for very large Kafka clusters, so that the Kafka brokers have enough time to transfer their work to another broker before they are terminated.
imagePullSecrets	Defines a list of references to OpenShift Secrets that can be used for pulling container images from private repositories. For more information about how to create a Secret with the credentials, see Pull an Image from a Private Registry. NOTE: When the **STRIMZI_IMAGE_PULL_SECRETS** environment variable in Cluster Operator and the **imagePullSecrets** option are specified, only the **imagePullSecrets** variable is used. The **STRIMZI_IMAGE_PULL_SECRETS** variable is ignored.
securityContext	Configures pod-level security attributes for containers running as part of a given Pod. For more information about configuring SecurityContext, see Configure a Security Context for a Pod or Container
priorityClassName	Configures the name of the Priority Class which will be used for given a Pod. For more information about Priority Classes, see Pod Priority and Preemption.

Field	Description
schedulerName	The name of the scheduler used to dispatch this **Pod**. If not specified, the default scheduler will be used.

These fields are effective on each type of cluster (Kafka and ZooKeeper; Kafka Connect and Kafka Connect with S2I support; and Kafka MirrorMaker).

The following example shows these customized fields on a **template** property:

```
# ...
template:
  pod:
    metadata:
      labels:
        label1: value1
    imagePullSecrets:
      - name: my-docker-credentials
    securityContext:
      runAsUser: 1000001
      fsGroup: 0
    terminationGracePeriodSeconds: 120
# ...
```

Additional resources

- For more information, see Section B.51, "**PodTemplate** schema reference".

3.9.4. Customizing containers with environment variables

You can set custom environment variables for a container by using the relevant **template** container property. The following table lists the AMQ Streams containers and the relevant template configuration property (defined under **spec**) for each custom resource.

Table 3.1. Table Container environment variable properties

AMQ Streams Element	Container	Configuration property
Kafka	Kafka Broker	**kafka.template.kafkaContainer.env**
Kafka	Kafka Broker TLS Sidecar	**kafka.template.tlsSidecarContainer.env**
Kafka	Kafka Initialization	**kafka.template.initContainer.env**
Kafka	ZooKeeper Node	**zookeeper.template.zookeeperContainer.env**

AMQ Streams Element	Container	Configuration property
Kafka	ZooKeeper TLS Sidecar	**zookeeper.template.tlsSidecarContainer.env**
Kafka	Topic Operator	**entityOperator.template.topicOperatorContainer.env**
Kafka	User Operator	**entityOperator.template.userOperatorContainer.env**
Kafka	Entity Operator TLS Sidecar	**entityOperator.template.tlsSidecarContainer.env**
KafkaConnect	Connect and ConnectS2I	**template.connectContainer.env**
KafkaMirrorMaker	MirrorMaker	**template.mirrorMakerContainer.env**
KafkaBridge	Bridge	**template.bridgeContainer.env**

The environment variables are defined under the **env** property as a list of objects with **name** and **value** fields. The following example shows two custom environment variables set for the Kafka broker containers:

```
# ...
kind: Kafka
spec:
  kafka:
    template:
      kafkaContainer:
        env:
          - name: TEST_ENV_1
            value: test.env.one
          - name: TEST_ENV_2
            value: test.env.two
# ...
```

Environment variables prefixed with **KAFKA_** are internal to AMQ Streams and should be avoided. If you set a custom environment variable that is already in use by AMQ Streams, it is ignored and a warning is recorded in the log.

Additional resources

- For more information, see Section B.55, "**ContainerTemplate** schema reference".

3.9.5. Customizing external Services

When exposing Kafka outside of OpenShift using loadbalancers or node ports, you can use additional customization properties in addition to labels and annotations. The properties for external services are described in the following table and affect how a Service is created.

Field	Description
externalTrafficPolicy	Specifies whether the service routes external traffic to node-local or cluster-wide endpoints. **Cluster** may cause a second hop to another node and obscures the client source IP. **Local** avoids a second hop for LoadBalancer and Nodeport type services and preserves the client source IP (when supported by the infrastructure). If unspecified, OpenShift will use **Cluster** as the default.
loadBalancerSourceRanges	A list of CIDR ranges (for example **10.0.0.0/8** or **130.211.204.1/32**) from which clients can connect to load balancer type listeners. If supported by the platform, traffic through the loadbalancer is restricted to the specified CIDR ranges. This field is applicable only for loadbalancer type services, and is ignored if the cloud provider does not support the feature. For more information, see https://kubernetes.io/docs/tasks/access-application-cluster/configure-cloud-provider-firewall/.

These properties are available for **externalBootstrapService** and **perPodService**. The following example shows these customized properties for a **template**:

```
# ...
template:
  externalBootstrapService:
    externalTrafficPolicy: Local
    loadBalancerSourceRanges:
      - 10.0.0.0/8
      - 88.208.76.87/32
  perPodService:
    externalTrafficPolicy: Local
    loadBalancerSourceRanges:
      - 10.0.0.0/8
      - 88.208.76.87/32
# ...
```

Additional resources

- For more information, see Section B.53, "**ExternalServiceTemplate** schema reference".

3.9.6. Customizing the image pull policy

AMQ Streams allows you to customize the image pull policy for containers in all pods deployed by the

Cluster Operator. The image pull policy is configured using the environment variable **STRIMZI_IMAGE_PULL_POLICY** in the Cluster Operator deployment. The **STRIMZI_IMAGE_PULL_POLICY** environment variable can be set to three different values:

Always

Container images are pulled from the registry every time the pod is started or restarted.

IfNotPresent

Container images are pulled from the registry only when they were not pulled before.

Never

Container images are never pulled from the registry.

The image pull policy can be currently customized only for all Kafka, Kafka Connect, and Kafka MirrorMaker clusters at once. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

Additional resources

- For more information about Cluster Operator configuration, see Section 4.1, "Cluster Operator".
- For more information about Image Pull Policies, see Disruptions.

3.9.7. Customizing Pod Disruption Budgets

AMQ Streams creates a pod disruption budget for every new **StatefulSet** or **Deployment**. By default, these pod disruption budgets only allow a single pod to be unavailable at a given time by setting the **maxUnavailable** value in the **PodDisruptionBudget.spec** resource to 1. You can change the amount of unavailable pods allowed by changing the default value of **maxUnavailable** in the pod disruption budget template. This template applies to each type of cluster (Kafka and ZooKeeper; Kafka Connect and Kafka Connect with S2I support; and Kafka MirrorMaker).

The following example shows customized **podDisruptionBudget** fields on a **template** property:

```
# ...
template:
    podDisruptionBudget:
        metadata:
            labels:
                key1: label1
                key2: label2
            annotations:
                key1: label1
                key2: label2
        maxUnavailable: 1
# ...
```

Additional resources

- For more information, see Section B.54, "**PodDisruptionBudgetTemplate** schema reference".
- The Disruptions chapter of the OpenShift documentation.

3.9.8. Customizing deployments

This procedure describes how to customize **Labels** of a Kafka cluster.

Prerequisites

- An OpenShift cluster.
- A running Cluster Operator.

Procedure

1. Edit the **template** property in the **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, or **KafkaMirrorMaker** resource. For example, to modify the labels for the Kafka broker **StatefulSet**, use:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  labels:
    app: my-cluster
spec:
  kafka:
    # ...
    template:
      statefulset:
        metadata:
          labels:
            mylabel: myvalue
    # ...
```

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

```
oc apply -f your-file
```

   Alternatively, use **oc edit**:

```
oc edit Resource ClusterName
```

3.10. EXTERNAL LOGGING

When setting the logging levels for a resource, you can specify them *inline* directly in the **spec.logging** property of the resource YAML:

```
spec:
  # ...
  logging:
    type: inline
    loggers:
      kafka.root.logger.level: "INFO"
```

Or you can specify *external* logging:

```
spec:
  # ...
  logging:
    type: external
    name: customConfigMap
```

With external logging, logging properties are defined in a ConfigMap. The name of the ConfigMap is referenced in the **spec.logging.name** property.

The advantages of using a ConfigMap are that the logging properties are maintained in one place and are accessible to more than one resource.

3.10.1. Creating a ConfigMap for logging

To use a ConfigMap to define logging properties, you create the ConfigMap and then reference it as part of the logging definition in the **spec** of a resource.

The ConfigMap must contain the appropriate logging configuration.

- **log4j.properties** for Kafka components, ZooKeeper, and the Kafka Bridge
- **log4j2.properties** for the Topic Operator and User Operator

The configuration must be placed under these properties.

Here we demonstrate how a ConfigMap defines a root logger for a Kafka resource.

Procedure

1. Create the ConfigMap.
   You can create the ConfigMap as a YAML file or from a properties file using **oc** at the command line.

   ConfigMap example with a root logger definition for Kafka:

   ```
   kind: ConfigMap
   apiVersion: kafka.strimzi.io/v1beta1
   metadata:
     name: logging-configmap
   data:
     log4j.properties:
       kafka.root.logger.level="INFO"
   ```

   From the command line, using a properties file:

   ```
   oc create configmap logging-configmap --from-file=log4j.properties
   ```

   The properties file defines the logging configuration:

   ```
   # Define the root logger
   kafka.root.logger.level="INFO"
   # ...
   ```

2. Define *external* logging in the **spec** of the resource, setting the **logging.name** to the name of the ConfigMap.

```
spec:
  # ...
  logging:
    type: external
    name: logging-configmap
```

3. Create or update the resource.

```
oc apply -f kafka.yaml
```

CHAPTER 4. OPERATORS

4.1. CLUSTER OPERATOR

Use the Cluster Operator to deploy a Kafka cluster and other Kafka components.

The Cluster Operator is deployed using YAML installation files. For information on deploying the Cluster Operator, see the Deploying the Cluster Operator.

For information on the deployment options available for Kafka, see Kafka Cluster configuration.

NOTE

On OpenShift, a Kafka Connect deployment can incorporate a Source2Image feature to provide a convenient way to add additional connectors.

4.1.1. Cluster Operator

AMQ Streams uses the Cluster Operator to deploy and manage clusters for:

- Kafka (including ZooKeeper, Entity Operator, Kafka Exporter, and Cruise Control)
- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

Custom resources are used to deploy the clusters.

For example, to deploy a Kafka cluster:

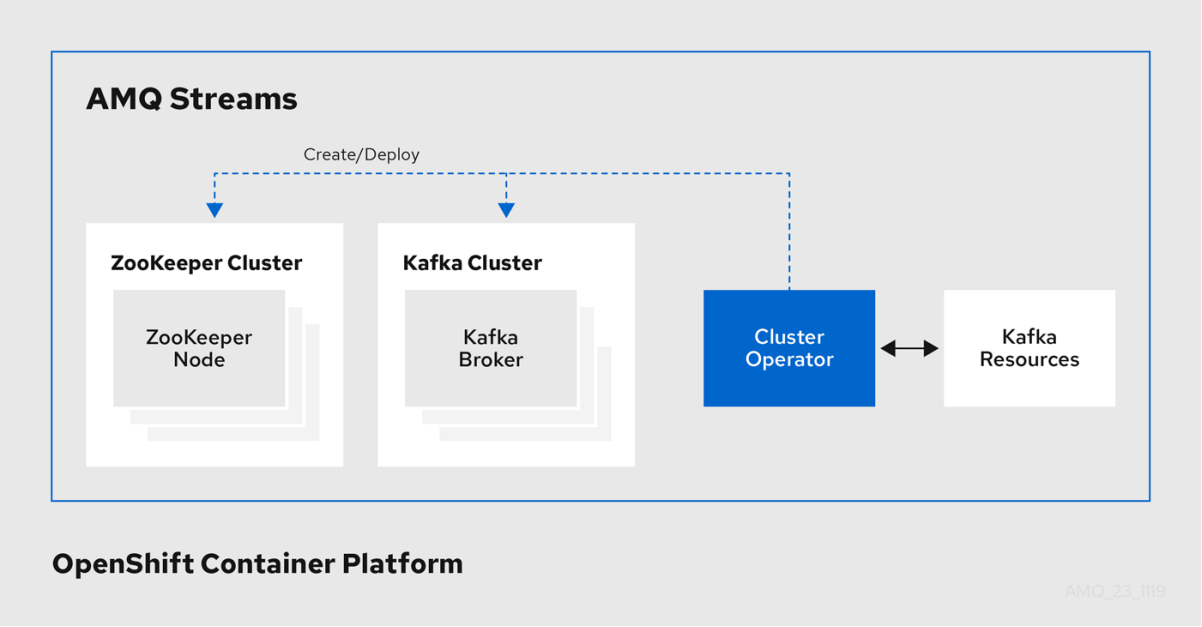
- A **Kafka** resource with the cluster configuration is created within the OpenShift cluster.
- The Cluster Operator deploys a corresponding Kafka cluster, based on what is declared in the **Kafka** resource.

The Cluster Operator can also deploy (through configuration of the **Kafka** resource):

- A Topic Operator to provide operator-style topic management through **KafkaTopic** custom resources
- A User Operator to provide operator-style user management through **KafkaUser** custom resources

The Topic Operator and User Operator function within the Entity Operator on deployment.

Example architecture for the Cluster Operator



4.1.2. Reconciliation

Although the operator reacts to all notifications about the desired cluster resources received from the OpenShift cluster, if the operator is not running, or if a notification is not received for any reason, the desired 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 desired resources with the current cluster deployments in order to have a consistent state across all of them. You can set the time interval for the periodic reconciliations using the [STRIMZI_FULL_RECONCILIATION_INTERVAL_MS] variable.

4.1.3. Cluster Operator Configuration

The Cluster Operator can be configured through the following supported environment variables:

STRIMZI_NAMESPACE

A comma-separated list of namespaces that the operator should operate in. When not set, set to empty string, or to * the Cluster Operator will operate in all namespaces. The Cluster Operator deployment might use the OpenShift Downward API to set this automatically to the namespace the Cluster Operator is deployed in. See the example below:

```
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_LOG_LEVEL

Optional, default **INFO**. The level for printing logging messages. The value can be set to: **ERROR**, **WARNING**, **INFO**, **DEBUG**, and **TRACE**.

STRIMZI_OPERATION_TIMEOUT_MS

Optional, default 300000 ms. The timeout for internal operations, in milliseconds. This value should be increased when using AMQ Streams on clusters where regular OpenShift operations take longer than usual (because of slow downloading of Docker images, for example).

STRIMZI_KAFKA_IMAGES

Required. This provides a mapping from Kafka version to the corresponding Docker image containing a Kafka broker of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.4.1=registry.redhat.io/amq7/amq-streams-kafka-24-rhel7:1.5.0, 2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0**. This is used when a **Kafka.spec.kafka.version** property is specified but not the **Kafka.spec.kafka.image**, as described in Section 3.1.19, "Container images".

STRIMZI_DEFAULT_KAFKA_INIT_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0**. The image name to use as default for the init container started before the broker for initial configuration work (that is, rack support), if no image is specified as the **kafka-init-image** in the Section 3.1.19, "Container images".

STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0**. The image name to use as the default when deploying the sidecar container which provides TLS support for Kafka, if no image is specified as the **Kafka.spec.kafka.tlsSidecar.image** in the Section 3.1.19, "Container images".

STRIMZI_KAFKA_CONNECT_IMAGES

Required. This provides a mapping from the Kafka version to the corresponding Docker image containing a Kafka connect of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.4.1=registry.redhat.io/amq7/amq-streams-kafka-24-rhel7:1.5.0, 2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0**. This is used when a **KafkaConnect.spec.version** property is specified but not the **KafkaConnect.spec.image**, as described in Section 3.2.12, "Container images".

STRIMZI_KAFKA_CONNECT_S2I_IMAGES

Required. This provides a mapping from the Kafka version to the corresponding Docker image containing a Kafka connect of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.4.1=registry.redhat.io/amq7/amq-streams-kafka-24-rhel7:1.5.0, 2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0**. This is used when a **KafkaConnectS2I.spec.version** property is specified but not the **KafkaConnectS2I.spec.image**, as described in Section 3.3.12, "Container images".

STRIMZI_KAFKA_MIRROR_MAKER_IMAGES

Required. This provides a mapping from the Kafka version to the corresponding Docker image containing a Kafka mirror maker of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.4.1=registry.redhat.io/amq7/amq-streams-kafka-24-rhel7:1.5.0, 2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0**. This is used when a **KafkaMirrorMaker.spec.version** property is specified but not the **KafkaMirrorMaker.spec.image**, as described in Section 3.4.2.14, "Container images".

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0**. 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 Section 3.1.19, "Container images" of the **Kafka** resource.

STRIMZI_DEFAULT_USER_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.5.0**. 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 Section 3.1.19, "Container images" of the **Kafka** resource.

STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.5.0**. The image name to use as the default when deploying the sidecar container which provides TLS support for the Entity Operator, if no image is specified as the **Kafka.spec.entityOperator.tlsSidecar.image** in the Section 3.1.19, "Container images".

STRIMZI_IMAGE_PULL_POLICY

Optional. The **ImagePullPolicy** which will be applied to containers in all pods managed by AMQ Streams Cluster Operator. The valid values are **Always**, **IfNotPresent**, and **Never**. If not specified, the OpenShift defaults will be 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 used in the **imagePullSecrets** field 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. See the example below:

```
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.

4.1.4. Role-Based Access Control (RBAC)

4.1.4.1. Provisioning Role-Based Access Control (RBAC) for the Cluster Operator

For the Cluster Operator to function it needs permission within the OpenShift cluster to interact with resources such as **Kafka**, **KafkaConnect**, and so on, as well as the managed resources, such as **ConfigMaps**, **Pods**, **Deployments**, **StatefulSets**, **Services**, and so on. Such permission is described in terms of OpenShift role-based access control (RBAC) resources:

- **ServiceAccount**,
- **Role** and **ClusterRole**,
- **RoleBinding** and **ClusterRoleBinding**.

In addition to running under its own **ServiceAccount** with a **ClusterRoleBinding**, the Cluster Operator manages some RBAC resources for the components that need access to OpenShift resources.

OpenShift also includes privilege escalation protections that prevent components operating under one **ServiceAccount** from granting other **ServiceAccounts** privileges that the granting **ServiceAccount** does not have. Because the Cluster Operator must be able to create the **ClusterRoleBindings**, and **RoleBindings** needed by resources it manages, the Cluster Operator must also have those same privileges.

4.1.4.2. Delegated privileges

When the Cluster Operator deploys resources for a desired **Kafka** resource it also creates **ServiceAccounts**, **RoleBindings**, and **ClusterRoleBindings**, as follows:

- The Kafka broker pods use a **ServiceAccount** called ***cluster-name*-kafka**
  - When the rack feature is used, the **strimzi-*cluster-name*-kafka-init ClusterRoleBinding** is used to grant this **ServiceAccount** access to the nodes within the cluster via a **ClusterRole** called **strimzi-kafka-broker**
  - When the rack feature is not used no binding is created
- The ZooKeeper pods use a **ServiceAccount** called ***cluster-name*-zookeeper**
- The Entity Operator pod uses a **ServiceAccount** called ***cluster-name*-entity-operator**
  - The Topic Operator produces OpenShift events with status information, so the **ServiceAccount** is bound to a **ClusterRole** called **strimzi-entity-operator** which grants this access via the **strimzi-entity-operator RoleBinding**
- The pods for **KafkaConnect** and **KafkaConnectS2I** resources use a **ServiceAccount** called ***cluster-name*-cluster-connect**
- The pods for **KafkaMirrorMaker** use a **ServiceAccount** called ***cluster-name*-mirror-maker**
- The pods for **KafkaBridge** use a **ServiceAccount** called ***cluster-name*-bridge**

4.1.4.3. 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 the the **strimzi-cluster-operator ServiceAccount** is specified as the **serviceAccountName**.

4.1.4.4. ClusterRoles

The Cluster Operator needs to operate using **ClusterRoles** that gives access to the necessary resources. Depending on the OpenShift cluster setup, a cluster administrator might be needed to create the **ClusterRoles**.

NOTE

Cluster administrator rights are only needed for the creation of the **ClusterRoles**. The Cluster Operator will not run under the cluster admin account.

The **ClusterRoles** follow the *principle of least privilege* and contain only those privileges needed by the Cluster Operator to operate Kafka, Kafka Connect, and ZooKeeper clusters. The first set of assigned privileges allow the Cluster Operator to manage OpenShift resources such as **StatefulSets**, **Deployments**, **Pods**, and **ConfigMaps**.

Cluster Operator uses ClusterRoles 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:
- apiGroups:
  - ""
  resources:
  # The cluster operator needs to access and manage service accounts to grant Strimzi components cluster permissions
  - serviceaccounts
  verbs:
  - get
  - create
  - delete
  - patch
  - update
- apiGroups:
  - "rbac.authorization.k8s.io"
  resources:
  # The cluster operator needs to access and manage rolebindings to grant Strimzi components cluster permissions
  - rolebindings
  verbs:
  - get
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  # The cluster operator needs to access and manage config maps for Strimzi components configuration
  - configmaps
  # The cluster operator needs to access and manage services to expose Strimzi components to network traffic
  - services
  # 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:
  - "kafka.strimzi.io"
  resources:
  # The cluster operator runs the KafkaAssemblyOperator, which needs to access and manage Kafka resources
  - kafkas
  - kafkas/status
  # The cluster operator runs the KafkaConnectAssemblyOperator, which needs to access and
```

```
  manage KafkaConnect resources
  - kafkaconnects
  - kafkaconnects/status
  # The cluster operator runs the KafkaConnectS2IAssemblyOperator, which needs to access and
  manage KafkaConnectS2I resources
  - kafkaconnects2is
  - kafkaconnects2is/status
  # The cluster operator runs the KafkaConnectorAssemblyOperator, which needs to access and
  manage KafkaConnector resources
  - kafkaconnectors
  - kafkaconnectors/status
  # The cluster operator runs the KafkaMirrorMakerAssemblyOperator, which needs to access and
  manage KafkaMirrorMaker resources
  - kafkamirrormakers
  - kafkamirrormakers/status
  # The cluster operator runs the KafkaBridgeAssemblyOperator, which needs to access and manage
  BridgeMaker resources
  - kafkabridges
  - kafkabridges/status
  # The cluster operator runs the KafkaMirrorMaker2AssemblyOperator, which needs to access and
  manage KafkaMirrorMaker2 resources
  - kafkamirrormaker2s
  - kafkamirrormaker2s/status
  # The cluster operator runs the KafkaRebalanceAssemblyOperator, which needs to access and
  manage KafkaRebalance resources
  - kafkarebalances
  - kafkarebalances/status
  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
  verbs:
  - get
  - list
  - watch
  - delete
- apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - get
  - list
  - watch
- apiGroups:
  # The cluster operator needs the extensions api as the operator supports Kubernetes version 1.11+
```

```
  # apps/v1 was introduced in Kubernetes 1.14
  - "extensions"
  resources:
  # The cluster operator needs to access and manage deployments to run deployment based Strimzi components
  - deployments
  - deployments/scale
  # The cluster operator needs to access replica sets to manage Strimzi components and to determine error states
  - replicasets
  # The cluster operator needs to access and manage replication controllers to manage replicasets
  - replicationcontrollers
  # 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:
  - "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:
  - ""
  resources:
  # The cluster operator needs to be able to create events and delegate permissions to do so
  - events
  verbs:
  - create
- apiGroups:
```

```
  # OpenShift S2I requirements
  - apps.openshift.io
  resources:
  - deploymentconfigs
  - deploymentconfigs/scale
  - deploymentconfigs/status
  - deploymentconfigs/finalizers
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  # OpenShift S2I requirements
  - build.openshift.io
  resources:
  - buildconfigs
  - builds
  verbs:
  - create
  - delete
  - get
  - list
  - patch
  - watch
  - update
- apiGroups:
  # OpenShift S2I requirements
  - image.openshift.io
  resources:
  - imagestreams
  - imagestreams/status
  verbs:
  - create
  - delete
  - get
  - list
  - watch
  - 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
  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
  - create
  - delete
  - patch
  - update
- 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
```

The second includes the permissions needed for cluster-scoped resources.

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
  - create
  - delete
  - patch
  - update
  - watch
```

```
- 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-kafka-broker ClusterRole** represents the access needed by the init container in Kafka pods that is used for the rack feature. As described in the Delegated privileges section, this role is also needed by the Cluster Operator in order to be able to delegate this access.

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-topic-operator ClusterRole** represents the access needed by the Topic Operator. As described in the Delegated privileges section, this role is also needed by the Cluster Operator in order to be able to delegate this access.

ClusterRole for the Cluster Operator allowing it to delegate access to events to the Topic Operator

```
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
  - create
  - patch
  - update
  - delete
```

4.1.4.5. ClusterRoleBindings

The operator needs **ClusterRoleBindings** and **RoleBindings** which associates its **ClusterRole** with its **ServiceAccount**: **ClusterRoleBindings** are needed for **ClusterRoles** 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
```

ClusterRoleBindings are also needed for the **ClusterRoles** needed for delegation:

Examples RoleBinding for the Cluster Operator

```
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
```

ClusterRoles containing only namespaced resources are bound using **RoleBindings** only.

```
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
```

```
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
```

4.2. TOPIC OPERATOR

The Topic Operator manages Kafka topics through custom resources.

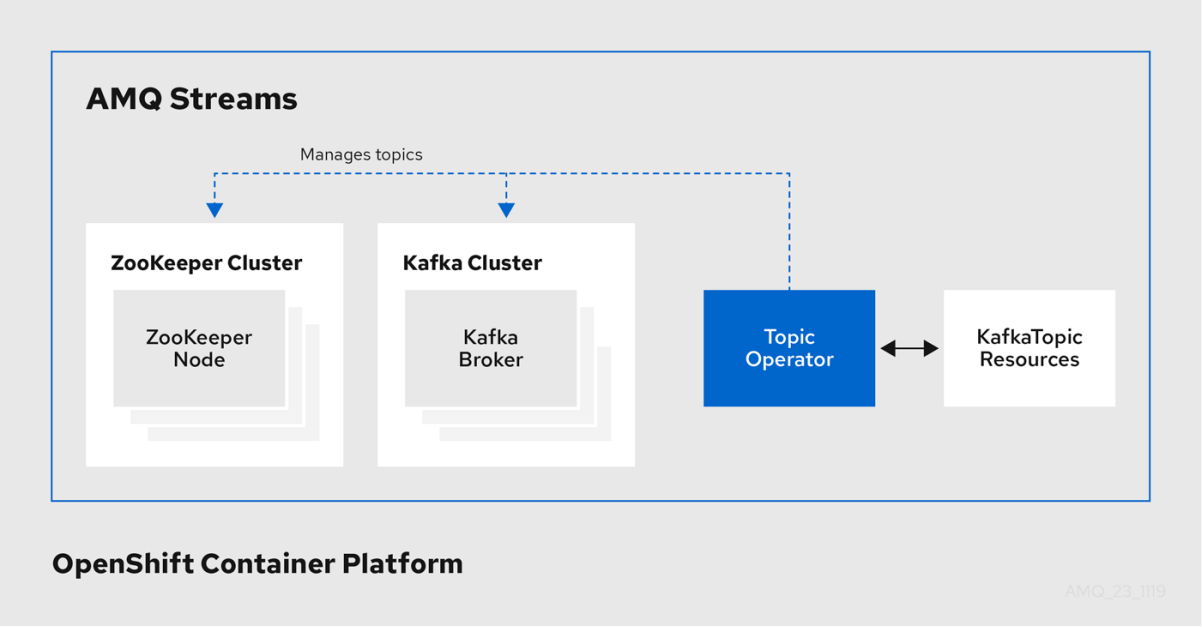
The Topic Operator is deployed:

- Using the Cluster Operator (recommended)
- Standalone to operate with Kafka clusters not managed by AMQ Streams

4.2.1. Topic Operator

The Topic Operator provides a way of managing topics in a Kafka cluster through OpenShift resources.

Example architecture for the Topic Operator



The role of the Topic Operator is to keep a set of **KafkaTopic** OpenShift resources describing Kafka topics in-sync with corresponding Kafka topics.

Specifically, if a **KafkaTopic** is:

- Created, the Topic Operator creates the topic
- Deleted, the Topic Operator deletes the topic
- Changed, the Topic Operator updates the topic

Working in the other direction, if a topic is:

- Created within the Kafka cluster, the Operator creates a **KafkaTopic**
- Deleted from the Kafka cluster, the Operator deletes the **KafkaTopic**
- Changed in the Kafka cluster, the Operator updates the **KafkaTopic**

This allows you to declare a **KafkaTopic** as part of your application's deployment and the Topic Operator will take care of creating the topic for you. Your application just needs to deal with producing or consuming from the necessary topics.

If the topic is reconfigured or reassigned to different Kafka nodes, the **KafkaTopic** will always be up to date.

4.2.2. Identifying a Kafka cluster for topic handling

A **KafkaTopic** 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/v1beta1
kind: KafkaTopic
metadata:
  name: my-topic
  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.

4.2.3. Understanding the Topic Operator

A fundamental problem that the operator has to solve is that there is no single source of truth: Both the **KafkaTopic** resource and the topic within Kafka can be modified independently of the operator. Complicating this, the Topic Operator might not always be able to observe changes at each end in real time (for example, the operator might be down).

To resolve this, the operator maintains its own private copy of the information about each topic. When a change happens either in the Kafka cluster, or in OpenShift, it looks at both the state of the other system and at its private copy in order to determine what needs to change to keep everything in sync. The same thing happens whenever the operator starts, and periodically while it is running.

For example, suppose the Topic Operator is not running, and a **KafkaTopic my-topic** gets created. When the operator starts it will lack a private copy of "my-topic", so it can infer that the **KafkaTopic** has been created since it was last running. The operator will create the topic corresponding to "my-topic" and also store a private copy of the metadata for "my-topic".

The private copy allows the operator to cope with scenarios where the topic configuration gets changed both in Kafka and in OpenShift, so long as the changes are not incompatible (for example, both changing the same topic config key, but to different values). In the case of incompatible changes, the Kafka configuration wins, and the **KafkaTopic** will be updated to reflect that.

The private copy is held in the same ZooKeeper ensemble used by Kafka itself. This mitigates availability concerns, because if ZooKeeper is not running then Kafka itself cannot run, so the operator will be no less available than it would even if it was stateless.

4.2.4. 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:
   Use **oc edit**:

   ```
   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/v1beta1
   kind: Kafka
   spec:
     # kafka and zookeeper sections...
     entityOperator:
       topicOperator:
         resources:
           request:
             cpu: "1"
             memory: 500Mi
           limit:
             cpu: "1"
             memory: 500Mi
   ```

3. Apply the new configuration to create or update the resource.
   Use **oc apply**:

   ```
   oc apply -f kafka.yaml
   ```

Additional resources

- For more information about the schema of the **resources** object, see {K8sResourceRequirementsAPI}.

4.3. USER OPERATOR

The User Operator manages Kafka users through custom resources.

The User Operator is deployed:

- Using the Cluster Operator (recommended)
- Standalone to operate with Kafka clusters not managed by AMQ Streams

4.3.1. User Operator

The User Operator manages Kafka users for a Kafka cluster by watching for **KafkaUser** resources that describe Kafka users, and ensuring that they are configured properly in the Kafka cluster.

For example, if a **KafkaUser** is:

- Created, the User Operator creates the user it describes
- Deleted, the User Operator deletes the user it describes
- Changed, the User Operator updates the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the OpenShift resources. Kafka topics can be created by applications directly in Kafka, but it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator.

The User Operator allows you to declare a **KafkaUser** resource as part of your application's deployment. You can specify the authentication and authorization mechanism for the user. You can also configure *user quotas* that control usage of Kafka resources to ensure, for example, that a user does not monopolize access to a broker.

When the user is created, the user credentials are created in a **Secret**. Your application needs to use the user and its credentials for authentication and to produce or consume messages.

In addition to managing credentials for authentication, the User Operator also manages authorization rules by including a description of the user's access rights in the **KafkaUser** declaration.

4.3.2. 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/v1beta1
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.

4.3.3. 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/v1beta1
kind: Kafka
spec:
  # kafka and zookeeper sections...
  entityOperator:
    userOperator:
      resources:
        request:
          cpu: "1"
          memory: 500Mi
        limit:
          cpu: "1"
          memory: 500Mi
```

   Save the file and exit the editor. The Cluster Operator will apply the changes automatically.

Additional resources

- For more information about the schema of the **resources** object, see {K8sResourceRequirementsAPI}.

4.4. MONITORING OPERATORS

4.4.1. Prometheus metrics

AMQ Streams operators expose Prometheus metrics. The metrics are automatically enabled and contain information about:

- Number of reconciliations
- Number of Custom Resources the operator is processing
- Duration of reconciliations
- JVM metrics from the operators

Additionally, we provide an example Grafana dashboard.

For more information about Prometheus, see the Introducing Metrics to Kafka .

CHAPTER 5. 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.

5.1. KAFKA TOPIC RESOURCE

The **KafkaTopic** resource is used to configure topics, including the number of partitions and replicas.

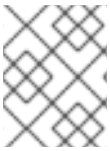
The full schema for **KafkaTopic** is described in **KafkaTopic** schema reference.

5.1.1. 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

Recommendations for identifiers and names in OpenShift are outlined in Identifiers and Names in OpenShift community article.

5.1.2. 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:

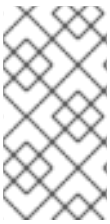
```
kind: KafkaTopic
metadata:
  name: topic-name-1
```

```
spec:
  topicName: topicName-1 (1)
  # ...
```

(1) Upper case is invalid in OpenShift.

cannot be changed to:

```
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 valid names based on the Kafka names. Invalid characters are replaced and a hash is appended to the name.

5.2. CONFIGURING A KAFKA TOPIC

Use the properties of the **KafkaTopic** resource to configure a Kafka topic.

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 making the following changes through the **KafkaTopic** resource:
  - Changing topic names using **spec.topicName**
  - Decreasing partition size using **spec.partitions**
- You cannot use **spec.replicas** to change the number of replicas that were initially specified.
- Increasing **spec.partitions** for topics with keys will change how records are partitioned, which can be particularly problematic when the topic uses *semantic partitioning*.

Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using TLS authentication and 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. Prepare a file containing the **KafkaTopic** to be created.

   An example KafkaTopic

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   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>
   ```

CHAPTER 6. 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.

6.1. KAFKA USER RESOURCE

The **KafkaUser** resource is used to configure the authentication mechanism, authorization mechanism, and access rights for a user.

The full schema for **KafkaUser** is described in **KafkaUser** schema reference.

6.1.1. User authentication

Authentication is configured using the **authentication** property in **KafkaUser.spec**. The authentication mechanism enabled for the user is specified using the **type** field.

Supported authentication mechanisms:

- TLS client authentication
- SCRAM-SHA-512 authentication

When no authentication mechanism is specified, the User Operator does not create the user or its credentials.

Additional resources

- When to use mutual TLS authentication for clients
- When to use SCRAM-SHA Authentication authentication for clients

6.1.1.1. TLS Client Authentication

To use TLS client authentication, you set the **type** field to **tls**.

An example KafkaUser with TLS client authentication enabled

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  # ...
```

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 TLS client authentication. The public key is contained in a user certificate, which is signed by the client Certificate Authority (CA).

All keys are in X.509 format.

Secrets provide private keys and certificates in PEM and PKCS #12 formats. For more information on securing Kafka communication with Secrets, see Chapter 12, *Security*.

An 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 of the client CA
  user.crt: # User certificate that contains the public key of the user
  user.key: # Private key of the user
  user.p12: # PKCS #12 archive file for storing certificates and keys
  user.password: # Password for protecting the PKCS #12 archive file
```

6.1.1.2. SCRAM-SHA-512 Authentication

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

An example KafkaUser with SCRAM-SHA-512 authentication enabled

```
apiVersion: kafka.strimzi.io/v1beta1
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.

An 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= # Generated password
```

Decoding the generated password:

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

6.1.2. User authorization

User authorization is configured using the **authorization** property in **KafkaUser.spec**. The authorization type enabled for a user is specified using the **type** field.

If no authorization is specified, the User Operator does not provision any access rights for the user.

To use simple authorization, you set the **type** property to **simple** in **KafkaUser.spec**. Simple authorization uses the default Kafka authorization plugin, **SimpleAclAuthorizer**.

Alternatively, if you are using OAuth 2.0 token based authentication, you can also configure OAuth 2.0 authorization.

ACL rules
SimpleAclAuthorizer uses ACL rules to manage access to Kafka brokers.

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

An **AclRule** is specified as a set of properties:

resource

The **resource** property specifies the resource that the rule applies to.
Simple authorization supports four resource types, which are specified in the **type** property:

- Topics (**topic**)
- Consumer Groups (**group**)
- Clusters (**cluster**)
- Transactional IDs (**transactionalId**)

For Topic, Group, and Transactional ID resources you can specify the name of the resource the rule applies to in the **name** property.

Cluster type resources have no name.

A name is specified as a **literal** or a **prefix** using the **patternType** property.

- Literal names are taken exactly as they are specified in the **name** field.
- Prefix names use the value from the **name** as a prefix, and will apply the rule to all resources with names starting with the value.

type

The **type** property specifies the type of ACL rule, **allow** or **deny**.
The **type** field is optional. If **type** is unspecified, the ACL rule is treated as an **allow** rule.

operation

The **operation** specifies the operation to allow or deny.

The following operations are supported:

- Read
- Write
- Delete
- Alter
- Describe
- All
- IdempotentWrite
- ClusterAction
- Create
- AlterConfigs
- DescribeConfigs

Only certain operations work with each resource.

For more details about **SimpleAclAuthorizer**, ACLs and supported combinations of resources and operations, see Authorization and ACLs.

host

The **host** property specifies a remote host from which the rule is allowed or denied.
Use an asterisk (*) to allow or deny the operation from all hosts. The **host** field is optional. If **host** is unspecified, the * value is used by default.

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

An example KafkaUser with authorization

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Read
      - resource:
```

```
      type: topic
      name: my-topic
      patternType: literal
    operation: Describe
  - resource:
      type: group
      name: my-group
      patternType: prefix
    operation: Read
```

6.1.2.1. 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.

For more information on configuring super users, see authentication and authorization of Kafka brokers .

6.1.3. User quotas

You can configure the **spec** for the **KafkaUser** resource to enforce quotas so that a user does not exceed access to Kafka brokers based on a byte threshold or a time limit of CPU utilization.

An example KafkaUser with user quotas

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  quotas:
    producerByteRate: 1048576 (1)
    consumerByteRate: 2097152 (2)
    requestPercentage: 55 (3)
```

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

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

6.2. CONFIGURING A KAFKA USER

Use the properties of the **KafkaUser** resource to configure a Kafka user.

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>***

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 TLS 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 TLS if it is not also enabled in the Kafka configuration.

Prerequisites

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

If you are using SCRAM-SHA authentication, you need a running Kafka cluster configured with a Kafka broker listener using SCRAM-SHA authentication.

Procedure

1. Prepare a YAML file containing the **KafkaUser** to be created.

   An example KafkaUser

```
apiVersion: kafka.strimzi.io/v1beta1
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
```

```
      operation: Read
    - resource:
        type: topic
        name: my-topic
        patternType: literal
      operation: Describe
    - resource:
        type: group
        name: my-group
        patternType: literal
      operation: 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 the **KafkaUser** resource in OpenShift.

```
oc apply -f <user-config-file>
```

3. Use the credentials from the **my-user** secret in your client application.

CHAPTER 7. KAFKA BRIDGE

This chapter provides an overview of the AMQ Streams Kafka Bridge and helps you get started using its REST API to interact with AMQ Streams. To try out the Kafka Bridge in your local environment, see the Section 7.2, "Kafka Bridge quickstart" later in this chapter.

7.1. KAFKA BRIDGE OVERVIEW

You can use the Kafka Bridge as an interface to make specific types of request to the Kafka cluster.

7.1.1. Kafka Bridge interface

AMQ Streams Kafka Bridge provides a RESTful interface that allows HTTP-based clients to interact with a Kafka cluster. Kafka Bridge offers the advantages of a web API connection to AMQ Streams, without the need for client applications to interpret the Kafka protocol.

The API has two main resources — **consumers** and **topics** — that are exposed and made accessible through endpoints to interact with consumers and producers in your Kafka cluster. The resources relate only to the Kafka Bridge, not the consumers and producers connected directly to Kafka.

7.1.1.1. HTTP requests

The Kafka Bridge supports HTTP requests to a Kafka cluster, with methods to:

- Send messages to a topic.
- Retrieve messages from topics.
- Create and delete consumers.
- Subscribe consumers to topics, so that they start receiving messages from those topics.
- Retrieve a list of topics that a consumer is subscribed to.
- Unsubscribe consumers from topics.
- Assign partitions to consumers.
- Commit a list of consumer offsets.
- Seek on a partition, so that a consumer starts receiving messages from the first or last offset position, or a given offset position.

The methods provide JSON responses and HTTP response code error handling. Messages can be sent in JSON or binary formats.

Clients can produce and consume messages without the requirement to use the native Kafka protocol.

Additional resources

- To view the API documentation, including example requests and responses, see the Kafka Bridge API reference on the AMQ Streams website.

7.1.2. Supported clients for the Kafka Bridge

You can use the Kafka Bridge to integrate both *internal* and *external* HTTP client applications with your Kafka cluster.

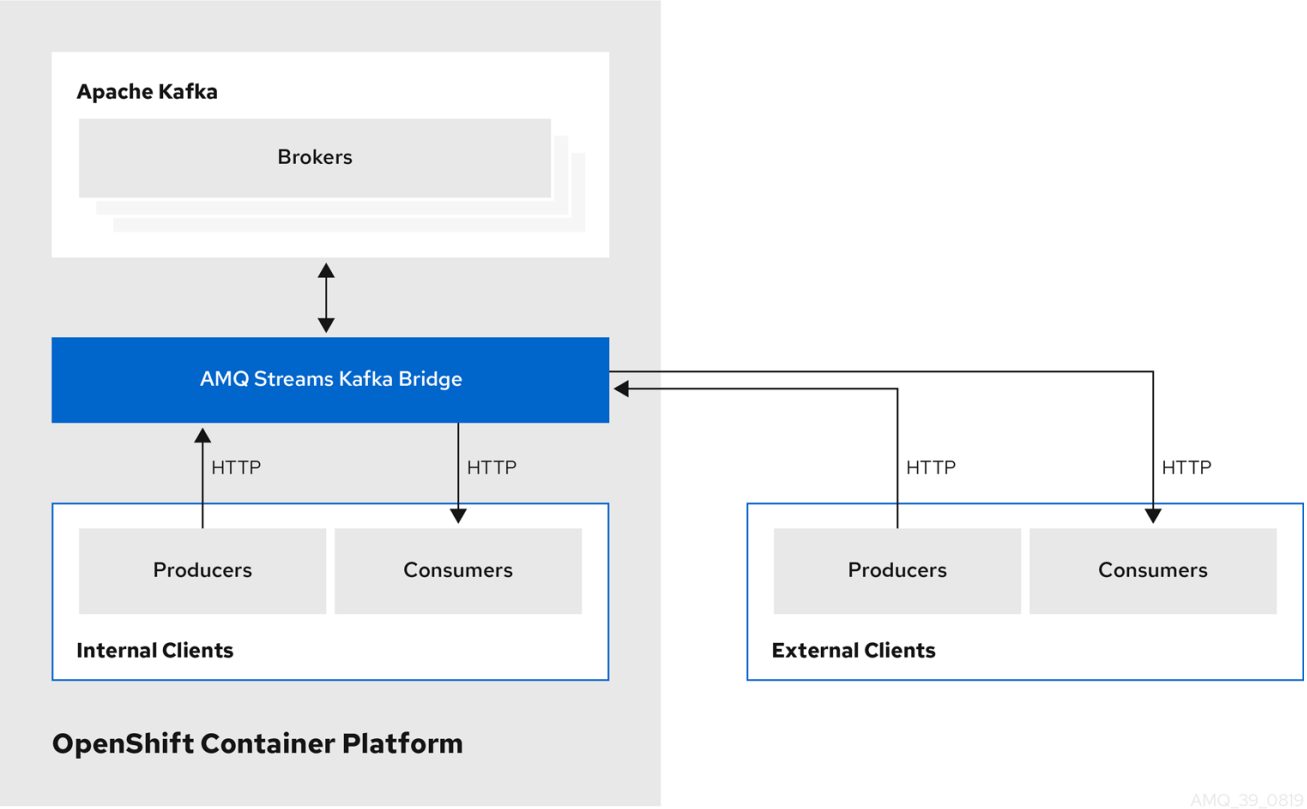
Internal clients

Internal clients are container-based HTTP clients running in *the same* OpenShift cluster as the Kafka Bridge itself. Internal clients can access the Kafka Bridge on the host and port defined in the **KafkaBridge** custom resource.

External clients

External clients are HTTP clients running *outside* the OpenShift cluster in which the Kafka Bridge is deployed and running. External clients can access the Kafka Bridge through an OpenShift Route, a loadbalancer service, or using an Ingress.

HTTP internal and external client integration



7.1.3. Securing the Kafka Bridge

AMQ Streams does not currently provide any encryption, authentication, or authorization for the Kafka Bridge. This means that requests sent from external clients to the Kafka Bridge are:

- Not encrypted, and must use HTTP rather than HTTPS
- Sent without authentication

However, you can secure the Kafka Bridge using other methods, such as:

- OpenShift Network Policies that define which pods can access the Kafka Bridge.
- Reverse proxies with authentication or authorization, for example, OAuth2 proxies.
- API Gateways.

- Ingress or OpenShift Routes with TLS termination.

The Kafka Bridge supports TLS encryption and TLS and SASL authentication when connecting to the Kafka Brokers. Within your OpenShift cluster, you can configure:

- TLS or SASL-based authentication between the Kafka Bridge and your Kafka cluster
- A TLS-encrypted connection between the Kafka Bridge and your Kafka cluster.

For more information, see Section 3.6.4.1, “Authentication support in Kafka Bridge” .

You can use ACLs in Kafka brokers to restrict the topics that can be consumed and produced using the Kafka Bridge.

7.1.4. 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 using one of the following features:

- Services of types LoadBalancer or NodePort
- Ingress resources
- OpenShift Routes

If you decide to create Services, use the following labels in the **selector** 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.

7.1.5. Requests to the Kafka Bridge

Specify data formats and HTTP headers to ensure valid requests are submitted to the Kafka Bridge.

7.1.5.1. Content Type headers

API request and response bodies are always encoded as JSON.

- When performing consumer operations, **POST** requests must provide the following **Content-Type** header if there is a non-empty body:

```
Content-Type: application/vnd.kafka.v2+json
```

- When performing producer operations, **POST** requests must provide **Content-Type** headers specifying the desired *embedded data format*, either **json** or **binary**, as shown in the following table.

Embedded data format	Content-Type header
JSON	**Content-Type: application/vnd.kafka.json.v2+json**
Binary	**Content-Type: application/vnd.kafka.binary.v2+json**

You set the embedded data format when creating a consumer using the **consumers/*groupid*** endpoint —for more information, see the next section.

The **Content-Type** must not be set if the **POST** request has an empty body. An empty body can be used to create a consumer with the default values.

7.1.5.2. Embedded data format

The embedded data format is the format of the Kafka messages that are transmitted, over HTTP, from a producer to a consumer using the Kafka Bridge. Two embedded data formats are supported: JSON and binary.

When creating a consumer using the /**consumers/*groupid*** endpoint, the **POST** request body must specify an embedded data format of either JSON or binary. This is specified in the **format** field, for example:

```
{
  "name": "my-consumer",
  "format": "binary", (1)
...
}
```

(1) A binary embedded data format.

The embedded data format specified when creating a consumer must match the data format of the Kafka messages it will consume.

If you choose to specify a binary embedded data format, subsequent producer requests must provide the binary data in the request body as Base64-encoded strings. For example, when sending messages using the /**topics/*topicname*** endpoint, **records.value** must be encoded in Base64:

```
{
  "records": [
    {
      "key": "my-key",
      "value": "ZWR3YXJkdGhldGhyZWVsZWdnZWRjYXQ="
    },
  ]
}
```

Producer requests must also provide a **Content-Type** header that corresponds to the embedded data format, for example, **Content-Type: application/vnd.kafka.binary.v2+json**.

7.1.5.3. Accept headers

After creating a consumer, all subsequent GET requests must provide an **Accept** header in the following format:

```
Accept: application/vnd.kafka.embedded-data-format.v2+json
```

The **embedded-data-format** is either **json** or **binary**.

For example, when retrieving records for a subscribed consumer using an embedded data format of JSON, include this Accept header:

```
Accept: application/vnd.kafka.json.v2+json
```

7.1.6. Kafka Bridge API resources

For the full list of REST API endpoints and descriptions, including example requests and responses, see the Kafka Bridge API reference on the AMQ Streams website.

7.1.7. Kafka Bridge deployment

You deploy the Kafka Bridge into your OpenShift cluster by using the Cluster Operator.

After the Kafka Bridge is deployed, the Cluster Operator creates Kafka Bridge objects in your OpenShift cluster. Objects include the *deployment*, *service*, and *pod*, each named after the name given in the custom resource for the Kafka Bridge.

Additional resources

- For deployment instructions, see Section 2.5.1, "Deploying Kafka Bridge to your OpenShift cluster".
- For detailed information on configuring the Kafka Bridge, see Section 3.6, "Kafka Bridge configuration"
- For information on configuring the host and port for the **KafkaBridge** resource, see Section 3.6.5.3, "Kafka Bridge HTTP configuration" .
- For information on integrating external clients, see Section 7.1.4, "Accessing the Kafka Bridge outside of OpenShift".

7.2. KAFKA BRIDGE QUICKSTART

Use this quickstart to try out the AMQ Streams Kafka Bridge in your local development environment. You will learn how to:

- Deploy the Kafka Bridge to your OpenShift cluster
- Expose the Kafka Bridge service to your local machine by using port-forwarding
- Produce messages to topics and partitions in your Kafka cluster

- Create a Kafka Bridge consumer
- Perform basic consumer operations, such as subscribing the consumer to topics and retrieving the messages that you produced

In this quickstart, HTTP requests are formatted as curl commands that you can copy and paste to your terminal. Access to an OpenShift cluster is required; to run and manage a local OpenShift cluster, use a tool such as Minikube, CodeReady Containers, or MiniShift.

Ensure you have the prerequisites and then follow the tasks in the order provided in this chapter.

About data formats

In this quickstart, you will produce and consume messages in JSON format, not binary. For more information on the data formats and HTTP headers used in the example requests, see Section 7.1.5, "Requests to the Kafka Bridge".

Prerequisites for the quickstart

- Cluster administrator access to a local or remote OpenShift cluster.
- AMQ Streams is installed.
- A running Kafka cluster, deployed by the Cluster Operator, in an OpenShift namespace.
- The Entity Operator is deployed and running as part of the Kafka cluster.

7.2.1. Deploying the Kafka Bridge to your OpenShift cluster

AMQ Streams includes a YAML example that specifies the configuration of the AMQ Streams Kafka Bridge. Make some minimal changes to this file and then deploy an instance of the Kafka Bridge to your OpenShift cluster.

Procedure

1. Edit the **examples/bridge/kafka-bridge.yaml** file.

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: quickstart (1)
spec:
  replicas: 1
  bootstrapServers: <cluster-name>-kafka-bootstrap:9092 (2)
  http:
    port: 8080
```

(1) When the Kafka Bridge is deployed, **-bridge** is appended to the name of the deployment and other related resources. In this example, the Kafka Bridge deployment is named **quickstart-bridge** and the accompanying Kafka Bridge service is named **quickstart-bridge-service**.

(2) In the **bootstrapServers** property, enter the name of the Kafka cluster as the **<cluster-name>**.

2. Deploy the Kafka Bridge to your OpenShift cluster:

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

A **quickstart-bridge** deployment, service, and other related resources are created in your OpenShift cluster.

3. Verify that the Kafka Bridge was successfully deployed:

```
oc get deployments
```

```
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
quickstart-bridge          1/1     1            1           34m
my-cluster-connect         1/1     1            1           24h
my-cluster-entity-operator 1/1     1            1           24h
#...
```

What to do next

After deploying the Kafka Bridge to your OpenShift cluster, expose the Kafka Bridge service to your local machine.

Additional resources

- For more detailed information about configuring the Kafka Bridge, see Section 3.6, "Kafka Bridge configuration".

7.2.2. Exposing the Kafka Bridge service to your local machine

Next, 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/quickstart-bridge-589d78784d-9jcnr
pod/strimzi-cluster-operator-76bcf9bc76-8dnfm
```

2. Connect to the **quickstart-bridge** pod on port **8080**:

```
oc port-forward pod/quickstart-bridge-589d78784d-9jcnr 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.

7.2.3. Producing messages to topics and partitions

Next, produce messages to topics in JSON format by using the topics endpoint. You can specify destination partitions for messages in the request body, as shown here. The partitions endpoint provides an alternative method for specifying a single destination partition for all messages as a path parameter.

Procedure

1. In a text editor, create a YAML definition for a Kafka topic with three partitions.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: bridge-quickstart-topic
  labels:
    strimzi.io/cluster: <kafka-cluster-name> 1
spec:
  partitions: 3 2
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
```

1. The name of the Kafka cluster in which the Kafka Bridge is deployed.
2. The number of partitions for the topic.

2. Save the file to the **examples/topic** directory as **bridge-quickstart-topic.yaml**.
3. Create the topic in your OpenShift cluster:

```
oc apply -f examples/topic/bridge-quickstart-topic.yaml
```

4. Using the Kafka Bridge, produce three messages to the topic you created:

```
curl -X POST \
  http://localhost:8080/topics/bridge-quickstart-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
  -d '{
    "records": [
        {
            "key": "my-key",
            "value": "sales-lead-0001"
        },
```

```
        {
          "value": "sales-lead-0002",
          "partition": 2
        },
        {
          "value": "sales-lead-0003"
        }
    ]
}'
```

- **sales-lead-0001** is sent to a partition based on the hash of the key.
- **sales-lead-0002** is sent directly to partition 2.
- **sales-lead-0003** is sent to a partition in the **bridge-quickstart-topic** topic using a round-robin method.

5. If the request is successful, the Kafka Bridge returns an **offsets** array, along with a **200** code and a **content-type** header of **application/vnd.kafka.v2+json**. For each message, the **offsets** array describes:
   - The partition that the message was sent to
   - The current message offset of the partition

   Example response

```
#...
{
  "offsets":[
    {
      "partition":0,
      "offset":0
    },
    {
      "partition":2,
      "offset":0
    },
    {
      "partition":0,
      "offset":1
    }
  ]
}
```

What to do next

After producing messages to topics and partitions, create a Kafka Bridge consumer .

Additional resources

- POST /topics/{topicname} in the API reference documentation.
- POST /topics/{topicname}/partitions/{partitionid} in the API reference documentation.

7.2.4. Creating a Kafka Bridge consumer

Before you can perform any consumer operations in the Kafka cluster, you must first create a consumer by using the consumers endpoint. The consumer is referred to as a *Kafka Bridge consumer*.

Procedure

1. Create a Kafka Bridge consumer in a new consumer group named **bridge-quickstart-consumer-group**:

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "name": "bridge-quickstart-consumer",
    "auto.offset.reset": "earliest",
    "format": "json",
    "enable.auto.commit": false,
    "fetch.min.bytes": 512,
    "consumer.request.timeout.ms": 30000
  }'
```

   - The consumer is named **bridge-quickstart-consumer** and the embedded data format is set as **json**.
   - Some basic configuration settings are defined.
   - The consumer will not commit offsets to the log automatically because the **enable.auto.commit** setting is **false**. You will commit the offsets manually later in this quickstart.
     If the request is successful, the Kafka Bridge returns the consumer ID (**instance_id**) and base URL (**base_uri**) in the response body, along with a **200** code.

     Example response

```
#...
{
  "instance_id": "bridge-quickstart-consumer",
  "base_uri":"http://<bridge-name>-bridge-service:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer"
}
```

2. Copy the base URL (**base_uri**) to use in the other consumer operations in this quickstart.

What to do next

Now that you have created a Kafka Bridge consumer, you can subscribe it to topics .

Additional resources

- POST /consumers/{groupid} in the API reference documentation.

7.2.5. Subscribing a Kafka Bridge consumer to topics

After you have created a Kafka Bridge consumer, subscribe it to one or more topics by using the subscription endpoint. Once subscribed, the consumer starts receiving all messages that are produced to the topic.

Procedure

- Subscribe the consumer to the **bridge-quickstart-topic** topic that you created earlier, in Producing messages to topics and partitions :

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/subscription \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "topics": [
        "bridge-quickstart-topic"
    ]
}'
```

The **topics** array can contain a single topic (as shown here) or multiple topics. If you want to subscribe the consumer to multiple topics that match a regular expression, you can use the **topic_pattern** string instead of the **topics** array.

If the request is successful, the Kafka Bridge returns a **204** (No Content) code only.

What to do next

After subscribing a Kafka Bridge consumer to topics, you can retrieve messages from the consumer .

Additional resources

- POST /consumers/{groupid}/instances/{name}/subscription in the API reference documentation.

7.2.6. Retrieving the latest messages from a Kafka Bridge consumer

Next, retrieve the latest messages from the Kafka Bridge consumer by requesting data from the records endpoint. In production, HTTP clients can call this endpoint repeatedly (in a loop).

Procedure

1. Produce additional messages to the Kafka Bridge consumer, as described in Producing messages to topics and partitions.
2. Submit a **GET** request to the **records** endpoint:

```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/records \
  -H 'accept: application/vnd.kafka.json.v2+json'
```

After creating and subscribing to a Kafka Bridge consumer, a first GET request will return an empty response because the poll operation starts a rebalancing process to assign partitions.

3. Repeat step two to retrieve messages from the Kafka Bridge consumer.
The Kafka Bridge returns an array of messages – describing the topic name, key, value, partition, and offset – in the response body, along with a **200** code. Messages are retrieved from the latest offset by default.

```
HTTP/1.1 200 OK
content-type: application/vnd.kafka.json.v2+json
```

```
#...
[
  {
    "topic":"bridge-quickstart-topic",
    "key":"my-key",
    "value":"sales-lead-0001",
    "partition":0,
    "offset":0
  },
  {
    "topic":"bridge-quickstart-topic",
    "key":null,
    "value":"sales-lead-0003",
    "partition":0,
    "offset":1
  },
#...
```

NOTE

If an empty response is returned, produce more records to the consumer as described in Producing messages to topics and partitions , and then try retrieving messages again.

What to do next

After retrieving messages from a Kafka Bridge consumer, try committing offsets to the log .

Additional resources

- GET /consumers/{groupid}/instances/{name}/records in the API reference documentation.

7.2.7. Commiting offsets to the log

Next, use the offsets endpoint to manually commit offsets to the log for all messages received by the Kafka Bridge consumer. This is required because the Kafka Bridge consumer that you created earlier, in Creating a Kafka Bridge consumer , was configured with the **enable.auto.commit** setting as **false**.

Procedure

- Commit offsets to the log for the **bridge-quickstart-consumer**:

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-
group/instances/bridge-quickstart-consumer/offsets
```

Because no request body is submitted, offsets are committed for all the records that have been received by the consumer. Alternatively, the request body can contain an array (OffsetCommitSeekList) that specifies the topics and partitions that you want to commit offsets for.

If the request is successful, the Kafka Bridge returns a **204** code only.

What to do next

After committing offsets to the log, try out the endpoints for seeking to offsets.

Additional resources

- POST /consumers/{groupid}/instances/{name}/offsets in the API reference documentation.

7.2.8. Seeking to offsets for a partition

Next, use the positions endpoints to configure the Kafka Bridge consumer to retrieve messages for a partition from a specific offset, and then from the latest offset. This is referred to in Apache Kafka as a seek operation.

Procedure

1. Seek to a specific offset for partition 0 of the **quickstart-bridge-topic** topic:

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/positions \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "offsets": [
        {
            "topic": "bridge-quickstart-topic",
            "partition": 0,
            "offset": 2
        }
    ]
}'
```

   If the request is successful, the Kafka Bridge returns a **204** code only.

2. Submit a **GET** request to the **records** endpoint:

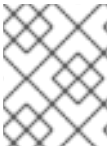
```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/records \
  -H 'accept: application/vnd.kafka.json.v2+json'
```

   The Kafka Bridge returns messages from the offset that you seeked to.

3. Restore the default message retrieval behavior by seeking to the last offset for the same partition. This time, use the positions/end endpoint.

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/positions/end \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "partitions": [
        {
            "topic": "bridge-quickstart-topic",
            "partition": 0
        }
    ]
}'
```

   If the request is successful, the Kafka Bridge returns another **204** code.



NOTE

You can also use the positions/beginning endpoint to seek to the first offset for one or more partitions.

What to do next

In this quickstart, you have used the AMQ Streams Kafka Bridge to perform several common operations on a Kafka cluster. You can now delete the Kafka Bridge consumer that you created earlier.

Additional resources

- POST /consumers/{groupid}/instances/{name}/positions in the API reference documentation.
- POST /consumers/{groupid}/instances/{name}/positions/beginning in the API reference documentation.
- POST /consumers/{groupid}/instances/{name}/positions/end in the API reference documentation.

7.2.9. Deleting a Kafka Bridge consumer

Finally, delete the Kafa Bridge consumer that you used throughout this quickstart.

Procedure

- Delete the Kafka Bridge consumer by sending a **DELETE** request to the instances endpoint.

```
curl -X DELETE http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer
```

  If the request is successful, the Kafka Bridge returns a **204** code only.

Additional resources

- DELETE /consumers/{groupid}/instances/{name} in the API reference documentation.

CHAPTER 8. USING THE KAFKA BRIDGE WITH 3SCALE

You can deploy and integrate Red Hat 3scale API Management with the AMQ Streams Kafka Bridge.

8.1. USING THE KAFKA BRIDGE WITH 3SCALE

With a plain deployment of the Kafka Bridge, there is no provision for authentication or authorization, and no support for a TLS encrypted connection to external clients.

3scale can secure the Kafka Bridge with TLS, and provide authentication and authorization. Integration with 3scale also means that additional features like metrics, rate limiting and billing are available.

With 3scale, you can use different types of authentication for requests from external clients wishing to access AMQ Streams. 3scale supports the following types of authentication:

Standard API Keys

Single randomized strings or hashes acting as an identifier and a secret token.

Application Identifier and Key pairs

Immutable identifier and mutable secret key strings.

OpenID Connect

Protocol for delegated authentication.

Using an existing 3scale deployment?

If you already have 3scale deployed to OpenShift and you wish to use it with the Kafka Bridge, ensure that you have the correct setup.

Setup is described in Section 8.2, “Deploying 3scale for the Kafka Bridge” .

8.1.1. Kafka Bridge service discovery

3scale is integrated using service discovery, which requires that 3scale is deployed to the same OpenShift cluster as AMQ Streams and the Kafka Bridge.

Your AMQ Streams Cluster Operator deployment must have the following environment variables set:

- STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_LABELS
- STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_ANNOTATIONS

When the Kafka Bridge is deployed, the service that exposes the REST interface of the Kafka Bridge uses the annotations and labels for discovery by 3scale.

- A **discovery.3scale.net=true** label is used by 3scale to find a service.
- Annotations provide information about the service.

You can check your configuration in the OpenShift console by navigating to **Services** for the Kafka Bridge instance. Under **Annotations** you will see the endpoint to the OpenAPI specification for the Kafka Bridge.

8.1.2. 3scale APIcast gateway policies

3scale is used in conjunction with 3scale APIcast, an API gateway deployed with 3scale that provides a single point of entry for the Kafka Bridge.

APIcast policies provide a mechanism to customize how the gateway operates. 3scale provides a set of standard policies for gateway configuration. You can also create your own policies.

For more information on APIcast policies, see Administering the API Gateway in the 3scale documentation.

APIcast policies for the Kafka Bridge

A sample policy configuration for 3scale integration with the Kafka Bridge is provided with the **policies_config.json** file, which defines:

- Anonymous access
- Header modification
- Routing
- URL rewriting

Gateway policies are enabled or disabled through this file.

You can use this sample as a starting point for defining your own policies.

Anonymous access

The anonymous access policy exposes a service without authentication, providing default credentials (for anonymous access) when a HTTP client does not provide them. The policy is not mandatory and can be disabled or removed if authentication is always needed.

Header modification

The header modification policy allows existing HTTP headers to be modified, or new headers added to requests or responses passing through the gateway. For 3scale integration, the policy adds headers to every request passing through the gateway from a HTTP client to the Kafka Bridge. When the Kafka Bridge receives a request for creating a new consumer, it returns a JSON payload containing a **base_uri** field with the URI that the consumer must use for all the subsequent requests. For example:

```
{
  "instance_id": "consumer-1",
  "base_uri":"http://my-bridge:8080/consumers/my-group/instances/consumer1"
}
```

When using APIcast, clients send all subsequent requests to the gateway and not to the Kafka Bridge directly. So the URI requires the gateway hostname, not the address of the Kafka Bridge behind the gateway.

Using header modification policies, headers are added to requests from the HTTP client so that the Kafka Bridge uses the gateway hostname.

For example, by applying a **Forwarded: host=my-gateway:80;proto=http** header, the Kafka Bridge delivers the following to the consumer.

```
{
   "instance_id": "consumer-1",
   "base_uri":"http://my-gateway:80/consumers/my-group/instances/consumer1"
```

```
}
```

An **X-Forwarded-Path** header carries the original path contained in a request from the client to the gateway. This header is strictly related to the routing policy applied when a gateway supports more than one Kafka Bridge instance.

Routing

A routing policy is applied when there is more than one Kafka Bridge instance. Requests must be sent to the same Kafka Bridge instance where the consumer was initially created, so a request must specify a route for the gateway to forward a request to the appropriate Kafka Bridge instance.
A routing policy names each bridge instance, and routing is performed using the name. You specify the name in the **KafkaBridge** custom resource when you deploy the Kafka Bridge.

For example, each request (using **X-Forwarded-Path**) from a consumer to:

http://my-gateway:80/my-bridge-1/consumers/my-group/instances/consumer1

is forwarded to:

http://my-bridge-1-bridge-service:8080/consumers/my-group/instances/consumer1

URL rewriting policy removes the bridge name, as it is not used when forwarding the request from the gateway to the Kafka Bridge.

URL rewriting

The URL rewiring policy ensures that a request to a specific Kafka Bridge instance from a client does not contain the bridge name when forwarding the request from the gateway to the Kafka Bridge.
The bridge name is not used in the endpoints exposed by the bridge.

8.1.3. TLS validation

You can set up APIcast for TLS validation, which requires a self-managed deployment of APIcast using a template. The **apicast** service is exposed as a route.

You can also apply a TLS policy to the Kafka Bridge API.

For more information on TLS configuration, see Administering the API Gateway in the 3scale documentation.

8.1.4. 3scale documentation

The procedure to deploy 3scale for use with the Kafka Bridge assumes some understanding of 3scale.

For more information, refer to the 3scale product documentation:

- Product Documentation for Red Hat 3scale API Management

8.2. DEPLOYING 3SCALE FOR THE KAFKA BRIDGE

In order to use 3scale with the Kafka Bridge, you first deploy it and then configure it to discover the Kafka Bridge API.

You will also use 3scale APIcast and 3scale toolbox.

- APIcast is provided by 3scale as an NGINX-based API gateway for HTTP clients to connect to the Kafka Bridge API service.
- 3scale toolbox is a configuration tool that is used to import the OpenAPI specification for the Kafka Bridge service to 3scale.

In this scenario, you run AMQ Streams, Kafka, the Kafka Bridge and 3scale/APIcast in the same OpenShift cluster.

NOTE

If you already have 3scale deployed in the same cluster as the Kafka Bridge, you can skip the deployment steps and use your current deployment.

Prerequisites

- AMQ Streams and Kafka is running
- The Kafka Bridge is deployed

For the 3scale deployment:

- Check the Red Hat 3scale API Management supported configurations .
- Installation requires a user with **cluster-admin** role, such as **system:admin**.
- You need access to the JSON files describing the:
  - Kafka Bridge OpenAPI specification (**openapiv2.json**)
  - Header modification and routing policies for the Kafka Bridge (**policies_config.json**)
    Find the JSON files on GitHub.

Procedure

1. Deploy 3scale API Management to the OpenShift cluster.
   a. Create a new project or use an existing project.

```
oc new-project my-project \
  --description="description" --display-name="display_name"
```

   b. Deploy 3scale.
   Use the information provided in the Installing 3scale guide to deploy 3scale on OpenShift using a template or operator.

   Whichever approach you use, make sure that you set the *WILDCARD_DOMAIN* parameter to the domain of your OpenShift cluster.

   Make a note of the URLS and credentials presented for accessing the 3scale Admin Portal.

2. Grant authorization for 3scale to discover the Kafka Bridge service:

```
oc adm policy add-cluster-role-to-user view system:serviceaccount:my-project:amp
```

3. Verify that 3scale was successfully deployed to the Openshift cluster from the OpenShift console or CLI.
   For example:

```
oc get deployment 3scale-operator
```

4. Set up 3scale toolbox.

   a. Use the information provided in the Operating 3scale guide to install 3scale toolbox.

   b. Set environment variables to be able to interact with 3scale:

```
export REMOTE_NAME=strimzi-kafka-bridge (1)
export SYSTEM_NAME=strimzi_http_bridge_for_apache_kafka (2)
export TENANT=strimzi-kafka-bridge-admin (3)
export PORTAL_ENDPOINT=$TENANT.3scale.net (4)
export TOKEN=3scale access token (5)
```

   (1) **REMOTE_NAME** is the name assigned to the remote address of the 3scale Admin Portal.

   (2) **SYSTEM_NAME** is the name of the 3scale service/API created by importing the OpenAPI specification through the 3scale toolbox.

   (3) **TENANT** is the tenant name of the 3scale Admin Portal (that is, **https://$TENANT.3scale.net**).

   (4) **PORTAL_ENDPOINT** is the endpoint running the 3scale Admin Portal.

   (5) **TOKEN** is the access token provided by the 3scale Admin Portal for interaction through the 3scale toolbox or HTTP requests.

   c. Configure the remote web address of the 3scale toolbox:

```
3scale remote add $REMOTE_NAME https://$TOKEN@$PORTAL_ENDPOINT/
```

   Now the endpoint address of the 3scale Admin portal does not need to be specified every time you run the toolbox.

5. Check that your Cluster Operator deployment has the labels and annotations properties required for the Kafka Bridge service to be discovered by 3scale.

```
#...
env:
- name: STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_LABELS
    value: |
    discovery.3scale.net=true
- name: STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_ANNOTATIONS
    value: |
    discovery.3scale.net/scheme=http
    discovery.3scale.net/port=8080
    discovery.3scale.net/path=/
    discovery.3scale.net/description-path=/openapi
#...
```

If not, add the properties through the OpenShift console or try redeploying the Cluster Operator and the Kafka Bridge.

6. Discover the Kafka Bridge API service through 3scale.

   a. Log in to the 3scale Admin portal using the credentials provided when 3scale was deployed.

   b. From the 3scale Admin Portal, navigate to **New API → Import from OpenShift** where you will see the Kafka Bridge service.

   c. Click **Create Service**.
   You may need to refresh the page to see the Kafka Bridge service.

   Now you need to import the configuration for the service. You do this from an editor, but keep the portal open to check the imports are successful.

7. Edit the **Host** field in the OpenAPI specification (JSON file) to use the base URL of the Kafka Bridge service:
   For example:

   ```
   "host": "my-bridge-bridge-service.my-project.svc.cluster.local:8080"
   ```

   Check the **host** URL includes the correct:

   - Kafka Bridge name (*my-bridge*)
   - Project name (*my-project*)
   - Port for the Kafka Bridge (*8080*)

8. Import the updated OpenAPI specification using the 3scale toolbox:

   ```
   3scale import openapi -k -d $REMOTE_NAME openapiv2.json -t myproject-my-bridge-bridge-service
   ```

9. Import the header modification and routing policies for the service (JSON file).

   a. Locate the ID for the service you created in 3scale.
   Here we use the `jq` utility:

   ```
   export SERVICE_ID=$(curl -k -s -X GET "https://$PORTAL_ENDPOINT/admin/api/services.json?access_token=$TOKEN" | jq ".services[] | select(.service.system_name | contains(\"$SYSTEM_NAME\")) | .service.id")
   ```

   You need the ID when importing the policies.

   b. Import the policies:

   ```
   curl -k -X PUT "https://$PORTAL_ENDPOINT/admin/api/services/$SERVICE_ID/proxy/policies.json" --data "access_token=$TOKEN" --data-urlencode policies_config@policies_config.json
   ```

10. From the 3scale Admin Portal, navigate to **Integration → Configuration** to check that the endpoints and policies for the Kafka Bridge service have loaded.

11. Navigate to **Applications → Create Application Plan** to create an application plan.
12. Navigate to **Audience → Developer → Applications → Create Application** to create an application.
    The application is required in order to obtain a user key for authentication.
13. (Production environment step) To make the API available to the production gateway, promote the configuration:

    ```
    3scale proxy-config promote $REMOTE_NAME $SERVICE_ID
    ```

14. Use an API testing tool to verify you can access the Kafka Bridge through the APIcast gateway using a call to create a consumer, and the user key created for the application.
    For example:

    ```
    https//my-project-my-bridge-bridge-service-3scale-apicast-
    staging.example.com:443/consumers/my-group?
    user_key=3dfc188650101010ecd7fdc56098ce95
    ```

    If a payload is returned from the Kafka Bridge, the consumer was created successfully.

    ```
    {
      "instance_id": "consumer1",
      "base uri": "https//my-project-my-bridge-bridge-service-3scale-apicast-
    staging.example.com:443/consumers/my-group/instances/consumer1"
    }
    ```

    The base URI is the address that the client will use in subsequent requests.

CHAPTER 9. CRUISE CONTROL FOR CLUSTER REBALANCING

IMPORTANT

Cruise Control for cluster rebalancing is a Technology Preview only. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

You can deploy Cruise Control to your AMQ Streams cluster and use it to *rebalance* the Kafka cluster.

Cruise Control is an open source system for automating Kafka operations, such as monitoring cluster workload, rebalancing a cluster based on predefined constraints, and detecting and fixing anomalies. It 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 multiple *optimization goals*.
- Rebalancing a Kafka cluster based on an optimization proposal.

Other Cruise Control features are not currently supported, including anomaly detection, notifications, write-your-own goals, and changing the topic replication factor.

Example YAML files for Cruise Control are provided in **examples/cruise-control/**.

9.1. WHY USE CRUISE CONTROL?

Cruise Control reduces the time and effort involved in running an efficient and balanced Kafka cluster.

A typical cluster can become unevenly loaded over time. Partitions that handle large amounts of message traffic might be unevenly 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.

When you approve an optimization proposal, Cruise Control applies it to your Kafka cluster. When the cluster rebalancing operation is complete, the broker pods are used more effectively and the Kafka cluster is more evenly balanced.

Additional resources

- Cruise Control Wiki

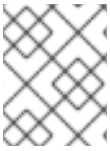
9.2. OPTIMIZATION GOALS OVERVIEW

To rebalance a Kafka cluster, Cruise Control uses optimization goals to generate optimization proposals, which you can approve or reject.

Optimization goals are constraints on workload redistribution and resource utilization across a Kafka cluster. With a few exceptions, AMQ Streams supports all 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. Replica capacity
3. Capacity: Disk capacity, Network inbound capacity, Network outbound capacity
4. Replica distribution
5. Potential network output
6. Resource distribution: Disk utilization distribution, Network inbound utilization distribution, Network outbound utilization distribution
7. Leader bytes-in rate distribution
8. Topic replica distribution
9. Leader replica distribution
10. Preferred leader election

For more information on each optimization goal, see Goals in the Cruise Control Wiki.



NOTE

CPU goals, intra-broker disk goals, "Write your own" goals, and Kafka assigner goals are not yet supported.

Goals configuration in AMQ Streams custom resources

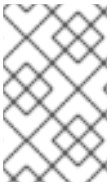
You configure optimization goals in **Kafka** and **KafkaRebalance** custom resources. Cruise Control has configurations for hard optimization goals that must be satisfied, as well as master, default, and user-provided optimization goals. Optimization goals are subject to any capacity limits on broker resources.

The following sections describe each goal configuration in more detail.

Hard goals and soft 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 master optimization goals are preset as hard goals:

```
RackAwareGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal;
NetworkOutboundCapacityGoal
```

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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      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.

Master optimization goals

The *master optimization goals* are available to all users. Goals that are not listed in the master 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 master optimization goals from Cruise Control, in descending priority order:

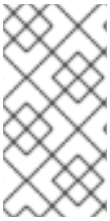
```
RackAwareGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal;
NetworkOutboundCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal;
DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal;
NetworkOutboundUsageDistributionGoal; TopicReplicaDistributionGoal;
LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal
```

Five of these goals are preset as hard goals.

To reduce complexity, we recommend that you use the inherited master optimization goals, unless you need to *completely* exclude one or more goals from use in **KafkaRebalance** resources. The priority order of the master optimization goals can be modified, if desired, in the configuration for default optimization goals.

You configure master optimization goals, if necessary, in the Cruise Control deployment configuration: **Kafka.spec.cruiseControl.config.goals**

- To accept the inherited master optimization goals, do not specify the **goals** property in **Kafka.spec.cruiseControl.config**.
- If you need to modify the inherited master optimization goals, specify a list of goals, in descending priority order, in the **goals** configuration option.



NOTE

If you change the inherited master optimization goals, you must ensure that the hard goals, if configured in the **hard.goals** property in **Kafka.spec.cruiseControl.config**, are a subset of the master optimization goals that you configured. Otherwise, errors will occur when generating optimization proposals.

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 9.3, "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 master optimization goals are used as the default optimization goals. In this case, the cached optimization proposal is generated using the master optimization goals.

- To use the master 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 master optimization goals.

Example Kafka configuration for default optimization goals

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
      # ...
```

If no default optimization goals are specified, the cached proposal is generated using the master optimization goals.

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 the **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 master optimization goals

To ignore the configured hard goals in an optimization proposal, add the **skipHardGoalCheck: true** option to the **KafkaRebalance** custom resource.

Additional resources

- Section 9.5, “Cruise Control configuration”
- Configurations in the Cruise Control Wiki.

9.3. OPTIMIZATION PROPOSALS OVERVIEW

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.

An optimization proposal is contained in the **Status.Optimization Result** property of a **KafkaRebalance** custom resource. The information provided is a summary of the full optimization proposal. Use the summary to decide whether to:

- Approve the optimization proposal. This instructs Cruise Control to apply the proposal to the Kafka cluster and start a cluster rebalance operation.
- Reject the optimization proposal. You can change the optimization goals and then generate another proposal.

All optimization proposals are *dry runs*: you cannot approve a cluster rebalance without first generating an optimization proposal. There is no limit to the number of optimization proposals that can be generated.

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.

Contents of optimization proposals

The following table explains the properties contained in an optimization proposal:

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.
numReplicaMovements	The number of partition replicas that will be moved between separate brokers. **Performance impact during rebalance operation:** Relatively high.

JSON property	Description
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	Not yet supported. An empty list is returned.
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.
excludedBrokersForReplicaMove	Not yet supported. An empty list is returned.

Additional resources

- Section 9.2, "Optimization goals overview"

- Section 9.6, “Generating optimization proposals”
- Section 9.7, “Approving an optimization proposal”

9.4. DEPLOYING CRUISE CONTROL

To deploy Cruise Control to your AMQ Streams cluster, define the configuration using the **cruiseControl** property in the **Kafka** resource, and then create or update the resource.

Deploy one instance of Cruise Control per Kafka cluster.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **Kafka** resource and add the **cruiseControl** property.
   The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    brokerCapacity: 1
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
      # ...
    config: 2
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal
        # ...
      cpu.balance.threshold: 1.1
      metadata.max.age.ms: 300000
      send.buffer.bytes: 131072
      # ...
    resources: 3
      requests:
        cpu: 200m
        memory: 64Mi
      limits:
        cpu: 500m
        memory: 128Mi
    logging: 4
        type: inline
        loggers:
          cruisecontrol.root.logger: "INFO"
    template: 5
```

```
      pod:
        metadata:
          labels:
            label1: value1
        securityContext:
          runAsUser: 1000001
          fsGroup: 0
        terminationGracePeriodSeconds: 120
    readinessProbe: 6
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe: 7
      initialDelaySeconds: 15
      timeoutSeconds: 5
# ...
```

1. Specifies capacity limits for broker resources. For more information, see Capacity configuration.
2. Defines the Cruise Control configuration, including the default optimization goals (in **default.goals**) and any customizations to the master optimization goals (in **goals**) or the hard goals (in **hard.goals**). You can provide any standard Cruise Control configuration option apart from those managed directly by AMQ Streams. For more information on configuring optimization goals, see Section 9.2, "Optimization goals overview" .
3. CPU and memory resources reserved for Cruise Control. For more information, see Section 3.1.12, "CPU and memory resources".
4. Defined 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 **cruisecontrol.root.logger**. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF. For more information, see Logging configuration.
5. Customization of deployment templates and pods.
6. Healthcheck readiness probes.
7. Healthcheck liveness probes.

2. Create or update the resource:

```
oc apply -f kafka.yaml
```

3. Verify that Cruise Control was successfully deployed:

```
oc get deployments -l app.kubernetes.io/name=strimzi
```

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.

Auto-created topic	Created by	Function
strimzi.cruisecontrol.metrics	AMQ Streams Metrics Reporter	Stores the raw metrics from the Metrics Reporter in each Kafka broker.
strimzi.cruisecontrol.partitionmetricsamples	Cruise Control	Stores the derived metrics for each partition. These are created by the Metric Sample Aggregator.
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.

What to do next

After configuring and deploying Cruise Control, you can generate optimization proposals.

Additional resources

Section B.66, "**CruiseControlTemplate** schema reference".

9.5. CRUISE CONTROL CONFIGURATION

The **config** property in **Kafka.spec.cruiseControl** contains configuration options as keys with values as one of the following JSON types:

- String
- Number
- Boolean

NOTE

Strings that look like JSON or YAML will need to be explicitly quoted.

You can specify and configure all the options listed in the "Configurations" section of the Cruise Control documentation, apart from those managed directly by AMQ Streams. Specifically, you **cannot** modify configuration options with keys equal to or starting with one of the following strings:

- **bootstrap.servers**
- **zookeeper.**
- **ssl.**
- **security.**
- **failed.brokers.zk.path**

- **webserver.http.port**
- **webserver.http.address**
- **webserver.api.urlprefix**
- **metric.reporter.sampler.bootstrap.servers**
- **metric.reporter.topic**
- **metric.reporter.topic.pattern**
- **partition.metric.sample.store.topic**
- **broker.metric.sample.store.topic**
- **capacity.config.file**
- **skip.sample.store.topic.rack.awareness.check**
- **cruise.control.metrics.topic**
- **sasl.**

If restricted options are specified, they are ignored and a warning message is printed to the Cluster Operator log file. All the supported options are passed to Cruise Control.

An example Cruise Control configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    config:
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal
      cpu.balance.threshold: 1.1
      metadata.max.age.ms: 300000
      send.buffer.bytes: 131072
    # ...
```

Capacity configuration

Cruise Control uses *capacity limits* to determine if certain resource-based optimization goals are being broken.

You specify capacity limits for Kafka broker resources in the **brokerCapacity** property in **Kafka.spec.cruiseControl** . Capacity limits can be set for the following broker resources in the described units:

- **disk** - Disk storage in bytes
- **cpuUtilization** - CPU utilization as a percent (0-100)

- **inboundNetwork** - Inbound network throughput in bytes per second
- **outboundNetwork** - Outbound network throughput in bytes per second

Because AMQ Streams Kafka brokers are homogeneous, Cruise Control applies the same capacity limits to every broker it is monitoring.

An example Cruise Control brokerCapacity configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    brokerCapacity:
      disk: 100G
      cpuUtilization: 100
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    # ...
```

Additional resources

For more information, refer to the Section B.67, "**BrokerCapacity** schema reference".

Logging configuration

Cruise Control has its own configurable logger:

- **cruisecontrol.root.logger**

Cruise Control uses the Apache **log4j** logger implementation.

Use the **logging** property to configure loggers and logger levels.

You can set the log levels by specifying the logger and level directly (inline) or use a custom (external) ConfigMap. If a ConfigMap is used, you set **logging.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**.

Here we see examples of **inline** and **external** logging.

Inline logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
# ...
spec:
  cruiseControl:
    # ...
    logging:
      type: inline
```

```
    loggers:
      cruisecontrol.root.logger: "INFO"
  # ...
```

External logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
# ...
spec:
  cruiseControl:
    # ...
    logging:
      type: external
      name: customConfigMap
    # ...
```

9.6. 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 summary information in the optimization proposal and decide whether to approve it.

Prerequisites

- You have deployed Cruise Control to your AMQ Streams cluster.
- You have configured optimization goals and, optionally, capacity limits on broker resources.

Procedure

1. Create a **KafkaRebalance** resource:
   a. To use the *default optimization goals* defined in the **Kafka** resource, leave the **spec** property empty:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec: {}
```

   b. 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/v1alpha1
kind: KafkaRebalance
metadata:
  name: my-rebalance
```

```
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
```

2. Create or update the resource:

```
oc apply -f your-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.

3. Check the status of the **KafkaRebalance** resource:

```
oc describe kafkarebalance rebalance-cr-name
```

Cruise Control returns one of two statuses:

- **PendingProposal**: The rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.
- **ProposalReady**: The optimization proposal is ready for review and, if desired, approval. The optimization proposal is contained in the **Status.Optimization Result** property of the **KafkaRebalance** resource.

4. Review the optimization proposal.

```
oc describe kafkarebalance rebalance-cr-name
```

Here is an example 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.

What to do next

Section 9.7, "Approving an optimization proposal"

Additional resources

- Section 9.3, "Optimization proposals overview"

9.7. 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 **refresh**:

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=refresh
```

   b. Check the status of the **KafkaRebalance** resource:

```
oc describe kafkarebalance rebalance-cr-name
```

   c. Wait until the status changes to **ProposalReady**.
2. Approve the optimization proposal that you want Cruise Control to apply. Annotate the **KafkaRebalance** resource in OpenShift:

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=approve
```

3. The Cluster Operator detects the annotated resource and instructs Cruise Control to rebalance the Kafka cluster.

4. Check the status of the **KafkaRebalance** resource:

```
oc describe kafkarebalance rebalance-cr-name
```

5. Cruise Control returns one of three statuses:

   - Rebalancing: The cluster rebalance operation is in progress.
   - Ready: The cluster rebalancing operation completed successfully.
   - NotReady: An error occurred—see Section 9.9, "Fixing problems with a **KafkaRebalance** resource".

Additional resources

- Section 9.3, "Optimization proposals overview"
- Section 9.8, "Stopping a cluster rebalance"

9.8. 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 9.3, "Optimization proposals overview"

9.9. 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. Problems might include the following:

- A misconfigured parameter.
- 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 9.3, "Optimization proposals overview"

CHAPTER 10. MANAGING SCHEMAS WITH SERVICE REGISTRY

This chapter outlines how to deploy and integrate AMQ Streams with Red Hat Service Registry. You can use Service Registry as a centralized store of service schemas for data streaming.

Service Registry supports the storage and management of many standard artifact types. For example, for Kafka you can use schema definitions based on **AVRO** or **JSON**.

Service Registry provides a REST API and a Java REST client to register and query the schemas from client applications through server-side endpoints. You can also use the Service Registry web console to browse and update schemas directly. You can configure producer and consumer clients to use Service Registry.

A Maven plugin is also provided so that you can upload and download schemas as part of your build. The Maven plugin is useful for testing and validation, when checking that your schema updates are compatible with client applications.

Additional resources

- Service Registry documentation
- Service Registry is built on the Apicurio Registry open source community project available from GitHub: Apicurio/apicurio-registry
- A demo of Service Registry is also available from GitHub: Apicurio/apicurio-registry-demo
- Apache Avro

10.1. WHY USE SERVICE REGISTRY?

Using Service Registry decouples the process of managing schemas from the configuration of client applications. You enable an application to use a schema from the registry by specifying its URL in the client code.

For example, the schemas to serialize and deserialize messages can be stored in the registry, which are then referenced from the applications that use them to ensure that the messages that they send and receive are compatible with those schemas.

Kafka client applications can push or pull their schemas from Service Registry at runtime.

Schemas can evolve, so you can define rules in Service Registry, for example, to ensure that changes to a schema are valid and do not break previous versions used by applications. Service Registry checks for compatibility by comparing a modified schema with previous versions of schemas.

Service Registry provides full schema registry support for Avro schemas, which are used by client applications through Kafka client serializer/deserializer (SerDe) services provided by Service Registry.

10.2. PRODUCER SCHEMA CONFIGURATION

A producer client application uses a serializer to put the messages it sends to a specific broker topic into the correct data format.

To enable a producer to use Service Registry for serialization, you:

- Define and register your schema with Service Registry

- Configure the producer client code with the:
  - URL of Service Registry
  - Service Registry serializer services to use with the messages
  - *Strategy* to look up the schema used for serialization in Service Registry

After registering your schema, when you start Kafka and Service Registry, you can access the schema to format messages sent to the Kafka broker topic by the producer.

If a schema already exists, you can create a new version through the REST API based on compatibility rules defined in Service Registry. Versions are used for compatibility checking as a schema evolves. An artifact ID and schema version represents a unique tuple that identifies a schema.

10.3. CONSUMER SCHEMA CONFIGURATION

A consumer client application uses a deserializer to get the messages it consumes from a specific broker topic into the correct data format.

To enable a consumer to use Service Registry for deserialization, you:

- Define and register your schema with Service Registry
- Configure the consumer client code with the:
  - URL of Service Registry
  - Service Registry deserializer service to use with the messages
  - Input data stream for deserialization

The schema is then retrieved by the deserializer using a global ID written into the message being consumed. The message received must, therefore, include a global ID as well as the message data.

For example:

```
# ...
[MAGIC_BYTE]
[GLOBAL_ID]
[MESSAGE DATA]
```

Now, when you start Kafka and Service Registry, you can access the schema in order to format messages received from the Kafka broker topic.

10.4. STRATEGIES TO LOOKUP A SCHEMA

A Service Registry *strategy* is used by the Kafka client serializer/deserializer to determine the artifact ID or global ID under which the message schema is registered in Service Registry.

For a given topic and message, you can use implementations of the following Java classes:

- **ArtifactIdStrategy** to return an artifact ID
- **GlobalIdStrategy** to return a global ID

The artifact ID returned depends on whether the *key* or *value* in the message is being serialized.

The classes for each *strategy* are organized in the **io.apicurio.registry.utils.serde.strategy** package.

The default strategy is **TopicIdStrategy**, which looks for Service Registry artifacts with the same name as the Kafka topic receiving messages.

For example:

```
public String artifactId(String topic, boolean isKey, T schema) {
    return String.format("%s-%s", topic, isKey ? "key" : "value");
}
```

- The **topic** parameter is the name of the Kafka topic receiving the message.
- The **isKey** parameter is *true* when the message key is being serialized, and *false* when the message value is being serialized.
- The **schema** parameter is the schema of the message being serialized/deserialized.
- The **artifactID** returned is the ID under which the schema is registered in Service Registry.

What lookup strategy you use depends on how and where you store your schema. For example, you might use a strategy that uses a *record ID* if you have different Kafka topics with the same Avro message type.

Strategies to return an artifact ID

Strategies to return an artifact ID based on an implementation of **ArtifactIdStrategy**.

RecordIdStrategy

Avro-specific strategy that uses the full name of the schema.

TopicRecordIdStrategy

Avro-specific strategy that uses the topic name and the full name of the schema.

TopicIdStrategy

(Default) strategy that uses the topic name and **key** or **value** suffix.

SimpleTopicIdStrategy

Simple strategy that only uses the topic name.

Strategies to return a global ID

Strategies to return a global ID based on an implementation of **GlobalIdStrategy**.

FindLatestIdStrategy

Strategy that returns the global ID of the latest schema version, based on an artifact ID.

FindBySchemaIdStrategy

Strategy that matches schema content, based on an artifact ID, to return a global ID.

GetOrCreateIdStrategy

Strategy that tries to get the latest schema, based on an artifact ID, and if it does not exist, it creates a new schema.

AutoRegisterIdStrategy

Strategy that updates the schema, and uses the global ID of the updated schema.

10.5. SERVICE REGISTRY CONSTANTS

You can configure specific client SerDe services and schema lookup strategies directly into a client using the constants outlined here.

Alternatively, you can use specify the constants in a properties file, or a properties instance.

Constants for serializer/deserializer (SerDe) services

```
public abstract class AbstractKafkaSerDe<T extends AbstractKafkaSerDe<T>> implements
AutoCloseable {
    protected final Logger log = LoggerFactory.getLogger(getClass());

    public static final String REGISTRY_URL_CONFIG_PARAM = "apicurio.registry.url"; (1)
    public static final String REGISTRY_CACHED_CONFIG_PARAM = "apicurio.registry.cached";
(2)
    public static final String REGISTRY_ID_HANDLER_CONFIG_PARAM = "apicurio.registry.id-
handler"; (3)
    public static final String REGISTRY_CONFLUENT_ID_HANDLER_CONFIG_PARAM =
"apicurio.registry.as-confluent"; (4)
```

1. (Required) The URL of Service Registry.
2. Allows the client to make the request and look up the information from a cache of previous results, to improve processing time. If the cache is empty, the lookup is performed from Service Registry.
3. Extends ID handling to support other ID formats and make them compatible with Service Registry SerDe services. For example, changing the ID format from **Long** to **Integer** supports the Confluent ID format.
4. A flag to simplify the handling of Confluent IDs. If set to **true**, an **Integer** is used for the global ID lookup.

Constants for lookup strategies

```
public abstract class AbstractKafkaStrategyAwareSerDe<T, S extends
AbstractKafkaStrategyAwareSerDe<T, S>> extends AbstractKafkaSerDe<S> {
    public static final String REGISTRY_ARTIFACT_ID_STRATEGY_CONFIG_PARAM =
"apicurio.registry.artifact-id"; (1)
    public static final String REGISTRY_GLOBAL_ID_STRATEGY_CONFIG_PARAM =
"apicurio.registry.global-id"; (2)
```

1. ArtifactId strategy.
2. Global ID strategy.

Constants for converters

```
public class SchemalessConverter<T> extends AbstractKafkaSerDe<SchemalessConverter<T>>
implements Converter {
    public static final String REGISTRY_CONVERTER_SERIALIZER_PARAM =
```

```
"apicurio.registry.converter.serializer"; (1)
    public static final String REGISTRY_CONVERTER_DESERIALIZER_PARAM =
"apicurio.registry.converter.deserializer"; (2)
```

(1) (Required) Serializer to use with the converter.

(2) (Required) Deserializer to use with the converter.

Constants for Avro data providers

```
public interface AvroDatumProvider<T> {
    String REGISTRY_AVRO_DATUM_PROVIDER_CONFIG_PARAM = "apicurio.registry.avro-
datum-provider"; (1)
    String REGISTRY_USE_SPECIFIC_AVRO_READER_CONFIG_PARAM = "apicurio.registry.use-
specific-avro-reader"; (2)
```

(1) Avro Datum provider to write data to a schema, with or without reflection.

(2) Flag to set to use an Avro-specific datum reader.

```
DefaultAvroDatumProvider (io.apicurio.registry.utils.serde.avro) (1)
ReflectAvroDatumProvider (io.apicurio.registry.utils.serde.avro) (2)
```

(1) Default datum reader.

(2) Datum reader using reflection.

10.6. INSTALLING SERVICE REGISTRY

The instructions to install Service Registry with AMQ Streams storage are described in the Service Registry documentation.

You can install more than one instance of Service Registry depending on your cluster configuration. The number of instances depends on the storage type you use and how many schemas you need to handle.

10.7. REGISTERING A SCHEMA TO SERVICE REGISTRY

After you have defined a schema in the appropriate format, such as *Apache Avro*, you can add the schema to Service Registry.

You can add the schema through:

- The Service Registry web console
- A curl command using the Service Registry API
- A Maven plugin supplied with Service Registry
- Schema configuration added to your client code

Client applications cannot use Service Registry until you have registered your schemas.

Service Registry web console

Having installed Service Registry, you connect to the web console from the **ui** endpoint:

http://*MY-REGISTRY-URL*/ui

From the console, you can add, view and configure schemas. You can also create the rules that prevent invalid content being added to the registry.

For more information on using the Service Registry web console, see the Service Registry documentation.

Curl example

```
curl -X POST -H "Content-type: application/json; artifactType=AVRO" \
  -H "X-Registry-ArtifactId: prices-value" \
  --data '{ 1
     "type":"record",
     "name":"price",
     "namespace":"com.redhat",
     "fields":[{"name":"symbol","type":"string"},
     {"name":"price","type":"string"}]
   }'
  https://my-cluster-service-registry-myproject.example.com/api/artifacts -s 2
```

1 Avro schema

2 OpenShift route name that exposes Service Registry

Plugin example

```
<plugin>
<groupId>io.apicurio</groupId>
<artifactId>apicurio-registry-maven-plugin</artifactId>
<version>${registry.version}</version>
<executions>
  <execution>
    <phase>generate-sources</phase>
    <goals>
      <goal>register</goal>
    </goals>
    <configuration>
      <registryUrl>https://my-cluster-service-registry-myproject.example.com/api</registryUrl>
      <artifactType>AVRO</artifactType>
      <artifacts>
        <schema1>${project.basedir}/schemas/schema1.avsc</schema1>
      </artifacts>
    </configuration>
  </execution>
</executions>
</plugin>
```

Configuration through a (producer) client example

```
String registryUrl_node1 = PropertiesUtil.property(clientProperties, "registry.url.node1", 1
```

```
    "https://my-cluster-service-registry-myproject.example.com/api");
try (RegistryService service = RegistryClient.create(registryUrl_node1)) {
    String artifactId = ApplicationImpl.INPUT_TOPIC + "-value";
    try {
        service.getArtifactMetaData(artifactId); 2
    } catch (WebApplicationException e) {
        CompletionStage <ArtifactMetaData> csa = service.createArtifact(
            ArtifactType.AVRO,
            artifactId,
            new ByteArrayInputStream(LogInput.SCHEMA$.toString().getBytes())
        );
        csa.toCompletableFuture().get();
    }
}
```

1 The properties are registered. You can register properties against more than one node.

2 Check to see if the schema already exists based on the artifact ID.

10.8. USING A SERVICE REGISTRY SCHEMA FROM A PRODUCER CLIENT

This procedure describes how to configure a Java producer client to use a schema from Service Registry.

Prerequisites

- Service Registry is installed
- The schema is registered with Service Registry

Procedure

1. Configure the client with the URL of Service Registry.
   For example:

```
String registryUrl_node1 = PropertiesUtil.property(clientProperties, "registry.url.node1",
    "https://my-cluster-service-registry-myproject.example.com/api");
RegistryService service = RegistryClient.cached(registryUrl);
```

2. Configure the client with the serializer services, and the strategy to look up the schema in Service Registry.
   For example:

```
String registryUrl_node1 = PropertiesUtil.property(clientProperties, "registry.url.node1",
    "https://my-cluster-service-registry-myproject.example.com/api");

    clientProperties.put(CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG,
property(clientProperties, CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG, "my-
cluster-kafka-bootstrap:9092"));
    clientProperties.put(AbstractKafkaSerDe.REGISTRY_URL_CONFIG_PARAM,
registryUrl_node1); 1
    clientProperties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName()); 2
```

```
    clientProperties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
AvroKafkaSerializer.class.getName()); 3

clientProperties.put(AbstractKafkaSerializer.REGISTRY_GLOBAL_ID_STRATEGY_CONFIG_
PARAM, FindLatestIdStrategy.class.getName()); 4
```

1. The Service Registry URL.
2. The serializer service for the message *key* provided by Service Registry.
3. The serializer service for the message *value* provided by Service Registry.
4. Lookup strategy to find the global ID for the schema. Matches the schema of the message against its global ID (artifact ID and schema version) in Service Registry.

10.9. USING A SERVICE REGISTRY SCHEMA FROM A CONSUMER CLIENT

This procedure describes how to configure a Java consumer client to use a schema from Service Registry.

Prerequisites

- Service Registry is installed
- The schema is registered with Service Registry

Procedure

1. Configure the client with the URL of Service Registry.
   For example:

```
String registryUrl_node1 = PropertiesUtil.property(clientProperties, "registry.url.node1",
    "https://my-cluster-service-registry-myproject.example.com/api");
RegistryService service = RegistryClient.cached(registryUrl);
```

2. Configure the client with the Service Registry deserializer service.
   For example:

```
Deserializer<LogInput> deserializer = new AvroKafkaDeserializer <> ( 1
    service,
    new DefaultAvroDatumProvider<LogInput>().setUseSpecificAvroReader(true)
);
Serde<LogInput> logSerde = Serdes.serdeFrom( 2
    new AvroKafkaSerializer<>(service),
    deserializer
);
KStream<String, LogInput> input = builder.stream( 3
    INPUT_TOPIC,
    Consumed.with(Serdes.String(), logSerde)
);
```

1. The deserializer service provided by Service Registry.
2. The deserialization is in *Apache Avro* JSON format.
3. The input data for deserialization derived from the topic values consumed by the client.

CHAPTER 11. DISTRIBUTED TRACING

This chapter outlines the support for distributed tracing in AMQ Streams, using Jaeger.

How you configure distributed tracing varies by AMQ Streams client and component.

- You *instrument* Kafka Producer, Consumer, and Streams API applications for distributed tracing using an OpenTracing client library. This involves adding instrumentation code to these clients, which monitors the execution of individual transactions in order to generate trace data.
- Distributed tracing support is built in to the Kafka Connect, MirrorMaker, and Kafka Bridge components of AMQ Streams. To configure these components for distributed tracing, you configure and update the relevant custom resources.

Before configuring distributed tracing in AMQ Streams clients and components, you must first initialize and configure a Jaeger tracer in the Kafka cluster, as described in Initializing a Jaeger tracer for Kafka clients.

NOTE

Distributed tracing is not supported for Kafka brokers.

11.1. OVERVIEW OF DISTRIBUTED TRACING IN AMQ STREAMS

Distributed tracing allows developers and system administrators to track the progress of transactions between applications (and services in a microservice architecture) in a distributed system. This information is useful for monitoring application performance and investigating issues with target systems and end-user applications.

In AMQ Streams and data streaming platforms in general, distributed tracing facilitates the end-to-end tracking of messages: from source systems to the Kafka cluster and then to target systems and applications.

As an aspect of system observability, distributed tracing complements the metrics that are available to view in Grafana dashboards and the available loggers for each component.

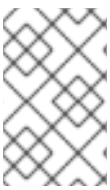
OpenTracing overview

Distributed tracing in AMQ Streams is implemented using the open source OpenTracing and Jaeger projects.

The OpenTracing specification defines APIs that developers can use to instrument applications for distributed tracing. It is independent from the tracing system.

When instrumented, applications generate *traces* for individual transactions. Traces are composed of *spans*, which define specific units of work.

To simplify the instrumentation of the Kafka Bridge and Kafka Producer, Consumer, and Streams API applications, AMQ Streams includes the OpenTracing Apache Kafka Client Instrumentation library.



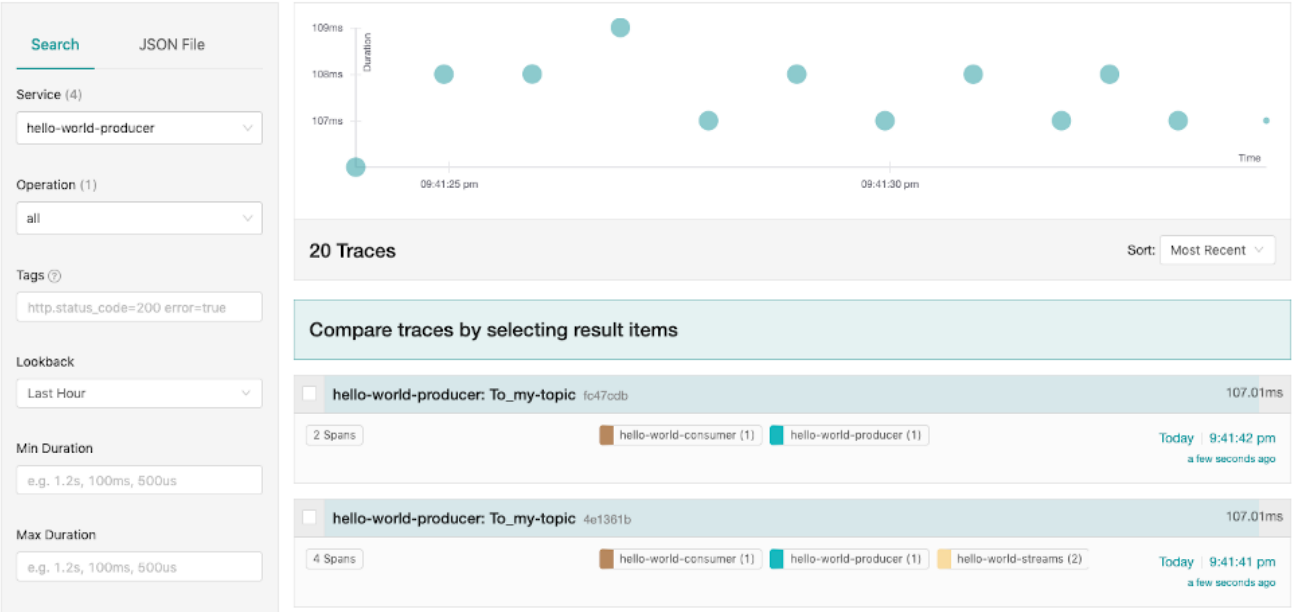
NOTE

The OpenTracing project is merging with the OpenCensus project. The new, combined project is named OpenTelemetry. OpenTelemetry will provide compatibility for applications that are instrumented using the OpenTracing APIs.

Jaeger overview

Jaeger, a tracing system, is an implementation of the OpenTracing APIs used for monitoring and troubleshooting microservices-based distributed systems. It consists of four main components and provides client libraries for instrumenting applications. You can use the Jaeger user interface to visualize, query, filter, and analyze trace data.

An example of a query in the Jaeger user interface



11.1.1. Distributed tracing support in AMQ Streams

In AMQ Streams, distributed tracing is supported in:

- Kafka Connect (including Kafka Connect with Source2Image support)
- MirrorMaker
- The AMQ Streams Kafka Bridge

You enable and configure distributed tracing for these components by setting template configuration properties in the relevant custom resource (for example, **KafkaConnect** and **KafkaBridge**).

To enable distributed tracing in Kafka Producer, Consumer, and Streams API applications, you can instrument application code using the OpenTracing Apache Kafka Client Instrumentation library. When instrumented, these clients generate traces for messages (for example, when producing messages or writing offsets to the log).

Traces are sampled according to a sampling strategy and then visualized in the Jaeger user interface. This trace data is useful for monitoring the performance of your Kafka cluster and debugging issues with target systems and applications.

Outline of procedures

To set up distributed tracing for AMQ Streams, follow these procedures:

- Initialize a Jaeger tracer for Kafka clients
- Instrument Kafka Producers and Consumers for tracing

- Instrument Kafka Streams applications for tracing
- Set up tracing for MirrorMaker, Kafka Connect, and the Kafka Bridge

This chapter covers setting up distributed tracing for AMQ Streams clients and components only. Setting up distributed tracing for applications and systems beyond AMQ Streams is outside the scope of this chapter. To learn more about this subject, see the OpenTracing documentation and search for "inject and extract".

Before you start

Before you set up distributed tracing for AMQ Streams, it is helpful to understand:

- The basics of OpenTracing, including key concepts such as traces, spans, and tracers. Refer to the OpenTracing documentation.
- The components of the Jaeger architecure.

Prerequisites

- The Jaeger backend components are deployed to your OpenShift cluster. For deployment instructions, see the Jaeger deployment documentation.

11.2. SETTING UP TRACING FOR KAFKA CLIENTS

This section describes how to initialize a Jaeger tracer to allow you to instrument your client applications for distributed tracing.

11.2.1. Initializing a Jaeger tracer for Kafka clients

Configure and initialize a Jaeger tracer using a set of tracing environment variables.

Procedure

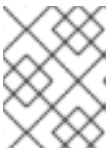
Perform the following steps for each client application.

1. Add Maven dependencies for Jaeger to the **pom.xml** file for the client application:

```
<dependency>
    <groupId>io.jaegertracing</groupId>
    <artifactId>jaeger-client</artifactId>
    <version>1.1.0.redhat-00002</version>
</dependency>
```

2. Define the configuration of the Jaeger tracer using the tracing environment variables.
3. Create the Jaeger tracer from the environment variables that you defined in step two:

```
Tracer tracer = Configuration.fromEnv().getTracer();
```



NOTE

For alternative ways to initialize a Jaeger tracer, see the Java OpenTracing library documentation.

4. Register the Jaeger tracer as a global tracer:

```
GlobalTracer.register(tracer);
```

A Jaeger tracer is now initialized for the client application to use.

11.2.2. Tracing environment variables

Use these environment variables when configuring a Jaeger tracer for Kafka clients.

NOTE

The tracing environment variables are part of the Jaeger project and are subject to change. For the latest environment variables, see the Jaeger documentation.

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.
JAEGER_ENDPOINT	No	The traces endpoint. Only define this variable if the client application will bypass the **jaeger-agent** and connect directly to the **jaeger-collector**.
JAEGER_AUTH_TOKEN	No	The authentication token to send to the endpoint as a bearer token.
JAEGER_USER	No	The username to send to the endpoint if using basic authentication.
JAEGER_PASSWORD	No	The password to send to the endpoint if using basic authentication.

Property	Required	Description
JAEGER_PROPAGATION	No	A comma-separated list of formats to use for propagating the trace context. Defaults to the standard Jaeger format. Valid values are **jaeger** and **b3**.
JAEGER_REPORTER_LOG_SPANS	No	Indicates whether the reporter should also log the spans.
JAEGER_REPORTER_MAX_QUEUE_SIZE	No	The reporter's maximum queue size.
JAEGER_REPORTER_FLUSH_INTERVAL	No	The reporter's flush interval, in ms. Defines how frequently the Jaeger reporter flushes span batches.
JAEGER_SAMPLER_TYPE	No	The sampling strategy to use for client traces: Constant, Probabilistic, Rate Limiting, or Remote (the default type). To sample all traces, use the Constant sampling strategy with a parameter of 1. For more information, see the Jaeger documentation.
JAEGER_SAMPLER_PARAM	No	The sampler parameter (number).
JAEGER_SAMPLER_MANAGER_HOST_PORT	No	The hostname and port to use if a Remote sampling strategy is selected.
JAEGER_TAGS	No	A comma-separated list of tracer-level tags that are added to all reported spans. The value can also refer to an environment variable using the format **${envVarName:default}**. **:default** is optional and identifies a value to use if the environment variable cannot be found.

Additional resources

- Section 11.2.1, "Initializing a Jaeger tracer for Kafka clients"

11.3. INSTRUMENTING KAFKA CLIENTS WITH TRACERS

This section describes how to instrument Kafka Producer, Consumer, and Streams API applications for distributed tracing.

11.3.1. Instrumenting Kafka Producers and Consumers for tracing

Use a Decorator pattern or Interceptors to instrument your Java Producer and Consumer application code for distributed tracing.

Procedure

Perform these steps in the application code of each Kafka Producer and Consumer application.

1. Add the Maven dependency for OpenTracing to the Producer or Consumer's **pom.xml** file.

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-kafka-client</artifactId>
    <version>0.1.12.redhat-00001</version>
</dependency>
```

2. Instrument your client application code using either a Decorator pattern or Interceptors.

   - If you prefer to use a Decorator pattern, use following example:

```
// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer:
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>
(producer,
        tracer);

// Send:
tracingProducer.send(...);

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer:
TracingKafkaConsumer<Integer, String> tracingConsumer = new
TracingKafkaConsumer<>(consumer,
        tracer);

// Subscribe:
tracingConsumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = tracingConsumer.poll(1000);

// Retrieve SpanContext from polled record (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

- If you prefer to use Interceptors, use the following example:

```
// Register the tracer with GlobalTracer:
GlobalTracer.register(tracer);

// Add the TracingProducerInterceptor to the sender properties:
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
        TracingProducerInterceptor.class.getName());

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Send:
producer.send(...);

// Add the TracingConsumerInterceptor to the consumer properties:
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
        TracingConsumerInterceptor.class.getName());

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Subscribe:
consumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = consumer.poll(1000);

// Retrieve the SpanContext from a polled message (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

11.3.1.1. Custom span names in a Decorator pattern

A *span* is a logical unit of work in Jaeger, with an operation name, start time, and duration.

If you use a Decorator pattern to instrument your Kafka Producer and Consumer applications, you can define custom span names by passing a **BiFunction** object as an additional argument when creating the **TracingKafkaProducer** and **TracingKafkaConsumer** objects. The OpenTracing Apache Kafka Client Instrumentation library includes several built-in span names, which are described below.

Example: Using custom span names to instrument client application code in a Decorator pattern

```
// Create a BiFunction for the KafkaProducer that operates on (String operationName,
ProducerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ProducerRecord, String> producerSpanNameProvider =
    (operationName, producerRecord) -> "CUSTOM_PRODUCER_NAME";

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer
```

```
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>(producer,
        tracer,
        producerSpanNameProvider);

// Spans created by the tracingProducer will now have "CUSTOM_PRODUCER_NAME" as the span
name.

// Create a BiFunction for the KafkaConsumer that operates on (String operationName,
ConsumerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ConsumerRecord, String> consumerSpanNameProvider =
    (operationName, consumerRecord) -> operationName.toUpperCase();

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer, passing in the consumerSpanNameProvider
BiFunction:

TracingKafkaConsumer<Integer, String> tracingConsumer = new TracingKafkaConsumer<>
(consumer,
        tracer,
        consumerSpanNameProvider);

// Spans created by the tracingConsumer will have the operation name as the span name, in upper-
case.
// "receive" -> "RECEIVE"
```

11.3.1.2. Built-in span names

When defining custom span names, you can use the following **BiFunctions** in the **ClientSpanNameProvider** class. If no **spanNameProvider** is specified, **CONSUMER_OPERATION_NAME** and **PRODUCER_OPERATION_NAME** are used.

BiFunction	Description
CONSUMER_OPERATION_NAME, PRODUCER_OPERATION_NAME	Returns the **operationName** as the span name: "receive" for Consumers and "send" for Producers.
CONSUMER_PREFIXED_OPERATION_NAME (String prefix), PRODUCER_PREFIXED_OPERATION_NAME(String prefix)	Returns a String concatenation of **prefix** and **operationName**.
CONSUMER_TOPIC, PRODUCER_TOPIC	Returns the name of the topic that the message was sent to or retrieved from in the format **(record.topic())**.
PREFIXED_CONSUMER_TOPIC(String prefix), PREFIXED_PRODUCER_TOPIC(String prefix)	Returns a String concatenation of **prefix** and the topic name in the format **(record.topic())**.

BiFunction	Description
CONSUMER_OPERATION_NAME_TOPIC, PRODUCER_OPERATION_NAME_TOPIC	Returns the operation name and the topic name: **"operationName - record.topic()"**.
CONSUMER_PREFIXED_OPERATION_NAME_TOPIC(String prefix), PRODUCER_PREFIXED_OPERATION_NAME_TOPIC(String prefix)	Returns a String concatenation of **prefix** and **"operationName - record.topic()"**.

11.3.2. Instrumenting Kafka Streams applications for tracing

This section describes how to instrument Kafka Streams API applications for distributed tracing.

Procedure

Perform the following steps for each Kafka Streams API application.

1. Add the **opentracing-kafka-streams** dependency to the pom.xml file for your Kafka Streams API application:

```
<dependency>
    <groupId>io.opentracing.contrib</groupId>
    <artifactId>opentracing-kafka-streams</artifactId>
    <version>0.1.12.redhat-00001</version>
</dependency>
```

2. Create an instance of the **TracingKafkaClientSupplier** supplier interface:

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
```

3. Provide the supplier interface to **KafkaStreams**:

```
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
supplier);
streams.start();
```

11.4. SETTING UP TRACING FOR MIRRORMAKER, KAFKA CONNECT, AND THE KAFKA BRIDGE

Distributed tracing is supported for MirrorMaker, Kafka Connect (including Kafka Connect with Source2Image support), and the AMQ Streams Kafka Bridge.

Tracing in MirrorMaker

For MirrorMaker, messages are traced from the source cluster to the target cluster; the trace data records messages entering and leaving the MirrorMaker component.

Tracing in Kafka Connect

Only messages produced and consumed by Kafka Connect itself are traced. To trace messages sent between Kafka Connect and external systems, you must configure tracing in the connectors for those systems. For more information, see Section 3.2, “Kafka Connect cluster configuration”.

Tracing in 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. In order to have end-to-end tracing, you must configure tracing in your HTTP clients.

11.4.1. Enabling tracing in MirrorMaker, Kafka Connect, and Kafka Bridge resources

Update the configuration of **KafkaMirrorMaker**, **KafkaConnect**, **KafkaConnectS2I**, and **KafkaBridge** custom resources to specify and configure a Jaeger tracer service for each resource. Updating a tracing-enabled resource in your OpenShift cluster triggers two events:

- Interceptor classes are updated in the integrated consumers and producers in MirrorMaker, Kafka Connect, or the AMQ Streams Kafka Bridge.
- For MirrorMaker and Kafka Connect, the tracing agent initializes a Jaeger tracer based on the tracing configuration defined in the resource.
- For the Kafka Bridge, a Jaeger tracer based on the tracing configuration defined in the resource is initialized by the Kafka Bridge itself.

Procedure

Perform these steps for each **KafkaMirrorMaker**, **KafkaConnect**, **KafkaConnectS2I**, and **KafkaBridge** resource.

1. In the **spec.template** property, configure the Jaeger tracer service. For example:

Jaeger tracer configuration for Kafka Connect

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  template:
    connectContainer: 1
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing: 2
    type: jaeger
  #...
```

Jaeger tracer configuration for MirrorMaker

```
apiVersion: kafka.strimzi.io/v1beta1
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
#...
```

Jaeger tracer configuration for the Kafka Bridge

```
apiVersion: kafka.strimzi.io/v1beta1
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
#...
```

1. Use the tracing environment variables as template configuration properties.
2. Set the **spec.tracing.type** property to **jaeger**.

2. Create or update the resource:

```
oc apply -f your-file
```

Additional resources

- Section 3.9.4, “Customizing containers with environment variables”
- Section 3.9.1, “Template properties”

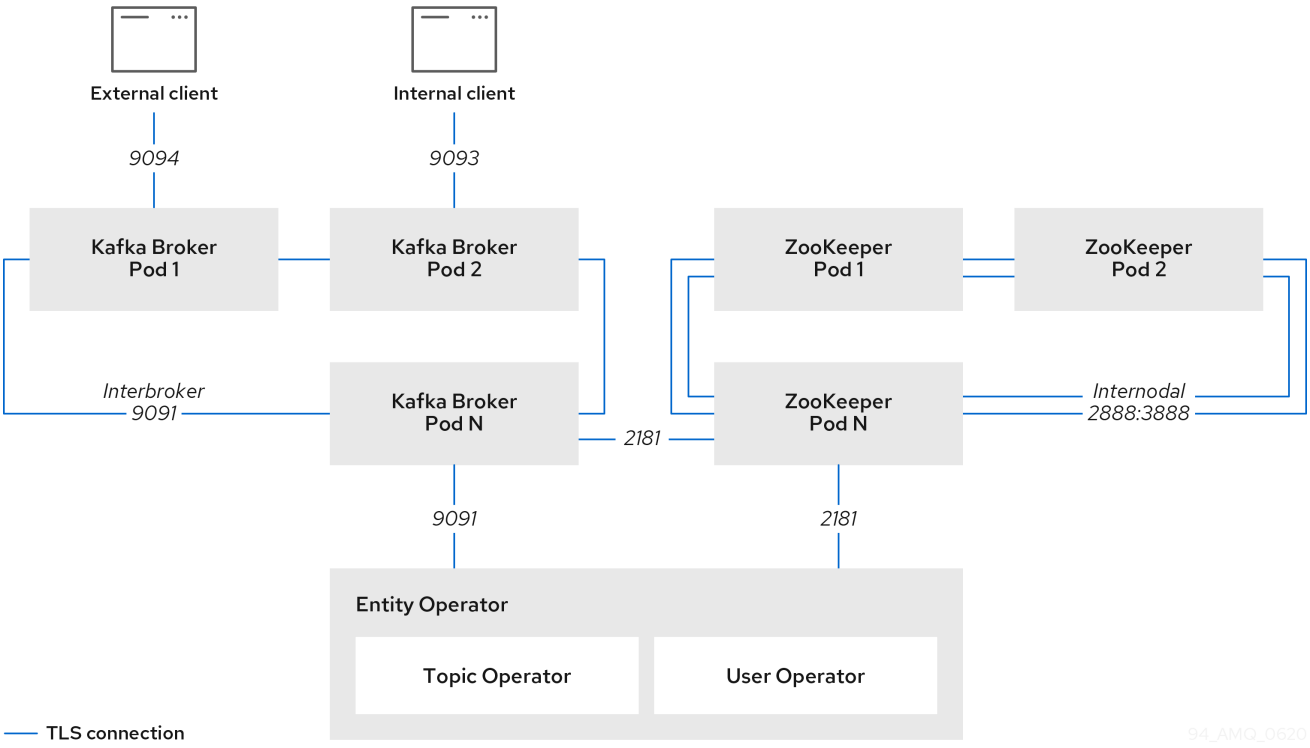
CHAPTER 12. SECURITY

AMQ Streams supports encrypted communication between the Kafka and AMQ Streams components using the TLS protocol. Communication between Kafka brokers (interbroker communication), between ZooKeeper nodes (internodal communication), and between these and the AMQ Streams operators is always encrypted. 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 TLS authentication between Kafka brokers and clients. Certificates provided by users are not renewed.

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 12.8, "Kafka listener certificates".

Figure 12.1. Example architecture diagram of the communication secured by TLS.



12.1. CERTIFICATE AUTHORITIES

To support encryption, each AMQ Streams component needs its own private keys and public key certificates. All component certificates are signed by an internal Certificate Authority (CA) called the *cluster CA*.

Similarly, each Kafka client application connecting to AMQ Streams using TLS client authentication needs to provide private keys and certificates. A second internal CA, named the *clients CA*, is used to sign certificates for the Kafka clients.

12.1.1. CA certificates

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 mutual TLS 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. Certificates provided by users are not renewed.

You can provide your own CA certificates for the cluster CA or clients CA. For more information, see Section 12.1.3, “Installing your own CA certificates” . If you provide your own certificates, you must manually renew them when needed.

12.1.2. Validity periods of CA certificates

CA certificate validity periods are expressed as a number of days after certificate generation. You can configure the validity period of:

- Cluster CA certificates in **Kafka.spec.clusterCa.validityDays**
- Client CA certificates in **Kafka.spec.clientsCa.validityDays**

12.1.3. Installing your own CA certificates

This procedure describes how to install your own CA certificates and private keys instead of using CA certificates and private keys generated by the Cluster Operator.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster is not yet deployed.
- 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 as a CA in the X509v3 Basic Constraints.

Procedure

1. Put your CA certificate in the corresponding **Secret** (***<cluster>*-cluster-ca-cert** for the cluster CA or ***<cluster>*-clients-ca-cert** for the clients CA):
   Run the following commands:

```
# Delete any existing secret (ignore "Not Exists" errors)
```

```
oc delete secret <ca-cert-secret>
# Create and label the new secret
oc create secret generic <ca-cert-secret> --from-file=ca.crt=<ca-cert-file>
```

2. Put your CA key in the corresponding **Secret** (***<cluster>*-cluster-ca** for the cluster CA or ***<cluster>*-clients-ca** for the clients CA):

```
# Delete the existing secret
oc delete secret <ca-key-secret>
# Create the new one
oc create secret generic <ca-key-secret> --from-file=ca.key=<ca-key-file>
```

3. Label both **Secrets** with the labels **strimzi.io/kind=Kafka** and **strimzi.io/cluster=*<my-cluster>***:

```
oc label secret <ca-cert-secret> strimzi.io/kind=Kafka strimzi.io/cluster=<my-cluster>
oc label secret <ca-key-secret> strimzi.io/kind=Kafka strimzi.io/cluster=<my-cluster>
```

4. 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/v1beta1
spec:
  # ...
  clusterCa:
    generateCertificateAuthority: false
```

Additional resources

- For the procedure for renewing CA certificates you have previously installed, see Section 12.3.4, “Renewing your own CA certificates”.
- Section 12.8.1, “Providing your own Kafka listener certificates” .

12.2. SECRETS

AMQ Streams uses *Secrets* to store private keys and certificates for Kafka cluster components and clients. Secrets are used for establishing TLS encrypted connections between Kafka brokers, and between brokers and clients. They are also used for mutual TLS authentication.

- A *Cluster Secret* contains a cluster CA certificate to sign Kafka broker certificates, and is used by a connecting client to establish a TLS encrypted connection with the Kafka cluster to validate broker identity.
- A *Client Secret* contains a client CA certificate for a user to sign its own client certificate to allow mutual authentication against the Kafka cluster. The broker validates the client identity through the client CA certificate itself.

- A *User Secret* contains a private key and certificate, which are generated and signed by the client CA certificate when a new user is created. The key and certificate are used for authentication and authorization when accessing the cluster.

Secrets provide private keys and certificates in PEM and PKCS #12 formats. Using private keys and certificates in PEM format means that users have to get them from the Secrets, and generate a corresponding truststore (or keystore) to use in their Java applications. PKCS #12 storage provides a truststore (or keystore) that can be used directly.

All keys are 2048 bits in size.

12.2.1. PKCS #12 storage

PKCS #12 defines an archive file format (**.p12**) for storing cryptography objects into a single file with password protection. You can use PKCS #12 to manage certificates and keys in one place.

Each Secret contains fields specific to PKCS #12.

- The **.p12** field contains the certificates and keys.
- The **.password** field is the password that protects the archive.

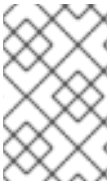
12.2.2. Cluster CA Secrets

Table 12.1. Cluster CA Secrets managed by the Cluster Operator in *<cluster>*

Secret name	Field within Secret	Description
***<cluster>*-cluster-ca**	**ca.key**	The current private key for the cluster CA.
***<cluster>*-cluster-ca-cert**	**ca.p12**	PKCS #12 archive file for storing certificates and keys.
	ca.password	Password for protecting the PKCS #12 archive file.
	ca.crt	The current certificate for the cluster CA.
***<cluster>*-kafka-brokers**	**<cluster>-kafka-<num>.p12**	PKCS #12 archive file for storing certificates and keys.
	<cluster>-kafka-<num>.password	Password for protecting the PKCS #12 archive file.
	***<cluster>*-kafka-*<num>*.crt**	Certificate for Kafka broker pod *<num>*. Signed by a current or former cluster CA private key in ***<cluster>*-cluster-ca**.
	***<cluster>*-kafka-*<num>*.key**	Private key for Kafka broker pod *<num>*.

Secret name	Field within Secret	Description
***<cluster>*-zookeeper-nodes**	**<cluster>-zookeeper-<num>.p12**	PKCS #12 archive file for storing certificates and keys.
	<cluster>-zookeeper-<num>.password	Password for protecting the PKCS #12 archive file.
	***<cluster>*-zookeeper-*<num>*.crt**	Certificate for ZooKeeper node *<num>*. Signed by a current or former cluster CA private key in ***<cluster>*-cluster-ca**.
	***<cluster>*-zookeeper-*<num>*.key**	Private key for ZooKeeper pod *<num>*.
***<cluster>*-entity-operator-certs**	**entity-operator_.p12**	PKCS #12 archive file for storing certificates and keys.
	entity-operator_.password	Password for protecting the PKCS #12 archive file.
	entity-operator_.crt	Certificate for TLS communication between the Entity Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster>*-cluster-ca**.
	entity-operator.key	Private key for TLS communication between the Entity Operator and Kafka or ZooKeeper

The CA certificates in ***<cluster>*-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.



NOTE

Only ***<cluster>*-cluster-ca-cert** needs to be used by clients. All other **Secrets** in the table above only need to be accessed by the AMQ Streams components. You can enforce this using OpenShift role-based access controls if necessary.

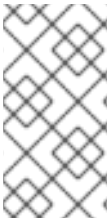
12.2.3. Client CA Secrets

Table 12.2. Clients CA Secrets managed by the Cluster Operator in*<cluster>*

Secret name	Field within Secret	Description
***<cluster>*-clients-ca**	**ca.key**	The current private key for the clients CA.

Secret name	Field within Secret	Description
***<cluster>*-clients-ca-cert**	**ca.p12**	PKCS #12 archive file for storing certificates and keys.
	ca.password	Password for protecting the PKCS #12 archive file.
	ca.crt	The current certificate for the clients CA.

The certificates in ***<cluster>*-clients-ca-cert** are those which the Kafka brokers trust.



NOTE

***<cluster>*-clients-ca** is used to sign certificates of client applications. It needs to 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.

12.2.4. User Secrets

Table 12.3. Secrets managed by the User Operator

Secret name	Field within Secret	Description
<user>	**user.p12**	PKCS #12 archive file for storing certificates and keys.
	user.password	Password for protecting the PKCS #12 archive file.
	user.crt	Certificate for the user, signed by the clients CA
	user.key	Private key for the user

12.3. CERTIFICATE RENEWAL

The 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 auto-generated CA certificates, you can configure the validity period in **Kafka.spec.clusterCa.validityDays** and **Kafka.spec.clientsCa.validityDays**. The default validity period for both certificates is 365 days. Manually-installed CA certificates should have their own validity period defined.

When a CA certificate expires, components and clients which still trust that certificate will not accept TLS connections from peers whose certificate 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 will initiate certificate renewal before the old CA certificates expire. You can configure the renewal period in

Kafka.spec.clusterCa.renewalDays and **Kafka.spec.clientsCa.renewalDays** (both default to 30 days). The renewal period is measured backwards, from the expiry date of the current certificate.

```
Not Before                    Not After
   |                             |
   |<--------------- validityDays --------------->|
                         <--- renewalDays --->|
```

The behavior of the Cluster Operator during the renewal period depends on whether the relevant setting is enabled, in either **Kafka.spec.clusterCa.generateCertificateAuthority** or **Kafka.spec.clientsCa.generateCertificateAuthority**.

12.3.1. Renewal process with generated CAs

The Cluster Operator performs the following process to renew CA certificates:

1. Generate a new CA certificate, but retain the existing key. The new certificate replaces the old one with the name **ca.crt** within the corresponding **Secret**.
2. Generate 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. Restart ZooKeeper nodes so that they will trust the new CA certificate and use the new client certificates.
4. Restart Kafka brokers so that they will trust the new CA certificate and use the new client certificates.
5. Restart the Topic and User Operators so that they will trust the new CA certificate and use the new client certificates.

12.3.2. Client applications

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.

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 .

12.3.2.1. Client certificate renewal

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.

12.3.3. Renewing CA certificates manually

Unless the **Kafka.spec.clusterCa.generateCertificateAuthority** and **Kafka.spec.clientsCa.generateCertificateAuthority** objects are set to **false**, the cluster and clients CA certificates will auto-renew at the start of their respective certificate renewal periods. You can manually renew one or both of these certificates before the certificate renewal period starts, if required for security reasons. A renewed certificate uses the same private key as the old certificate.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which CA certificates and private keys are installed.

Procedure

- Apply the **strimzi.io/force-renew** annotation to the **Secret** that contains the CA certificate that you want to renew.

Certificate	Secret	Annotate command
Cluster CA	*<cluster-name>*-cluster-ca-cert	**oc annotate secret *<cluster-name>*-cluster-ca-cert strimzi.io/force-renew=true**
Clients CA	*<cluster-name>*-clients-ca-cert	**oc annotate secret *<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.

Additional resources

- Section 12.2, "Secrets"
- Section 3.1.28, "Maintenance time windows for rolling updates"
- Section B.64, "**CertificateAuthority** schema reference"

12.3.4. Renewing your own CA certificates

This procedure describes how to renew CA certificates and private keys that you previously installed. You will need to follow this procedure during the renewal period in order to replace CA certificates which will soon expire.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which you previously installed your own CA certificates and private keys.
- New cluster and clients X.509 certificates and keys in PEM format. These could be generated using **openssl** using a command such as:

```
openssl req -x509 -new -days <validity> --nodes -out ca.crt -keyout ca.key
```

Procedure

1. Establish what CA certificates already exist in the **Secret**:
   Use the following commands:

   ```
   oc describe secret <ca-cert-secret>
   ```

2. Prepare a directory containing the existing CA certificates in the secret.

   ```
   mkdir new-ca-cert-secret
   cd new-ca-cert-secret
   ```

   For each certificate *<ca-certificate>* from the previous step, run:

   ```
   # Fetch the existing secret
   oc get secret <ca-cert-secret> -o 'jsonpath={.data.<ca-certificate>}' | base64 -d > <ca-certificate>
   ```

3. Rename the old **ca.crt** file to **ca_DATE.crt**, where *DATE* is the certificate expiry date in the format *<year>-<month>-<day>_T*<hour>_-<*minute*>-_<second>_Z, for example **ca-2018-09-27T17-32-00Z.crt**.

   ```
   mv ca.crt ca-$(date -u -d$(openssl x509 -enddate -noout -in ca.crt | sed 's/.*=//') +'%Y-%m-%dT%H-%M-%SZ').crt
   ```

4. Copy the new CA certificate into the directory, naming it **ca.crt**

   ```
   cp <path-to-new-cert> ca.crt
   ```

5. Replace the CA certificate **Secret** (***<cluster>*-cluster-ca** or ***<cluster>*-clients-ca**). This can be done using the following commands:

   ```
   # Delete the existing secret
   oc delete secret <ca-cert-secret>
   # Re-create the secret with the new private key
   oc create secret generic <ca-cert-secret> --from-file=.
   ```

   You can now delete the directory you created:

```
cd ..
rm -r new-ca-cert-secret
```

6. Replace the CA key **Secret** (***<cluster>*-cluster-ca** or ***<cluster>*-clients-ca**). This can be done using the following commands:

```
# Delete the existing secret
oc delete secret <ca-key-secret>
# Re-create the secret with the new private key
oc create secret generic <ca-key-secret> --from-file=ca.key=<ca-key-file>
```

12.4. REPLACING PRIVATE KEYS

You can replace the private keys used by the cluster CA and clients CA certificates. When a private key is replaced, the Cluster Operator generates a new CA certificate for the new private key.

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.

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 12.2, “Secrets”
- Section 3.1.28, “Maintenance time windows for rolling updates”

12.5. TLS CONNECTIONS

12.5.1. ZooKeeper communication

ZooKeeper does not support TLS itself. By deploying a TLS sidecar within every ZooKeeper pod, the Cluster Operator is able to provide data encryption and authentication between ZooKeeper nodes in a cluster. ZooKeeper only communicates with the TLS sidecar over the loopback interface. The TLS sidecar then proxies all ZooKeeper traffic, TLS decrypting data upon entry into a ZooKeeper pod, and TLS encrypting data upon departure from a ZooKeeper pod.

This TLS encrypting **stunnel** proxy is instantiated from the **spec.zookeeper.stunnelImage** specified in the Kafka resource.

12.5.2. Kafka interbroker communication

Communication between Kafka brokers is done through an internal listener on port 9091, which is encrypted by default and not accessible to Kafka clients.

Communication between Kafka brokers and ZooKeeper nodes uses a TLS sidecar, as described above.

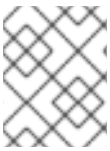
12.5.3. Topic and User Operators

Like the Cluster Operator, the Topic and User Operators each use a TLS sidecar when communicating with ZooKeeper. The Topic Operator connects to Kafka brokers on port 9091.

12.5.4. Kafka Client connections

Encrypted communication between Kafka brokers and clients running within the same OpenShift cluster can be provided by configuring the **spec.kafka.listeners.tls** listener, which listens on port 9093.

Encrypted communication between Kafka brokers and clients running outside the same OpenShift cluster can be provided by configuring the **spec.kafka.listeners.external** listener (the port of the **external** listener depends on its type).



NOTE

Unencrypted client communication with brokers can be configured by **spec.kafka.listeners.plain**, which listens on port 9092.

12.6. 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 the **tls** listener on port 9093 — 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 outside 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-cert
```

   Here we're mounting:

   - 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 TLS 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-cert
```

2. Use the certificate with clients that use certificates in X.509 format.

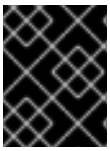
12.7. CONFIGURING EXTERNAL CLIENTS TO TRUST THE CLUSTER CA

This procedure describes how to configure a Kafka client that resides outside the OpenShift cluster – connecting to the **external** listener on port 9094 – 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** will contain 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 generated ***<cluster-name>*-cluster-ca-cert** Secret.

```
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
```

2. Configure the Kafka client with the following properties:
   - A security protocol option:
     - **security.protocol: SSL** when using TLS for encryption (with or without TLS 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. 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 generated ***<cluster-name>*-cluster-ca-cert** Secret.

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

2. Use the certificate with clients that use certificates in X.509 format.

12.8. KAFKA LISTENER CERTIFICATES

You can provide your own server certificates and private keys for the following types of listeners:

- TLS listeners for inter-cluster communication
- External listeners (**route**, **loadbalancer**, **ingress**, and **nodeport** types) which have TLS encryption enabled, for communication between Kafka clients and Kafka brokers

These user-provided certificates are called *Kafka listener certificates*.

Providing Kafka listener certificates for external listeners allows you to leverage existing security infrastructure, such as your organization's private CA or a public CA. Kafka clients will connect to Kafka brokers using Kafka listener certificates rather than certificates signed by the cluster CA or clients CA.

You must manually renew Kafka listener certificates when needed.

12.8.1. Providing your own Kafka listener certificates

This procedure shows how to configure a listener to use your own private key and server certificate, called a Kafka listener certificate.

Your client applications should use the CA public key as a trusted certificate in order to verify the identity of the Kafka broker.

Prerequisites

- An OpenShift cluster.
- The Cluster Operator is running.
- For each listener, a compatible server certificate signed by an external CA.
  - Provide an X.509 certificate in PEM format.
  - Specify the correct Subject Alternative Names (SANs) for each listener. For more information, see Section 12.8.2, "Alternative subjects in server certificates for Kafka listeners".
  - You can provide a certificate that includes the whole CA chain in the certificate file.

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:
  plain: {}
  external:
    type: loadbalancer
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
    tls: true
    authentication:
      type: tls
# ...
```

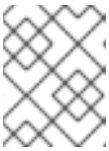
Example configuration for a TLS listener

```
# ...
listeners:
  plain: {}
  tls:
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.pem
        key: my-listener-key.key
    authentication:
      type: tls
# ...
```

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 TLS or external listener.

Additional resources

- Section 12.8.2, "Alternative subjects in server certificates for Kafka listeners"
- Section B.8, "**KafkaListeners** schema reference"
- Section 12.8, "Kafka listener certificates"

12.8.2. 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:

- 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.

12.8.2.1. TLS listener SAN examples

Use the following examples to help you specify hostnames of the SANs in your certificates for TLS listeners.

Wildcards example

```
//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

```
// 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
```

12.8.2.2. External listener SAN examples

For external listeners which have TLS encryption enabled, the hostnames you need to specify in certificates depends on the external listener **type**.

External listener type	In the SANs, specify...
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 12.8.1, “Providing your own Kafka listener certificates”

CHAPTER 13. MANAGING AMQ STREAMS

This chapter covers tasks to maintain a deployment of AMQ Streams.

13.1. 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
```

13.1.1. 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.

13.2. CHECKING THE STATUS OF A CUSTOM RESOURCE

The **status** property of a AMQ Streams custom resource publishes information about the resource to users and tools that need it.

13.2.1. AMQ Streams custom resource status information

Several resources have a **status** property, as described in the following table.

AMQ Streams resource	Schema reference	Publishes status information on...
Kafka	Section B.70, "**KafkaStatus** schema reference"	The Kafka cluster.
KafkaConnect	Section B.88, "**KafkaConnectStatus** schema reference"	The Kafka Connect cluster, if deployed.
KafkaConnectS2I	Section B.92, "**KafkaConnectS2IStatus** schema reference"	The Kafka Connect cluster with Source-to-Image support, if deployed.
KafkaConnector	Section B.126, "**KafkaConnectorStatus** schema reference"	**KafkaConnector** resources, if deployed.
KafkaMirrorMaker	Section B.114, "**KafkaMirrorMakerStatus** schema reference"	The Kafka MirrorMaker tool, if deployed.

AMQ Streams resource	Schema reference	Publishes status information on...
KafkaTopic	Section B.95, "**KafkaTopicStatus** schema reference"	Kafka topics in your Kafka cluster.
KafkaUser	Section B.107, "**KafkaUserStatus** schema reference"	Kafka users in your Kafka cluster.
KafkaBridge	Section B.123, "**KafkaBridgeStatus** schema reference"	The AMQ Streams Kafka Bridge, if deployed.

The **status** property of a resource provides information on the resource's:

- *Current state*, in the **status.conditions** property
- *Last observed generation*, in the **status.observedGeneration** property

The **status** property also provides resource-specific information. For example:

- **KafkaConnectStatus** provides the REST API endpoint for Kafka Connect connectors.
- **KafkaUserStatus** provides the user name of the Kafka user and the **Secret** in which their credentials are stored.
- **KafkaBridgeStatus** provides the HTTP address at which external client applications can access the Bridge service.

A resource's *current state* is useful for tracking progress related to the resource achieving its *desired state*, as defined by the **spec** property. The status conditions provide the time and reason the state of the resource changed and details of events preventing or delaying the operator from realizing the resource's desired state.

The *last observed generation* is the generation of the resource that was last reconciled by the Cluster Operator. If the value of **observedGeneration** is different from the value of **metadata.generation**, 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.

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** property specified for a Kafka custom resource.

Kafka custom resource with status

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
```

```
spec:
  # ...
status:
  conditions: 1
  - lastTransitionTime: 2019-07-23T23:46:57+0000
    status: "True"
    type: Ready 2
  observedGeneration: 4 3
  listeners: 4
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9092
    type: plain
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9093
    certificates:
    - |
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    type: tls
  - addresses:
    - host: 172.29.49.180
      port: 9094
    certificates:
    - |
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    type: external
    # ...
```

1. Status **conditions** describe criteria related to the status that cannot be deduced from the existing resource information, or are specific to the instance of a resource.

2. The **Ready** condition indicates whether the Cluster Operator currently considers the Kafka cluster able to handle traffic.

3. The **observedGeneration** indicates the generation of the **Kafka** custom resource that was last reconciled by the Cluster Operator.

4. The **listeners** describe the current Kafka bootstrap addresses by type.

IMPORTANT

The address in the custom resource status for external listeners with type **nodeport** is currently not supported.

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 13.2.2, "Finding the status of a custom resource".

13.2.2. 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 13.2.1, "AMQ Streams custom resource status information"
- For more information about using JSONPath, see JSONPath support.

13.3. 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

13.3.1. 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:

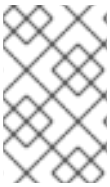
```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
  # ...
  persistentVolumeReclaimPolicy: Retain
```

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.

13.3.2. 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.

13.3.3. 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.
   This time you deploy your cluster without the Topic Operator, delete the Topic Operator data in ZooKeeper, and then redeploy it so that the Topic Operator can recreate the **KafkaTopic** resources from the corresponding topics.

NOTE

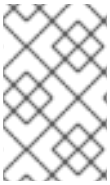
If the Topic Operator is not deployed, you only need to recover the **PersistentVolumeClaim** (PVC) resources.

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 Volume Claim naming
- JBOD and Persistent Volume Claims



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 apply -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. Run an **exec** command to one of the Kafka broker pods to open the ZooKeeper shell script.
   For example, where *my-cluster-kafka-0* is the name of the broker pod:

   ```
   oc exec my-cluster-kafka-0 bin/zookeeper-shell.sh localhost:2181
   ```

3. Delete the whole **/strimzi** path to remove the Topic Operator storage:

   ```
   deleteall /strimzi
   ```

4. Enable the Topic Operator by redeploying the Kafka cluster with the **topicOperator** property to recreate the **KafkaTopic** resources.
   For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   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 Section B.61, "**EntityTopicOperatorSpec** schema reference".

7. Verify the recovery by listing the **KafkaTopic** resources:

   ```
   oc get KafkaTopic
   ```

13.4. UNINSTALLING AMQ STREAMS

This procedure describes how to uninstall AMQ Streams and remove resources related to the deployment.

Prerequisites

In order to perform this procedure, 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**, **KafkaConnectS2I**, **KafkaMirrorMaker**, or **KafkaBridge** configuration.

Procedure

1. Delete the Cluster Operator **Deployment**, related **CustomResourceDefinitions**, and **RBAC** resources:

```
oc delete -f install/cluster-operator
```

WARNING

Deleting **CustomResourceDefinitions** results in the garbage collection of the corresponding custom resources (**Kafka**, **KafkaConnect**, **KafkaConnectS2I**, **KafkaMirrorMaker**, or **KafkaBridge**) and the resources dependent on them (Deployments, StatefulSets, and other dependent resources).

2. Delete the resources you identified in the prerequisites.

APPENDIX A. FREQUENTLY ASKED QUESTIONS

A.1. QUESTIONS RELATED TO THE CLUSTER OPERATOR

A.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?

A.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**.

A.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 AMQ Streams Administrators.

A.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.

A.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", "");
```

APPENDIX B. CUSTOM RESOURCE API REFERENCE

B.1. KAFKA SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka and ZooKeeper clusters, and Topic Operator.
KafkaSpec	
status	The status of the Kafka and ZooKeeper clusters, and Topic Operator.
KafkaStatus	

B.2. KAFKASPEC SCHEMA REFERENCE

Used in: **Kafka**

Property	Description
kafka	Configuration of the Kafka cluster.
KafkaClusterSpec	
zookeeper	Configuration of the ZooKeeper cluster.
ZookeeperClusterSpec	
topicOperator	**The property topicOperator has been deprecated. This feature should now be configured at path spec.entityOperator.topicOperator.** Configuration of the Topic Operator.
TopicOperatorSpec	
entityOperator	Configuration of the Entity Operator.
EntityOperatorSpec	
clusterCa	Configuration of the cluster certificate authority.
CertificateAuthority	
clientsCa	Configuration of the clients certificate authority.
CertificateAuthority	

Property	Description
cruiseControl **CruiseControlSpec**	Configuration for Cruise Control deployment. Deploys a Cruise Control instance when specified.
kafkaExporter **KafkaExporterSpec**	Configuration of the Kafka Exporter. Kafka Exporter can provide additional metrics, for example lag of consumer group at topic/partition.
maintenanceTimeWindows string array	A list of time windows for maintenance tasks (that is, certificates renewal). Each time window is defined by a cron expression.

B.3. KAFKACLUSTERSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
replicas integer	The number of pods in the cluster.
image string	The docker image for the pods. The default value depends on the configured **Kafka.spec.kafka.version**.
storage **EphemeralStorage**, **PersistentClaimStorage**, **JbodStorage**	Storage configuration (disk). Cannot be updated. The type depends on the value of the **storage.type** property within the given object, which must be one of [ephemeral, persistent-claim, jbod].
listeners **KafkaListeners**	Configures listeners of Kafka brokers.
authorization **KafkaAuthorizationSimple**, **KafkaAuthorizationOpa**, **KafkaAuthorizationKeycloak**	Authorization configuration for Kafka brokers. The type depends on the value of the **authorization.type** property within the given object, which must be one of [simple, opa, keycloak].

Property	Description
config map	Kafka broker config properties with the following prefixes cannot be set: listeners, advertised., broker., listener., host.name, port, inter.broker.listener.name, sasl., ssl., security., password., principal.builder.class, log.dir, zookeeper.connect, zookeeper.set.acl, authorizer., super.user, cruise.control.metrics.topic, cruise.control.metrics.reporter.bootstrap.servers (with the exception of: zookeeper.connection.timeout.ms, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols,cruise.control.metrics.topic.num.partitions, cruise.control.metrics.topic.replication.factor, cruise.control.metrics.topic.retention.ms).
rack **Rack**	Configuration of the **broker.rack** broker config.
brokerRackInitImage string	The image of the init container used for initializing the **broker.rack**.
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.kafka.template.pod.affinity.** The pod's affinity rules. See external documentation of core/v1 affinity.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.kafka.template.pod.tolerations.** The pod's tolerations. See external documentation of core/v1 toleration.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
jvmOptions **JvmOptions**	JVM Options for pods.
jmxOptions	JMX Options for Kafka brokers.

Property	Description
KafkaJmxOptions	
resources	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	
metrics	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
map	
logging	Logging configuration for Kafka. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
InlineLogging, **ExternalLogging**	
tlsSidecar	TLS sidecar configuration.
TlsSidecar	
template	Template for Kafka cluster resources. The template allows users to specify how are the **StatefulSet**, **Pods** and **Services** generated.
KafkaClusterTemplate	
version	The kafka broker version. Defaults to 2.5.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.
string	

B.4. EPHEMERALSTORAGE SCHEMA REFERENCE

Used in: **JbodStorage**, **KafkaClusterSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **EphemeralStorage** from **PersistentClaimStorage**. It must have the value **ephemeral** for the type **EphemeralStorage**.

Property	Description
id	Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'.
integer	
sizeLimit	When type=ephemeral, defines the total amount of local storage required for this EmptyDir volume (for example 1Gi).
string	
type	Must be **ephemeral**.

Property	Description
string	

B.5. PERSISTENTCLAIMSTORAGE SCHEMA REFERENCE

Used in: **JbodStorage**, **KafkaClusterSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **PersistentClaimStorage** from **EphemeralStorage**. It must have the value **persistent-claim** for the type **PersistentClaimStorage**.

Property	Description
type	Must be **persistent-claim**.
string	
size	When type=persistent-claim, defines the size of the persistent volume claim (i.e 1Gi). Mandatory when type=persistent-claim.
string	
selector	Specifies a specific persistent volume to use. It contains key:value pairs representing labels for selecting such a volume.
map	
deleteClaim	Specifies if the persistent volume claim has to be deleted when the cluster is un-deployed.
boolean	
class	The storage class to use for dynamic volume allocation.
string	
id	Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'.
integer	
overrides	Overrides for individual brokers. The **overrides** field allows to specify a different configuration for different brokers.
PersistentClaimStorageOverride array	

B.6. PERSISTENTCLAIMSTORAGEOVERRIDE SCHEMA REFERENCE

Used in: **PersistentClaimStorage**

Property	Description
class string	The storage class to use for dynamic volume allocation for this broker.
broker integer	Id of the kafka broker (broker identifier).

B.7. JBODSTORAGE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **JbodStorage** from **EphemeralStorage**, **PersistentClaimStorage**. It must have the value **jbod** for the type **JbodStorage**.

Property	Description
type string	Must be **jbod**.
volumes **EphemeralStorage**, **PersistentClaimStorage** array	List of volumes as Storage objects representing the JBOD disks array.

B.8. KAFKALISTENERS SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
plain **KafkaListenerPlain**	Configures plain listener on port 9092.
tls **KafkaListenerTls**	Configures TLS listener on port 9093.
external	Configures external listener on port 9094. The type depends on the value of the **external.type** property within the given object, which must be one of [route, loadbalancer, nodeport, ingress].

Property	Description
KafkaListenerExternalRoute, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalIngress**	

B.9. KAFKALISTENERPLAIN SCHEMA REFERENCE

Used in: **KafkaListeners**

Property	Description
authentication	Authentication configuration for this listener. Since this listener does not use TLS transport you cannot configure an authentication with **type: tls**. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, oauth].
KafkaListenerAuthenticationTls, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**	
networkPolicyPeers	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list. See external documentation of networking.k8s.io/v1 networkpolicypeer.
NetworkPolicyPeer array	

B.10. KAFKALISTENERAUTHENTICATIONTLS SCHEMA REFERENCE

Used in: **KafkaListenerExternalIngress**, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalRoute**, **KafkaListenerPlain**, **KafkaListenerTls**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerAuthenticationTls** from **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**. It must have the value **tls** for the type **KafkaListenerAuthenticationTls**.

Property	Description
type	Must be **tls**.
string	

B.11. KAFKALISTENERAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE

Used in: **KafkaListenerExternalIngress**, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalRoute**, **KafkaListenerPlain**, **KafkaListenerTls**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerAuthenticationScramSha512** from **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationOAuth**. It must have the value **scram-sha-512** for the type **KafkaListenerAuthenticationScramSha512**.

Property	Description
type	Must be **scram-sha-512**.
string	

B.12. KAFKALISTENERAUTHENTICATIONOAUTH SCHEMA REFERENCE

Used in: **KafkaListenerExternalIngress**, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalRoute**, **KafkaListenerPlain**, **KafkaListenerTls**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerAuthenticationOAuth** from **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**. It must have the value **oauth** for the type **KafkaListenerAuthenticationOAuth**.

Property	Description
accessTokenIsJwt	Configure whether the access token is treated as JWT. This must be set to **false** if the authorization server returns opaque tokens. Defaults to **true**.
boolean	
checkAccessTokenType	Configure whether the access token type check is performed or not. This should be set to **false** if the authorization server does not include 'typ' claim in JWT token. Defaults to **true**.
boolean	
checkIssuer	Enable or disable issuer checking. By default issuer is checked using the value configured by **validIssuerUri**. Default value is **true**.
boolean	
clientId	OAuth Client ID which the Kafka broker can use to authenticate against the authorization server and use the introspect endpoint URI.
string	
clientSecret	Link to OpenShift Secret containing the OAuth client secret which the Kafka broker can use to authenticate against the authorization server and use the introspect endpoint URI.
GenericSecretSource	

Property	Description
disableTlsHostnameVerification	Enable or disable TLS hostname verification. Default value is **false**.
boolean	
enableECDSA	Enable or disable ECDSA support by installing BouncyCastle crypto provider. Default value is **false**.
boolean	
fallbackUserNameClaim	The fallback username claim to be used for the user id if the claim specified by **userNameClaim** is not present. This is useful when **client_credentials** authentication only results in the client id being provided in another claim. It only takes effect if **userNameClaim** is set.
string	
fallbackUserNamePrefix	The prefix to use with the value of **fallbackUserNameClaim** to construct the user id. This only takes effect if **fallbackUserNameClaim** is true, and the value is present for the claim. Mapping usernames and client ids into the same user id space is useful in preventing name collisions.
string	
introspectionEndpointUri	URI of the token introspection endpoint which can be used to validate opaque non-JWT tokens.
string	
jwksEndpointUri	URI of the JWKS certificate endpoint, which can be used for local JWT validation.
string	
jwksExpirySeconds	Configures how often are the JWKS certificates considered valid. The expiry interval has to be at least 60 seconds longer then the refresh interval specified in **jwksRefreshSeconds**. Defaults to 360 seconds.
integer	
jwksRefreshSeconds	Configures how often are the JWKS certificates refreshed. The refresh interval has to be at least 60 seconds shorter then the expiry interval specified in **jwksExpirySeconds**. Defaults to 300 seconds.
integer	
tlsTrustedCertificates	Trusted certificates for TLS connection to the OAuth server.
CertSecretSource array	
type	Must be **oauth**.
string	

Property	Description
userInfoEndpointUri string	URI of the User Info Endpoint to use as a fallback to obtaining the user id when the Introspection Endpoint does not return information that can be used for the user id.
userNameClaim string	Name of the claim from the JWT authentication token, Introspection Endpoint response or User Info Endpoint response which will be used to extract the user id. Defaults to **sub**.
validIssuerUri string	URI of the token issuer used for authentication.
validTokenType string	Valid value for the **token_type** attribute returned by the Introspection Endpoint. No default value, and not checked by default.

B.13. GENERICSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaClientAuthenticationOAuth**, **KafkaListenerAuthenticationOAuth**

Property	Description
key string	The key under which the secret value is stored in the OpenShift Secret.
secretName string	The name of the OpenShift Secret containing the secret value.

B.14. CERTSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaAuthorizationKeycloak**, **KafkaBridgeTls**, **KafkaClientAuthenticationOAuth**, **KafkaConnectTls**, **KafkaListenerAuthenticationOAuth**, **KafkaMirrorMaker2Tls**, **KafkaMirrorMakerTls**

Property	Description
certificate string	The name of the file certificate in the Secret.

Property	Description
secretName	The name of the Secret containing the certificate.
string	

B.15. KAFKALISTENERTLS SCHEMA REFERENCE

Used in: **KafkaListeners**

Property	Description
authentication	Authentication configuration for this listener. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, oauth].
KafkaListenerAuthenticationTls, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**	
configuration	Configuration of TLS listener.
TlsListenerConfiguration	
networkPolicyPeers	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list. See external documentation of networking.k8s.io/v1 networkpolicypeer.
NetworkPolicyPeer array	

B.16. TLSLISTENERCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaListenerTls**

Property	Description
brokerCertChainAndKey	Reference to the **Secret** which holds the certificate and private key pair. The certificate can optionally contain the whole chain.
CertAndKeySecretSource	

B.17. CERTANDKEYSECRETSOURCE SCHEMA REFERENCE

Used in: **IngressListenerConfiguration**, **KafkaClientAuthenticationTls**, **KafkaListenerExternalConfiguration**, **NodePortListenerConfiguration**, **TlsListenerConfiguration**

Property	Description
certificate	The name of the file certificate in the Secret.
string	
key	The name of the private key in the Secret.
string	
secretName	The name of the Secret containing the certificate.
string	

B.18. KAFKALISTENEREXTERNALROUTE SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalRoute** from **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalIngress**. It must have the value **route** for the type **KafkaListenerExternalRoute**.

Property	Description
type	Must be **route**.
string	
authentication	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, oauth].
KafkaListenerAuthenticationTls, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**	
overrides	Overrides for external bootstrap and broker services and externally advertised addresses.
RouteListenerOverride	
configuration	External listener configuration.
KafkaListenerExternalConfiguration	

Property	Description
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list. See external documentation of networking.k8s.io/v1 networkpolicypeer.

B.19. ROUTELISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalRoute**

Property	Description
bootstrap **RouteListenerBootstrapOverride**	External bootstrap service configuration.
brokers **RouteListenerBrokerOverride** array	External broker services configuration.

B.20. ROUTELISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **RouteListenerOverride**

Property	Description
address string	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
host string	Host for the bootstrap route. This field will be used in the **spec.host** field of the OpenShift Route.

B.21. ROUTELISTENERBROKEROVERRIDE SCHEMA REFERENCE

Used in: **RouteListenerOverride**

Property	Description
broker integer	Id of the kafka broker (broker identifier).
advertisedHost string	The host name which will be used in the brokers' **advertised.brokers**.
advertisedPort integer	The port number which will be used in the brokers' **advertised.brokers**.
host string	Host for the broker route. This field will be used in the **spec.host** field of the OpenShift Route.

B.22. KAFKALISTENEREXTERNALCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalRoute**

Property	Description
brokerCertChainAndKey **CertAndKeySecretSource**	Reference to the **Secret** which holds the certificate and private key pair. The certificate can optionally contain the whole chain.

B.23. KAFKALISTENEREXTERNALLOADBALANCER SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalLoadBalancer** from **KafkaListenerExternalRoute**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalIngress**. It must have the value **loadbalancer** for the type **KafkaListenerExternalLoadBalancer**.

Property	Description
type string	Must be **loadbalancer**.

Property	Description
authentication **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, oauth].
overrides **LoadBalancerListenerOverride**	Overrides for external bootstrap and broker services and externally advertised addresses.
configuration **KafkaListenerExternalConfiguration**	External listener configuration.
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list. See external documentation of networking.k8s.io/v1 networkpolicypeer.
tls boolean	Enables TLS encryption on the listener. By default set to **true** for enabled TLS encryption.

B.24. LOADBALANCERLISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalLoadBalancer**

Property	Description
bootstrap **LoadBalancerListenerBootstrapOverride**	External bootstrap service configuration.
brokers **LoadBalancerListenerBrokerOverride** array	External broker services configuration.

B.25. LOADBALANCERLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **LoadBalancerListenerOverride**

Property	Description
address string	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
dnsAnnotations map	Annotations that will be added to the **Service** resource. You can use this field to configure DNS providers such as External DNS.
loadBalancerIP string	The loadbalancer is requested with the IP address specified in this field. This feature depends on whether the underlying cloud provider supports specifying the **loadBalancerIP** when a load balancer is created. This field is ignored if the cloud provider does not support the feature.

B.26. LOADBALANCERLISTENERBROKEROVERRIDE SCHEMA REFERENCE

Used in: **LoadBalancerListenerOverride**

Property	Description
broker integer	Id of the kafka broker (broker identifier).
advertisedHost string	The host name which will be used in the brokers' **advertised.brokers**.
advertisedPort integer	The port number which will be used in the brokers' **advertised.brokers**.
dnsAnnotations map	Annotations that will be added to the **Service** resources for individual brokers. You can use this field to configure DNS providers such as External DNS.
loadBalancerIP string	The loadbalancer is requested with the IP address specified in this field. This feature depends on whether the underlying cloud provider supports specifying the **loadBalancerIP** when a load balancer is created. This field is ignored if the cloud provider does not support the feature.

B.27. KAFKALISTENEREXTERNALNODEPORT SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalNodePort** from **KafkaListenerExternalRoute**, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalIngress**. It must have the value **nodeport** for the type **KafkaListenerExternalNodePort**.

Property	Description
type string	Must be **nodeport**.
authentication **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, oauth].
overrides **NodePortListenerOverride**	Overrides for external bootstrap and broker services and externally advertised addresses.
configuration **NodePortListenerConfiguration**	External listener configuration.
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list. See external documentation of networking.k8s.io/v1 networkpolicypeer.
tls boolean	Enables TLS encryption on the listener. By default set to **true** for enabled TLS encryption.

B.28. NODEPORTLISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalNodePort**

Property	Description
bootstrap	External bootstrap service configuration.
NodePortListenerBootstrapOverride	
brokers	External broker services configuration.
NodePortListenerBrokerOverride array	

B.29. NODEPORTLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **NodePortListenerOverride**

Property	Description
address	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
string	
dnsAnnotations	Annotations that will be added to the **Service** resource. You can use this field to configure DNS providers such as External DNS.
map	
nodePort	Node port for the bootstrap service.
integer	

B.30. NODEPORTLISTENERBROKEROVERRIDE SCHEMA REFERENCE

Used in: **NodePortListenerOverride**

Property	Description
broker	Id of the kafka broker (broker identifier).
integer	
advertisedHost	The host name which will be used in the brokers' **advertised.brokers**.
string	
advertisedPort	The port number which will be used in the brokers' **advertised.brokers**.
integer	

Property	Description
nodePort	Node port for the broker service.
integer	
dnsAnnotations	Annotations that will be added to the **Service** resources for individual brokers. You can use this field to configure DNS providers such as External DNS.
map	

B.31. NODEPORTLISTENERCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaListenerExternalNodePort**

Property	Description
brokerCertChainAndKey	Reference to the **Secret** which holds the certificate and private key pair. The certificate can optionally contain the whole chain.
CertAndKeySecretSource	
preferredAddressType	Defines which address type should be used as the node address. Available types are: **ExternalDNS**, **ExternalIP**, **InternalDNS**, **InternalIP** and **Hostname**. By default, the addresses will be used in the following order (the first one found will be used): * **ExternalDNS** * **ExternalIP** * **InternalDNS** * **InternalIP** * **Hostname** This field can be used to select the address type which will be used as the preferred type and checked first. In case no address will be found for this address type, the other types will be used in the default order..
string (one of [ExternalDNS, ExternalIP, Hostname, InternalIP, InternalDNS])	

B.32. KAFKALISTENEREXTERNALINGRESS SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalIngress** from **KafkaListenerExternalRoute**, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**. It must have the value **ingress** for the type **KafkaListenerExternalIngress**.

Property	Description
type	Must be **ingress**.
string	

Property	Description
authentication **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, oauth].
class string	Configures the **Ingress** class that defines which **Ingress** controller will be used. If not set, the **Ingress** class is set to **nginx**.
configuration **IngressListenerConfiguration**	External listener configuration.
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list. See external documentation of networking.k8s.io/v1 networkpolicypeer.

B.33. INGRESSLISTENERCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaListenerExternalIngress**

Property	Description
bootstrap **IngressListenerBootstrapConfiguration**	External bootstrap ingress configuration.
brokers **IngressListenerBrokerConfiguration** array	External broker ingress configuration.
brokerCertChainAndKey **CertAndKeySecretSource**	Reference to the **Secret** which holds the certificate and private key pair. The certificate can optionally contain the whole chain.

B.34. INGRESSLISTENERBOOTSTRAPCONFIGURATION SCHEMA REFERENCE

Used in: **IngressListenerConfiguration**

Property	Description
address string	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
dnsAnnotations map	Annotations that will be added to the **Ingress** resource. You can use this field to configure DNS providers such as External DNS.
host string	Host for the bootstrap route. This field will be used in the Ingress resource.

B.35. INGRESSLISTENERBROKERCONFIGURATION SCHEMA REFERENCE

Used in: **IngressListenerConfiguration**

Property	Description
broker integer	Id of the kafka broker (broker identifier).
advertisedHost string	The host name which will be used in the brokers' **advertised.brokers**.
advertisedPort integer	The port number which will be used in the brokers' **advertised.brokers**.
host string	Host for the broker ingress. This field will be used in the Ingress resource.
dnsAnnotations map	Annotations that will be added to the **Ingress** resources for individual brokers. You can use this field to configure DNS providers such as External DNS.

B.36. KAFKAAUTHORIZATIONSIMPLE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaAuthorizationSimple** from **KafkaAuthorizationOpa**, **KafkaAuthorizationKeycloak**. It must have the value **simple** for the type **KafkaAuthorizationSimple**.

Property	Description
type	Must be **simple**.
string	
superUsers	List of super users. Should contain list of user principals which should get unlimited access rights.
string array	

B.37. KAFKAAUTHORIZATIONOPA SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

To use Open Policy Agent authorization, set the **type** property in the **authorization** section to the value **opa**. The Open Policy Agent authorizer has several configuration options:

url

The URL used to connect to the Open Policy Agent server. The URL has to include the policy which will be queried by the authorizer. **Required.**

allowOnError

Defines whether a Kafka client should be allowed or denied by default when the authorizer fails to query the Open Policy Agent, for example, when it is temporarily unavailable. Defaults to **false** - all actions will be denied.

initialCacheCapacity

Initial capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request. Defaults to **5000**.

maximumCacheSize

Maximum capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request. Defaults to **50000**.

expireAfterMs

The expiration of the records kept in the local cache to avoid querying the Open Policy Agent for every request. Defines how often the cached authorization decisions are reloaded from the Open Policy Agent server. In milliseconds. Defaults to **3600000** milliseconds (1 hour).

superUsers

A list of user principals treated as super users, so that they are always allowed without querying the open Policy Agent policy. For more information see Super users.

An example of Open Policy Agent authorizer configuration

```
authorization:
  type: opa
  url: http://opa:8181/v1/data/kafka/allow
  allowOnError: false
  initialCacheCapacity: 1000
```

```
maximumCacheSize: 10000
expireAfterMs: 60000
superUsers:
  - CN=fred
  - sam
  - CN=edward
```

The **type** property is a discriminator that distinguishes the use of the type **KafkaAuthorizationOpa** from **KafkaAuthorizationSimple**, **KafkaAuthorizationKeycloak**. It must have the value **opa** for the type **KafkaAuthorizationOpa**.

Property	Description
type string	Must be **opa**.
url string	The URL used to connect to the Open Policy Agent server. The URL has to include the policy which will be queried by the authorizer. This option is required.
allowOnError boolean	Defines whether a Kafka client should be allowed or denied by default when the authorizer fails to query the Open Policy Agent, for example, when it is temporarily unavailable). Defaults to **false** - all actions will be denied.
initialCacheCapacity integer	Initial capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request Defaults to **5000**.
maximumCacheSize integer	Maximum capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request. Defaults to **50000**.
expireAfterMs integer	The expiration of the records kept in the local cache to avoid querying the Open Policy Agent for every request. Defines how often the cached authorization decisions are reloaded from the Open Policy Agent server. In milliseconds. Defaults to **3600000**.
superUsers string array	List of super users, which is specifically a list of user principals that have unlimited access rights.

B.38. KAFKAAUTHORIZATIONKEYCLOAK SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaAuthorizationKeycloak** from **KafkaAuthorizationSimple**, **KafkaAuthorizationOpa**. It must have the value **keycloak** for the type **KafkaAuthorizationKeycloak**.

Property	Description
type	Must be **keycloak**.
string	
clientId	OAuth Client ID which the Kafka client can use to authenticate against the OAuth server and use the token endpoint URI.
string	
tokenEndpointUri	Authorization server token endpoint URI.
string	
tlsTrustedCertificates	Trusted certificates for TLS connection to the OAuth server.
CertSecretSource array	
disableTlsHostnameVerification	Enable or disable TLS hostname verification. Default value is **false**.
boolean	
delegateToKafkaAcls	Whether authorization decision should be delegated to the 'Simple' authorizer if DENIED by Red Hat Single Sign-On Authorization Services policies.Default value is **false**.
boolean	
superUsers	List of super users. Should contain list of user principals which should get unlimited access rights.
string array	

B.39. RACK SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
topologyKey	A key that matches labels assigned to the OpenShift cluster nodes. The value of the label is used to set the broker's **broker.rack** config.
string	

B.40. PROBE SCHEMA REFERENCE

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaExporterSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **TlsSidecar**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Property	Description
failureThreshold	Minimum consecutive failures for the probe to be considered failed after having succeeded. Defaults to 3. Minimum value is 1.
integer	
initialDelaySeconds	The initial delay before first the health is first checked.
integer	
periodSeconds	How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
integer	
successThreshold	Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.
integer	
timeoutSeconds	The timeout for each attempted health check.
integer	

B.41. JVMOPTIONS SCHEMA REFERENCE

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Property	Description
-XX	A map of -XX options to the JVM.
map	
-Xms	-Xms option to to the JVM.
string	
-Xmx	-Xmx option to to the JVM.
string	

Property	Description
gcLoggingEnabled boolean	Specifies whether the Garbage Collection logging is enabled. The default is false.
javaSystemProperties **SystemProperty** array	A map of additional system properties which will be passed using the **-D** option to the JVM.

B.42. SYSTEMPROPERTY SCHEMA REFERENCE

Used in: **JvmOptions**

Property	Description
name string	The system property name.
value string	The system property value.

B.43. KAFKAJMXOPTIONS SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
authentication **KafkaJmxAuthenticationPassword**	Authentication configuration for connecting to the Kafka JMX port. The type depends on the value of the **authentication.type** property within the given object, which must be one of [password].

B.44. KAFKAJMXAUTHENTICATIONPASSWORD SCHEMA REFERENCE

Used in: **KafkaJmxOptions**

The **type** property is a discriminator that distinguishes the use of the type **KafkaJmxAuthenticationPassword** from other subtypes which may be added in the future. It must have the value **password** for the type **KafkaJmxAuthenticationPassword**.

Property	Description
type	Must be **password**.
string	

B.45. INLINELOGGING SCHEMA REFERENCE

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **InlineLogging** from **ExternalLogging**. It must have the value **inline** for the type **InlineLogging**.

Property	Description
type	Must be **inline**.
string	
loggers	A Map from logger name to logger level.
map	

B.46. EXTERNALLOGGING SCHEMA REFERENCE

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **ExternalLogging** from **InlineLogging**. It must have the value **external** for the type **ExternalLogging**.

Property	Description
type	Must be **external**.
string	
name	The name of the **ConfigMap** from which to get the logging configuration.
string	

B.47. TLSSIDECAR SCHEMA REFERENCE

Used in: **CruiseControlSpec**, **EntityOperatorSpec**, **KafkaClusterSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Property	Description
image	The docker image for the container.
string	
livenessProbe	Pod liveness checking.
Probe	
logLevel	The log level for the TLS sidecar. Default value is **notice**.
string (one of [emerg, debug, crit, err, alert, warning, notice, info])	
readinessProbe	Pod readiness checking.
Probe	
resources	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	

B.48. KAFKACLUSTERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
statefulset	Template for Kafka **StatefulSet**.
StatefulSetTemplate	
pod	Template for Kafka **Pods**.
PodTemplate	
bootstrapService	Template for Kafka bootstrap **Service**.
ResourceTemplate	
brokersService	Template for Kafka broker **Service**.

Property	Description
ResourceTemplate	
externalBootstrapService	Template for Kafka external bootstrap **Service**.
ExternalServiceTemplate	
perPodService	Template for Kafka per-pod **Services** used for access from outside of OpenShift.
ExternalServiceTemplate	
externalBootstrapRoute	Template for Kafka external bootstrap **Route**.
ResourceTemplate	
perPodRoute	Template for Kafka per-pod **Routes** used for access from outside of OpenShift.
ResourceTemplate	
externalBootstrapIngress	Template for Kafka external bootstrap **Ingress**.
ResourceTemplate	
perPodIngress	Template for Kafka per-pod **Ingress** used for access from outside of OpenShift.
ResourceTemplate	
persistentVolumeClaim	Template for all Kafka **PersistentVolumeClaims**.
ResourceTemplate	
podDisruptionBudget	Template for Kafka **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	
kafkaContainer	Template for the Kafka broker container.
ContainerTemplate	
tlsSidecarContainer	Template for the Kafka broker TLS sidecar container.
ContainerTemplate	
initContainer	Template for the Kafka init container.
ContainerTemplate	

B.49. STATEFULSETTEMPLATE SCHEMA REFERENCE

Used in: **KafkaClusterTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata	Metadata which should be applied to the resource.
MetadataTemplate	
podManagementPolicy	PodManagementPolicy which will be used for this StatefulSet. Valid values are **Parallel** and **OrderedReady**. Defaults to **Parallel**.
string (one of [OrderedReady, Parallel])	

B.50. METADATATEMPLATE SCHEMA REFERENCE

Used in: **ExternalServiceTemplate**, **PodDisruptionBudgetTemplate**, **PodTemplate**, **ResourceTemplate**, **StatefulSetTemplate**

Property	Description
labels	Labels which should be added to the resource template. Can be applied to different resources such as **StatefulSets**, **Deployments**, **Pods**, and **Services**.
map	
annotations	Annotations which should be added to the resource template. Can be applied to different resources such as **StatefulSets**, **Deployments**, **Pods**, and **Services**.
map	

B.51. PODTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata	Metadata applied to the resource.
MetadataTemplate	
imagePullSecrets	List of references to secrets in the same namespace to use for pulling any of the images used by this Pod. See external documentation of core/v1 localobjectreference.
LocalObjectReference array	

Property	Description
securityContext PodSecurityContext	Configures pod-level security attributes and common container settings. See external documentation of core/v1 podsecuritycontext.
terminationGracePeriodSeconds integer	The grace period is the duration in seconds after the processes running in the pod are sent a termination signal and the time when the processes are forcibly halted with a kill signal. Set this value longer than the expected cleanup time for your process.Value must be non-negative integer. The value zero indicates delete immediately. Defaults to 30 seconds.
affinity Affinity	The pod's affinity rules. See external documentation of core/v1 affinity.
priorityClassName string	The name of the Priority Class to which these pods will be assigned.
schedulerName string	The name of the scheduler used to dispatch this **Pod**. If not specified, the default scheduler will be used.
tolerations Toleration array	The pod's tolerations. See external documentation of core/v1 toleration.

B.52. RESOURCETEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata **MetadataTemplate**	Metadata which should be applied to the resource.

B.53. EXTERNALSERVICETEMPLATE SCHEMA REFERENCE

Used in: **KafkaClusterTemplate**

Property	Description
metadata **MetadataTemplate**	Metadata which should be applied to the resource.
externalTrafficPolicy string (one of [Local, Cluster])	Specifies whether the service routes external traffic to node-local or cluster-wide endpoints. **Cluster** may cause a second hop to another node and obscures the client source IP. **Local** avoids a second hop for LoadBalancer and Nodeport type services and preserves the client source IP (when supported by the infrastructure). If unspecified, OpenShift will use **Cluster** as the default.
loadBalancerSourceRanges string array	A list of CIDR ranges (for example **10.0.0.0/8** or **130.211.204.1/32**) from which clients can connect to load balancer type listeners. If supported by the platform, traffic through the loadbalancer is restricted to the specified CIDR ranges. This field is applicable only for loadbalancer type services and is ignored if the cloud provider does not support the feature. For more information, see https://kubernetes.io/docs/tasks/access-application-cluster/configure-cloud-provider-firewall/.

B.54. PODDISRUPTIONBUDGETTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata **MetadataTemplate**	Metadata to apply to the **PodDistruptionBugetTemplate** resource.
maxUnavailable integer	Maximum number of unavailable pods to allow automatic Pod eviction. A Pod eviction is allowed when the **maxUnavailable** number of pods or fewer are unavailable after the eviction. Setting this value to 0 prevents all voluntary evictions, so the pods must be evicted manually. Defaults to 1.

B.55. CONTAINERTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Property	Description
env	Environment variables which should be applied to the container.
ContainerEnvVar array	
securityContext	Security context for the container. See external documentation of core/v1 securitycontext.
SecurityContext	

B.56. CONTAINERENVVAR SCHEMA REFERENCE

Used in: **ContainerTemplate**

Property	Description
name	The environment variable key.
string	
value	The environment variable value.
string	

B.57. ZOOKEEPERCLUSTERSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
replicas	The number of pods in the cluster.
integer	
image	The docker image for the pods.
string	
storage	Storage configuration (disk). Cannot be updated. The type depends on the value of the **storage.type** property within the given object, which must be one of [ephemeral, persistent-claim].
EphemeralStorage, **PersistentClaimStorage**	

Property	Description
config map	The ZooKeeper broker config. Properties with the following prefixes cannot be set: server., dataDir, dataLogDir, clientPort, authProvider, quorum.auth, requireClientAuthScheme, snapshot.trust.empty, standaloneEnabled, reconfigEnabled, 4lw.commands.whitelist, secureClientPort, ssl., serverCnxnFactory, sslQuorum (with the exception of: ssl.protocol, ssl.quorum.protocol, ssl.enabledProtocols, ssl.quorum.enabledProtocols, ssl.ciphersuites, ssl.quorum.ciphersuites, ssl.hostnameVerification, ssl.quorum.hostnameVerification).
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.zookeeper.template.pod.affinity.** The pod's affinity rules. See external documentation of core/v1 affinity.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.zookeeper.template.pod.tolerations.** The pod's tolerations. See external documentation of core/v1 toleration.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
jvmOptions **JvmOptions**	JVM Options for pods.
resources ResourceRequirements	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.

Property	Description
logging **InlineLogging**, **ExternalLogging**	Logging configuration for ZooKeeper. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
template **ZookeeperClusterTemplate**	Template for ZooKeeper cluster resources. The template allows users to specify how are the **StatefulSet**, **Pods** and **Services** generated.
tlsSidecar **TlsSidecar**	**The property tlsSidecar has been deprecated.** TLS sidecar configuration. The TLS sidecar is not used anymore and this option will be ignored.

B.58. ZOOKEEPERCLUSTERTEMPLATE SCHEMA REFERENCE

Used in: **ZookeeperClusterSpec**

Property	Description
statefulset **StatefulSetTemplate**	Template for ZooKeeper **StatefulSet**.
pod **PodTemplate**	Template for ZooKeeper **Pods**.
clientService **ResourceTemplate**	Template for ZooKeeper client **Service**.
nodesService **ResourceTemplate**	Template for ZooKeeper nodes **Service**.
persistentVolumeClaim **ResourceTemplate**	Template for all ZooKeeper **PersistentVolumeClaims**.
podDisruptionBudget **PodDisruptionBudgetTemplate**	Template for ZooKeeper **PodDisruptionBudget**.

Property	Description
zookeeperContainer **ContainerTemplate**	Template for the ZooKeeper container.
tlsSidecarContainer **ContainerTemplate**	**The property tlsSidecarContainer has been deprecated.** Template for the Zookeeper server TLS sidecar container. The TLS sidecar is not used anymore and this option will be ignored.

B.59. TOPICOPERATORSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
watchedNamespace string	The namespace the Topic Operator should watch.
image string	The image to use for the Topic Operator.
reconciliationIntervalSeconds integer	Interval between periodic reconciliations.
zookeeperSessionTimeoutSeconds integer	Timeout for the ZooKeeper session.
affinity Affinity	Pod affinity rules. See external documentation of core/v1 affinity.
resources ResourceRequirements	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
topicMetadataMaxAttempts integer	The number of attempts at getting topic metadata.
tlsSidecar	TLS sidecar configuration.

Property	Description
TlsSidecar	
logging	Logging configuration. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
InlineLogging, **ExternalLogging**	
jvmOptions	JVM Options for pods.
JvmOptions	
livenessProbe	Pod liveness checking.
Probe	
readinessProbe	Pod readiness checking.
Probe	

B.60. ENTITYOPERATORSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
topicOperator	Configuration of the Topic Operator.
EntityTopicOperatorSpec	
userOperator	Configuration of the User Operator.
EntityUserOperatorSpec	
affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules. See external documentation of core/v1 affinity.
Affinity	
tolerations	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations. See external documentation of core/v1 toleration.
Toleration array	
tlsSidecar	TLS sidecar configuration.

Property	Description
TlsSidecar	
template	Template for Entity Operator resources. The template allows users to specify how is the **Deployment** and **Pods** generated.
EntityOperatorTemplate	

B.61. ENTITYTOPICOPERATORSPEC SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
watchedNamespace	The namespace the Topic Operator should watch.
string	
image	The image to use for the Topic Operator.
string	
reconciliationIntervalSeconds	Interval between periodic reconciliations.
integer	
zookeeperSessionTimeoutSeconds	Timeout for the ZooKeeper session.
integer	
livenessProbe	Pod liveness checking.
Probe	
readinessProbe	Pod readiness checking.
Probe	
resources	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	
topicMetadataMaxAttempts	The number of attempts at getting topic metadata.
integer	

Property	Description
logging **InlineLogging**, **ExternalLogging**	Logging configuration. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
jvmOptions **JvmOptions**	JVM Options for pods.

B.62. ENTITYUSEROPERATORSPEC SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
watchedNamespace string	The namespace the User Operator should watch.
image string	The image to use for the User Operator.
reconciliationIntervalSeconds integer	Interval between periodic reconciliations.
zookeeperSessionTimeoutSeconds integer	Timeout for the ZooKeeper session.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
resources ResourceRequirements	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.

Property	Description
logging **InlineLogging**, **ExternalLogging**	Logging configuration. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
jvmOptions **JvmOptions**	JVM Options for pods.

B.63. ENTITYOPERATORTEMPLATE SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
deployment **ResourceTemplate**	Template for Entity Operator **Deployment**.
pod **PodTemplate**	Template for Entity Operator **Pods**.
tlsSidecarContainer **ContainerTemplate**	Template for the Entity Operator TLS sidecar container.
topicOperatorContainer **ContainerTemplate**	Template for the Entity Topic Operator container.
userOperatorContainer **ContainerTemplate**	Template for the Entity User Operator container.

B.64. CERTIFICATEAUTHORITY SCHEMA REFERENCE

Used in: **KafkaSpec**

Configuration of how TLS certificates are used within the cluster. This applies to certificates used for both internal communication within the cluster and to certificates used for client access via **Kafka.spec.kafka.listeners.tls**.

Property	Description
generateCertificateAuthority boolean	If true then Certificate Authority certificates will be generated automatically. Otherwise the user will need to provide a Secret with the CA certificate. Default is true.
validityDays integer	The number of days generated certificates should be valid for. The default is 365.
renewalDays integer	The number of days in the certificate renewal period. This is the number of days before the a certificate expires during which renewal actions may be performed. When **generateCertificateAuthority** is true, this will cause the generation of a new certificate. When **generateCertificateAuthority** is true, this will cause extra logging at WARN level about the pending certificate expiry. Default is 30.
certificateExpirationPolicy string (one of [replace-key, renew-certificate])	How should CA certificate expiration be handled when **generateCertificateAuthority=true**. The default is for a new CA certificate to be generated reusing the existing private key.

B.65. CRUISECONTROLSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
image string	The docker image for the pods.
config map	The Cruise Control configuration. For a full list of configuration options refer to https://github.com/linkedin/cruise-control/wiki/Configurations. Note that properties with the following prefixes cannot be set: bootstrap.servers, client.id, zookeeper., network., security., failed.brokers.zk.path,webserver.http., webserver.api.urlprefix, webserver.session.path, webserver.accesslog., two.step., request.reason.required,metric.reporter.sampler.bootstrap.servers, metric.reporter.topic, partition.metric.sample.store.topic, broker.metric.sample.store.topic,capacity.config.file, self.healing., anomaly.detection., ssl.

Property	Description
livenessProbe	Pod liveness checking for the Cruise Control container.
Probe	
readinessProbe	Pod readiness checking for the Cruise Control container.
Probe	
jvmOptions	JVM Options for the Cruise Control container.
JvmOptions	
resources	CPU and memory resources to reserve for the Cruise Control container. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	
logging	Logging configuration (log4j1) for Cruise Control. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
InlineLogging, **ExternalLogging**	
tlsSidecar	TLS sidecar configuration.
TlsSidecar	
template	Template to specify how Cruise Control resources, **Deployments** and **Pods**, are generated.
CruiseControlTemplate	
brokerCapacity	The Cruise Control **brokerCapacity** configuration.
BrokerCapacity	

B.66. CRUISECONTROLTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlSpec**

Property	Description
deployment	Template for Cruise Control **Deployment**.
ResourceTemplate	
pod	Template for Cruise Control **Pods**.

Property	Description
PodTemplate	
apiService	Template for Cruise Control API **Service**.
ResourceTemplate	
podDisruptionBudget	Template for Cruise Control **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	
cruiseControlContainer	Template for the Cruise Control container.
ContainerTemplate	
tlsSidecarContainer	Template for the Cruise Control TLS sidecar container.
ContainerTemplate	

B.67. BROKERCAPACITY SCHEMA REFERENCE

Used in: **CruiseControlSpec**

Property	Description
disk	Broker capacity for disk in bytes, for example, 100Gi.
string	
cpuUtilization	Broker capacity for CPU resource utilization as a percentage (0 - 100).
integer	
inboundNetwork	Broker capacity for inbound network throughput in bytes per second, for example, 10000KB/s.
string	
outboundNetwork	Broker capacity for outbound network throughput in bytes per second, for example 10000KB/s.
string	

B.68. KAFKAEXPORTERSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
image string	The docker image for the pods.
groupRegex string	Regular expression to specify which consumer groups to collect. Default value is **.***.
topicRegex string	Regular expression to specify which topics to collect. Default value is **.***.
resources ResourceRequirements	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
logging string	Only log messages with the given severity or above. Valid levels: [**debug**, **info**, **warn**, **error**, **fatal**]. Default log level is **info**.
enableSaramaLogging boolean	Enable Sarama logging, a Go client library used by the Kafka Exporter.
template **KafkaExporterTemplate**	Customization of deployment templates and pods.
livenessProbe **Probe**	Pod liveness check.
readinessProbe **Probe**	Pod readiness check.

B.69. KAFKAEXPORTERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaExporterSpec**

Property	Description
deployment	Template for Kafka Exporter **Deployment**.
ResourceTemplate	
pod	Template for Kafka Exporter **Pods**.
PodTemplate	
service	Template for Kafka Exporter **Service**.
ResourceTemplate	
container	Template for the Kafka Exporter container.
ContainerTemplate	

B.70. KAFKASTATUS SCHEMA REFERENCE

Used in: **Kafka**

Property	Description
conditions	List of status conditions.
Condition array	
observedGeneration	The generation of the CRD that was last reconciled by the operator.
integer	
listeners	Addresses of the internal and external listeners.
ListenerStatus array	

B.71. CONDITION SCHEMA REFERENCE

Used in: **KafkaBridgeStatus**, **KafkaConnectorStatus**, **KafkaConnectS2IStatus**, **KafkaConnectStatus**, **KafkaMirrorMaker2Status**, **KafkaMirrorMakerStatus**, **KafkaRebalanceStatus**, **KafkaStatus**, **KafkaTopicStatus**, **KafkaUserStatus**

Property	Description

Property	Description
type string	The unique identifier of a condition, used to distinguish between other conditions in the resource.
status string	The status of the condition, either True, False or Unknown.
lastTransitionTime string	Last time the condition of a type changed from one status to another. The required format is 'yyyy-MM-ddTHH:mm:ssZ', in the UTC time zone.
reason string	The reason for the condition's last transition (a single word in CamelCase).
message string	Human-readable message indicating details about the condition's last transition.

B.72. LISTENERSTATUS SCHEMA REFERENCE

Used in: **KafkaStatus**

Property	Description
type string	The type of the listener. Can be one of the following three types: **plain**, **tls**, and **external**.
addresses **ListenerAddress** array	A list of the addresses for this listener.
bootstrapServers string	A comma-separated list of **host:port** pairs for connecting to the Kafka cluster using this listener.
certificates string array	A list of TLS certificates which can be used to verify the identity of the server when connecting to the given listener. Set only for **tls** and **external** listeners.

B.73. LISTENERADDRESS SCHEMA REFERENCE

Used in: **ListenerStatus**

Property	Description
host string	The DNS name or IP address of the Kafka bootstrap service.
port integer	The port of the Kafka bootstrap service.

B.74. KAFKACONNECT SCHEMA REFERENCE

Property	Description
spec **KafkaConnectSpec**	The specification of the Kafka Connect cluster.
status **KafkaConnectStatus**	The status of the Kafka Connect cluster.

B.75. KAFKACONNECTSPEC SCHEMA REFERENCE

Used in: **KafkaConnect**

Property	Description
replicas integer	The number of pods in the Kafka Connect group.
version string	The Kafka Connect version. Defaults to 2.5.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.
image string	The docker image for the pods.
bootstrapServers	Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>:<port>* pairs.

Property	Description
string	
tls	TLS configuration.
KafkaConnectTls	
authentication	Authentication configuration for Kafka Connect. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, plain, oauth].
KafkaClientAuthenticationTls, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	
config	The Kafka Connect configuration. Properties with the following prefixes cannot be set: ssl., sasl., security., listeners, plugin.path, rest., bootstrap.servers, consumer.interceptor.classes, producer.interceptor.classes (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols).
map	
resources	The maximum limits for CPU and memory resources and the requested initial resources. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	
livenessProbe	Pod liveness checking.
Probe	
readinessProbe	Pod readiness checking.
Probe	
jvmOptions	JVM Options for pods.
JvmOptions	
affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules. See external documentation of core/v1 affinity.
Affinity	
tolerations	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations. See external documentation of core/v1 toleration.
Toleration array	

Property	Description
logging **InlineLogging**, **ExternalLogging**	Logging configuration for Kafka Connect. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
tracing **JaegerTracing**	The configuration of tracing in Kafka Connect. The type depends on the value of the **tracing.type** property within the given object, which must be one of [jaeger].
template **KafkaConnectTemplate**	Template for Kafka Connect and Kafka Connect S2I resources. The template allows users to specify how the **Deployment**, **Pods** and **Service** are generated.
externalConfiguration **ExternalConfiguration**	Pass data from Secrets or ConfigMaps to the Kafka Connect pods and use them to configure connectors.

B.76. KAFKACONNECTTLS SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

Property	Description
trustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection.

B.77. KAFKACLIENTAUTHENTICATIONTLS SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.

NOTE

TLS client authentication can only be used with TLS connections.

An example TLS client authentication configuration

```
authentication:
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: public.crt
    key: private.key
```

The **type** property is a discriminator that distinguishes the use of the type **KafkaClientAuthenticationTls** from **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**. It must have the value **tls** for the type **KafkaClientAuthenticationTls**.

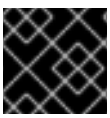
Property	Description
certificateAndKey	Reference to the **Secret** which holds the certificate and private key pair.
CertAndKeySecretSource	
type	Must be **tls**.
string	

B.78. KAFKACLIENTAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

To configure SASL-based SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. The SCRAM-SHA-512 authentication mechanism requires a username and password.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.



IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based SCRAM-SHA-512 client authentication configuration

```
authentication:
  type: scram-sha-512
```

```
username: my-connect
passwordSecret:
  secretName: my-connect
  password: password
```

The **type** property is a discriminator that distinguishes the use of the type **KafkaClientAuthenticationScramSha512** from **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**. It must have the value **scram-sha-512** for the type **KafkaClientAuthenticationScramSha512**.

Property	Description
passwordSecret	Reference to the **Secret** which holds the password.
PasswordSecretSource	
type	Must be **scram-sha-512**.
string	
username	Username used for the authentication.
string	

B.79. PASSWORDSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationScramSha512**

Property	Description
password	The name of the key in the Secret under which the password is stored.
string	
secretName	The name of the Secret containing the password.
string	

B.80. KAFKACLIENTAUTHENTICATIONPLAIN SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

To configure SASL-based PLAIN authentication, set the **type** property to **plain**. SASL PLAIN authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of such a **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based PLAIN client authentication configuration

```
authentication:
  type: plain
  username: my-connect
  passwordSecret:
    secretName: my-connect
    password: password
```

The **type** property is a discriminator that distinguishes the use of the type **KafkaClientAuthenticationPlain** from **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationOAuth**. It must have the value **plain** for the type **KafkaClientAuthenticationPlain**.

Property	Description
passwordSecret	Reference to the **Secret** which holds the password.
PasswordSecretSource	
type	Must be **plain**.
string	
username	Username used for the authentication.
string	

B.81. KAFKACLIENTAUTHENTICATIONOAUTH SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

To use OAuth client authentication, set the **type** property to the value **oauth**. OAuth authentication can be configured using:

- Client ID and secret
- Client ID and refresh token
- Access token
- TLS

Client ID and secret

You can configure the address of your authorization server in the **tokenEndpointUri** property together with the client ID and client secret used in authentication. The OAuth client will connect to the OAuth server, authenticate using the client ID and secret and get an access token which it will use to authenticate with the Kafka broker. In the **clientSecret** property, specify a link to a **Secret** containing the client secret.

An example of OAuth client authentication using client ID and client secret

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  clientSecret:
    secretName: my-client-oauth-secret
    key: client-secret
```

Client ID and refresh token

You can configure the address of your OAuth server in the **tokenEndpointUri** property together with the OAuth client ID and refresh token. The OAuth client will connect to the OAuth server, authenticate using the client ID and refresh token and get an access token which it will use to authenticate with the Kafka broker. In the **refreshToken** property, specify a link to a **Secret** containing the refresh token.

An example of OAuth client authentication using client ID and refresh token

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
```

Access token

You can configure the access token used for authentication with the Kafka broker directly. In this case, you do not specify the **tokenEndpointUri**. In the **accessToken** property, specify a link to a **Secret** containing the access token.

An example of OAuth client authentication using only an access token

```
authentication:
  type: oauth
  accessToken:
    secretName: my-access-token-secret
    key: access-token
```

TLS

Accessing the OAuth server using the HTTPS protocol does not require any additional configuration as long as the TLS certificates used by it are signed by a trusted certification authority and its hostname is listed in the certificate.

If your OAuth server is using certificates which are self-signed or are signed by a certification authority which is not trusted, you can configure a list of trusted certificates in the custom resoruce. The **tlsTrustedCertificates** property contains a list of secrets with key names under which the certificates are stored. The certificates must be stored in X509 format.

An example of TLS certificates provided

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
  tlsTrustedCertificates:
    - secretName: oauth-server-ca
      certificate: tls.crt
```

The OAuth client will by default verify that the hostname of your OAuth server matches either the certificate subject or one of the alternative DNS names. If it is not required, you can disable the hostname verification.

An example of disabled TLS hostname verification

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
  disableTlsHostnameVerification: true
```

The **type** property is a discriminator that distinguishes the use of the type **KafkaClientAuthenticationOAuth** from **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**. It must have the value **oauth** for the type **KafkaClientAuthenticationOAuth**.

Property	Description
accessToken **GenericSecretSource**	Link to OpenShift Secret containing the access token which was obtained from the authorization server.
accessTokenIsJwt boolean	Configure whether access token should be treated as JWT. This should be set to **false** if the authorization server returns opaque tokens. Defaults to **true**.
clientId string	OAuth Client ID which the Kafka client can use to authenticate against the OAuth server and use the token endpoint URI.
clientSecret **GenericSecretSource**	Link to OpenShift Secret containing the OAuth client secret which the Kafka client can use to authenticate against the OAuth server and use the token endpoint URI.
disableTlsHostnameVerification boolean	Enable or disable TLS hostname verification. Default value is **false**.
maxTokenExpirySeconds integer	Set or limit time-to-live of the access tokens to the specified number of seconds. This should be set if the authorization server returns opaque tokens.
refreshToken **GenericSecretSource**	Link to OpenShift Secret containing the refresh token which can be used to obtain access token from the authorization server.
scope string	OAuth scope to use when authenticating against the authorization server. Some authorization servers require this to be set. The possible values depend on how authorization server is configured. By default **scope** is not specified when doing the token endpoint request.
tlsTrustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection to the OAuth server.
tokenEndpointUri string	Authorization server token endpoint URI.
type	Must be **oauth**.

Property	Description
string	

B.82. JAEGERTRACING SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**

The **type** property is a discriminator that distinguishes the use of the type **JaegerTracing** from other subtypes which may be added in the future. It must have the value **jaeger** for the type **JaegerTracing**.

Property	Description
type	Must be **jaeger**.
string	

B.83. KAFKACONNECTTEMPLATE SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**

Property	Description
deployment	Template for Kafka Connect **Deployment**.
ResourceTemplate	
pod	Template for Kafka Connect **Pods**.
PodTemplate	
apiService	Template for Kafka Connect API **Service**.
ResourceTemplate	
connectContainer	Template for the Kafka Connect container.
ContainerTemplate	
podDisruptionBudget	Template for Kafka Connect **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

B.84. EXTERNALCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**

Property	Description
env **ExternalConfigurationEnv** array	Allows to pass data from Secret or ConfigMap to the Kafka Connect pods as environment variables.
volumes **ExternalConfigurationVolumeSource** array	Allows to pass data from Secret or ConfigMap to the Kafka Connect pods as volumes.

B.85. EXTERNALCONFIGURATIONENV SCHEMA REFERENCE

Used in: **ExternalConfiguration**

Property	Description
name string	Name of the environment variable which will be passed to the Kafka Connect pods. The name of the environment variable cannot start with **KAFKA_** or **STRIMZI_**.
valueFrom **ExternalConfigurationEnvVarSource**	Value of the environment variable which will be passed to the Kafka Connect pods. It can be passed either as a reference to Secret or ConfigMap field. The field has to specify exactly one Secret or ConfigMap.

B.86. EXTERNALCONFIGURATIONENVVARSOURCE SCHEMA REFERENCE

Used in: **ExternalConfigurationEnv**

Property	Description
configMapKeyRef ConfigMapKeySelector	Refernce to a key in a ConfigMap. See external documentation of core/v1 configmapkeyselector.
secretKeyRef SecretKeySelector	Reference to a key in a Secret. See external documentation of core/v1 secretkeyselector.

B.87. EXTERNALCONFIGURATIONVOLUMESOURCE SCHEMA REFERENCE

Used in: **ExternalConfiguration**

Property	Description
configMap ConfigMapVolumeSource	Reference to a key in a ConfigMap. Exactly one Secret or ConfigMap has to be specified. See external documentation of core/v1 configmapvolumesource.
name string	Name of the volume which will be added to the Kafka Connect pods.
secret SecretVolumeSource	Reference to a key in a Secret. Exactly one Secret or ConfigMap has to be specified. See external documentation of core/v1 secretvolumesource.

B.88. KAFKACONNECTSTATUS SCHEMA REFERENCE

Used in: **KafkaConnect**

Property	Description
conditions **Condition** array	List of status conditions.
observedGeneration integer	The generation of the CRD that was last reconciled by the operator.
url string	The URL of the REST API endpoint for managing and monitoring Kafka Connect connectors.
connectorPlugins **ConnectorPlugin** array	The list of connector plugins available in this Kafka Connect deployment.
podSelector LabelSelector	Label selector for pods providing this resource. See external documentation of meta/v1 labelselector.
replicas integer	The current number of pods being used to provide this resource.

B.89. CONNECTORPLUGIN SCHEMA REFERENCE

Used in: **KafkaConnectS2IStatus**, **KafkaConnectStatus**, **KafkaMirrorMaker2Status**

Property	Description
type	The type of the connector plugin. The available types are **sink** and **source**.
string	
version	The version of the connector plugin.
string	
class	The class of the connector plugin.
string	

B.90. KAFKACONNECTS2I SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Connect Source-to-Image (S2I) cluster.
KafkaConnectS2ISpec	
status	The status of the Kafka Connect Source-to-Image (S2I) cluster.
KafkaConnectS2IStatus	

B.91. KAFKACONNECTS2ISPEC SCHEMA REFERENCE

Used in: **KafkaConnectS2I**

Property	Description
replicas	The number of pods in the Kafka Connect group.
integer	
image	The docker image for the pods.
string	
buildResources	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	

Property	Description
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
jvmOptions **JvmOptions**	JVM Options for pods.
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules. See external documentation of core/v1 affinity.
logging **InlineLogging**, **ExternalLogging**	Logging configuration for Kafka Connect. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
template **KafkaConnectTemplate**	Template for Kafka Connect and Kafka Connect S2I resources. The template allows users to specify how the **Deployment**, **Pods** and **Service** are generated.
authentication **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	Authentication configuration for Kafka Connect. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, plain, oauth].
bootstrapServers string	Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>:<port>* pairs.

Property	Description
config map	The Kafka Connect configuration. Properties with the following prefixes cannot be set: ssl., sasl., security., listeners, plugin.path, rest., bootstrap.servers, consumer.interceptor.classes, producer.interceptor.classes (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols).
externalConfiguration **ExternalConfiguration**	Pass data from Secrets or ConfigMaps to the Kafka Connect pods and use them to configure connectors.
insecureSourceRepository boolean	When true this configures the source repository with the 'Local' reference policy and an import policy that accepts insecure source tags.
resources ResourceRequirements	The maximum limits for CPU and memory resources and the requested initial resources. See external documentation of core/v1 resourcerequirements.
tls **KafkaConnectTls**	TLS configuration.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations. See external documentation of core/v1 toleration.
tracing **JaegerTracing**	The configuration of tracing in Kafka Connect. The type depends on the value of the **tracing.type** property within the given object, which must be one of [jaeger].
version string	The Kafka Connect version. Defaults to 2.5.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.

B.92. KAFKACONNECTS2ISTATUS SCHEMA REFERENCE

Used in: **KafkaConnectS2I**

Property	Description
conditions	List of status conditions.
Condition array	
observedGeneration	The generation of the CRD that was last reconciled by the operator.
integer	
url	The URL of the REST API endpoint for managing and monitoring Kafka Connect connectors.
string	
connectorPlugins	The list of connector plugins available in this Kafka Connect deployment.
ConnectorPlugin array	
buildConfigName	The name of the build configuration.
string	
podSelector	Label selector for pods providing this resource. See external documentation of meta/v1 labelselector.
LabelSelector	
replicas	The current number of pods being used to provide this resource.
integer	

B.93. KAFKATOPIC SCHEMA REFERENCE

Property	Description
spec	The specification of the topic.
KafkaTopicSpec	
status	The status of the topic.
KafkaTopicStatus	

B.94. KAFKATOPICSPEC SCHEMA REFERENCE

Used in: **KafkaTopic**

Property	Description
partitions integer	The number of partitions the topic should have. This cannot be decreased after topic creation. It can be increased after topic creation, but it is important to understand the consequences that has, especially for topics with semantic partitioning.
replicas integer	The number of replicas the topic should have.
config map	The topic configuration.
topicName string	The name of the topic. When absent this will default to the metadata.name of the topic. It is recommended to not set this unless the topic name is not a valid OpenShift resource name.

B.95. KAFKATOPICSTATUS SCHEMA REFERENCE

Used in: **KafkaTopic**

Property	Description
conditions **Condition** array	List of status conditions.
observedGeneration integer	The generation of the CRD that was last reconciled by the operator.

B.96. KAFKAUSER SCHEMA REFERENCE

Property	Description
spec **KafkaUserSpec**	The specification of the user.
status **KafkaUserStatus**	The status of the Kafka User.

B.97. KAFKAUSERSPEC SCHEMA REFERENCE

Used in: **KafkaUser**

Property	Description
authentication **KafkaUserTlsClientAuthentication**, **KafkaUserScramSha512ClientAuthentication**	Authentication mechanism enabled for this Kafka user. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
authorization **KafkaUserAuthorizationSimple**	Authorization rules for this Kafka user. The type depends on the value of the **authorization.type** property within the given object, which must be one of [simple].
quotas **KafkaUserQuotas**	Quotas on requests to control the broker resources used by clients. Network bandwidth and request rate quotas can be enforced.Kafka documentation for Kafka User quotas can be found at http://kafka.apache.org/documentation/#design_quotas.

B.98. KAFKAUSERTLSCLIENTAUTHENTICATION SCHEMA REFERENCE

Used in: **KafkaUserSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaUserTlsClientAuthentication** from **KafkaUserScramSha512ClientAuthentication**. It must have the value **tls** for the type **KafkaUserTlsClientAuthentication**.

Property	Description
type string	Must be **tls**.

B.99. KAFKAUSERSCRAMSHA512CLIENTAUTHENTICATION SCHEMA REFERENCE

Used in: **KafkaUserSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaUserScramSha512ClientAuthentication** from **KafkaUserTlsClientAuthentication**. It must have the value **scram-sha-512** for the type **KafkaUserScramSha512ClientAuthentication**.

Property	Description
type	Must be **scram-sha-512**.

Property	Description
string	

B.100. KAFKAUSERAUTHORIZATIONSIMPLE SCHEMA REFERENCE

Used in: **KafkaUserSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaUserAuthorizationSimple** from other subtypes which may be added in the future. It must have the value **simple** for the type **KafkaUserAuthorizationSimple**.

Property	Description
type	Must be **simple**.
string	
acls	List of ACL rules which should be applied to this user.
AclRule array	

B.101. ACLRULE SCHEMA REFERENCE

Used in: **KafkaUserAuthorizationSimple**

Property	Description
host	The host from which the action described in the ACL rule is allowed or denied.
string	
operation	Operation which will be allowed or denied. Supported operations are: Read, Write, Create, Delete, Alter, Describe, ClusterAction, AlterConfigs, DescribeConfigs, IdempotentWrite and All.
string (one of [Read, Write, Delete, Alter, Describe, All, IdempotentWrite, ClusterAction, Create, AlterConfigs, DescribeConfigs])	
resource	Indicates the resource for which given ACL rule applies. The type depends on the value of the **resource.type** property within the given object, which must be one of [topic, group, cluster, transactionalId].
AclRuleTopicResource, **AclRuleGroupResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource**	

Property	Description
type string (one of [allow, deny])	The type of the rule. Currently the only supported type is **allow**. ACL rules with type**allow** are used to allow user to execute the specified operations. Default value is **allow**.

B.102. ACLRULETOPICRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleTopicResource** from **AclRuleGroupResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource**. It must have the value **topic** for the type **AclRuleTopicResource**.

Property	Description
type string	Must be **topic**.
name string	Name of resource for which given ACL rule applies. Can be combined with **patternType** field to use prefix pattern.
patternType string (one of [prefix, literal])	Describes the pattern used in the resource field. The supported types are **literal** and **prefix**. With **literal** pattern type, the resource field will be used as a definition of a full topic name. With **prefix** pattern type, the resource name will be used only as a prefix. Default value is **literal**.

B.103. ACLRULEGROUPRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleGroupResource** from **AclRuleTopicResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource**. It must have the value **group** for the type **AclRuleGroupResource**.

Property	Description
type string	Must be **group**.

Property	Description
name	Name of resource for which given ACL rule applies. Can be combined with **patternType** field to use prefix pattern.
string	
patternType	Describes the pattern used in the resource field. The supported types are **literal** and **prefix**. With **literal** pattern type, the resource field will be used as a definition of a full topic name. With **prefix** pattern type, the resource name will be used only as a prefix. Default value is **literal**.
string (one of [prefix, literal])	

B.104. ACLRULECLUSTERRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleClusterResource** from **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleTransactionalIdResource**. It must have the value **cluster** for the type **AclRuleClusterResource**.

Property	Description
type	Must be **cluster**.
string	

B.105. ACLRULETRANSACTIONALIDRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleTransactionalIdResource** from **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleClusterResource**. It must have the value **transactionalId** for the type **AclRuleTransactionalIdResource**.

Property	Description
type	Must be **transactionalId**.
string	
name	Name of resource for which given ACL rule applies. Can be combined with **patternType** field to use prefix pattern.
string	

Property	Description
patternType string (one of [prefix, literal])	Describes the pattern used in the resource field. The supported types are **literal** and **prefix**. With **literal** pattern type, the resource field will be used as a definition of a full name. With **prefix** pattern type, the resource name will be used only as a prefix. Default value is **literal**.

B.106. KAFKAUSERQUOTAS SCHEMA REFERENCE

Used in: **KafkaUserSpec**

Kafka allows a user to enforce certain quotas to control usage of resources by clients. Quotas split into two categories:

- *Network usage* quotas, which are defined as the byte rate threshold for each group of clients sharing a quota
- *CPU utilization* quotas, which are defined as the percentage of time a client can utilize on request handler I/O threads and network threads of each broker within a quota window

Using quotas for Kafka clients might be useful in a number of situations. Consider a wrongly configured Kafka producer which is sending requests at too high a rate. Such misconfiguration can cause a denial of service to other clients, so the problematic client ought to be blocked. By using a network limiting quota, it is possible to prevent this situation from significantly impacting other clients.

AMQ Streams supports user-level quotas, but not client-level quotas.

An example Kafka user quotas

```
spec:
  quotas:
    producerByteRate: 1048576
    consumerByteRate: 2097152
    requestPercentage: 55
```

For more info about Kafka user quotas visit Apache Kafka documentation.

Property	Description
consumerByteRate integer	A quota on the maximum bytes per-second that each client group can fetch from a broker before the clients in the group are throttled. Defined on a per-broker basis.

Property	Description
producerByteRate integer	A quota on the maximum bytes per-second that each client group can publish to a broker before the clients in the group are throttled. Defined on a per-broker basis.
requestPercentage integer	A quota on the maximum CPU utilization of each client group as a percentage of network and I/O threads.

B.107. KAFKAUSERSTATUS SCHEMA REFERENCE

Used in: **KafkaUser**

Property	Description
conditions **Condition** array	List of status conditions.
observedGeneration integer	The generation of the CRD that was last reconciled by the operator.
username string	Username.
secret string	The name of **Secret** where the credentials are stored.

B.108. KAFKAMIRRORMAKER SCHEMA REFERENCE

Property	Description
spec **KafkaMirrorMakerSpec**	The specification of Kafka MirrorMaker.
status **KafkaMirrorMakerStatus**	The status of Kafka MirrorMaker.

B.109. KAFKAMIRRORMAKERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker**

Property	Description
replicas	The number of pods in the **Deployment**.
integer	
image	The docker image for the pods.
string	
whitelist	List of topics which are included for mirroring. This option allows any regular expression using Java-style regular expressions. Mirroring two topics named A and B is achieved by using the whitelist **'A|B'**. Or, as a special case, you can mirror all topics using the whitelist '*'. You can also specify multiple regular expressions separated by commas.
string	
consumer	Configuration of source cluster.
KafkaMirrorMakerConsumerSpec	
producer	Configuration of target cluster.
KafkaMirrorMakerProducerSpec	
resources	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	
affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules. See external documentation of core/v1 affinity.
Affinity	
tolerations	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations. See external documentation of core/v1 toleration.
Toleration array	
jvmOptions	JVM Options for pods.
JvmOptions	

Property	Description
logging **InlineLogging**, **ExternalLogging**	Logging configuration for MirrorMaker. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
metrics map	The Prometheus JMX Exporter configuration. See JMX Exporter documentation for details of the structure of this configuration.
tracing **JaegerTracing**	The configuration of tracing in Kafka MirrorMaker. The type depends on the value of the **tracing.type** property within the given object, which must be one of [jaeger].
template **KafkaMirrorMakerTemplate**	Template to specify how Kafka MirrorMaker resources, **Deployments** and **Pods**, are generated.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
version string	The Kafka MirrorMaker version. Defaults to 2.5.0. Consult the documentation to understand the process required to upgrade or downgrade the version.

B.110. KAFKAMIRRORMAKERCONSUMERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example SSL configuration

```
spec:
  consumer:
    config:
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 1
      ssl.enabled.protocols: "TLSv1.2" 2
      ssl.protocol: "TLSv1.2" 3
```

1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

Property	Description
numStreams integer	Specifies the number of consumer stream threads to create.
offsetCommitInterval integer	Specifies the offset auto-commit interval in ms. Default value is 60000.
groupId string	A unique string that identifies the consumer group this consumer belongs to.
bootstrapServers string	A list of host:port pairs for establishing the initial connection to the Kafka cluster.
authentication **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	Authentication configuration for connecting to the cluster. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, plain, oauth].
config map	The MirrorMaker consumer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, group.id, sasl., security., interceptor.classes (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols).
tls **KafkaMirrorMakerTls**	TLS configuration for connecting MirrorMaker to the cluster.

B.111. KAFKAMIRRORMAKERTLS SCHEMA REFERENCE

Used in: **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Use the **tls** property to configure TLS encryption. Provide a list of secrets with key names under which the certificates are stored in X.509 format.

An example TLS encryption configuration

```
tls:
  trustedCertificates:
    - secretName: my-cluster-cluster-ca-cert
      certificate: ca.crt
```

Property	Description
trustedCertificates	Trusted certificates for TLS connection.
CertSecretSource array	

B.112. KAFKAMIRRORMAKERPRODUCERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example SSL configuration

```
spec:
  producer:
    config:
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 1
      ssl.enabled.protocols: "TLSv1.2" 2
      ssl.protocol: "TLSv1.2" 3
```

1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

Property	Description
bootstrapServers	A list of host:port pairs for establishing the initial connection to the Kafka cluster.
string	
abortOnSendFailure	Flag to set the MirrorMaker to exit on a failed send. Default value is **true**.
boolean	

Property	Description
authentication	Authentication configuration for connecting to the cluster. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, plain, oauth].
KafkaClientAuthenticationTls, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	
config	The MirrorMaker producer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, sasl., security., interceptor.classes (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols).
map	
tls	TLS configuration for connecting MirrorMaker to the cluster.
KafkaMirrorMakerTls	

B.113. KAFKAMIRRORMAKERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Property	Description
deployment	Template for Kafka MirrorMaker **Deployment**.
ResourceTemplate	
pod	Template for Kafka MirrorMaker **Pods**.
PodTemplate	
mirrorMakerContainer	Template for Kafka MirrorMaker container.
ContainerTemplate	
podDisruptionBudget	Template for Kafka MirrorMaker **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

B.114. KAFKAMIRRORMAKERSTATUS SCHEMA REFERENCE

Used in: **KafkaMirrorMaker**

Property	Description
conditions	List of status conditions.
Condition array	
observedGeneration	The generation of the CRD that was last reconciled by the operator.
integer	
podSelector	Label selector for pods providing this resource. See external documentation of meta/v1 labelselector.
LabelSelector	
replicas	The current number of pods being used to provide this resource.
integer	

B.115. KAFKABRIDGE SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Bridge.
KafkaBridgeSpec	
status	The status of the Kafka Bridge.
KafkaBridgeStatus	

B.116. KAFKABRIDGESPEC SCHEMA REFERENCE

Used in: **KafkaBridge**

Property	Description
replicas	The number of pods in the **Deployment**.
integer	
image	The docker image for the pods.
string	

Property	Description
bootstrapServers string	A list of host:port pairs for establishing the initial connection to the Kafka cluster.
tls **KafkaBridgeTls**	TLS configuration for connecting Kafka Bridge to the cluster.
authentication **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	Authentication configuration for connecting to the cluster. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, plain, oauth].
http **KafkaBridgeHttpConfig**	The HTTP related configuration.
consumer **KafkaBridgeConsumerSpec**	Kafka consumer related configuration.
producer **KafkaBridgeProducerSpec**	Kafka producer related configuration.
resources ResourceRequirements	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
jvmOptions **JvmOptions**	**Currently not supported** JVM Options for pods.
logging **InlineLogging**, **ExternalLogging**	Logging configuration for Kafka Bridge. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
metrics map	**Currently not supported** The Prometheus JMX Exporter configuration. See JMX Exporter documentation for details of the structure of this configuration.
livenessProbe	Pod liveness checking.

Property	Description
Probe	
readinessProbe	Pod readiness checking.
Probe	
template	Template for Kafka Bridge resources. The template allows users to specify how is the **Deployment** and **Pods** generated.
KafkaBridgeTemplate	
tracing	The configuration of tracing in Kafka Bridge. The type depends on the value of the **tracing.type** property within the given object, which must be one of [jaeger].
JaegerTracing	

B.117. KAFKABRIDGETLS SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
trustedCertificates	Trusted certificates for TLS connection.
CertSecretSource array	

B.118. KAFKABRIDGEHTTPCONFIG SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
port	The port which is the server listening on.
integer	
cors	CORS configuration for the HTTP Bridge.
KafkaBridgeHttpCors	

B.119. KAFKABRIDGEHTTPCORS SCHEMA REFERENCE

Used in: **KafkaBridgeHttpConfig**

Property	Description
allowedOrigins string array	List of allowed origins. Java regular expressions can be used.
allowedMethods string array	List of allowed HTTP methods.

B.120. KAFKABRIDGECONSUMERSPEC SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
config map	The Kafka consumer configuration used for consumer instances created by the bridge. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, group.id, sasl., security. (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols).

B.121. KAFKABRIDGEPRODUCERSPEC SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
config map	The Kafka producer configuration used for producer instances created by the bridge. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, sasl., security. (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols).

B.122. KAFKABRIDGETEMPLATE SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
deployment	Template for Kafka Bridge **Deployment**.

Property	Description
ResourceTemplate	
pod	Template for Kafka Bridge **Pods**.
PodTemplate	
apiService	Template for Kafka Bridge API **Service**.
ResourceTemplate	
bridgeContainer	Template for the Kafka Bridge container.
ContainerTemplate	
podDisruptionBudget	Template for Kafka Bridge **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

B.123. KAFKABRIDGESTATUS SCHEMA REFERENCE

Used in: **KafkaBridge**

Property	Description
conditions	List of status conditions.
Condition array	
observedGeneration	The generation of the CRD that was last reconciled by the operator.
integer	
url	The URL at which external client applications can access the Kafka Bridge.
string	
podSelector	Label selector for pods providing this resource. See external documentation of meta/v1 labelselector.
LabelSelector	
replicas	The current number of pods being used to provide this resource.
integer	

B.124. KAFKACONNECTOR SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Connector.
KafkaConnectorSpec	
status	The status of the Kafka Connector.
KafkaConnectorStatus	

B.125. KAFKACONNECTORSPEC SCHEMA REFERENCE

Used in: **KafkaConnector**

Property	Description
class	The Class for the Kafka Connector.
string	
tasksMax	The maximum number of tasks for the Kafka Connector.
integer	
config	The Kafka Connector configuration. The following properties cannot be set: connector.class, tasks.max.
map	
pause	Whether the connector should be paused. Defaults to false.
boolean	

B.126. KAFKACONNECTORSTATUS SCHEMA REFERENCE

Used in: **KafkaConnector**

Property	Description
conditions	List of status conditions.
Condition array	

Property	Description
observedGeneration	The generation of the CRD that was last reconciled by the operator.
integer	
connectorStatus	The connector status, as reported by the Kafka Connect REST API.
map	
tasksMax	The maximum number of tasks for the Kafka Connector.
integer	

B.127. KAFKAMIRRORMAKER2 SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka MirrorMaker 2.0 cluster.
KafkaMirrorMaker2Spec	
status	The status of the Kafka MirrorMaker 2.0 cluster.
KafkaMirrorMaker2Status	

B.128. KAFKAMIRRORMAKER2SPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2**

Property	Description
replicas	The number of pods in the Kafka Connect group.
integer	
version	The Kafka Connect version. Defaults to 2.5.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.
string	
image	The docker image for the pods.
string	

Property	Description
connectCluster string	The cluster alias used for Kafka Connect. The alias must match a cluster in the list at **spec.clusters**.
clusters **KafkaMirrorMaker2ClusterSpec** array	Kafka clusters for mirroring.
mirrors **KafkaMirrorMaker2MirrorSpec** array	Configuration of the MirrorMaker 2.0 connectors.
resources ResourceRequirements	The maximum limits for CPU and memory resources and the requested initial resources. See external documentation of core/v1 resourcerequirements.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
jvmOptions **JvmOptions**	JVM Options for pods.
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules. See external documentation of core/v1 affinity.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations. See external documentation of core/v1 toleration.
logging **InlineLogging**, **ExternalLogging**	Logging configuration for Kafka Connect. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
metrics	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.

Property	Description
map	
tracing **JaegerTracing**	The configuration of tracing in Kafka Connect. The type depends on the value of the **tracing.type** property within the given object, which must be one of [jaeger].
template **KafkaConnectTemplate**	Template for Kafka Connect and Kafka Connect S2I resources. The template allows users to specify how the **Deployment**, **Pods** and **Service** are generated.
externalConfiguration **ExternalConfiguration**	Pass data from Secrets or ConfigMaps to the Kafka Connect pods and use them to configure connectors.

B.129. KAFKAMIRRORMAKER2CLUSTERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2Spec**

Use the three allowed **ssl** configuration options to run external listeners with a specific *cipher suite* for a TLS version. A *cipher suite* combines algorithms for secure connection and data transfer.

Example SSL configuration

```
spec:
  clusters:
    config:
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (1)
      ssl.enabled.protocols: "TLSv1.2" (2)
      ssl.protocol: "TLSv1.2" (3)
```

1. The cipher suite for TLS using a combination of **ECDHE** key exchange mechanism, **RSA** authentication algorithm, **AES** bulk encyption algorithm and **SHA384** MAC algorithm.
2. The SSl protocol **TLSv1.2** is enabled.
3. Specifies the **TLSv1.2** protocol to generate the SSL context. Allowed values are **TLSv1.1** and **TLSv1.2**.

Property	Description
alias	Alias used to reference the Kafka cluster.

Property	Description
string	
bootstrapServers	A comma-separated list of **host:port** pairs for establishing the connection to the Kafka cluster.
string	
config	The MirrorMaker 2.0 cluster config. Properties with the following prefixes cannot be set: ssl., sasl., security., listeners, plugin.path, rest., bootstrap.servers, consumer.interceptor.classes, producer.interceptor.classes (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols).
map	
tls	TLS configuration for connecting MirrorMaker 2.0 connectors to a cluster.
KafkaMirrorMaker2Tls	
authentication	Authentication configuration for connecting to the cluster. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512, plain, oauth].
KafkaClientAuthenticationTls, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	

B.130. KAFKAMIRRORMAKER2TLS SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2ClusterSpec**

Property	Description
trustedCertificates	Trusted certificates for TLS connection.
CertSecretSource array	

B.131. KAFKAMIRRORMAKER2MIRRORSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2Spec**

Property	Description
sourceCluster	The alias of the source cluster used by the Kafka MirrorMaker 2.0 connectors. The alias must match a cluster in the list at **spec.clusters**.
string	

Property	Description
targetCluster string	The alias of the target cluster used by the Kafka MirrorMaker 2.0 connectors. The alias must match a cluster in the list at **spec.clusters**.
sourceConnector **KafkaMirrorMaker2ConnectorSpec**	The specification of the Kafka MirrorMaker 2.0 source connector.
checkpointConnector **KafkaMirrorMaker2ConnectorSpec**	The specification of the Kafka MirrorMaker 2.0 checkpoint connector.
heartbeatConnector **KafkaMirrorMaker2ConnectorSpec**	The specification of the Kafka MirrorMaker 2.0 heartbeat connector.
topicsPattern string	A regular expression matching the topics to be mirrored, for example, "topic1|topic2|topic3". Comma-separated lists are also supported.
topicsBlacklistPattern string	A regular expression matching the topics to exclude from mirroring. Comma-separated lists are also supported.
groupsPattern string	A regular expression matching the consumer groups to be mirrored. Comma-separated lists are also supported.
groupsBlacklistPattern string	A regular expression matching the consumer groups to exclude from mirroring. Comma-separated lists are also supported.

B.132. KAFKAMIRRORMAKER2CONNECTORSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2MirrorSpec**

Property	Description
tasksMax integer	The maximum number of tasks for the Kafka Connector.
config	The Kafka Connector configuration. The following properties cannot be set: connector.class, tasks.max.

Property	Description
map	
pause boolean	Whether the connector should be paused. Defaults to false.

B.133. KAFKAMIRRORMAKER2STATUS SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2**

Property	Description
conditions **Condition** array	List of status conditions.
observedGeneration integer	The generation of the CRD that was last reconciled by the operator.
url string	The URL of the REST API endpoint for managing and monitoring Kafka Connect connectors.
connectorPlugins **ConnectorPlugin** array	The list of connector plugins available in this Kafka Connect deployment.
connectors map array	List of MirrorMaker 2.0 connector statuses, as reported by the Kafka Connect REST API.
podSelector LabelSelector	Label selector for pods providing this resource. See external documentation of meta/v1 labelselector.
replicas integer	The current number of pods being used to provide this resource.

B.134. KAFKAREBALANCE SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka rebalance.
KafkaRebalanceSpec	
status	The status of the Kafka rebalance.
KafkaRebalanceStatus	

B.135. KAFKAREBALANCESPEC SCHEMA REFERENCE

Used in: **KafkaRebalance**

Property	Description
goals	A list of goals, ordered by decreasing priority, to use for generating and executing the rebalance proposal. The supported goals are available at https://github.com/linkedin/cruise-control#goals. If an empty goals list is provided, the goals declared in the default.goals Cruise Control configuration parameter are used.
string array	
skipHardGoalCheck	Whether to allow the hard goals specified in the Kafka CR to be skipped in rebalance proposal generation. This can be useful when some of those hard goals are preventing a balance solution being found. Default is false.
boolean	

B.136. KAFKAREBALANCESTATUS SCHEMA REFERENCE

Used in: **KafkaRebalance**

Property	Description
conditions	List of status conditions.
Condition array	
observedGeneration	The generation of the CRD that was last reconciled by the operator.
integer	

Property	Description
sessionId string	The session identifier for requests to Cruise Control pertaining to this KafkaRebalance resource. This is used by the Kafka Rebalance operator to track the status of ongoing rebalancing operations.
optimizationResult map	A JSON object describing the optimization result.

APPENDIX C. 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 **Red Hat AMQ Streams** entries in the **JBOSS INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ Streams product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Revised on 2020-07-09 06:57:11 UTC