Red Hat AMQ 2020.Q4

Using AMQ Streams on OpenShift

For use with AMQ Streams 1.6 on OpenShift Container Platform

Last Updated: 2022-02-01

Red Hat AMQ 2020.Q4 Using AMQ Streams on OpenShift

For use with AMQ Streams 1.6 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 provides instructions for configuring Kafka components and 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.

You can configure your deployment using AMQ Streams custom resources. The Custom resource API reference describes the properties you can use in your configuration.

NOTE

Looking to get started with AMQ Streams? For step-by-step deployment instructions, see the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

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

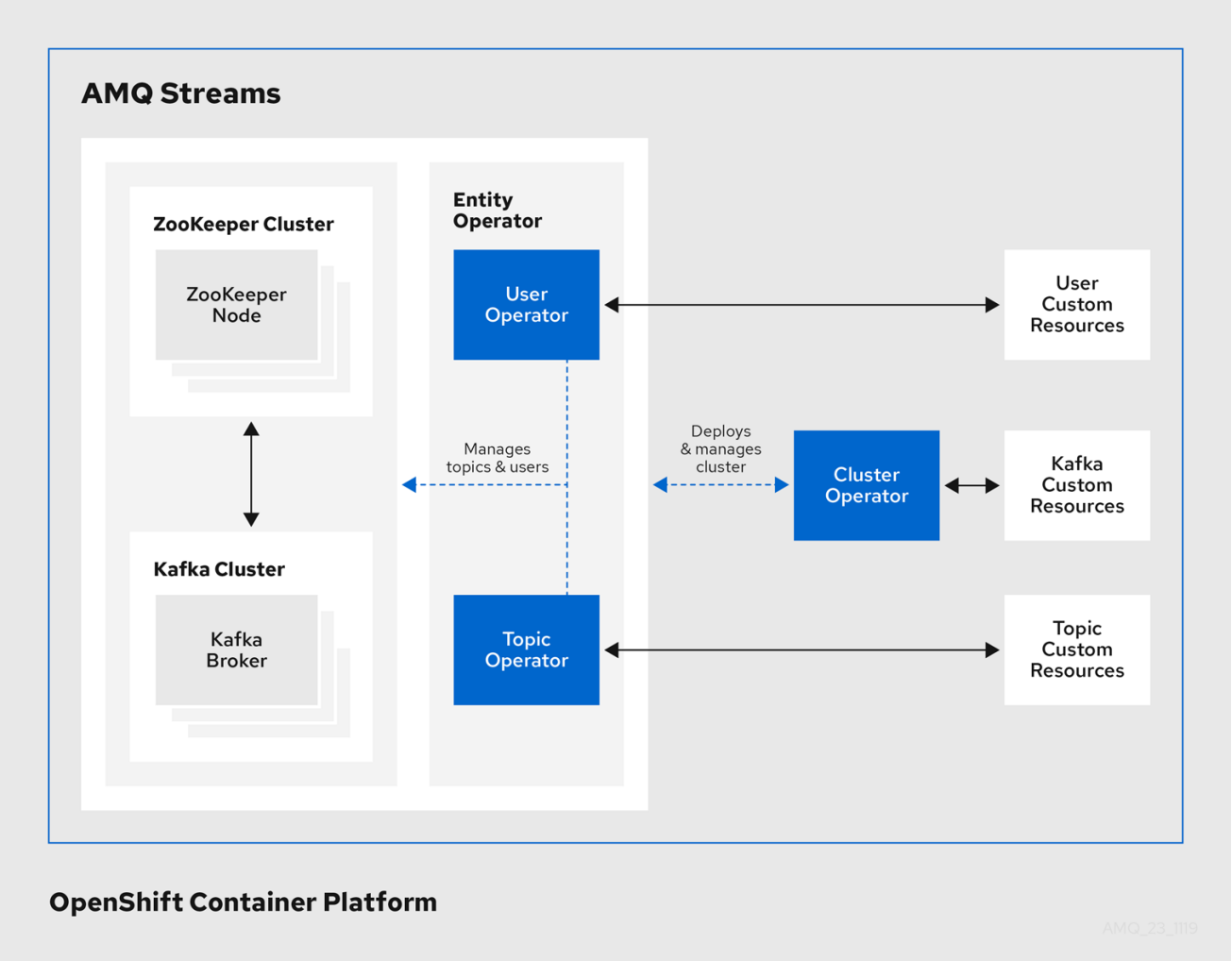
Manages Kafka topics

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.

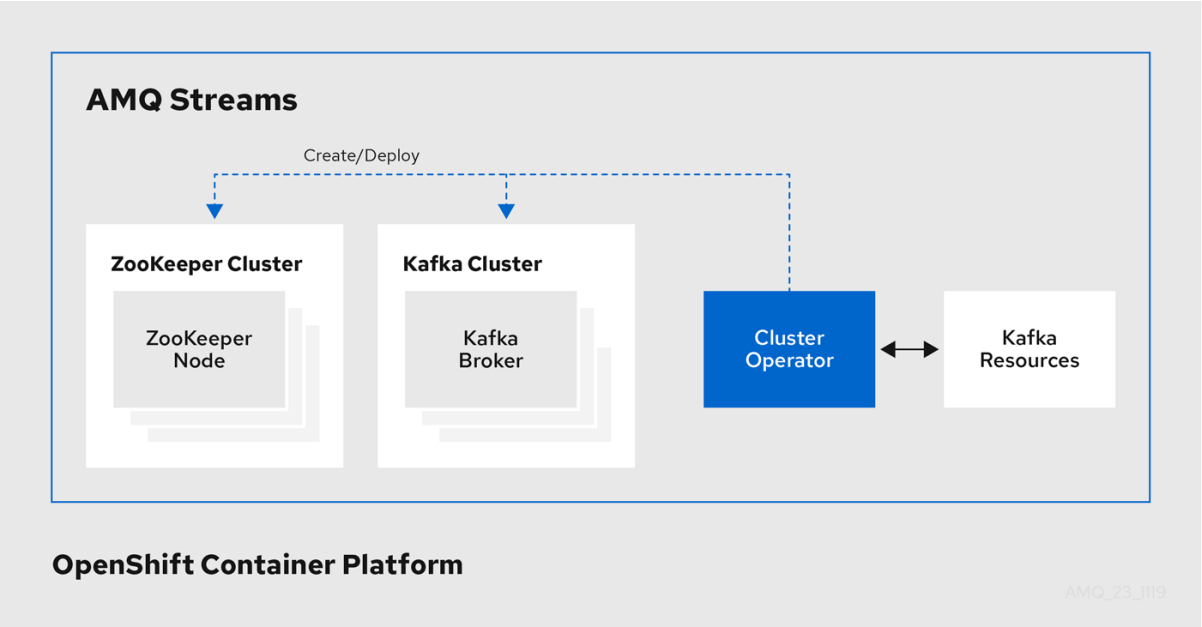
The Cluster Operator can also deploy (through configuration of the **Kafka** resource):

- 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.

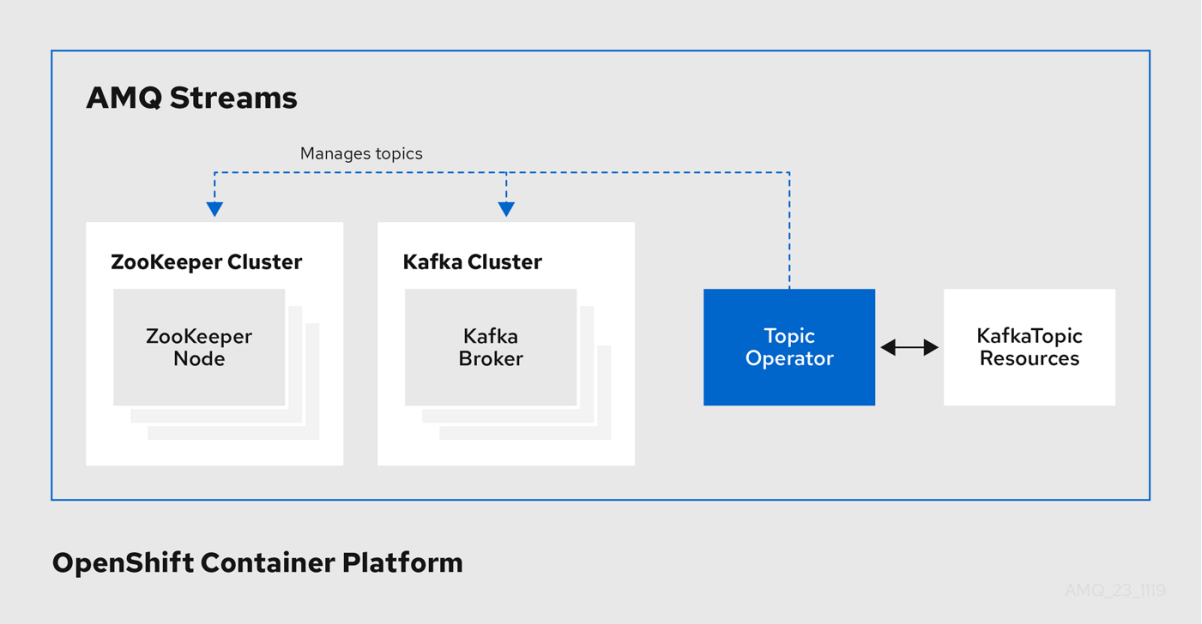
Example architecture for the Cluster Operator



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. For more information, see Designating AMQ Streams administrators in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

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. LISTENER CONFIGURATION

Listeners are used to connect to Kafka brokers.

AMQ Streams provides a generic **GenericKafkaListener** schema with properties to configure listeners through the **Kafka** resource.

The **GenericKafkaListener** provides a flexible approach to listener configuration.

You can specify properties to configure *internal* listeners for connecting within the OpenShift cluster, or *external* listeners for connecting outside the OpenShift cluster.

Generic listener configuration

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

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

Generic listener configuration replaces the previous approach to listener configuration using the **KafkaListeners** schema reference, which is **deprecated**. However, you can convert the old format into the new format with backwards compatibility.

The **KafkaListeners** schema uses sub-properties for **plain**, **tls** and **external** listeners, with fixed ports for each. Because of the limits inherent in the architecture of the schema, it is only possible to configure three listeners, with configuration options limited to the type of listener.

With the **GenericKafkaListener** schema, you can configure as many listeners as required, as long as their names and ports are unique.

You might want to configure multiple external listeners, for example, to handle access from networks that require different authentication mechanisms. Or you might need to join your OpenShift network to an outside network. In which case, you can configure internal listeners (using the **useServiceDnsDomain** property) so that the OpenShift service DNS domain (typically **.cluster.local**) is not used.

Configuring listeners to secure access to Kafka brokers

You can configure listeners for secure connection using authentication. For more information on securing access to Kafka brokers, see Managing access to Kafka.

Configuring external listeners for client access outside OpenShift

You can configure external listeners for client access outside an OpenShift environment using a specified connection mechanism, such as a loadbalancer. For more information on the configuration options for connecting an external client, see Configuring external listeners.

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 Kafka listener certificates.

1.7. 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. 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

2.1. KAFKA CLUSTER CONFIGURATION

The full schema of the **Kafka** resource is described in the Section B.2, "**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.

2.1.1. 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: 1.6 (2)
```

```
  resources: 3
    requests:
      memory: 64Gi
      cpu: "8"
    limits: 4
      memory: 64Gi
      cpu: "12"
  jvmOptions: 5
    -Xms: 8192m
    -Xmx: 8192m
  listeners: 6
    - name: plain 7
      port: 9092 8
      type: internal 9
      tls: false 10
      configuration:
        useServiceDnsDomain: true 11
    - name: tls
      port: 9093
      type: internal
      tls: true
      authentication: 12
        type: tls
    - name: external 13
      port: 9094
      type: route
      tls: true
      configuration:
        brokerCertChainAndKey: 14
          secretName: my-secret
          certificate: my-certificate.crt
          key: my-key.key
  authorization: 15
    type: simple
  config: 16
    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" 17
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
  storage: 18
    type: persistent-claim 19
    size: 10000Gi 20
  rack: 21
    topologyKey: topology.kubernetes.io/zone
  metrics: 22
    lowercaseOutputName: true
    rules: 23
    # 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: 24
  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: 25
  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: 26
  # ...
cruiseControl: 27
  # ...
```

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 *internal* or *external* listeners for connection inside or outside the OpenShift cluster .

7 Name to identify the listener. Must be unique within the Kafka cluster.

8 Port number used by the listener inside Kafka. The port number has to be unique within a given Kafka cluster. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX. Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.

9 Listener type specified as **internal**, or for external listeners, as **route**, **loadbalancer**, **nodeport** or **ingress**.

10 Enables TLS encryption for each listener. Default is **false**. TLS encryption is not required for **route** listeners.

11 Defines whether the fully-qualified DNS names including the cluster service suffix (usually **.cluster.local**) are assigned.

12 Listener authentication mechanism specified as mutual TLS, SCRAM-SHA-512 or token-based OAuth 2.0.

13 External listener configuration specifies how the Kafka cluster is exposed outside OpenShift, such as through a **route**, **loadbalancer** or **nodeport**.

14 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.

15 Authorization enables simple, OAUTH 2.0 or OPA authorization on the Kafka broker. Simple authorization uses the **AclAuthorizer** Kafka plugin.

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

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

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

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

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

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

22 Kafka metrics configuration for use with Prometheus.

23 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.

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

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

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

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

2.1.2. 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.

2.1.2.1. 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.

2.1.2.2. Apache Kafka and ZooKeeper storage

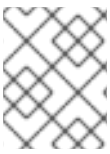
Use separate disks for Apache Kafka and ZooKeeper.

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.



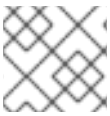
NOTE

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

2.1.3. 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.

2.1.3.1. Ephemeral storage

Ephemeral storage uses the **emptyDir** 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
    # ...
```

2.1.3.1.1. 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**.

2.1.3.2. 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
# ...
```

2.1.3.2.1. Storage class overrides

You can specify a different storage class for one or more Kafka brokers or ZooKeeper nodes, 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
  # ...
  zookeeper:
    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 volumes use the following storage classes:

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

- The persistent volumes of Kafka 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.

2.1.3.2.2. 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***.

2.1.3.2.3. 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**.

2.1.3.3. 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.

2.1.3.4. 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.

2.1.3.4.1. 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.

2.1.3.4.2. 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***.

2.1.3.4.3. Log directories

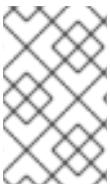
The JBOD volumes will be used by the Kafka brokers as log directories mounted into the following path:

/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**.

2.1.3.5. 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 KAFKA-CONFIG-FILE
```

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

Additional resources

For more information about reassigning topics, see Section 2.1.24.2, “Partition reassignment”.

2.1.3.6. 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 2.1.24.2, “Partition reassignment”.

2.1.4. 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.

2.1.4.1. 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 2.1.24, “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 2.1.24, “Scaling clusters”.

2.1.5. 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.

For more information on broker configuration, see the **KafkaClusterSpec** schema.

Listener configuration

You configure listeners for connecting to Kafka brokers. For more information on configuring listeners, see Listener configuration

Authorizing access to Kafka

You can configure your Kafka cluster to allow or decline actions executed by users. For more information on securing access to Kafka brokers, see Managing access to Kafka.

2.1.5.1. 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**.

2.1.6. Listener configuration

Listeners are used to connect to Kafka brokers.

AMQ Streams provides a generic **GenericKafkaListener** schema with properties to configure listeners through the **Kafka** resource.

The **GenericKafkaListener** provides a flexible approach to listener configuration.

You can specify properties to configure *internal* listeners for connecting within the OpenShift cluster, or *external* listeners for connecting outside the OpenShift cluster.

Generic listener configuration

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

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

Generic listener configuration replaces the previous approach to listener configuration using the **KafkaListeners** schema reference, which is **deprecated**. However, you can convert the old format into the new format with backwards compatibility.

The **KafkaListeners** schema uses sub-properties for **plain**, **tls** and **external** listeners, with fixed ports for each. Because of the limits inherent in the architecture of the schema, it is only possible to configure three listeners, with configuration options limited to the type of listener.

With the **GenericKafkaListener** schema, you can configure as many listeners as required, as long as their names and ports are unique.

You might want to configure multiple external listeners, for example, to handle access from networks that require different authentication mechanisms. Or you might need to join your OpenShift network to an outside network. In which case, you can configure internal listeners (using the **useServiceDnsDomain** property) so that the OpenShift service DNS domain (typically **.cluster.local**) is not used.

Configuring listeners to secure access to Kafka brokers

You can configure listeners for secure connection using authentication. For more information on securing access to Kafka brokers, see Managing access to Kafka.

Configuring external listeners for client access outside OpenShift

You can configure external listeners for client access outside an OpenShift environment using a specified connection mechanism, such as a loadbalancer. For more information on the configuration options for connecting an external client, see Configuring external listeners.

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 Kafka listener certificates.

2.1.7. 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.

2.1.7.1. 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
    # ...
```

2.1.7.2. 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**.

2.1.8. 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.

2.1.8.1. 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 for client connection using 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**.

2.1.8.2. 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**.

2.1.9. 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, you can use a terminal inside a ZooKeeper container and connect to **localhost:12181** as the ZooKeeper address.

2.1.9.1. Connecting to ZooKeeper from a terminal

Most Kafka CLI tools can connect directly to Kafka. So you should under normal circumstances not need to connect to ZooKeeper. In case it is needed, you can follow this procedure. Open a terminal inside a ZooKeeper 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-zookeeper-0 -- bin/kafka-topics.sh --list --zookeeper localhost:12181
```

   Be sure to use **localhost:12181**.

   You can now run Kafka commands to ZooKeeper.

2.1.10. 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.

2.1.10.1. 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 2.1.19, "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 2.6, "Customizing OpenShift resources".

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.

2.1.10.2. 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 2.1.18, “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 2.1.11, “CPU and memory resources”.

logging

The **logging** property configures the logging of the Topic Operator. For more details, see Section 2.1.10.4, "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
    # ...
```

2.1.10.3. 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 2.1.18, "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 2.1.11, "CPU and memory resources".

logging

The **logging** property configures the logging of the User Operator. For more details, see Section 2.1.10.4, "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
  # ...
```

2.1.10.4. 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 2.1.17.1, “JVM configuration”
- For more information about log levels, see Apache logging services.

2.1.10.5. 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
```

2.1.11. 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.

2.1.11.1. Resource limits and requests

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

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

2.1.11.1.1. 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
# ...
```

2.1.11.1.2. 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
# ...
```

2.1.11.1.3. 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.

2.1.11.1.4. 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.

2.1.11.2. 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 ResourceRequirements API reference.

2.1.12. Kafka loggers

Kafka has its own configurable loggers:

- **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.

Operators use the Apache **log4j2** logger implementation, so the logging configuration is described inside the ConfigMap using **log4j2.properties**. For more information, see Section 2.1.10.4, "Operator loggers".

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:
  # ...
  kafka:
    # ...
    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
  # ...
```

Changes to both external and inline logging levels will be applied to Kafka brokers without a restart.

Additional resources

- Garbage collector (GC) logging can also be enabled (or disabled). For more information on garbage collection, see Section 2.1.17.1, “JVM configuration”
- For more information about log levels, see Apache logging services.

2.1.13. 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.

2.1.13.1. 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 **topology.kubernetes.io/zone** label (or **failure-domain.beta.kubernetes.io/zone** on older OpenShift versions) that can be used as the **topologyKey** value. For more information about OpenShift node labels, see Well-Known Labels, Annotations and Taints. 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: topology.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 2.1.18, “Container images”.

2.1.14. 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.

2.1.14.1. Healthcheck configurations

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

- **Kafka.spec.kafka**
- **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.45, "**Probe** schema reference".

An example of liveness and readiness probe configuration

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

2.1.14.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **livenessProbe** or **readinessProbe** property in the **Kafka** 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
```

2.1.15. 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 in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

2.1.15.1. 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:
    # ...
```

2.1.15.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **metrics** property in the **Kafka** 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
```

2.1.16. 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.

2.1.16.1. 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.

2.1.17. 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.

2.1.17.1. JVM configuration

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.

2.1.17.2. 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
```

2.1.18. 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.

2.1.18.1. Container image configurations

Use the **image** property to specify which container image to use .

WARNING

Overriding container images is recommended only in special situations.

2.1.18.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **image** property in the **Kafka** 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
```

2.1.19. 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.

The TLS sidecar is used in:

- Entity Operator
- Cruise Control

2.1.19.1. 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 2.1.18, “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 2.1.14.1, “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:
    # ...
```

2.1.19.2. 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:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    tlsSidecar:
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
    # ...
  cruiseControl:
    # ...
    tlsSidecar:
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
    # ...
```

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

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

2.1.20. 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.

2.1.20.1. Scheduling pods based on other applications

2.1.20.1.1. 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.

2.1.20.1.2. 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 .

2.1.20.1.3. 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
```

2.1.20.2. Scheduling pods to specific nodes

2.1.20.2.1. 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.

2.1.20.2.2. 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 .

2.1.20.2.3. 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
```

2.1.20.3. Using dedicated nodes

2.1.20.3.1. 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.

2.1.20.3.2. 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 .

2.1.20.3.3. 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 .

2.1.20.3.4. 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
```

2.1.21. 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 setting up Kafka Exporter and why it is important to monitor consumer lag for performance, see Kafka Exporter in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

2.1.22. 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

See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:

- Cluster Operator
- Kafka cluster

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**.

2.1.23. 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

See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:

- Cluster Operator
- Kafka cluster

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**.

2.1.24. Scaling clusters

2.1.24.1. Scaling Kafka clusters

2.1.24.1.1. 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.

2.1.24.1.2. 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.

2.1.24.2. 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.

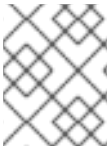
2.1.24.2.1. Reassignment JSON file

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.

2.1.24.2.2. 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" ]
}
```

2.1.24.3. 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 --bootstrap-server localhost:9092 \
  --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 --bootstrap-server localhost:9092 \
--topics-to-move-json-file /tmp/topics.json \
--broker-list 4,7 \
--generate
```

2.1.24.4. Creating reassignment JSON files manually

You can manually create the reassignment JSON file if you want to move specific partitions.

2.1.24.5. 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.

2.1.24.6. 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 --bootstrap-server localhost:9092 \
 --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 --bootstrap-server localhost:9092 \
 --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 --bootstrap-server localhost:9092 \
 --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 --bootstrap-server localhost:9092 \
 --reassignment-json-file /tmp/reassignment.json \
 --verify
```

   For example,

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
 bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 \
 --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.

2.1.24.7. 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 --bootstrap-server localhost:9092 \
  --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 --bootstrap-server localhost:9092 \
  --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 --bootstrap-server localhost:9092 \
```

```
  --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 --bootstrap-server localhost:9092 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

For example,

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  bin/kafka-reassign-partitions.sh --bootstrap-server localhost:9092 \
  --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)**.

2.1.25. 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

See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:

- Cluster Operator
- Kafka cluster

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.

2.1.26. 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

See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:

- Cluster Operator
- Kafka cluster

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.

2.1.27. 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.

2.1.27.1. 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.

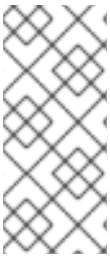
2.1.27.2. Maintenance time window definition

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 5.1.1, "Cluster Operator configuration".

2.1.27.3. 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 2.1.22, "Performing a rolling update of a Kafka cluster"
- Performing a rolling update of a ZooKeeper cluster, see Section 2.1.23, "Performing a rolling update of a ZooKeeper cluster"

2.1.28. Renewing CA certificates manually

Cluster and clients CA certificates auto-renew at the start of their respective certificate renewal periods. If **Kafka.spec.clusterCa.generateCertificateAuthority** and **Kafka.spec.clientsCa.generateCertificateAuthority** are set to **false**, the CA certificates do not auto-renew.

You can manually renew one or both of these certificates before the certificate renewal period starts. You might do this for security reasons, or if you have changed the renewal or validity periods for the certificates.

A renewed certificate uses the same private key as the old certificate.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which CA certificates and private keys are installed.

Procedure

1. Apply the **strimzi.io/force-renew** annotation to the **Secret** that contains the CA certificate that you want to renew.

   Table 2.1. Annotation for the Secret that forces renewal of certificates

Certificate	Secret	Annotate command
Cluster CA	*KAFKA-CLUSTER-NAME*-cluster-ca-cert	**oc annotate secret *KAFKA-CLUSTER-NAME*-cluster-ca-cert strimzi.io/force-renew=true**
Clients CA	*KAFKA-CLUSTER-NAME*-clients-ca-cert	**oc annotate secret *KAFKA-CLUSTER-NAME*-clients-ca-cert strimzi.io/force-renew=true**

   At the next reconciliation the Cluster Operator will generate a new CA certificate for the **Secret** that you annotated. If maintenance time windows are configured, the Cluster Operator will generate the new CA certificate at the first reconciliation within the next maintenance time window.

   Client applications must reload the cluster and clients CA certificates that were renewed by the Cluster Operator.

2. Check the period the CA certificate is valid:
   For example, using an **openssl** command:

```
oc get secret CA-CERTIFICATE-SECRET -o 'jsonpath={.data.CA-CERTIFICATE}' | base64 -d | openssl x509 -subject -issuer -startdate -enddate -noout
```

CA-CERTIFICATE-SECRET is the name of the **Secret**, which is ***KAFKA-CLUSTER-NAME*-cluster-ca-cert** for the cluster CA certificate and ***KAFKA-CLUSTER-NAME*-clients-ca-cert** for the clients CA certificate.

CA-CERTIFICATE is the name of the CA certificate, such as **jsonpath={.data.ca\.crt}**.

The command returns a **notBefore** and **notAfter** date, which is the validity period for the CA certificate.

For example, for a cluster CA certificate:

```
subject=O = io.strimzi, CN = cluster-ca v0
issuer=O = io.strimzi, CN = cluster-ca v0
notBefore=Jun 30 09:43:54 2020 GMT
notAfter=Jun 30 09:43:54 2021 GMT
```

3. Delete old certificates from the Secret.
When components are using the new certificates, older certificates might still be active. Delete the old certificates to remove any potential security risk.

Additional resources

- Section 11.2, "Secrets"
- Section 2.1.27, "Maintenance time windows for rolling updates"
- Section B.69, "**CertificateAuthority** schema reference"

2.1.29. 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.

Table 2.2. Commands for replacing private keys

Private key for	Secret	Annotate command
Cluster CA	*<cluster-name>*-cluster-ca	**oc annotate secret *<cluster-name>*-cluster-ca strimzi.io/force-replace=true**

Private key for	Secret	Annotate command
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 11.2, "Secrets"
- Section 2.1.27, "Maintenance time windows for rolling updates"

2.1.30. List of resources created as part of Kafka cluster

The following resources are created by the Cluster Operator in the OpenShift cluster:

Shared resources

***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 private key used to sign user certiticates

***cluster-name*-clients-ca-cert**

Secret with the Clients CA public key. This key can be used to verify the identity of the Kafka users.

***cluster-name*-cluster-operator-certs**

Secret with Cluster operators keys for communication with Kafka and ZooKeeper.

Zookeeper nodes

***cluster-name*-zookeeper**

StatefulSet which is in charge of managing the ZooKeeper node pods.

cluster-name*-zookeeper-*idx

Pods created by the Zookeeper StatefulSet.

***cluster-name*-zookeeper-nodes**

Headless 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 that 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**

Service account used by the Zookeeper nodes.

***cluster-name*-zookeeper**

Pod Disruption Budget configured for the ZooKeeper nodes.

***cluster-name*-network-policy-zookeeper**

Network policy managing access to the ZooKeeper services.

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.

Kafka brokers

***cluster-name*-kafka**

StatefulSet which is in charge of managing the Kafka broker pods.

cluster-name*-kafka-*idx

Pods created by the Kafka StatefulSet.

***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.

***cluster-name*-network-policy-kafka**

Network policy managing access to the Kafka services.

strimzi-*namespace-name*-*cluster-name*-kafka-init

Cluster role binding used by the Kafka brokers.

***cluster-name*-jmx**

Secret with JMX username and password used to secure the Kafka broker port. This resource will be created only when JMX is enabled in Kafka.

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.

Entity Operator

These resources are only created if the Entity Operator is deployed using the Cluster Operator.

***cluster-name*-entity-operator**

Deployment with Topic and User Operators.

cluster-name*-entity-operator-*random-string

Pod created by the Entity Operator deployment.

***cluster-name*-entity-topic-operator-config**

ConfigMap with ancillary configuration for Topic Operators.

***cluster-name*-entity-user-operator-config**

ConfigMap with ancillary configuration for User Operators.

***cluster-name*-entity-operator-certs**

Secret with Entity Operator keys for communication with Kafka and ZooKeeper.

***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.

Kafka Exporter

These resources are only created if the Kafka Exporter is deployed using the Cluster Operator.

***cluster-name*-kafka-exporter**

Deployment with Kafka Exporter.

cluster-name*-kafka-exporter-*random-string

Pod created by the Kafka Exporter deployment.

***cluster-name*-kafka-exporter**

Service used to collect consumer lag metrics.

***cluster-name*-kafka-exporter**

Service account used by the Kafka Exporter.

Cruise Control

These resources are only created only if Cruise Control was deployed using the Cluster Operator.

***cluster-name*-cruise-control**

Deployment with Cruise Control.

cluster-name*-cruise-control-*random-string

Pod created by the Cruise Control deployment.

***cluster-name*-cruise-control-config**

ConfigMap that contains the Cruise Control ancillary configuration, and is mounted as a volume by the Cruise Control pods.

***cluster-name*-cruise-control-certs**

Secret with Cruise Control keys for communication with Kafka and ZooKeeper.

***cluster-name*-cruise-control**

Service used to communicate with Cruise Control.

***cluster-name*-cruise-control**

Service account used by Cruise Control.

***cluster-name*-network-policy-cruise-control**

Network policy managing access to the Cruise Control service.

JMXTrans

These resources are only created if JMXTrans is deployed using the Cluster Operator.

***cluster-name*-jmxtrans**

Deployment with JMXTrans.

cluster-name*-jmxtrans-*random-string

Pod created by the JMXTrans deployment.

***cluster-name*-jmxtrans-config**

ConfigMap that contains the JMXTrans ancillary configuration, and is mounted as a volume by the JMXTrans pods.

***cluster-name*-jmxtrans**

Service account used by JMXTrans.

2.2. KAFKA CONNECT/S2I CLUSTER CONFIGURATION

This section describes how to configure a Kafka Connect or Kafka Connect with Source-to-Image (S2I) deployment in your AMQ Streams cluster.

Kafka Connect is an integration toolkit for streaming data between Kafka brokers and other systems using **Connector** plugins. Kafka Connect provides a framework for integrating Kafka with an external data source or target, such as a database, for import or export of data using connectors. Connectors are plugins that provide the connection configuration needed.

If you are using Kafka Connect, you configure either the **KafkaConnect** or the **KafkaConnectS2I** resource. Use the **KafkaConnectS2I** resource if you are using the Source-to-Image (S2I) framework to deploy Kafka Connect.

- The full schema of the **KafkaConnect** resource is described in Section B.79, "**KafkaConnect** schema reference".
- The full schema of the **KafkaConnectS2I** resource is described in Section B.95, "**KafkaConnectS2I** schema reference".

Additional resources

- Creating and managing connectors
- Deploying a **KafkaConnector** resource to Kafka Connect

2.2.1. Configuring Kafka Connect

Use Kafka Connect to set up external data connections to your Kafka cluster.

Use the properties of the **KafkaConnect** or **KafkaConnectS2I** resource to configure your Kafka Connect deployment. The example shown in this procedure is for the **KafkaConnect** resource, but the properties are the same for the **KafkaConnectS2I** resource.

Kafka connector configuration

KafkaConnector resources allow you to create and manage connector instances for Kafka Connect in an OpenShift-native way.

In the configuration, you enable **KafkaConnectors** for a Kafka Connect cluster by adding the **strimzi.io/use-connector-resources** annotation. You can also specify external configuration for Kafka Connect connectors through the **externalConfiguration** property.

Connectors are created, reconfigured, and deleted using the Kafka Connect HTTP REST interface, or by using **KafkaConnectors**. For more information on these methods, see Creating and managing connectors in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

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. 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.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:

- Cluster Operator
- Kafka cluster

Procedure

1. Edit the **spec** properties for the **KafkaConnect** or **KafkaConnectS2I** resource. The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect (1)
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (2)
spec:
  replicas: 3 (3)
  authentication: (4)
    type: tls
    certificateAndKey:
      certificate: source.crt
      key: source.key
      secretName: my-user-source
  bootstrapServers: my-cluster-kafka-bootstrap:9092 (5)
  tls: (6)
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        certificate: ca.crt
      - secretName: my-cluster-cluster-cert
        certificate: ca2.crt
  config: (7)
    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
  externalConfiguration: (8)
    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
resources: 9
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
logging: 10
  type: inline
  loggers:
    log4j.rootLogger: "INFO"
readinessProbe: 11
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
metrics: 12
  lowercaseOutputName: true
  lowercaseOutputLabelNames: true
  rules:
    - pattern: kafka.connect<type=connect-worker-metrics><>([a-z-]+)
      name: kafka_connect_worker_$1
      help: "Kafka Connect JMX metric worker"
      type: GAUGE
    - pattern: kafka.connect<type=connect-worker-rebalance-metrics><>([a-z-]+)
      name: kafka_connect_worker_rebalance_$1
      help: "Kafka Connect JMX metric rebalance information"
      type: GAUGE
jvmOptions: 13
  "-Xmx": "1g"
  "-Xms": "1g"
image: my-org/my-image:latest 14
template: 15
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
  connectContainer: 16
    env:
      - name: JAEGER_SERVICE_NAME
        value: my-jaeger-service
      - name: JAEGER_AGENT_HOST
        value: jaeger-agent-name
      - name: JAEGER_AGENT_PORT
        value: "6831"
```

1. Use **KafkaConnect** or **KafkaConnectS2I**, as required.
2. Enables **KafkaConnectors** for the Kafka Connect cluster.
3. The number of replica nodes.
4. Authentication for the Kafka Connect cluster, using the TLS mechanism, as shown here, using OAuth bearer tokens, or a SASL-based SCRAM-SHA-512 or PLAIN mechanism. By default, Kafka Connect connects to Kafka brokers using a plain text connection.
5. Bootstrap server for connection to the Kafka Connect cluster.
6. TLS encryption with key names under which TLS certificates are stored in X.509 format for the cluster. If certificates are stored in the same secret, it can be listed multiple times.
7. Kafka Connect configuration of *workers* (not connectors). Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by AMQ Streams.
8. External configuration for Kafka connectors using environment variables, as shown here, or volumes.
9. Requests for reservation of supported resources, currently **cpu** and **memory**, and limits to specify the maximum resources that can be consumed.
10. Specified Kafka Connect 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. For the Kafka Connect **log4j.rootLogger** logger, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
11. Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
12. Prometheus metrics, which are enabled with configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using **metrics: {}**.
13. JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka Connect.
14. ADVANCED OPTION: Container image configuration, which is recommended only in special situations.
15. Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.
16. Environment variables are also set for distributed tracing using Jaeger.

2. Create or update the resource:

```
oc apply -f KAFKA-CONNECT-CONFIG-FILE
```

3. If authorization is enabled for Kafka Connect, configure Kafka Connect users to enable access to the Kafka Connect consumer group and topics.

2.2.2. 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.

2.2.3. 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 read/write access rights to the consumer group and the internal topics of Kafka Connect.

The properties for the consumer group and internal topics are automatically configured by AMQ Streams, or they can be specified explicitly in the **spec** of the **KafkaConnect** or **KafkaConnectS2I** resource.

Example configuration properties in the KafkaConnect resource

```
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.

This procedure shows how access is provided when **simple** authorization is being used.

Simple authorization uses ACL rules, handled by the Kafka **AclAuthorizer** plugin, to provide the right level of access. For more information on configuring a **KafkaUser** resource to use simple authorization, see the **AclRule** schema reference.

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 KAFKA-USER-CONFIG-FILE
```

2.2.4. List of Kafka Connect cluster resources

The following resources are 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.

2.2.5. List of Kafka Connect (S2I) cluster resources

The following resources are 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.

2.2.6. 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.

2.3. KAFKA MIRRORMAKER CLUSTER 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

The full schema of the **KafkaMirrorMaker** resource is described in the KafkaMirrorMaker schema reference.

2.3.1. 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

- See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:
  - Cluster Operator
  - Kafka cluster
- Source and target Kafka clusters must be 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"
      ssl.endpoint.identification.algorithm: HTTPS (10)
  producer:
    bootstrapServers: my-target-cluster-kafka-bootstrap:9092
    abortOnSendFailure: false (11)
    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" (12)
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
      ssl.endpoint.identification.algorithm: HTTPS (13)
  whitelist: "my-topic|other-topic" (14)
  resources: (15)
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  logging: (16)
    type: inline
```

```
    loggers:
      mirrormaker.root.logger: "INFO"
  readinessProbe: 17
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  metrics: 18
    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: 19
    "-Xmx": "1g"
    "-Xms": "1g"
  image: my-org/my-image:latest 20
  template: 21
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: application
                    operator: In
                    values:
                      - postgresql
                      - mongodb
              topologyKey: "kubernetes.io/hostname"
    connectContainer: 22
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing: 23
    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. If certificates are stored in the same secret, it can be listed multiple

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 Hostname verification is enabled by setting to **HTTPS**. An empty string disables the verification.

11 If the **abortOnSendFailure** property is set to **true**, Kafka MirrorMaker will exit and the container will restart following a send failure for a message.

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

13 Hostname verification is enabled by setting to **HTTPS**. An empty string disables the verification.

14 A *whitelist* of topics mirrored from source to target Kafka cluster.

15 Requests for reservation of supported resources, currently **cpu** and **memory**, and limits to specify the maximum resources that can be consumed.

16 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.

17 Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).

18 Prometheus metrics, which are enabled with configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using **metrics: {}**.

19 JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka MirrorMaker.

20 ADVANCED OPTION: Container image configuration, which is recommended only in special situations.

21 Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.

22 Environment variables are also set for distributed tracing using Jaeger.

23 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>
```

2.3.2. List of Kafka MirrorMaker cluster resources

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 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.

2.4. KAFKA MIRRORMAKER 2.0 CLUSTER 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.

2.4.1. 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:

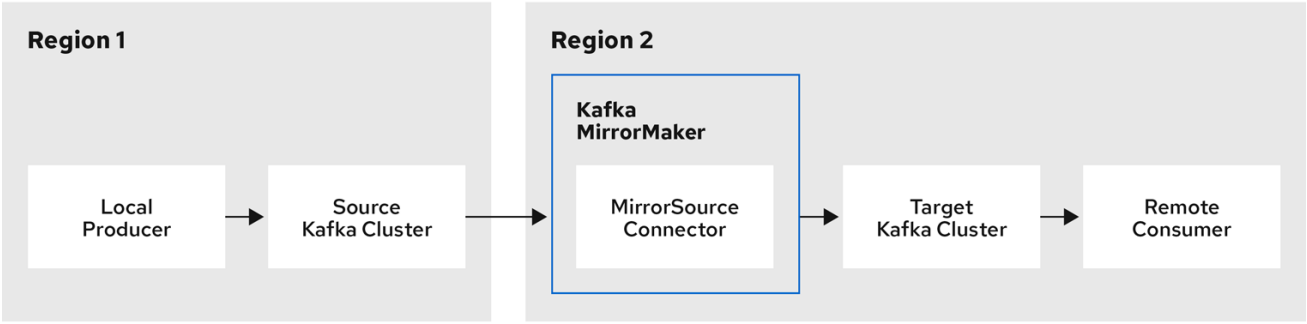
- Source cluster configuration to consume data from the source cluster
- Target cluster configuration to output data to the target cluster

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 2.1. Replication across two clusters



2.4.2. Cluster configuration

You can use MirrorMaker 2.0 in *active/passive* or *active/active* cluster configurations.

- 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.
- 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.

The expectation is that producers and consumers connect to active clusters only.

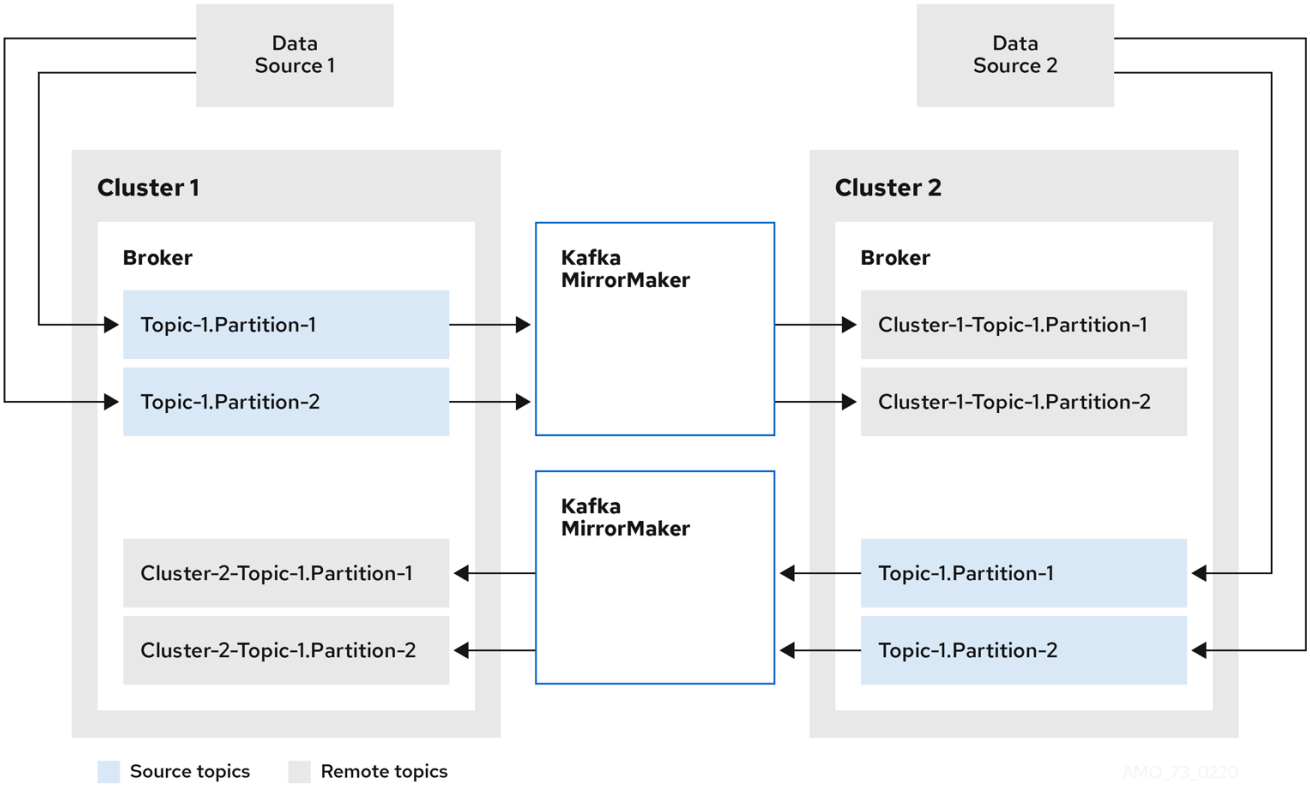
A MirrorMaker 2.0 cluster is required at each target destination.

2.4.2.1. Bidirectional replication (active/active)

The MirrorMaker 2.0 architecture supports bidirectional replication in an *active/active* cluster configuration.

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. The name of the originating cluster is prepended to the name of the topic.

Figure 2.2. 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.

2.4.2.2. Unidirectional replication (active/passive)

The MirrorMaker 2.0 architecture supports unidirectional replication in an *active/passive* cluster configuration.

You can use an *active/passive* cluster configuration to make backups or migrate data to another cluster. In this situation, you might not want automatic renaming of remote topics.

You can override automatic renaming by adding **IdentityReplicationPolicy** to the source connector configuration of the **KafkaMirrorMaker2** resource. With this configuration applied, topics retain their original names.

2.4.2.3. Topic configuration synchronization

Topic configuration is automatically synchronized between source and target clusters. By synchronizing configuration properties, the need for rebalancing is reduced.

2.4.2.4. 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.

2.4.2.5. 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.

2.4.2.6. 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.

2.4.3. ACL rules synchronization

ACL access to remote topics is possible if you are **not** using the User Operator.

If **AclAuthorizer** 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.

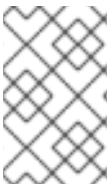
2.4.4. Synchronizing data between Kafka clusters using MirrorMaker 2.0

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: 2.6.0
  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

- See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:
  - Cluster Operator
  - Kafka cluster
- Source and target Kafka clusters must be 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: 2.6.0 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"
      ssl.endpoint.identification.algorithm: HTTPS 14
    tls: 15
      trustedCertificates:
      - certificate: ca.crt
        secretName: my-cluster-target-cluster-ca-cert
  mirrors: 16
  - sourceCluster: "my-cluster-source" 17
    targetCluster: "my-cluster-target" 18
    sourceConnector: 19
      config:
        replication.factor: 1 20
        offset-syncs.topic.replication.factor: 1 21
        sync.topic.acls.enabled: "false" 22
        replication.policy.separator: "" 23
```

```
      replication.policy.class: "io.strimzi.kafka.connect.mirror.IdentityReplicationPolicy" 24
  heartbeatConnector: 25
    config:
      heartbeats.topic.replication.factor: 1 26
  checkpointConnector: 27
    config:
      checkpoints.topic.replication.factor: 1 28
  topicsPattern: ".*" 29
  groupsPattern: "group1|group2|group3" 30
resources: 31
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
logging: 32
  type: inline
  loggers:
    connect.root.logger.level: "INFO"
readinessProbe: 33
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
jvmOptions: 34
  "-Xmx": "1g"
  "-Xms": "1g"
image: my-org/my-image:latest 35
template: 36
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
  connectContainer: 37
    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 38
externalConfiguration: 39
```

```
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. Cluster alias for Kafka Connect.
4. Specification for the Kafka clusters being synchronized.
5. 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. If certificates are stored in the same secret, it can be listed multiple times.
9. 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. Hostname verification is enabled by setting to **HTTPS**. An empty string disables the verification.
15. TLS encryption for the target Kafka cluster is configured in the same way as for the source Kafka cluster.
16. MirrorMaker 2.0 connectors.
17. Cluster alias for the source cluster used by the MirrorMaker 2.0 connectors.
18. Cluster alias for the target cluster used by the MirrorMaker 2.0 connectors.
19. Configuration for the **MirrorSourceConnector** that creates remote topics. The **config** overrides the default configuration options.

20 Replication factor for mirrored topics created at the target cluster.

21 Replication factor for the **MirrorSourceConnector offset-syncs** internal topic that maps the offsets of the source and target clusters.

22 When ACL rules synchronization is enabled, ACLs are applied to synchronized topics. The default is **true**.

23 Defines the separator used for the renaming of remote topics.

24 Adds a policy that overrides the automatic renaming of remote topics. Instead of prepending the name with the name of the source cluster, the topic retains its original name. This optional setting is useful for active/passive backups and data migration.

25 Configuration for the **MirrorHeartbeatConnector** that performs connectivity checks. The **config** overrides the default configuration options.

26 Replication factor for the heartbeat topic created at the target cluster.

27 Configuration for the **MirrorCheckpointConnector** that tracks offsets. The **config** overrides the default configuration options.

28 Replication factor for the checkpoints topic created at the target cluster.

29 Topic replication from the source cluster defined as regular expression patterns . Here we request all topics.

30 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.

31 Requests for reservation of supported resources, currently **cpu** and **memory**, and limits to specify the maximum resources that can be consumed.

32 Specified Kafka Connect 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. For the Kafka Connect **log4j.rootLogger** logger, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

33 Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).

34 JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka MirrorMaker.

35 ADVANCED OPTION: Container image configuration, which is recommended only in special situations.

36 Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.

37 Environment variables are also set for distributed tracing using Jaeger.

38 Distributed tracing is enabled for Jaeger.

39 External configuration for an OpenShift Secret mounted to Kafka MirrorMaker as an environment variable.

2. Create or update the resource:

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

2.5. KAFKA BRIDGE CLUSTER CONFIGURATION

This section describes how to configure a Kafka Bridge deployment in your AMQ Streams cluster.

Kafka Bridge provides an API for integrating HTTP-based clients with a Kafka cluster.

If you are using the Kafka Bridge, you configure the **KafkaBridge** resource.

The full schema of the **KafkaBridge** resource is described in Section B.121, "**KafkaBridge** schema reference".

2.5.1. Configuring the Kafka Bridge

Use the Kafka Bridge to make HTTP-based requests to the Kafka cluster.

Use the properties of the **KafkaBridge** resource to configure your Kafka Bridge deployment.

In order to prevent issues arising when client consumer requests are processed by different Kafka Bridge instances, address-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.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

See the *Deploying and Upgrading AMQ Streams on OpenShift* guide for instructions on running a:

- Cluster Operator
- Kafka cluster

Procedure

1. Edit the **spec** properties for the **KafkaBridge** resource.
   The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  replicas: 3 (1)
  bootstrapServers: my-cluster-kafka-bootstrap:9092 (2)
  tls: (3)
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        certificate: ca.crt
```

```
    - secretName: my-cluster-cluster-cert
      certificate: ca2.crt
authentication: 4
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: public.crt
    key: private.key
http: 5
  port: 8080
  cors: 6
    allowedOrigins: "https://strimzi.io"
    allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
consumer: 7
  config:
    auto.offset.reset: earliest
producer: 8
  config:
    delivery.timeout.ms: 300000
resources: 9
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
logging: 10
  type: inline
  loggers:
    logger.bridge.level: "INFO"
    # enabling DEBUG just for send operation
    logger.send.name: "http.openapi.operation.send"
    logger.send.level: "DEBUG"
jvmOptions: 11
  "-Xmx": "1g"
  "-Xms": "1g"
readinessProbe: 12
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
image: my-org/my-image:latest 13
template: 14
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
```

```
          topologyKey: "kubernetes.io/hostname"
  bridgeContainer: 15
    env:
      - name: JAEGER_SERVICE_NAME
        value: my-jaeger-service
      - name: JAEGER_AGENT_HOST
        value: jaeger-agent-name
      - name: JAEGER_AGENT_PORT
        value: "6831"
```

1. The number of replica nodes.
2. Bootstrap server for connection to the target Kafka cluster.
3. TLS encryption with key names under which TLS certificates are stored in X.509 format for the source Kafka cluster. If certificates are stored in the same secret, it can be listed multiple times.
4. Authentication for the Kafka Bridge cluster, using the TLS mechanism, as shown here, using OAuth bearer tokens, or a SASL-based SCRAM-SHA-512 or PLAIN mechanism. By default, the Kafka Bridge connects to Kafka brokers without authentication.
5. HTTP access to Kafka brokers.
6. CORS access specifying selected resources and access methods. Additional HTTP headers in requests describe the origins that are permitted access to the Kafka cluster.
7. Consumer configuration options.
8. Producer configuration options.
9. Requests for reservation of supported resources, currently **cpu** and **memory**, and limits to specify the maximum resources that can be consumed.
10. Specified Kafka Bridge 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. For the Kafka Bridge loggers, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
11. JVM configuration options to optimize performance for the Virtual Machine (VM) running the Kafka Bridge.
12. Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
13. ADVANCED OPTION: Container image configuration, which is recommended only in special situations.
14. Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.
15. Environment variables are also set for distributed tracing using Jaeger.

2. Create or update the resource:

```
oc apply -f KAFKA-BRIDGE-CONFIG-FILE
```

2.5.2. List of Kafka Bridge cluster resources

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.

2.6. CUSTOMIZING OPENSHIFT RESOURCES

AMQ Streams creates several OpenShift resources, such as **Deployments**, **StatefulSets**, **Pods**, and **Services**, which are managed by AMQ Streams 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 such changes using the **template** property in the AMQ Streams custom resources. The **template** property is supported in the following resources. The API reference provides more details about the customizable fields.

Kafka.spec.kafka

See Section B.53, "**KafkaClusterTemplate** schema reference"

Kafka.spec.zookeeper

See Section B.63, "**ZookeeperClusterTemplate** schema reference"

Kafka.spec.entityOperator

See Section B.68, "**EntityOperatorTemplate** schema reference"

Kafka.spec.kafkaExporter

See Section B.74, "**KafkaExporterTemplate** schema reference"

Kafka.spec.cruiseControl

See Section B.71, "**CruiseControlTemplate** schema reference"

KafkaConnect.spec

See Section B.88, "**KafkaConnectTemplate** schema reference"

KafkaConnectS2I.spec

See Section B.88, "**KafkaConnectTemplate** schema reference"

KafkaMirrorMaker.spec

See Section B.119, "**KafkaMirrorMakerTemplate** schema reference"

KafkaMirrorMaker2.spec

See Section B.88, "**KafkaConnectTemplate** schema reference"

KafkaBridge.spec

See Section B.128, "**KafkaBridgeTemplate** schema reference"

KafkaUser.spec

See Section B.112, "**KafkaUserTemplate** schema reference"

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
    # ...
```

2.6.1. 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 5.1, "Using the Cluster Operator".
- For more information about Image Pull Policies, see Disruptions.

2.7. 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.

2.7.1. 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 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 3. CONFIGURING EXTERNAL LISTENERS

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

Specify the connection **type** to expose Kafka in the external listener configuration.

- **nodeport** uses **NodePort** type **Services**
- **loadbalancer** uses **Loadbalancer** type **Services**
- **ingress** uses Kubernetes **Ingress** and the NGINX Ingress Controller for Kubernetes
- **route** uses OpenShift **Routes** and the HAProxy router

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

NOTE

route is only supported on OpenShift

Additional resources

- Accessing Apache Kafka in Strimzi

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

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: nodeport
        tls: true
```

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

2. Create or update the resource.

```
oc apply -f KAFKA-CONFIG-FILE
```

NodePort type services are created for each Kafka broker, as well as an external *bootstrap service*. The bootstrap service routes external traffic to the Kafka brokers. Node addresses used for connection are propagated to the **status** of the Kafka custom resource.

The cluster CA certificate to verify the identity of the kafka brokers is also created with the same name as the **Kafka** resource.

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

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?
(@.type=="external")].bootstrapServers}{"\n"}'
```

4. If TLS encryption is enabled, extract the public certificate of the broker certification authority.

```
oc get secret KAFKA-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.

3.2. ACCESSING KAFKA USING LOADBALANCERS

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

To connect to a broker, you need the address of the *bootstrap loadbalancer*, as well as the certificate used for TLS encryption.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

```
kafka:
  # ...
  listeners:
    - name: external
      port: 9094
      type: loadbalancer
      tls: true
      # ...
  # ...
zookeeper:
  # ...
```

2. Create or update the resource.

```
oc apply -f KAFKA-CONFIG-FILE
```

loadbalancer type services and loadbalancers are created for each Kafka broker, as well as an external *bootstrap service*. The bootstrap service routes external traffic to all Kafka brokers. DNS names and IP addresses used for connection are propagated to the **status** of each service.

The cluster CA certificate to verify the identity of the kafka brokers is also created with the same name as the **Kafka** resource.

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

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?
(@.type=="external")].bootstrapServers}{"\n"}'
```

4. If TLS encryption is enabled, extract the public certificate of the broker certification authority.

```
oc get secret KAFKA-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.

3.3. ACCESSING KAFKA USING INGRESS

This procedure shows how to access a AMQ Streams Kafka cluster from an external client outside of OpenShift using Nginx Ingress.

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

For access using Ingress, the port is always 443.

TLS passthrough

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. Ensure TLS passthrough is enabled in your NGINX Ingress Controller for Kubernetes deployment.

Because it is using the TLS passthrough functionality, TLS encryption cannot be disabled when exposing Kafka using **Ingress**.

For more information about enabling TLS passthrough, see TLS passthrough documentation.

Prerequisites

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

Procedure

1. Configure a **Kafka** resource with an external listener set to the **ingress** type. Specify the Ingress hosts for the bootstrap service and Kafka brokers.

   For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: ingress
        tls: true
        authentication:
          type: tls
        configuration: (1)
          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:
    # ...
```

   (1) Ingress hosts for the bootstrap service and Kafka brokers.

2. Create or update the resource.

```
oc apply -f KAFKA-CONFIG-FILE
```

   ClusterIP type services are created for each Kafka broker, as well as an additional *bootstrap service*. These services are used by the Ingress controller to route traffic to the Kafka brokers.

An **Ingress** resource is also created for each service to expose them using the Ingress controller. The Ingress hosts are propagated to the **status** of each service.

The cluster CA certificate to verify the identity of the kafka brokers is also created with the same name as the **Kafka** resource.

Use the address for the bootstrap host you specified in the **configuration** and port 443 (*BOOTSTRAP-HOST:443*) in your Kafka client as the *bootstrap address* to connect to the Kafka cluster.

3. Extract the public certificate of the broker certificate authority.

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

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.

3.4. ACCESSING KAFKA USING OPENSHIFT ROUTES

This procedure describes how to access a AMQ Streams Kafka cluster from an external client outside of OpenShift using routes.

To connect to a broker, you need a hostname for the route *bootstrap address*, as well as the certificate used for TLS encryption.

For access using routes, the port is always 443.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: listener1
        port: 9094
        type: route
        tls: true
        # ...
```

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

WARNING

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

2. Create or update the resource.

```
oc apply -f KAFKA-CONFIG-FILE
```

ClusterIP type services are created for each Kafka broker, as well as an external *bootstrap service*. The services route the traffic from the OpenShift Routes to the Kafka brokers. An OpenShift **Route** resource is also created for each service to expose them using the HAProxy load balancer. DNS addresses used for connection are propagated to the **status** of each service.

The cluster CA certificate to verify the identity of the kafka brokers is also created with the same name as the **Kafka** resource.

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

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?
(@.type=="external")].bootstrapServers}{"\n"}'
```

4. Extract the public certificate of the broker certification authority.

```
oc get secret KAFKA-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.

CHAPTER 4. MANAGING SECURE ACCESS TO KAFKA

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

A secure connection between Kafka brokers and clients can encompass:

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

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

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

4.1. SECURITY OPTIONS FOR KAFKA

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

4.1.1. Listener authentication

For clients inside the OpenShift cluster, you can create **plain** (without encryption) or **tls** *internal* listeners.

For clients outside the OpenShift cluster, you create *external* listeners and specify a connection mechanism, which can be **nodeport**, **loadbalancer**, **ingress** or **route** (on OpenShift).

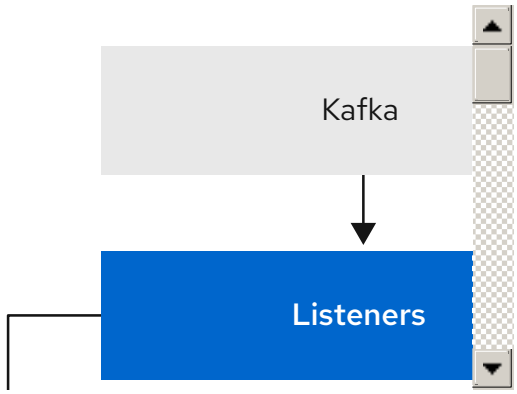
For more information on the configuration options for connecting an external client, see Configuring external listeners.

Supported authentication options:

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

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

Figure 4.1. Kafka listener authentication options



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

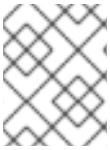
If no **authentication** property is specified then the listener does not authenticate clients which connect through that listener. The listener will accept all connections without authentication.

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

The following example shows:

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

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



NOTE

Listeners cannot be configured to use the ports set aside for interbroker communication (9091) and metrics (9404).

An example showing listener authentication configuration

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: true
    authentication:
      type: scram-sha-512
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication:
      type: tls
  - name: external
    port: 9094
    type: loadbalancer
```

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

4.1.1.1. Mutual TLS authentication

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

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

NOTE

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

4.1.1.2. SCRAM-SHA-512 authentication

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

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

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

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

When 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.

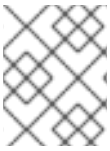
4.1.1.3. Network policies

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** property.

Use network policies as part of the listener authentication configuration. Each listener can have a different **networkPolicyPeers** configuration.

For more information, refer to the Listener network policies section and the NetworkPolicyPeer API reference.



NOTE

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

4.1.1.4. Additional listener configuration options

You can use the properties of the GenericKafkaListenerConfiguration schema to add further configuration to listeners.

4.1.2. Kafka 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 and clients have no restrictions. When enabled, authorization is applied to all enabled listeners. The authorization method is defined in the **type** field.

Supported authorization options:

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

Figure 4.2. Kafka cluster authorization options

4.1.2.1. Super users

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

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

An example configuration with super users

```
authorization:
  type: simple
  superUsers:
    - CN=client_1
    - user_2
    - CN=client_3
```

4.2. SECURITY OPTIONS FOR KAFKA CLIENTS

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

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

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

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

4.2.1. Identifying a Kafka cluster for user handling

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

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

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

4.2.2. User authentication

User 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 or SCRAM-SHA Authentication authentication for clients

4.2.2.1. 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 11, *Managing TLS certificates*.

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
```

4.2.2.2. SCRAM-SHA-512 Authentication

To use the 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= (1)
```

(1) The generated password, base64 encoded.

Decoding the generated password:

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

4.2.3. 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.

To use simple authorization, you set the **type** property to **simple** in **KafkaUser.spec.authorization**. Simple authorization uses the default Kafka authorization plugin, **AclAuthorizer**.

Alternatively, you can use OPA authorization, or if you are already using OAuth 2.0 token based authentication, you can also use OAuth 2.0 authorization.

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

4.2.3.1. ACL rules

AclAuthorizer uses ACL rules to manage access to Kafka brokers.

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

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

4.2.3.2. Super user access to Kafka brokers

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

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

4.2.3.3. 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.

4.3. SECURING ACCESS TO KAFKA BROKERS

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

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

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

- Listener authentication

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

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

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

You can replace the certificates generated by the Cluster Operator by installing your own certificates. You can also configure your listener to use a Kafka listener certificate managed by an external Certificate Authority. Certificates are available in PKCS #12 format (.p12) and PEM (.crt) formats.

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

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

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

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

Additional resources

For more information about the schema for:

- **Kafka**, see the **Kafka** schema reference.
- **KafkaUser**, see the **KafkaUser** schema reference.

4.3.1. Securing Kafka brokers

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

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

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

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

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

Procedure

1. Configure the **Kafka** resource.

   a. Configure the **authorization** property for authorization.

   b. Configure the **listeners** property to create a listener with authentication.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    authorization: 1
      type: simple
      superUsers: 2
        - CN=client_1
        - user_2
        - CN=client_3
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls 3
    # ...
  zookeeper:
    # ...
```

   1 Authorization enables **simple** authorization on the Kafka broker using the **AclAuthorizer** Kafka plugin.

   2 List of user principals with unlimited access to Kafka. *CN* is the common name from the client certificate when TLS authentication is used.

   3 Listener authentication mechanisms may be configured for each listener, and specified as mutual TLS, SCRAM-SHA-512 or token-based OAuth 2.0.

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

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

```
oc apply -f KAFKA-CONFIG-FILE
```

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

   A service is created for each Kafka broker pod.

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

   The cluster CA certificate to verify the identity of the kafka brokers is also created with the same name as the **Kafka** resource.

4.3.2. Securing user access to Kafka

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:

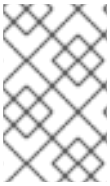
- **oc apply -f *USER-CONFIG-FILE***
- **oc delete KafkaUser *USER-NAME***

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).

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

Procedure

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

```
apiVersion: kafka.strimzi.io/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 or update the **KafkaUser** resource.

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

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

For information on configuring a Kafka client with properties for secure connection to Kafka brokers, see Setting up access for clients outside of OpenShift in the *Deploying AMQ Streams Guide*.

4.3.3. Restricting access to Kafka listeners using network policies

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

Prerequisites

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

Procedure

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

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

```
kafka:
  # ...
  listeners:
    - name: tls
      port: 9093
      type: internal
      tls: true
      authentication:
        type: tls
      networkPolicyPeers:
        - podSelector:
            matchLabels:
              app: kafka-client
  # ...
zookeeper:
  # ...
```

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

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

Additional resources

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

4.4. 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 **simple** or OPA-based Kafka authorization.

Using OAuth 2.0 token-based authentication, application clients can access resources on application servers (called *resource servers*) without exposing account credentials.

The application client passes an access token as a means of authenticating, which application servers can also use to determine the level of access to grant. The authorization server handles the granting of access and inquiries about access.

In the context of AMQ Streams:

- Kafka brokers act as OAuth 2.0 resource servers
- Kafka clients act as OAuth 2.0 application clients

Kafka clients authenticate to Kafka brokers. The brokers and clients communicate with the OAuth 2.0 authorization server, as necessary, to obtain or validate access tokens.

For a deployment of AMQ Streams, OAuth 2.0 integration provides:

- Server-side OAuth 2.0 support for Kafka brokers
- Client-side OAuth 2.0 support for Kafka MirrorMaker, Kafka Connect and the Kafka Bridge

Additional resources

- OAuth 2.0 site

4.4.1. 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.

4.4.2. OAuth 2.0 Kafka broker configuration

Kafka broker configuration for OAuth 2.0 involves:

- Creating the OAuth 2.0 client in the authorization server
- Configuring OAuth 2.0 authentication in the Kafka custom resource

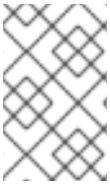
NOTE

In relation to the authorization server, Kafka brokers and Kafka clients are both regarded as OAuth 2.0 clients.

4.4.2.1. OAuth 2.0 client configuration on an authorization server

To configure a Kafka broker to validate the token received during session initiation, the recommended approach is to create an OAuth 2.0 *client* definition in an authorization server, configured as *confidential*, with the following client credentials enabled:

- Client ID of **kafka** (for example)
- Client ID and Secret as the authentication mechanism



NOTE

You only need to use a client ID and secret when using a non-public introspection endpoint of the authorization server. The credentials are not typically required when using public authorization server endpoints, as with fast local JWT token validation.

4.4.2.2. OAuth 2.0 authentication configuration in the Kafka cluster

To use OAuth 2.0 authentication in the Kafka cluster, you specify, for example, a TLS listener configuration for your Kafka cluster custom resource with the authentication method **oauth**:

Assigining the authentication method type for OAuth 2.0

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
      #...
```

You can configure **plain**, **tls** and **external** 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:
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: oauth
    #...
```

The **tls** property is *false* by default, so it must be enabled.

When you have defined the type of authentication as OAuth 2.0, you add configuration based on the type of validation, either as fast local JWT validation or token validation using an introspection endpoint.

The procedure to configure OAuth 2.0 for listeners, with descriptions and examples, is described in Configuring OAuth 2.0 support for Kafka brokers.

4.4.2.3. Fast local JWT token validation configuration

Fast local JWT token validation checks a JWT token signature locally.

The local check ensures that a token:

- Conforms to type by containing a (*typ*) claim value of **Bearer** for an access token
- Is valid (not expired)
- Has an issuer that matches a **validIssuerURI**

You specify a **validIssuerURI** attribute when you configure the listener, so that any tokens not issued by the authorization server are rejected.

The authorization server does not need to be contacted during fast local JWT token validation. You activate fast local JWT token validation by specifying a **jwksEndpointUri** attribute, the endpoint exposed by the OAuth 2.0 authorization server. The endpoint contains the public keys used to validate signed JWT tokens, which are sent as credentials by Kafka clients.

NOTE

All communication with the authorization server should be performed using TLS encryption.

You can configure a certificate truststore as an OpenShift Secret in your AMQ Streams project namespace, and use a **tlsTrustedCertificates** attribute to point to the OpenShift Secret containing the truststore file.

You might want to configure a **userNameClaim** to properly extract a username from the JWT token. If you want to use Kafka ACL authorization, you need to identify the user by their username during authentication. (The **sub** claim in JWT tokens is typically a unique ID, not a username.)

Example configuration for fast local JWT token validation

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    #...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          jwksEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-connect/certs>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
    #...
```

4.4.2.4. OAuth 2.0 introspection endpoint configuration

Token validation using an OAuth 2.0 introspection endpoint treats a received access token as opaque. The Kafka broker sends an access token to the introspection endpoint, which responds with the token information necessary for validation. Importantly, it returns up-to-date information if the specific access token is valid, and also information about when the token expires.

To configure OAuth 2.0 introspection-based validation, you specify an **introspectionEndpointUri** attribute rather than the **jwksEndpointUri** attribute specified for fast local JWT token validation. Depending on the authorization server, you typically have to specify a **clientId** and **clientSecret**, because the introspection endpoint is usually protected.

Example configuration for an introspection endpoint

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          clientId: kafka-broker
          clientSecret:
            secretName: my-cluster-oauth
            key: clientSecret
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          introspectionEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-connect/token/introspect>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
```

4.4.3. Session re-authentication for Kafka brokers

The Kafka *SASL OAUTHBEARER* mechanism, which is used for OAuth 2.0 authentication in AMQ Streams, supports a Kafka feature called the *re-authentication* mechanism.

When the re-authentication mechanism is enabled in the configuration of an **oauth** type listener, the broker's authenticated session expires when the access token expires. The client must then re-authenticate to the existing session by sending a new, valid access token to the broker, without dropping the connection.

If token validation is successful, a new client session is started using the existing connection. If the client fails to re-authenticate, the broker will close the connection if further attempts are made to send or receive messages. Java clients that use Kafka client library 2.2 or later automatically re-authenticate if the re-authentication mechanism is enabled on the broker.

You enable session re-authentication in the **Kafka** resource. Set the **maxSecondsWithoutReauthentication** property for a TLS listener with **type: oauth** authentication. Session re-authentication is supported for both types of token validation (fast local JWT and introspection endpoint). For an example configuration, see Section 4.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers".

For more information about the re-authentication mechanism, which was added in Kafka version 2.2, see KIP-368.

Additional resources

- Section 4.4.2, "OAuth 2.0 Kafka broker configuration"
- Section 4.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers"

- KafkaListenerAuthenticationOAuth schema reference

4.4.4. OAuth 2.0 Kafka client configuration

A Kafka client is configured with either:

- The credentials required to obtain a valid access token from an authorization server (client ID and Secret)
- A valid long-lived access token or refresh token, obtained using tools provided by an authorization server

The only information ever sent to the Kafka broker is an access token. The credentials used to authenticate with the authorization server to obtain the access token are never sent to the broker.

When a client obtains an access token, no further communication with the authorization server is needed.

The simplest mechanism is authentication with a client ID and Secret. Using a long-lived access token, or a long-lived refresh token, adds more complexity because there is an additional dependency on authorization server tools.

NOTE

If you are using long-lived access tokens, you may need to configure the client in the authorization server to increase the maximum lifetime of the token.

If the Kafka client is not configured with an access token directly, the client exchanges credentials for an access token during Kafka session initiation by contacting the authorization server. The Kafka client exchanges either:

- Client ID and Secret
- Client ID, refresh token, and (optionally) a Secret

4.4.5. 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:

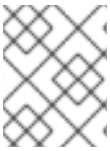
- Fast local token validation based on JWT signature checking and local token introspection, without contacting the authorization server
- An OAuth 2.0 introspection endpoint provided by the authorization server

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

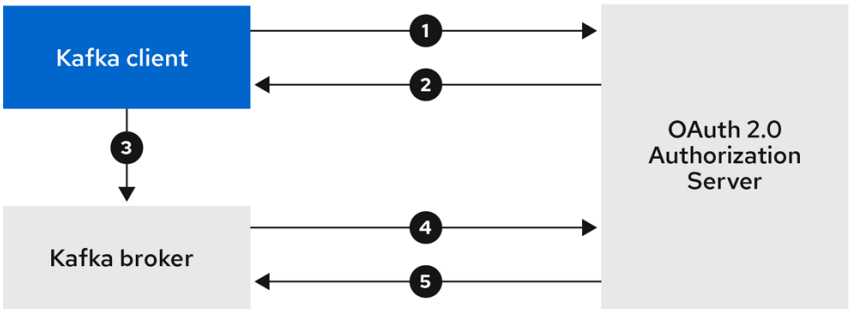
An authorization server might only allow the use of opaque access tokens, which means that local token validation is not possible.

4.4.5.1. Example client authentication flows

Here you can see the communication flows, for different configurations of Kafka clients and brokers, during Kafka session authentication.

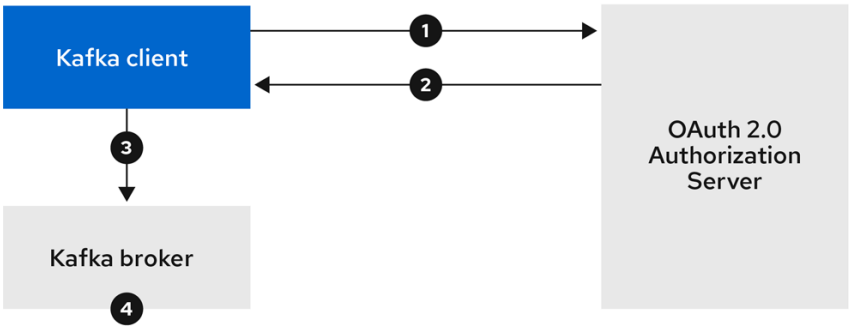
- Client using client ID and secret, with broker delegating validation to authorization server
- Client using client ID and secret, with broker performing fast local token validation
- Client using long-lived access token, with broker delegating validation to authorization server
- Client using long-lived access token, with broker performing fast local validation

Client using client ID and secret, with broker delegating validation to authorization server



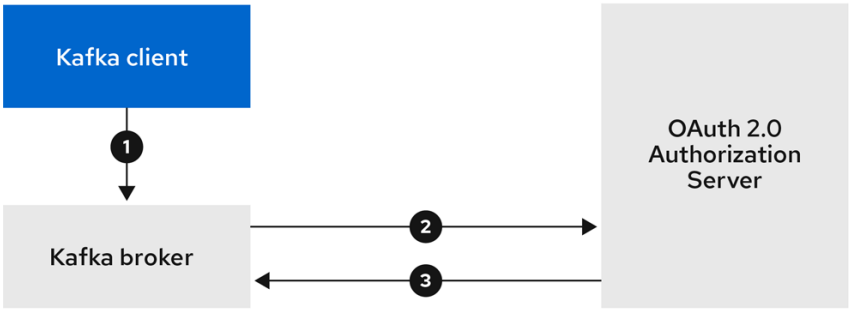
1. Kafka client requests access token from authorization server, using client ID and secret, and optionally a refresh token.
2. Authorization server generates a new access token.
3. Kafka client authenticates with the Kafka broker using the *SASL OAUTHBEARER* mechanism to pass the access token.
4. Kafka 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.

4.4.6. Configuring OAuth 2.0 authentication

OAuth 2.0 is used for interaction between Kafka clients and AMQ Streams components.

In order to use OAuth 2.0 for AMQ Streams, you must:

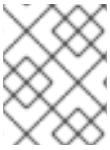
1. Deploy an authorization server and configure the deployment to integrate with AMQ Streams
2. Deploy or update the Kafka cluster with Kafka broker listeners configured to use OAuth 2.0
3. Update your Java-based Kafka clients to use OAuth 2.0
4. Update Kafka component clients to use OAuth 2.0

4.4.6.1. 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 .

4.4.6.2. Configuring OAuth 2.0 support for Kafka brokers

This procedure describes how to configure Kafka brokers so that the broker listeners are enabled to use OAuth 2.0 authentication using an authorization server.

We advise use of OAuth 2.0 over an encrypted interface through 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 may need to configure the truststore with prober certificates or disable the certificate hostname validation.

When configuring the Kafka broker you have two options for the mechanism used to validate the access token during OAuth 2.0 authentication of the newly connected Kafka client:

- Configuring fast local JWT token validation
- Configuring token validation using an introspection endpoint

Before you start

For more information on the configuration of OAuth 2.0 authentication for Kafka broker listeners, see:

- KafkaListenerAuthenticationOAuth schema reference
- Managing access to Kafka

Prerequisites

- AMQ Streams and Kafka are running
- An OAuth 2.0 authorization server is deployed

Procedure

1. Update the Kafka broker configuration (**Kafka.spec.kafka**) of your **Kafka** resource in an editor.

```
oc edit kafka my-cluster
```

2. Configure the Kafka broker **listeners** configuration.
The configuration for each type of listener does not have to be the same, as they are independent.

   The examples here show the configuration options as configured for external listeners.

   Example 1: Configuring fast local JWT token validation

```
#...
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth 1
    validIssuerUri: <https://<auth-server-address>/auth/realms/external> 2
    jwksEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-connect/certs> 3
```

```
      userNameClaim: preferred_username 4
      maxSecondsWithoutReauthentication: 3600 5
      tlsTrustedCertificates: 6
      - secretName: oauth-server-cert
        certificate: ca.crt
      disableTlsHostnameVerification: true 7
      jwksExpirySeconds: 360 8
      jwksRefreshSeconds: 300 9
      jwksMinRefreshPauseSeconds: 1 10
      enableECDSA: "true" 11
```

1. Listener type set to **oauth**.
2. URI of the token issuer used for authentication.
3. URI of the JWKS certificate endpoint used for local JWT validation.
4. The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The **userNameClaim** value will depend on the authentication flow and the authorization server used.
5. (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.
6. (Optional) Trusted certificates for TLS connection to the authorization server.
7. (Optional) Disable TLS hostname verification. Default is **false**.
8. The duration the JWKS certificates are considered valid before they expire. Default is **360** seconds. If you specify a longer time, consider the risk of allowing access to revoked certificates.
9. The period between refreshes of JWKS certificates. The interval must be at least 60 seconds shorter than the expiry interval. Default is **300** seconds.
10. The minimum pause in seconds between consecutive attempts to refresh JWKS public keys. When an unknown signing key is encountered, the JWKS keys refresh is scheduled outside the regular periodic schedule with at least the specified pause since the last refresh attempt. The refreshing of keys follows the rule of exponential backoff, retrying on unsuccessful refreshes with ever increasing pause, until it reaches **jwksRefreshSeconds**. The default value is 1.
11. (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 library. Default is **false**.

Example 2: Configuring token validation using an introspection endpoint

```
- name: external
  port: 9094
  type: loadbalancer
```

```
tls: true
authentication:
  type: oauth
  validIssuerUri: <https://<auth-server-address>/auth/realms/external>
  introspectionEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-connect/token/introspect> (1)
  clientId: kafka-broker (2)
  clientSecret: (3)
    secretName: my-cluster-oauth
    key: clientSecret
  userNameClaim: preferred_username (4)
  maxSecondsWithoutReauthentication: 3600 (5)
```

1. URI of the token introspection endpoint.
2. Client ID to identify the client.
3. Client Secret and client ID is used for authentication.
4. The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The **userNameClaim** value will depend on the authorization server used.
5. (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.

Depending on how you apply OAuth 2.0 authentication, and the type of authorization server, there are additional (optional) configuration settings you can use:

```
# ...
authentication:
  type: oauth
  # ...
  checkIssuer: false (1)
  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

4.4.6.3. 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>0.6.1.redhat-00003</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

4.4.6.4. Configuring OAuth 2.0 for Kafka components

This procedure describes how to configure Kafka components to use OAuth 2.0 authentication using an authorization server.

You can configure authentication for:

- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

In this scenario, the Kafka component and the authorization server are running in the same cluster.

Before you start

For more information on the configuration of OAuth 2.0 authentication for Kafka components, see:

- 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.

4.5. USING OAUTH 2.0 TOKEN-BASED AUTHORIZATION

If you are using OAuth 2.0 with Red Hat Single Sign-On for token-based authentication, you can also use Red Hat Single Sign-On to configure authorization rules to constrain client access to Kafka brokers. Authentication establishes the identity of a user. Authorization decides the level of access for that user.

AMQ Streams supports the use of OAuth 2.0 token-based authorization through Red Hat Single Sign-On Authorization Services, which allows you to manage security policies and permissions centrally.

Security policies and permissions defined in Red Hat Single Sign-On are used to grant access to resources on Kafka brokers. Users and clients are matched against policies that permit access to perform specific actions on Kafka brokers.

Kafka allows all users full access to brokers by default, and also provides the **AclAuthorizer** plugin to configure authorization based on Access Control Lists (ACLs).

ZooKeeper stores ACL rules that grant or deny access to resources based on *username*. However, OAuth 2.0 token-based authorization with Red Hat Single Sign-On offers far greater flexibility on how you wish to implement access control to Kafka brokers. In addition, you can configure your Kafka brokers to use OAuth 2.0 authorization and ACLs.

Additional resources

- Using OAuth 2.0 token-based authentication
- Kafka Authorization
- Red Hat Single Sign-On documentation

4.5.1. OAuth 2.0 authorization mechanism

OAuth 2.0 authorization in AMQ Streams uses Red Hat Single Sign-On server Authorization Services REST endpoints to extend token-based authentication with Red Hat Single Sign-On by applying defined security policies on a particular user, and providing a list of permissions granted on different resources for that user. Policies use roles and groups to match permissions to users. OAuth 2.0 authorization enforces permissions locally based on the received list of grants for the user from Red Hat Single Sign-On Authorization Services.

4.5.1.1. Kafka broker custom authorizer

A Red Hat Single Sign-On *authorizer* (**KeycloakRBACAuthorizer**) is provided with AMQ Streams. To be able to use the Red Hat Single Sign-On REST endpoints for Authorization Services provided by Red Hat Single Sign-On, you configure a custom authorizer on the Kafka broker.

The authorizer fetches a list of granted permissions from the authorization server as needed, and enforces authorization locally on the Kafka Broker, making rapid authorization decisions for each client request.

4.5.2. Configuring OAuth 2.0 authorization support

This procedure describes how to configure Kafka brokers to use OAuth 2.0 authorization using Red Hat Single Sign-On Authorization Services.

Before you begin

Consider the access you require or want to limit for certain users. You can use a combination of Red Hat Single Sign-On *groups*, *roles*, *clients*, and *users* to configure access in Red Hat Single Sign-On.

Typically, groups are used to match users based on organizational departments or geographical locations. And roles are used to match users based on their function.

With Red Hat Single Sign-On, you can store users and groups in LDAP, whereas clients and roles cannot be stored this way. Storage and access to user data may be a factor in how you choose to configure authorization policies.

NOTE

Super users always have unconstrained access to a Kafka broker regardless of the authorization implemented on the Kafka broker.

Prerequisites

- AMQ Streams must be configured to use OAuth 2.0 with Red Hat Single Sign-On for token-based authentication. You use the same Red Hat Single Sign-On server endpoint when you set up authorization.
- OAuth 2.0 authentication must be configured with the **maxSecondsWithoutReauthentication** option to enable re-authentication.
- 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
    grantsRefreshPeriodSeconds: 60 8
    grantsRefreshPoolSize: 5 9
  #...
```

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 **AclAuthorizer** if access is denied by Red Hat Single Sign-On Authorization Services policies. Default is **false**.
5. (Optional) Disable TLS hostname verification. Default is **false**.
6. (Optional) Designated super users.
7. (Optional) Trusted certificates for TLS connection to the authorization server.
8. (Optional) The time between two consecutive grants refresh runs. That is the maximum time for active sessions to detect any permissions changes for the user on Red Hat Single Sign-On. The default value is 60.
9. (Optional) The number of threads to use to refresh (in parallel) the grants for the active sessions. The default value is 5.

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.

CHAPTER 5. USING AMQ STREAMS OPERATORS

Use the AMQ Streams operators to manage your Kafka cluster, and Kafka topics and users.

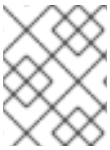
5.1. USING THE CLUSTER OPERATOR

The Cluster Operator is used 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 Deploying the Cluster Operator in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

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.

5.1.1. Cluster Operator configuration

The Cluster Operator can be configured through the following supported environment variables and through the logging configuration.

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_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 **2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.6.7, 2.6.0=registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.7**. This is used when a **Kafka.spec.kafka.version** property is specified but not the **Kafka.spec.kafka.image**, as described in Section 2.1.18, “Container images”.

STRIMZI_DEFAULT_KAFKA_INIT_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.6.7**. 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 2.1.18, "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 **2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.6.7, 2.6.0=registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.7**. This is used when a **KafkaConnect.spec.version** property is specified but not the **KafkaConnect.spec.image**, as described in Section B.1.6, "**image**".

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 **2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.6.7, 2.6.0=registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.7**. This is used when a **KafkaConnectS2I.spec.version** property is specified but not the **KafkaConnectS2I.spec.image**, as described in Section B.1.6, "**image**".

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 **2.5.0=registry.redhat.io/amq7/amq-streams-kafka-25-rhel7:1.6.7, 2.6.0=registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.7**. This is used when a **KafkaMirrorMaker.spec.version** property is specified but not the **KafkaMirrorMaker.spec.image**, as described in Section B.1.6, "**image**".

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.6.7**. 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 2.1.18, "Container images" of the **Kafka** resource.

STRIMZI_DEFAULT_USER_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.6.7**. 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 2.1.18, "Container images" of the **Kafka** resource.

STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-kafka-26-rhel7:1.6.7**. 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 2.1.18, "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.

Configuration by ConfigMap

The Cluster Operator's logging is configured by the **strimzi-cluster-operator ConfigMap**.

A **ConfigMap** containing logging configuration is created when installing the Cluster Operator. This **ConfigMap** is described in the file **install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml**. You configure Cluster Operator logging by changing the data field **log4j2.properties** in this **ConfigMap**.

To update the logging configuration, you can edit the **050-ConfigMap-strimzi-cluster-operator.yaml** file and then run the following command:

```
oc apply -f install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml
```

Alternatively, edit the **ConfigMap** directly:

```
oc edit cm strimzi-cluster-operator
```

To change the frequency of the reload interval, set a time in seconds in the **monitorInterval** option in the created **ConfigMap**.

If the **ConfigMap** is missing when the Cluster Operator is deployed, the default logging values are used.

If the **ConfigMap** is accidentally deleted after the Cluster Operator is deployed, the most recently loaded logging configuration is used. Create a new **ConfigMap** to load a new logging configuration.

NOTE

Do not remove the monitorInterval option from the ConfigMap.

5.1.1.1. Periodic reconciliation

Although the Cluster 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.

5.1.2. Provisioning Role-Based Access Control (RBAC)

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** and **Services**. 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.

5.1.2.1. 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 **KafkaMirrorMaker2** use a **ServiceAccount** called ***cluster-name*-mirrormaker2**
- The pods for **KafkaBridge** use a **ServiceAccount** called ***cluster-name*-bridge**

5.1.2.2. 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 **strimzi-cluster-operator ServiceAccount** is specified as the **serviceAccountName**.

5.1.2.3. 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
```

The **strimzi-kafka-client ClusterRole** represents the access needed by the components based on Kafka clients which use the client rack-awareness. 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 client based pods

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-kafka-client
  labels:
    app: strimzi
rules:
  - apiGroups:
      - ""
    resources:
      # The Kafka clients (Connect, Mirror Maker, etc.) require "get" permissions to view the node they
are on
      # This information is used to generate a Rack ID (client.rack option) that is used for consuming
from the closest
      # replicas when enabled
      - nodes
    verbs:
      - get
```

5.1.2.4. 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:

Example ClusterRoleBinding for the Cluster Operator for the Kafka broker rack-awarness

```
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
```

and

Example ClusterRoleBinding for the Cluster Operator for the Kafka client rack-awarness

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator-kafka-client-delegation
  labels:
    app: strimzi
# The Kafka clients cluster role must be bound to the cluster operator service account so that it can delegate the
# cluster role to the Kafka clients using it for consuming from closest replica.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-client
  apiGroup: rbac.authorization.k8s.io
```

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
```

5.2. 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.

The *Deploying and Upgrading AMQ Streams on OpenShift* guide provides instructions to deploy the Topic Operator:

- Using the Cluster Operator (recommended)
- Standalone to operate with Kafka clusters not managed by AMQ Streams

5.2.1. 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.

5.2.1.1. 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.

For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
```

```
name: topic-name-1
labels:
  strimzi.io/cluster: my-cluster
```

The label is used by the Topic Operator to identify the **KafkaTopic** resource and create a new topic, and also in subsequent handling of the topic.

If the label does not match the Kafka cluster, the Topic Operator cannot identify the **KafkaTopic** and the topic is not created.

5.2.1.2. Handling changes to topics

A fundamental problem that the Topic Operator has to solve is that there is no single source of truth: Both the **KafkaTopic** resource and the Kafka topic can be modified independently of the 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.

5.2.1.3. 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.

5.2.1.4. Kafka topic naming conventions

Kafka and OpenShift impose their own validation rules for the naming of topics in Kafka and **KafkaTopic.metadata.name** respectively. There are valid names for each which are invalid in the other.

Using the **spec.topicName** property, it is possible to create a valid topic in Kafka with a name that would be invalid for the Kafka topic in OpenShift.

The **spec.topicName** property inherits Kafka naming validation rules:

- The name must not be longer than 249 characters.
- Valid characters for Kafka topics are ASCII alphanumerics, **.**, **_**, and **-**.
- The name cannot be **.** or **..**, though **.** can be used in a name, such as **exampleTopic.** or **.exampleTopic**.

spec.topicName must not be changed.

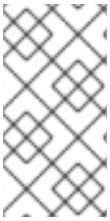
For example:

```
apiVersion: {KafkaApiVersion}
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: topicName-1 (1)
  # ...
```

(1) Upper case is invalid in OpenShift.

cannot be changed to:

```
apiVersion: {KafkaApiVersion}
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.

5.2.2. 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
   ```

5.2.3. Configuring the Topic Operator with resource requests and limits

You can allocate resources, such as CPU and memory, to the Topic Operator and set a limit on the amount of resources it can consume.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Update the Kafka cluster configuration in an editor, as required:

```
oc edit kafka MY-CLUSTER
```

2. In the **spec.entityOperator.topicOperator.resources** property in the **Kafka** resource, set the resource requests and limits for the Topic Operator.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # Kafka and ZooKeeper sections...
  entityOperator:
    topicOperator:
      resources:
        requests:
          cpu: "1"
          memory: 500Mi
        limits:
          cpu: "1"
          memory: 500Mi
```

3. Apply the new configuration to create or update the resource.

```
oc apply -f KAFKA-CONFIG-FILE
```

5.3. 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.

The *Deploying and Upgrading AMQ Streams on OpenShift* guide provides instructions to deploy the User Operator:

- Using the Cluster Operator (recommended)
- Standalone to operate with Kafka clusters not managed by AMQ Streams

For more information about the schema, see **KafkaUser** schema reference.

Authenticating and authorizing access to Kafka

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

For more information on using **KafkUser** to manage users and secure access to Kafka brokers, see Securing access to Kafka brokers .

5.3.1. 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:
        requests:
          cpu: "1"
          memory: 500Mi
        limits:
          cpu: "1"
          memory: 500Mi
```

Save the file and exit the editor. The Cluster Operator applies the changes automatically.

5.4. MONITORING OPERATORS USING 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 in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

CHAPTER 6. 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 6.2, "Kafka Bridge quickstart" later in this chapter.
- For detailed configuration steps, see Section 2.5, "Kafka Bridge cluster configuration" .
- To view the API documentation, see the Kafka Bridge API reference .

6.1. KAFKA BRIDGE OVERVIEW

You can use the AMQ Streams Kafka Bridge as an interface to make specific types of HTTP requests to the Kafka cluster.

6.1.1. Kafka Bridge interface

The Kafka Bridge provides a RESTful interface that allows HTTP-based clients to interact with a Kafka cluster. It 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.

6.1.1.1. HTTP requests

The Kafka Bridge supports HTTP requests to a Kafka cluster, with methods to:

- Send messages to a topic.
- Retrieve messages from topics.
- Retrieve a list of partitions for a topic.
- 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.

6.1.2. 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.

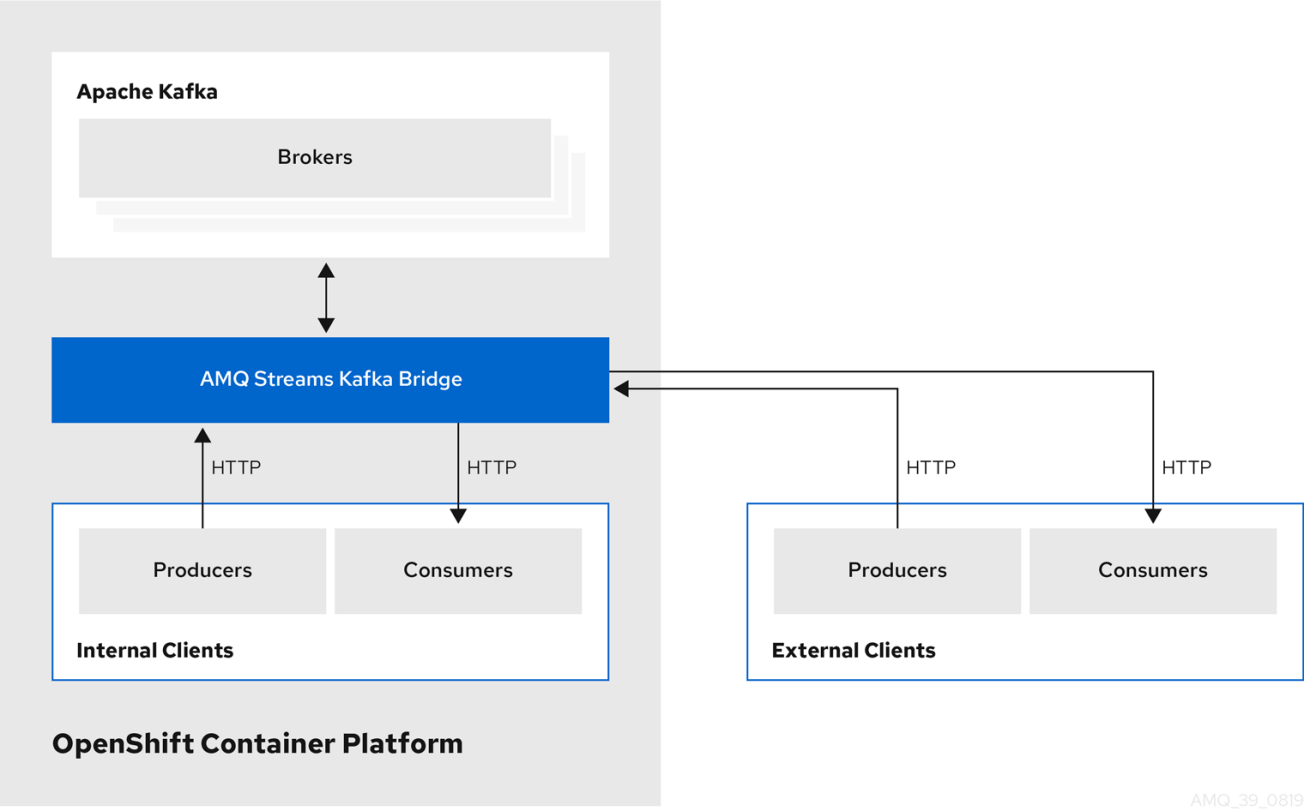
Internal clients

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



6.1.3. 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 2.5.1, “Configuring the Kafka Bridge” .

You can use ACLs in Kafka brokers to restrict the topics that can be consumed and produced using the Kafka Bridge.

6.1.4. 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.

6.1.5. Requests to the Kafka Bridge

Specify data formats and HTTP headers to ensure valid requests are submitted to the Kafka Bridge.

6.1.5.1. 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 *embedded data format* of the messages produced. This can be either **json** or **binary**.

Embedded data format	Content-Type header
JSON	**Content-Type: application/vnd.kafka.json.v2+json**
Binary	**Content-Type: application/vnd.kafka.binary.v2+json**

The embedded data format is set per consumer, as described in 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.

6.1.5.2. 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**.

6.1.5.3. Message format

When sending messages using the **/topics** endpoint, you enter the message payload in the request body, in the **records** parameter.

The **records** parameter can contain any of these optional fields:

- Message **headers**
- Message **key**
- Message **value**
- Destination **partition**

Example POST request to /topics

```
curl -X POST \
  http://localhost:8080/topics/my-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
  -d '{
    "records": [
        {
            "key": "my-key",
            "value": "sales-lead-0001"
            "partition": 2
            "headers": [
              {
                "key": "key1",
                "value": "QXBhY2hlIEthZmthIGlzIHRoZSBib21iIQ==" 1
              }
            ]
        },
    ]
}'
```

1 The header value in binary format and encoded as Base64.

6.1.5.4. 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
```

6.1.6. CORS

Cross-Origin Resource Sharing (CORS) allows you to specify allowed methods and originating URLs for accessing the Kafka cluster in your Kafka Bridge HTTP configuration .

Example CORS configuration for Kafka Bridge

```
# ...
cors:
  allowedOrigins: "https://strimzi.io"
  allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  # ...
```

CORS allows for *simple* and *preflighted* requests between origin sources on different domains.

Simple requests are suitable for standard requests using **GET**, **HEAD**, **POST** methods.

A preflighted request sends a *HTTP OPTIONS* request as an initial check that the actual request is safe to send. On confirmation, the actual request is sent. Preflight requests are suitable for methods that require greater safeguards, such as **PUT** and **DELETE**, and use non-standard headers.

All requests require an **Origin** value in their header, which is the source of the HTTP request.

6.1.6.1. Simple request

For example, this simple request header specifies the origin as **https://strimzi.io**.

```
Origin: https://strimzi.io
```

The header information is added to the request.

```
curl -v -X GET HTTP-ADDRESS/bridge-consumer/records \
-H 'Origin: https://strimzi.io'\
-H 'content-type: application/vnd.kafka.v2+json'
```

In the response from the Kafka Bridge, an **Access-Control-Allow-Origin** header is returned.

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: * (1)
```

(1) Returning an asterisk (*) shows the resource can be accessed by any domain.

6.1.6.2. Preflighted request

An initial preflight request is sent to Kafka Bridge using an **OPTIONS** method. The *HTTP OPTIONS* request sends header information to check that Kafka Bridge will allow the actual request.

Here the preflight request checks that a **POST** request is valid from **https://strimzi.io**.

```
OPTIONS /my-group/instances/my-user/subscription HTTP/1.1
```

```
Origin: https://strimzi.io
Access-Control-Request-Method: POST (1)
Access-Control-Request-Headers: Content-Type (2)
```

1. Kafka Bridge is alerted that the actual request is a **POST** request.
2. The actual request will be sent with a **Content-Type** header.

OPTIONS is added to the header information of the preflight request.

```
curl -v -X OPTIONS -H 'Origin: https://strimzi.io' \
-H 'Access-Control-Request-Method: POST' \
-H 'content-type: application/vnd.kafka.v2+json'
```

Kafka Bridge responds to the initial request to confirm that the request will be accepted. The response header returns allowed origins, methods and headers.

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://strimzi.io
Access-Control-Allow-Methods: GET,POST,PUT,DELETE,OPTIONS,PATCH
Access-Control-Allow-Headers: content-type
```

If the origin or method is rejected, an error message is returned.

The actual request does not require **Access-Control-Request-Method** header, as it was confirmed in the preflight request, but it does require the origin header.

```
curl -v -X POST HTTP-ADDRESS/topics/bridge-topic \
-H 'Origin: https://strimzi.io' \
-H 'content-type: application/vnd.kafka.v2+json'
```

The response shows the originating URL is allowed.

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://strimzi.io
```

Additional resources

Fetch CORS specification

6.1.7. 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 .

6.1.8. 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 Deploying Kafka Bridge to your OpenShift cluster in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.
- For detailed information on configuring the Kafka Bridge, see Section 2.5, "Kafka Bridge cluster configuration"
- For information on configuring the host and port for the **KafkaBridge** resource, see Section 2.5.1, "Configuring the Kafka Bridge".
- For information on integrating external clients, see Section 6.1.4, "Accessing the Kafka Bridge outside of OpenShift".

6.2. 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 6.1.5, "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.

6.2.1. 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 2.5, "Kafka Bridge cluster configuration".

6.2.2. 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

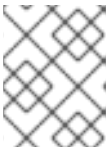
1. List the names of the pods in your OpenShift cluster:

```
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.

6.2.3. 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.

6.2.4. 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.

6.2.5. 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.

6.2.6. 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.

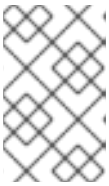
2. Submit a **GET** request to the **records** endpoint:

```
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.

6.2.7. 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.

6.2.8. 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.

6.2.9. 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 7. USING THE KAFKA BRIDGE WITH 3SCALE

You can deploy and integrate Red Hat 3scale API Management with the AMQ Streams Kafka Bridge.

7.1. 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 7.2, “Deploying 3scale for the Kafka Bridge” .

7.1.1. 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.

7.1.2. 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.

7.1.3. 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.

7.1.4. 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

7.2. 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 8. CRUISE CONTROL FOR CLUSTER REBALANCING

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/**.

8.1. 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

8.2. 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. AMQ Streams supports most of the optimization goals developed in the Cruise Control project. The supported goals, in the default descending order of priority, are as follows:

1. Rack-awareness
2. Replica capacity
3. *Capacity*: Disk capacity, Network inbound capacity, Network outbound capacity, CPU capacity

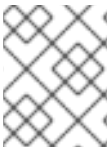
4. Replica distribution
5. Potential network output
6. *Resource distribution*: Disk utilization distribution, Network inbound utilization distribution, Network outbound utilization distribution, CPU utilization distribution

NOTE

The resource distribution goals are controlled using capacity limits on broker resources.

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

Intra-broker disk goals, "Write your own" goals, and Kafka assigner goals are not yet supported.

Goals configuration in AMQ Streams custom resources

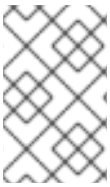
You configure optimization goals in **Kafka** and **KafkaRebalance** custom resources. Cruise Control has configurations for hard optimization goals that must be satisfied, as well as master, default, and user-provided optimization goals. Optimization goals for resource distribution (disk, network inbound, network outbound, and CPU) are subject to 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; CpuCapacityGoal
```

You configure hard goals in the Cruise Control deployment configuration, by editing the **hard.goals** property in **Kafka.spec.cruiseControl.config**.

- To inherit the preset hard goals from Cruise Control, do not specify the **hard.goals** property in **Kafka.spec.cruiseControl.config**
- To change the preset hard goals, specify the desired goals in the **hard.goals** property, using their fully-qualified domain names.

Example Kafka configuration for hard optimization goals

```
apiVersion: kafka.strimzi.io/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; CpuCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal;
DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal;
```

```
NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal;
LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal
```

Six 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 8.3, "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 a **KafkaRebalance** custom resource:

```
KafkaRebalance.spec.goals
```

User-provided optimization goals can generate optimization proposals for different scenarios. For example, you might want to optimize leader replica distribution across the Kafka cluster without considering disk capacity or disk utilization. So, you create a **KafkaRebalance** custom resource containing a single user-provided goal for leader replica distribution.

User-provided optimization goals must:

- Include all configured hard goals, or an error occurs
- Be a subset of the master optimization goals

To ignore the configured hard goals when generating an optimization proposal, add the **skipHardGoalCheck: true** property to the **KafkaRebalance** custom resource. See Section 8.7, "Generating optimization proposals".

Additional resources

- Section 8.5, "Cruise Control configuration"
- Configurations in the Cruise Control Wiki.

8.3. 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 describes the properties contained in an optimization proposal:

Table 8.1. 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.
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.

JSON property	Description
intraBrokerDataToMoveMB	The sum of the size of each partition replica that will be moved between disks on the same broker (see also **numIntraBrokerReplicaMovements**). **Performance impact during rebalance operation**: Variable. The larger the number, the longer the cluster rebalance will take to complete. Moving a large amount of data between disks on the same broker has less impact than between separate brokers (see **dataToMoveMB**).
recentWindows	The number of metrics windows upon which the optimization proposal is based.
dataToMoveMB	The sum of the size of each partition replica that will be moved to a separate broker (see also **numReplicaMovements**). **Performance impact during rebalance operation**: Variable. The larger the number, the longer the cluster rebalance will take to complete.
monitoredPartitionsPercentage	The percentage of partitions in the Kafka cluster covered by the optimization proposal. Affected by the number of **excludedTopics**.
excludedTopics	If you specified a regular expression in the **spec.excludedTopicsRegex** property in the **KafkaRebalance** resource, all topic names matching that expression are listed here. These topics are excluded from the calculation of partition replica/leader movements in the optimization proposal.
numLeaderMovements	The number of partitions whose leaders will be switched to different replicas. This involves a change to ZooKeeper configuration. **Performance impact during rebalance operation**: Relatively low.
excludedBrokersForReplicaMove	Not yet supported. An empty list is returned.

Additional resources

- Section 8.2, "Optimization goals overview"
- Section 8.7, "Generating optimization proposals"
- Section 8.8, "Approving an optimization proposal"

8.4. REBALANCE PERFORMANCE TUNING OVERVIEW

You can adjust several performance tuning options for cluster rebalances. These options control how partition replica and leadership movements in a rebalance are executed, as well as the bandwidth that is allocated to a rebalance operation.

Partition reassignment commands

Optimization proposals are comprised of separate partition reassignment commands. When you approve a proposal, the Cruise Control server applies these commands to the Kafka cluster.

A partition reassignment command consists of either of the following types of operations:

- Partition movement: Involves transferring the partition replica and its data to a new location. Partition movements can take one of two forms:
  - Inter-broker movement: The partition replica is moved to a log directory on a different broker.
  - Intra-broker movement: The partition replica is moved to a different log directory on the same broker.
- Leadership movement: This involves switching the leader of the partition's replicas.

Cruise Control issues partition reassignment commands to the Kafka cluster in batches. The performance of the cluster during the rebalance is affected by the number of each type of movement contained in each batch.

Replica movement strategies

Cluster rebalance performance is also influenced by the *replica movement strategy* that is applied to the batches of partition reassignment commands. By default, Cruise Control uses the **BaseReplicaMovementStrategy**, which simply applies the commands in the order they were generated. However, if there are some very large partition reassignments early in the proposal, this strategy can slow down the application of the other reassignments.

Cruise Control provides three alternative replica movement strategies that can be applied to optimization proposals:

- **PrioritizeSmallReplicaMovementStrategy**: Order reassignments in order of ascending size.
- **PrioritizeLargeReplicaMovementStrategy**: Order reassignments in order of descending size.
- **PostponeUrpReplicaMovementStrategy**: Prioritize reassignments for replicas of partitions which have no out-of-sync replicas.

These strategies can be configured as a sequence. The first strategy attempts to compare two partition reassignments using its internal logic. If the reassignments are equivalent, then it passes them to the next strategy in the sequence to decide the order, and so on.

Rebalance tuning options

Cruise Control provides several configuration options for tuning the rebalance parameters discussed above. You can set these tuning options at either the Cruise Control server or optimization proposal levels:

- The Cruise Control server setting can be set in the Kafka custom resource under **Kafka.spec.cruiseControl.config**.
- The individual rebalance performance configurations can be set under **KafkaRebalance.spec**.

The relevant configurations are summarized below:

Server and **KafkaRebalance** Configuration	Description	Default Value
num.concurrent.partition.movements.per.broker **concurrentPartitionMovementsPerBroker**	The maximum number of inter-broker partition movements in each partition reassignment batch	5
num.concurrent.intra.broker.partition.movements **concurrentIntraBrokerPartitionMovements**	The maximum number of intra-broker partition movements in each partition reassignment batch	2
num.concurrent.leader.movements **concurrentLeaderMovements**	The maximum number of partition leadership changes in each partition reassignment batch	1000
default.replication.throttle **replicationThrottle**	The bandwidth (in bytes per second) to be assigned to the reassigning of partitions	No Limit
default.replica.movement.strategies **replicaMovementStrategies**	The list of strategies (in priority order) used to determine the order in which partition reassignment commands are executed for generated proposals. For the server setting, use a comma separated string with the fully qualified names of the strategy class (add **com.linkedin.kafka.cruisecontrol.executor.strategy.** to the start of each class name). For the **KafkaRebalance** resource setting use a YAML array of strategy class names.	**BaseReplicaMovementStrategy**

Changing the default settings affects the length of time that the rebalance takes to complete, as well as the load placed on the Kafka cluster during the rebalance. Using lower values reduces the load but increases the amount of time taken, and vice versa.

Additional resources

- Section B.70, "**CruiseControlSpec** schema reference".

- Section B.141, "**KafkaRebalanceSpec** schema reference".

8.5. 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 keys mentioned here.

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 optimization goals for resource distribution are being broken. There are four goals of this type:

- **DiskUsageDistributionGoal** - Disk utilization distribution
- **CpuUsageDistributionGoal** - CPU utilization distribution
- **NetworkInboundUsageDistributionGoal** - Network inbound utilization distribution
- **NetworkOutboundUsageDistributionGoal** - Network outbound utilization distribution

You specify capacity limits for Kafka broker resources in the **brokerCapacity** property in **Kafka.spec.cruiseControl** . They are enabled by default and you can change their default values. Capacity limits can be set for the following broker resources, using the standard OpenShift byte units (K, M, G and T) or their bibyte (power of two) equivalents (Ki, Mi, Gi and Ti):

- **disk** - Disk storage per broker (Default: 100000Mi)
- **cpuUtilization** - CPU utilization as a percentage (Default: 100)
- **inboundNetwork** - Inbound network throughput in byte units per second (Default: 10000KiB/s)
- **outboundNetwork** - Outbound network throughput in byte units per second (Default: 10000KiB/s)

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 using bibyte units

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    brokerCapacity:
      disk: 100Gi
      cpuUtilization: 100
      inboundNetwork: 10000KiB/s
      outboundNetwork: 10000KiB/s
    # ...
```

Additional resources

For more information, refer to the Section B.72, "**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
    # ...
```

8.6. 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 8.2, “Optimization goals overview” .

3. CPU and memory resources reserved for Cruise Control. For more information, see Section 2.1.11, “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.

Table 8.2. Auto-created topics

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.71, "**CruiseControlTemplate** schema reference".

8.7. GENERATING OPTIMIZATION PROPOSALS

When you create or update a **KafkaRebalance** resource, Cruise Control generates an optimization proposal for the Kafka cluster based on the configured optimization goals.

Analyze the information in the optimization proposal and decide whether to approve it.

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
   ```

   c. To ignore the configured hard goals, add the **skipHardGoalCheck: true** property:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     goals:
       - RackAwareGoal
       - ReplicaCapacityGoal
   skipHardGoalCheck: true
   ```

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 8.8, “Approving an optimization proposal”

Additional resources

- Section 8.3, “Optimization proposals overview”

8.8. 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. The **KafkaRebalance** custom resource cannot be reused.

- NotReady: An error occurred—see Section 8.10, "Fixing problems with a **KafkaRebalance** resource".

Additional resources

- Section 8.3, "Optimization proposals overview"
- Section 8.9, "Stopping a cluster rebalance"

8.9. 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 8.3, "Optimization proposals overview"

8.10. FIXING PROBLEMS WITH A KAFKAREBALANCE RESOURCE

If an issue occurs when creating a **KafkaRebalance** resource or interacting with Cruise Control, the error is reported in the resource status, along with details of how to fix it. The resource also moves to the **NotReady** state.

To continue with the cluster rebalance operation, you must fix the problem in the **KafkaRebalance** resource itself or with the overall Cruise Control deployment. Problems might include the following:

- A misconfigured parameter in the **KafkaRebalance** resource.
- The **strimzi.io/cluster** label for specifying the Kafka cluster in the **KafkaRebalance** resource is missing.
- The Cruise Control server is not deployed as the **cruiseControl** property in the **Kafka** resource is missing.
- The Cruise Control server is not reachable.

After fixing the issue, you need to add the **refresh** annotation to the **KafkaRebalance** resource. During a "refresh", a new optimization proposal is requested from the Cruise Control server.

Prerequisites

- You have approved an optimization proposal .
- The status of the **KafkaRebalance** custom resource for the rebalance operation is **NotReady**.

Procedure

1. Get information about the error from the **KafkaRebalance** status:

```
oc describe kafkarebalance rebalance-cr-name
```

2. Attempt to resolve the issue in the **KafkaRebalance** resource.
3. Annotate the **KafkaRebalance** resource in OpenShift:

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=refresh
```

4. Check the status of the **KafkaRebalance** resource:

```
oc describe kafkarebalance rebalance-cr-name
```

5. Wait until the status changes to **PendingProposal**, or directly to **ProposalReady**.

Additional resources

- Section 8.3, "Optimization proposals overview"

CHAPTER 9. 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

9.1. 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.

9.2. 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.

9.3. 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.

9.4. 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.

9.5. 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.

9.6. 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.

9.7. 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.

9.8. 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.

9.9. 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 10. DISTRIBUTED TRACING

Distributed tracing allows you to track the progress of transactions between applications in a distributed system. In a microservices architecture, tracing tracks the progress of transactions between services. Trace data is useful for monitoring application performance and investigating issues with target systems and end-user applications.

In AMQ Streams, tracing facilitates the end-to-end tracking of messages: from source systems to Kafka, and then from Kafka to target systems and applications. It complements the metrics that are available to view in Grafana dashboards, as well as the component loggers.

How AMQ Streams supports tracing

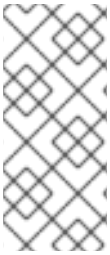
Support for tracing is built in to the following components:

- Kafka Connect (including Kafka Connect with Source2Image support)
- MirrorMaker
- MirrorMaker 2.0
- AMQ Streams Kafka Bridge

You enable and configure tracing for these components using template configuration properties in their custom resources.

To enable tracing in Kafka producers, consumers, and Kafka Streams API applications, you *instrument* application code using the OpenTracing Apache Kafka Client Instrumentation library (included with AMQ Streams). When instrumented, clients generate trace data; 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.



NOTE

Tracing is not supported for Kafka brokers.

Setting up tracing for applications and systems beyond AMQ Streams is outside the scope of this chapter. To learn more about this subject, search for "inject and extract" in the OpenTracing documentation.

Outline of procedures

To set up tracing for AMQ Streams, follow these procedures in order:

- Set up tracing for clients:
  - Initialize a Jaeger tracer for Kafka clients
- Instrument clients with tracers:
  - Instrument producers and consumers for tracing
  - Instrument Kafka Streams applications for tracing
- Set up tracing for MirrorMaker, Kafka Connect, and the Kafka Bridge

Prerequisites

- The Jaeger backend components are deployed to your OpenShift cluster. For deployment instructions, see the Jaeger deployment documentation.

10.1. OVERVIEW OF OPENTRACING AND JAEGER

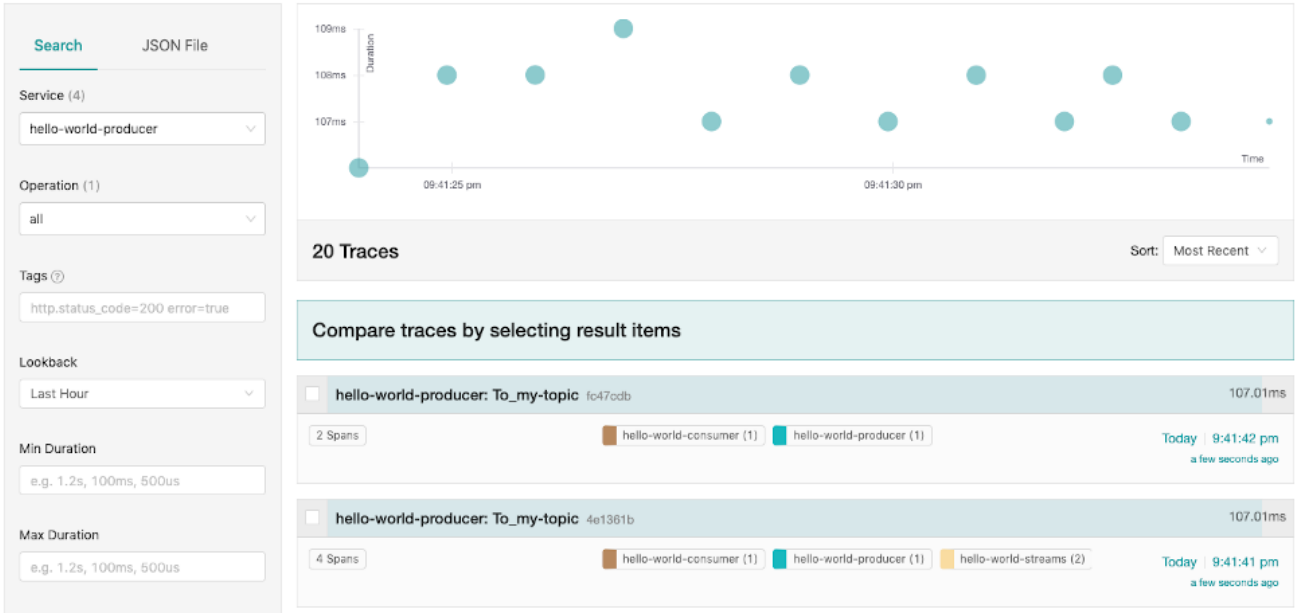
AMQ Streams uses the OpenTracing and Jaeger projects.

OpenTracing is an API specification that is independent from the tracing or monitoring system.

- The OpenTracing APIs are used to *instrument* application code
- Instrumented applications generate *traces* for individual transactions across the distributed system
- Traces are composed of *spans* that define specific units of work over time

Jaeger is a tracing system for microservices-based distributed systems.

- Jaeger implements the OpenTracing APIs and provides client libraries for instrumentation
- The Jaeger user interface allows you to query, filter, and analyze trace data



Additional resources

- OpenTracing
- Jaeger

10.2. SETTING UP TRACING FOR KAFKA CLIENTS

Initialize a Jaeger tracer to instrument your client applications for distributed tracing.

10.2.1. Initializing a Jaeger tracer for Kafka clients

Configure and initialize a Jaeger tracer using a set of tracing environment variables.

Procedure

In each client application:

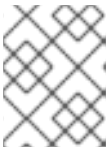
1. Add Maven dependencies for Jaeger to the **pom.xml** file for the client application:

```
<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.

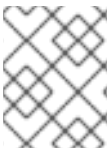
4. Register the Jaeger tracer as a global tracer:

```
GlobalTracer.register(tracer);
```

A Jaeger tracer is now initialized for the client application to use.

10.2.2. Environment variables for tracing

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.

Property	Required	Description
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.
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**, **b3**, and **w3c**.
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.

Property	Required	Description
JAEGER_SAMPLER_TYPE	No	The sampling strategy to use for client traces: • Constant • Probabilistic • Rate Limiting • Remote (the default) 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 10.2.1, "Initializing a Jaeger tracer for Kafka clients"

10.3. INSTRUMENTING KAFKA CLIENTS WITH TRACERS

Instrument Kafka producer and consumer clients, and Kafka Streams API applications for distributed tracing.

10.3.1. Instrumenting producers and consumers for tracing

Use a Decorator pattern or Interceptors to instrument your Java producer and consumer application code for tracing.

Procedure

In the application code of each 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.

- To use a Decorator pattern:

```
// 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);
```

- To use Interceptors:

```
// 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);
```

10.3.1.1. 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.

To use a Decorator pattern to instrument your producer and consumer applications, 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.

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"
```

10.3.1.2. 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())**.
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()"**.

10.3.2. Instrumenting Kafka Streams applications for tracing

This section describes how to instrument Kafka Streams API applications for distributed tracing.

Procedure

In 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();
```

10.4. SETTING UP TRACING FOR MIRRORMAKER, KAFKA CONNECT, AND THE KAFKA BRIDGE

Distributed tracing is supported for MirrorMaker, MirrorMaker 2.0, Kafka Connect (including Kafka Connect with Source2Image support), and the AMQ Streams Kafka Bridge.

Tracing in MirrorMaker and MirrorMaker 2.0

For MirrorMaker and MirrorMaker 2.0, messages are traced from the source cluster to the target cluster. The trace data records messages entering and leaving the MirrorMaker or MirrorMaker 2.0 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 2.2.1, “Configuring Kafka Connect” .

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. To have end-to-end tracing, you must configure tracing in your HTTP clients.

10.4.1. Enabling tracing in MirrorMaker, Kafka Connect, and Kafka Bridge resources

Update the configuration of **KafkaMirrorMaker**, **KafkaMirrorMaker2**, **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, MirrorMaker 2.0, Kafka Connect, or the AMQ Streams Kafka Bridge.
- For MirrorMaker, MirrorMaker 2.0, 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**, **KafkaMirrorMaker2**, **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 MirrorMaker 2.0

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

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 B.60, "**ContainerTemplate** schema reference"
- Section 2.6, "Customizing OpenShift resources"

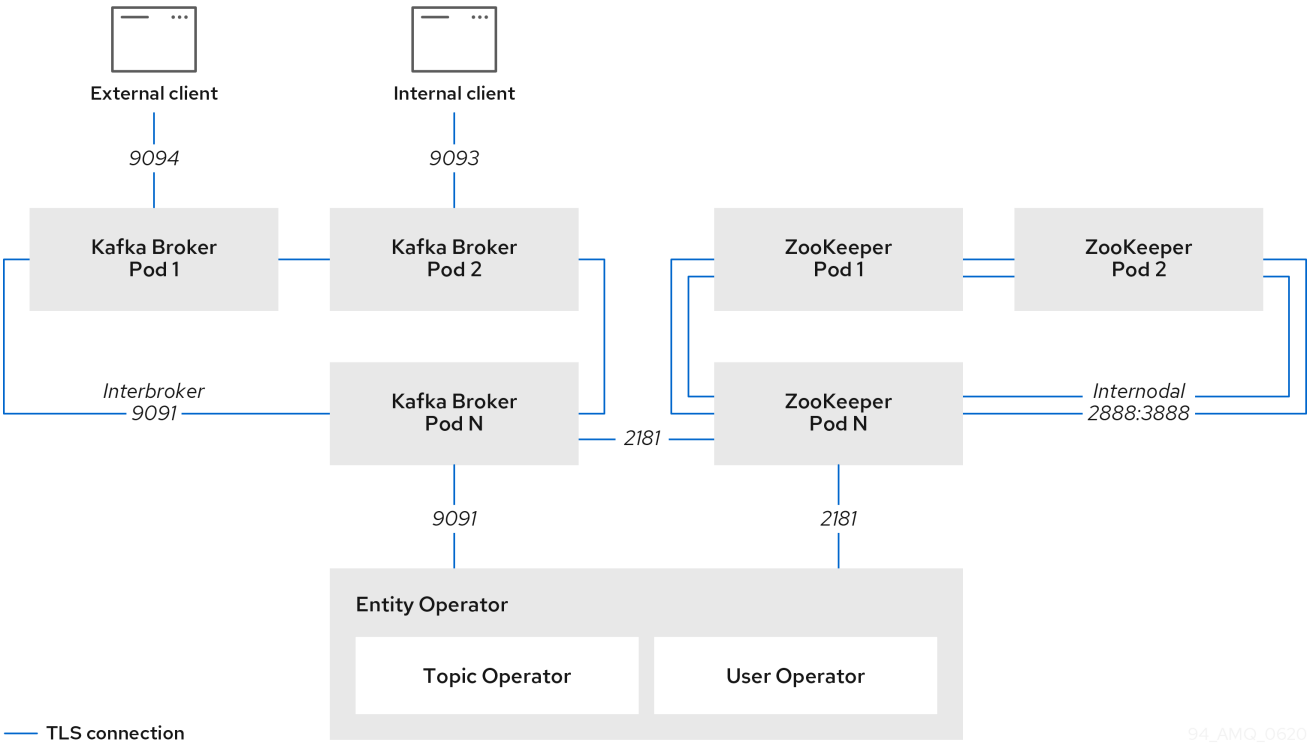
CHAPTER 11. MANAGING TLS CERTIFICATES

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 11.8, “Kafka listener certificates”.

Figure 11.1. Example architecture of the communication secured by TLS



11.1. 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.

11.1.1. 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 11.1.2, “Installing your own CA certificates” . If you provide your own certificates, you must manually renew them when needed.

11.1.2. Installing your own CA certificates

This procedure describes how to install your own CA certificates and keys instead of using the CA certificates and private keys generated by the Cluster Operator.

You can use this procedure to install your own cluster or client CA certificates.

The procedure describes renewal of CA certificates in PEM format. You can also use certificates in PKCS #12 format.

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**.
   a. Delete the existing secret:

   ```
   oc delete secret CA-CERTIFICATE-SECRET
   ```

   CA-CERTIFICATE-SECRET is the name of the **Secret**, which is ***CLUSTER-NAME*-cluster-ca-cert** for the cluster CA certificate and ***CLUSTER-NAME*-clients-ca-cert** for the clients CA certificate.

Ignore any "Not Exists" errors.

b. Create and label the new secret

```
oc create secret generic CA-CERTIFICATE-SECRET --from-file=ca.crt=CA-CERTIFICATE-FILENAME
```

2. Put your CA key in the corresponding **Secret**.

a. Delete the existing secret:

```
oc delete secret CA-KEY-SECRET
```

CA-KEY-SECRET is the name of CA key, which is ***CLUSTER-NAME*-cluster-ca** for the cluster CA key and ***CLUSTER-NAME*-clients-ca** for the clients CA key.

b. Create the new secret:

```
oc create secret generic CA-KEY-SECRET --from-file=ca.key=CA-KEY-SECRET-FILENAME
```

3. Label the secrets with the labels **strimzi.io/kind=Kafka** and **strimzi.io/cluster=*CLUSTER-NAME***:

```
oc label secret CA-CERTIFICATE-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=CLUSTER-NAME
oc label secret CA-KEY-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=CLUSTER-NAME
```

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

- To renew CA certificates you have previously installed, see Section 11.3.4, "Renewing your own CA certificates".
- Section 11.8.1, "Providing your own Kafka listener certificates" .

11.2. 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.

11.2.1. 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.

11.2.2. Cluster CA Secrets

The following tables describe the Cluster Secrets that are managed by the Cluster Operator in a Kafka cluster.

Only the ***<cluster>*-cluster-ca-cert** Secret needs to be used by clients. All other **Secrets** described only need to be accessed by the AMQ Streams components. You can enforce this using OpenShift role-based access controls, if necessary.

Table 11.1. Fields in the *<cluster>*-cluster-ca Secret

Field	Description
ca.key	The current private key for the cluster CA.

Table 11.2. Fields in the *<cluster>*-cluster-ca-cert Secret

Field	Description
ca.p12	PKCS #12 archive file for storing certificates and keys.
ca.password	Password for protecting the PKCS #12 archive file.

Field	Description
ca.crt	The current certificate for the cluster CA.

NOTE

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.

Table 11.3. Fields in the ***<cluster>*-kafka-brokers** Secret

Field	Description
***<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>***.

Table 11.4. Fields in the ***<cluster>*-zookeeper-nodes** Secret

Field	Description
***<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>***.

Table 11.5. Fields in the ***<cluster>*-entity-operator-certs** Secret

Field	Description
entity-operator_.p12	PKCS #12 archive file for storing certificates and keys.
entity-operator_.password	Password for protecting the PKCS #12 archive file.

Field	Description
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.

11.2.3. Client CA Secrets

Table 11.6. 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.
***<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.

11.2.4. User Secrets

Table 11.7. 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.

Secret name	Field within Secret	Description
	user.crt	Certificate for the user, signed by the clients CA
	user.key	Private key for the user

11.3. CERTIFICATE RENEWAL AND VALIDITY PERIODS

Cluster CA and clients CA certificates are only valid for a limited time period, known as the validity period. This is usually defined as a number of days since the certificate was generated.

For auto-generated CA certificates, you can configure the validity period of:

- Cluster CA certificates in **Kafka.spec.clusterCa.validityDays**
- Client CA certificates in **Kafka.spec.clientsCa.validityDays**

The default validity period for both certificates is 365 days. Manually-installed CA certificates should have their own validity periods defined.

When a CA certificate expires, components and clients that still trust that certificate will not accept TLS connections from peers whose certificates were signed by the CA private key. The components and clients need to trust the *new* CA certificate instead.

To allow the renewal of CA certificates without a loss of service, the Cluster Operator will initiate certificate renewal before the old CA certificates expire.

You can configure the renewal period of:

- Cluster CA certificates in **Kafka.spec.clusterCa.renewalDays**
- Client CA certificates in **Kafka.spec.clientsCa.renewalDays**

The default renewal period for both certificates is 30 days.

The renewal period is measured backwards, from the expiry date of the current certificate.

Validity period against renewal period

```
Not Before                    Not After
   |                             |
   |<--------------- validityDays --------------->|
                     <--- renewalDays --->|
```

To make a change to the validity and renewal periods after creating the Kafka cluster, you configure and apply the **Kafka** custom resource, and manually renew the CA certificates. If you do not manually renew the certificates, the new periods will be used the next time the certificate is renewed automatically.

Example Kafka configuration for certificate validity and renewal periods

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

```
# ...
spec:
# ...
  clusterCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
  clientsCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
# ...
```

The behavior of the Cluster Operator during the renewal period depends on whether the relevant setting is enabled in either **Kafka.spec.clusterCa.generateCertificateAuthority** or **Kafka.spec.clientsCa.generateCertificateAuthority**.

11.3.1. 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.

11.3.2. 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 .

11.3.2.1. 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.

11.3.3. Renewing CA certificates manually

Cluster and clients CA certificates auto-renew at the start of their respective certificate renewal periods. If **Kafka.spec.clusterCa.generateCertificateAuthority** and **Kafka.spec.clientsCa.generateCertificateAuthority** are set to **false**, the CA certificates do not auto-renew.

You can manually renew one or both of these certificates before the certificate renewal period starts. You might do this for security reasons, or if you have changed the renewal or validity periods for the certificates.

A renewed certificate uses the same private key as the old certificate.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which CA certificates and private keys are installed.

Procedure

1. Apply the **strimzi.io/force-renew** annotation to the **Secret** that contains the CA certificate that you want to renew.

   Table 11.8. Annotation for the Secret that forces renewal of certificates

Certificate	Secret	Annotate command
Cluster CA	*KAFKA-CLUSTER-NAME*-cluster-ca-cert	**oc annotate secret *KAFKA-CLUSTER-NAME*-cluster-ca-cert strimzi.io/force-renew=true**
Clients CA	*KAFKA-CLUSTER-NAME*-clients-ca-cert	**oc annotate secret *KAFKA-CLUSTER-NAME*-clients-ca-cert strimzi.io/force-renew=true**

   At the next reconciliation the Cluster Operator will generate a new CA certificate for the **Secret** that you annotated. If maintenance time windows are configured, the Cluster Operator will generate the new CA certificate at the first reconciliation within the next maintenance time window.

   Client applications must reload the cluster and clients CA certificates that were renewed by the Cluster Operator.

2. Check the period the CA certificate is valid:
   For example, using an **openssl** command:

   ```
   oc get secret CA-CERTIFICATE-SECRET -o 'jsonpath={.data.CA-CERTIFICATE}' | base64 -d | openssl x509 -subject -issuer -startdate -enddate -noout
   ```

   CA-CERTIFICATE-SECRET is the name of the **Secret**, which is ***KAFKA-CLUSTER-NAME*-cluster-ca-cert** for the cluster CA certificate and ***KAFKA-CLUSTER-NAME*-clients-ca-cert** for the clients CA certificate.

   CA-CERTIFICATE is the name of the CA certificate, such as **jsonpath={.data.ca\.crt}**.

   The command returns a **notBefore** and **notAfter** date, which is the validity period for the CA certificate.

   For example, for a cluster CA certificate:

   ```
   subject=O = io.strimzi, CN = cluster-ca v0
   issuer=O = io.strimzi, CN = cluster-ca v0
   notBefore=Jun 30 09:43:54 2020 GMT
   notAfter=Jun 30 09:43:54 2021 GMT
   ```

3. Delete old certificates from the Secret.
   When components are using the new certificates, older certificates might still be active. Delete the old certificates to remove any potential security risk.

Additional resources

- Section 11.2, “Secrets”
- Section 2.1.27, “Maintenance time windows for rolling updates”
- Section B.69, “**CertificateAuthority** schema reference”

11.3.4. Renewing your own CA certificates

This procedure describes how to renew CA certificates and keys that you previously installed. You need to follow this procedure during the renewal period in order to replace CA certificates which will soon expire.

You can use this procedure to install your own cluster or client CA certificates.

The procedure describes renewal of CA certificates in PEM format. You can also use certificates in PKCS #12 format.

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 an **openssl** command, such as:

```
openssl req -x509 -new -days NUMBER-OF-DAYS-VALID --nodes -out ca.crt -keyout ca.key
```

Procedure

1. Check the details of the current CA certificates in the **Secret**:

   ```
   oc describe secret CA-CERTIFICATE-SECRET
   ```

   CA-CERTIFICATE-SECRET is the name of the **Secret**, which is ***KAFKA-CLUSTER-NAME*-cluster-ca-cert** for the cluster CA certificate and ***KAFKA-CLUSTER-NAME*-clients-ca-cert** for the clients CA certificate.

2. Create a directory to contain the existing CA certificates in the secret.

   ```
   mkdir new-ca-cert-secret
   cd new-ca-cert-secret
   ```

3. Fetch the secret for each CA certificate you wish to renew:

   ```
   oc get secret CA-CERTIFICATE-SECRET -o 'jsonpath={.data.CA-CERTIFICATE}' | base64 -d > CA-CERTIFICATE
   ```

   Replace *CA-CERTIFICATE* with the name of each CA certificate.

4. Rename the old **ca.crt** file as **ca-*DATE*.crt**, where *DATE* is the certificate expiry date in the format *YEAR-MONTH-DAYTHOUR-MINUTE-SECONDZ*.
   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
   ```

5. Copy your new CA certificate into the directory, naming it **ca.crt**:

   ```
   cp PATH-TO-NEW-CERTIFICATE ca.crt
   ```

6. Put your CA certificate in the corresponding **Secret**.

   a. Delete the existing secret:

      ```
      oc delete secret CA-CERTIFICATE-SECRET
      ```

      CA-CERTIFICATE-SECRET is the name of the **Secret**, as returned in the first step.

      Ignore any "Not Exists" errors.

   b. Recreate the secret:

      ```
      oc create secret generic CA-CERTIFICATE-SECRET --from-file=.
      ```

7. Delete the directory you created:

   ```
   cd ..
   rm -r new-ca-cert-secret
   ```

8. Put your CA key in the corresponding **Secret**.

   a. Delete the existing secret:

   ```
   oc delete secret CA-KEY-SECRET
   ```

   CA-KEY-SECRET is the name of CA key, which is ***KAFKA-CLUSTER-NAME*-cluster-ca** for the cluster CA key and ***KAFKA-CLUSTER-NAME*-clients-ca** for the clients CA key.

   b. Recreate the secret with the new CA key:

   ```
   oc create secret generic CA-KEY-SECRET --from-file=ca.key=CA-KEY-SECRET-FILENAME
   ```

9. Label the secrets with the labels **strimzi.io/kind=Kafka** and **strimzi.io/cluster=*KAFKA-CLUSTER-NAME***:

   ```
   oc label secret CA-CERTIFICATE-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=KAFKA-CLUSTER-NAME
   oc label secret CA-KEY-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=KAFKA-CLUSTER-NAME
   ```

11.4. 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.

  Table 11.9. Commands for replacing private keys

Private key for	Secret	Annotate command
Cluster CA	*<cluster-name>*-cluster-ca	**oc annotate secret *<cluster-name>*-cluster-ca strimzi.io/force-replace=true**
Clients CA	*<cluster-name>*-clients-ca	**oc annotate secret *<cluster-name>*-clients-ca strimzi.io/force-replace=true**

At the next reconciliation the Cluster Operator will:

- Generate a new private key for the **Secret** that you annotated
- Generate a new CA certificate

If maintenance time windows are configured, the Cluster Operator will generate the new private key and CA certificate at the first reconciliation within the next maintenance time window.

Client applications must reload the cluster and clients CA certificates that were renewed by the Cluster Operator.

Additional resources

- Section 11.2, “Secrets”
- Section 2.1.27, “Maintenance time windows for rolling updates”

11.5. TLS CONNECTIONS

11.5.1. ZooKeeper communication

Communication between the ZooKeeper nodes on all ports as well as between clients and ZooKeeper is encrypted.

11.5.2. 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 is also encrypted.

11.5.3. Topic and User Operators

All Operators use encryption for communication with both Kafka and ZooKeeper. In Topic and User Operators, a TLS sidecar is used when communicating with ZooKeeper.

11.5.4. Cruise Control

Cruise Control uses encryption for communication with both Kafka and ZooKeeper. A TLS sidecar is used when communicating with ZooKeeper.

11.5.5. Kafka Client connections

Encrypted or unencrypted communication between Kafka brokers and clients is configured using the **tls** property for **spec.kafka.listeners**.

11.6. 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 a TLS listener — to trust the cluster CA certificate.

The easiest way to achieve this for an internal client is to use a volume mount to access the **Secrets** containing the necessary certificates and keys.

Follow the steps to configure trust certificates that are signed by the cluster CA for Java-based Kafka Producer, Consumer, and Streams APIs.

Choose the steps to follow according to the certificate format of the cluster CA: PKCS #12 (**.p12**) or PEM (**.crt**).

The steps describe how to mount the Cluster Secret that verifies the identity of the Kafka cluster to the client pod.

Prerequisites

- The Cluster Operator must be running.
- There needs to be a **Kafka** resource within the OpenShift cluster.
- You need a Kafka client application inside the OpenShift cluster that will connect using TLS, and needs to trust the cluster CA certificate.
- The client application must be running in the same namespace as the **Kafka** resource.

Using PKCS #12 format (.p12)

1. Mount the cluster Secret as a volume when defining the client pod.
   For example:

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
  - name: client-name
    image: client-name
    volumeMounts:
    - name: secret-volume
      mountPath: /data/p12
    env:
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: my-password
  volumes:
  - name: secret-volume
    secret:
      secretName: my-cluster-cluster-ca-cert
```

   Here we're mounting:

   - The 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-ca-cert
```

2. Use the certificate with clients that use certificates in X.509 format.

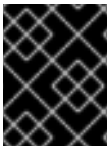
11.7. CONFIGURING EXTERNAL CLIENTS TO TRUST THE CLUSTER CA

This procedure describes how to configure a Kafka client that resides outside the OpenShift cluster – connecting to an **external** listener – to trust the cluster CA certificate. Follow this procedure when setting up the client and during the renewal period, when the old clients CA certificate is replaced.

Follow the steps to configure trust certificates that are signed by the cluster CA for Java-based Kafka Producer, Consumer, and Streams APIs.

Choose the steps to follow according to the certificate format of the cluster CA: PKCS #12 (**.p12**) or PEM (**.crt**).

The steps describe how to obtain the certificate from the Cluster Secret that verifies the identity of the Kafka cluster.



IMPORTANT

The ***<cluster-name>*-cluster-ca-cert Secret** 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.

11.8. KAFKA LISTENER CERTIFICATES

You can provide your own server certificates and private keys for the following types of listeners:

- Internal TLS listeners for communication within the OpenShift cluster
- 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.

11.8.1. 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 11.8.2, "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:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
```

```
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
# ...
```

Example configuration for a TLS listener

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication:
      type: tls
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
# ...
```

3. Apply the new configuration to create or update the resource:

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

The Cluster Operator starts a rolling update of the Kafka cluster, which updates the configuration of the listeners.

NOTE

A rolling update is also started if you update a Kafka listener certificate in a **Secret** that is already used by a TLS or external listener.

Additional resources

- Alternative subjects in server certificates for Kafka listeners
- **GenericKafkaListener** schema reference
- Kafka listener certificates

11.8.2. 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.

11.8.2.1. 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
```

11.8.2.2. 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**.

Table 11.10. SANs for each type of external listener

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.

External listener type	In the SANs, specify...
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 11.8.1, “Providing your own Kafka listener certificates”

CHAPTER 12. MANAGING AMQ STREAMS

This chapter covers tasks to maintain a deployment of AMQ Streams.

12.1. WORKING WITH CUSTOM RESOURCES

You can use **oc** commands to retrieve information and perform other operations on AMQ Streams custom resources.

Using **oc** with the **status** subresource of a custom resource allows you to get the information about the resource.

12.1.1. Performing oc operations on custom resources

Use **oc** commands, such as **get**, **describe**, **edit**, or **delete**, to perform operations on resource types. For example, **oc get kafkatopics** retrieves a list of all Kafka topics and **oc get kafkas** retrieves all deployed Kafka clusters.

When referencing resource types, you can use both singular and plural names: **oc get kafkas** gets the same results as **oc get kafka**.

You can also use the *short name* of the resource. Learning short names can save you time when managing AMQ Streams. The short name for **Kafka** is **k**, so you can also run **oc get k** to list all Kafka clusters.

```
oc get k

NAME       DESIRED KAFKA REPLICAS   DESIRED ZK REPLICAS
my-cluster 3                        3
```

Table 12.1. Long and short names for each AMQ Streams resource

AMQ Streams resource	Long name	Short name
Kafka	kafka	k
Kafka Topic	kafkatopic	kt
Kafka User	kafkauser	ku
Kafka Connect	kafkaconnect	kc
Kafka Connect S2I	kafkaconnects2i	kcs2i
Kafka Connector	kafkaconnector	kctr
Kafka Mirror Maker	kafkamirrormaker	kmm
Kafka Mirror Maker 2	kafkamirrormaker2	kmm2

AMQ Streams resource	Long name	Short name
Kafka Bridge	kafkabridge	kb
Kafka Rebalance	kafkarebalance	kr

12.1.1.1. Resource categories

Categories of custom resources can also be used in **oc** commands.

All AMQ Streams custom resources belong to the category **strimzi**, so you can use **strimzi** to get all the AMQ Streams resources with one command.

For example, running **oc get strimzi** lists all AMQ Streams custom resources in a given namespace.

```
oc get strimzi

NAME                                DESIRED KAFKA REPLICAS DESIRED ZK REPLICAS
kafka.kafka.strimzi.io/my-cluster   3                      3

NAME                                PARTITIONS REPLICATION FACTOR
kafkatopic.kafka.strimzi.io/kafka-apps 3          3

NAME                                AUTHENTICATION AUTHORIZATION
kafkauser.kafka.strimzi.io/my-user  tls            simple
```

The **oc get strimzi -o name** command returns all resource types and resource names. The **-o name** option fetches the output in the *type/name* format

```
oc get strimzi -o name

kafka.kafka.strimzi.io/my-cluster
kafkatopic.kafka.strimzi.io/kafka-apps
kafkauser.kafka.strimzi.io/my-user
```

You can combine this **strimzi** command with other commands. For example, you can pass it into a **oc delete** command to delete all resources in a single command.

```
oc delete $(oc get strimzi -o name)

kafka.kafka.strimzi.io "my-cluster" deleted
kafkatopic.kafka.strimzi.io "kafka-apps" deleted
kafkauser.kafka.strimzi.io "my-user" deleted
```

Deleting all resources in a single operation might be useful, for example, when you are testing new AMQ Streams features.

12.1.1.2. Querying the status of sub-resources

There are other values you can pass to the **-o** option. For example, by using **-o yaml** you get the output in YAML format. Usng **-o json** will return it as JSON.

You can see all the options in **oc get --help**.

One of the most useful options is the JSONPath support, which allows you to pass JSONPath expressions to query the Kubernetes API. A JSONPath expression can extract or navigate specific parts of any resource.

For example, you can use the JSONPath expression **{.status.listeners[?(@.type=="tls")].bootstrapServers}** to get the bootstrap address from the status of the Kafka custom resource and use it in your Kafka clients.

Here, the command finds the **bootstrapServers** value of the **tls** listeners.

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

my-cluster-kafka-bootstrap.myproject.svc:9093
```

By changing the type condition to **@.type=="external"** or **@.type=="plain"** you can also get the address of the other Kafka listeners.

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

192.168.1.247:9094
```

You can use **jsonpath** to extract any other property or group of properties from any custom resource.

12.1.2. AMQ Streams custom resource status information

Several resources have a **status** property, as described in the following table.

Table 12.2. Custom resource status properties

AMQ Streams resource	Schema reference	Publishes status information on...
Kafka	Section B.75, "**KafkaStatus** schema reference"	The Kafka cluster.
KafkaConnect	Section B.93, "**KafkaConnectStatus** schema reference"	The Kafka Connect cluster, if deployed.
KafkaConnectS2I	Section B.97, "**KafkaConnectS2IStatus** schema reference"	The Kafka Connect cluster with Source-to-Image support, if deployed.
KafkaConnector	Section B.132, "**KafkaConnectorStatus** schema reference"	**KafkaConnector** resources, if deployed.
KafkaMirrorMaker	Section B.120, "**KafkaMirrorMakerStatus** schema reference"	The Kafka MirrorMaker tool, if deployed.

AMQ Streams resource	Schema reference	Publishes status information on...
KafkaTopic	Section B.100, "**KafkaTopicStatus** schema reference"	Kafka topics in your Kafka cluster.
KafkaUser	Section B.113, "**KafkaUserStatus** schema reference"	Kafka users in your Kafka cluster.
KafkaBridge	Section B.129, "**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

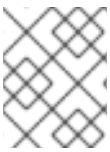
```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
spec:
```

```
  # ...
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 12.1.3, “Finding the status of a custom resource” .

12.1.3. Finding the status of a custom resource

This procedure describes how to find the status of a custom resource.

Prerequisites

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

Procedure

- Specify the custom resource and use the **-o jsonpath** option to apply a standard JSONPath expression to select the **status** property:

  ```
  oc get kafka <kafka_resource_name> -o jsonpath='{.status}'
  ```

  This expression returns all the status information for the specified custom resource. You can use dot notation, such as **status.listeners** or **status.observedGeneration**, to fine-tune the status information you wish to see.

Additional resources

- Section 12.1.2, “AMQ Streams custom resource status information”
- For more information about using JSONPath, see JSONPath support.

12.2. 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
```

12.2.1. 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.

12.3. 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

12.3.1. 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.

12.3.2. 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.

12.3.3. Recovering a deleted cluster from persistent volumes

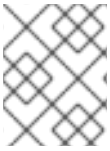
This procedure describes how to recover a deleted cluster from persistent volumes (PVs).

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 -ti my-cluster-zookeeper-0 -- bin/zookeeper-shell.sh localhost:12181
      ```

   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.66, "**EntityTopicOperatorSpec** schema reference".

7. Verify the recovery by listing the **KafkaTopic** resources:

```
oc get KafkaTopic
```

12.4. TUNING CLIENT CONFIGURATION

Use configuration properties to optimize the performance of Kafka producers and consumers.

A minimum set of configuration properties is required, but you can add or adjust properties to change how producers and consumers interact with Kafka. For example, for producers you can tune latency and throughput of messages so that clients can respond to data in real time. Or you can change the configuration to provide stronger message durability guarantees.

You might start by analyzing client metrics to gauge where to make your initial configurations, then make incremental changes and further comparisons until you have the configuration you need.

12.4.1. Kafka producer configuration tuning

Use a basic producer configuration with optional properties that are tailored to specific use cases.

Adjusting your configuration to maximize throughput might increase latency or vice versa. You will need to experiment and tune your producer configuration to get the balance you need.

12.4.1.1. Basic producer configuration

Connection and serializer properties are required for every producer. Generally, it is good practice to add a client id for tracking, and use compression on the producer to reduce batch sizes in requests.

In a basic producer configuration:

- The order of messages in a partition is not guaranteed.
- The acknowledgment of messages reaching the broker does not guarantee durability.

```
# ...
bootstrap.servers=localhost:9092 1
key.serializer=org.apache.kafka.common.serialization.StringSerializer 2
value.serializer=org.apache.kafka.common.serialization.StringSerializer 3
client.id=my-client 4
compression.type=gzip 5
# ...
```

1. (Required) Tells the producer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The producer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a server is down, but it's not necessary to provide a list of all the brokers in the cluster.

2. (Required) Serializer to transform the key of each message to bytes prior to them being sent to a broker.

3. (Required) Serializer to transform the value of each message to bytes prior to them being sent to a broker.

4. (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request.

or a request.

5 (Optional) The codec for compressing messages, which are sent and might be stored in compressed format and then decompressed when reaching a consumer. Compression is useful for improving throughput and reducing the load on storage, but might not be suitable for low latency applications where the cost of compression or decompression could be prohibitive.

12.4.1.2. Data durability

You can apply greater data durability, to minimize the likelihood that messages are lost, using message delivery acknowledgments.

```
# ...
acks=all 1
# ...
```

1 Specifying **acks=all** forces a partition leader to replicate messages to a certain number of followers before acknowledging that the message request was successfully received. Because of the additional checks, **acks=all** increases the latency between the producer sending a message and receiving acknowledgment.

The number of brokers which need to have appended the messages to their logs before the acknowledgment is sent to the producer is determined by the topic's **min.insync.replicas** configuration. A typical starting point is to have a topic replication factor of 3, with two in-sync replicas on other brokers. In this configuration, the producer can continue unaffected if a single broker is unavailable. If a second broker becomes unavailable, the producer won't receive acknowledgments and won't be able to produce more messages.

Topic configuration to support acks=all

```
# ...
min.insync.replicas=2 1
# ...
```

1 Use **2** in-sync replicas. The default is **1**.

NOTE

If the system fails, there is a risk of unsent data in the buffer being lost.

12.4.1.3. Ordered delivery

Idempotent producers avoid duplicates as messages are delivered exactly once. IDs and sequence numbers are assigned to messages to ensure the order of delivery, even in the event of failure. If you are using **acks=all** for data consistency, enabling idempotency makes sense for ordered delivery.

Ordered delivery with idempotency

```
# ...
enable.idempotence=true 1
max.in.flight.requests.per.connection=5 2
```

```
acks=all (3)
retries=2147483647 (4)
# ...
```

1. Set to **true** to enable the idempotent producer.
2. With idempotent delivery the number of in-flight requests may be greater than 1 while still providing the message ordering guarantee. The default is 5 in-flight requests.
3. Set **acks** to **all**.
4. Set the number of attempts to resend a failed message request.

If you are not using **acks=all** and idempotency because of the performance cost, set the number of in-flight (unacknowledged) requests to 1 to preserve ordering. Otherwise, a situation is possible where *Message-A* fails only to succeed after *Message-B* was already written to the broker.

Ordered delivery without idempotency

```
# ...
enable.idempotence=false (1)
max.in.flight.requests.per.connection=1 (2)
retries=2147483647
# ...
```

1. Set to **false** to disable the idempotent producer.
2. Set the number of in-flight requests to exactly **1**.

12.4.1.4. Reliability guarantees

Idempotence is useful for exactly once writes to a single partition. Transactions, when used with idempotence, allow exactly once writes across multiple partitions.

Transactions guarantee that messages using the same transactional ID are produced once, and either *all* are successfully written to the respective logs or *none* of them are.

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID (1)
transaction.timeout.ms=900000 (2)
# ...
```

1. Specify a unique transactional ID.
2. Set the maximum allowed time for transactions in milliseconds before a timeout error is returned. The default is **900000** or 15 minutes.

The choice of **transactional.id** is important in order that the transactional guarantee is maintained. Each

transactional id should be used for a unique set of topic partitions. For example, this can be achieved using an external mapping of topic partition names to transactional ids, or by computing the transactional id from the topic partition names using a function that avoids collisions.

12.4.1.5. Optimizing throughput and latency

Usually, the requirement of a system is to satisfy a particular throughput target for a proportion of messages within a given latency. For example, targeting 500,000 messages per second with 95% of messages being acknowledged within 2 seconds.

It's likely that the messaging semantics (message ordering and durability) of your producer are defined by the requirements for your application. For instance, it's possible that you don't have the option of using **acks=0** or **acks=1** without breaking some important property or guarantee provided by your application.

Broker restarts have a significant impact on high percentile statistics. For example, over a long period the 99th percentile latency is dominated by behavior around broker restarts. This is worth considering when designing benchmarks or comparing performance numbers from benchmarking with performance numbers seen in production.

Depending on your objective, Kafka offers a number of configuration parameters and techniques for tuning producer performance for throughput and latency.

Message batching (`linger.ms` and `batch.size`)

Message batching delays sending messages in the hope that more messages destined for the same broker will be sent, allowing them to be batched into a single produce request. Batching is a compromise between higher latency in return for higher throughput. Time-based batching is configured using **linger.ms**, and size-based batching is configured using **batch.size**.

Compression (`compression.type`)

Message compression adds latency in the producer (CPU time spent compressing the messages), but makes requests (and potentially disk writes) smaller, which can increase throughput. Whether compression is worthwhile, and the best compression to use, will depend on the messages being sent. Compression happens on the thread which calls **KafkaProducer.send()**, so if the latency of this method matters for your application you should consider using more threads.

Pipelining (`max.in.flight.requests.per.connection`)

Pipelining means sending more requests before the response to a previous request has been received. In general more pipelining means better throughput, up to a threshold at which other effects, such as worse batching, start to counteract the effect on throughput.

Lowering latency

When your application calls **KafkaProducer.send()** the messages are:

- Processed by any interceptors
- Serialized
- Assigned to a partition
- Compressed
- Added to a batch of messages in a per-partition queue

At which point the **send()** method returns. So the time **send()** is blocked is determined by:

- The time spent in the interceptors, serializers and partitioner
- The compression algorithm used
- The time spent waiting for a buffer to use for compression

Batches will remain in the queue until one of the following occurs:

- The batch is full (according to **batch.size**)
- The delay introduced by **linger.ms** has passed
- The sender is about to send message batches for other partitions to the same broker, and it is possible to add this batch too
- The producer is being flushed or closed

Look at the configuration for batching and buffering to mitigate the impact of **send()** blocking on latency.

```
# ...
linger.ms=100 1
batch.size=16384 2
buffer.memory=33554432 3
# ...
```

1. The **linger** property adds a delay in milliseconds so that larger batches of messages are accumulated and sent in a request. The default is **0'.**
2. If a maximum **batch.size** in bytes is used, a request is sent when the maximum is reached, or messages have been queued for longer than **linger.ms** (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.
3. The buffer size must be at least as big as the batch size, and be able to accommodate buffering, compression and in-flight requests.

Increasing throughput

Improve throughput of your message requests by adjusting the maximum time to wait before a message is delivered and completes a send request.

You can also direct messages to a specified partition by writing a custom partitioner to replace the default.

```
# ...
delivery.timeout.ms=120000 1
partitioner.class=my-custom-partitioner 2

# ...
```

1. The maximum time in milliseconds to wait for a complete send request. You can set the value to **MAX_LONG** to delegate to Kafka an indefinite number of retries. The default is **120000** or 2 minutes.
2. Specify the class name of the custom partitioner.

12.4.2. Kafka consumer configuration tuning

Use a basic consumer configuration with optional properties that are tailored to specific use cases.

When tuning your consumers your primary concern will be ensuring that they cope efficiently with the amount of data ingested. As with the producer tuning, be prepared to make incremental changes until the consumers operate as expected.

12.4.2.1. Basic consumer configuration

Connection and deserializer properties are required for every consumer. Generally, it is good practice to add a client id for tracking.

In a consumer configuration, irrespective of any subsequent configuration:

- The consumer fetches from a given offset and consumes the messages in order, unless the offset is changed to skip or re-read messages.
- The broker does not know if the consumer processed the responses, even when committing offsets to Kafka, because the offsets might be sent to a different broker in the cluster.

```
# ...
bootstrap.servers=localhost:9092 1
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer 2
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer 3
client.id=my-client 4
group.id=my-group-id 5
# ...
```

1. (Required) Tells the consumer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The consumer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a server is down, but it is not necessary to provide a list of all the brokers in the cluster. If you are using a loadbalancer service to expose the Kafka cluster, you only need the address for the service because the availability is handled by the loadbalancer.
2. (Required) Deserializer to transform the bytes fetched from the Kafka broker into message keys.
3. (Required) Deserializer to transform the bytes fetched from the Kafka broker into message values.
4. (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request. The id can also be used to throttle consumers based on processing time quotas.
5. (Conditional) A group id is *required* for a consumer to be able to join a consumer group.

Consumer groups are used to share a typically large data stream generated by multiple producers from a given topic. Consumers are grouped using a **group.id**, allowing messages to be spread across the members.

12.4.2.2. Scaling data consumption using consumer groups

Consumer groups share a typically large data stream generated by one or multiple producers from a given topic. Consumers with the same **group.id** property are in the same group. One of the consumers in the group is elected leader and decides how the partitions are assigned to the consumers in the

group. Each partition can only be assigned to a single consumer.

If you do not already have as many consumers as partitions, you can scale data consumption by adding more consumer instances with the same **group.id**. Adding more consumers to a group than there are partitions will not help throughput, but it does mean that there are consumers on standby should one stop functioning. If you can meet throughput goals with fewer consumers, you save on resources.

Consumers within the same consumer group send offset commits and heartbeats to the same broker. So the greater the number of consumers in the group, the higher the request load on the broker.

```
# ...
group.id=my-group-id 1
# ...
```

1 Add a consumer to a consumer group using a group id.

12.4.2.3. Message ordering guarantees

Kafka brokers receive fetch requests from consumers that ask the broker to send messages from a list of topics, partitions and offset positions.

A consumer observes messages in a single partition in the same order that they were committed to the broker, which means that Kafka **only** provides ordering guarantees for messages in a single partition. Conversely, if a consumer is consuming messages from multiple partitions, the order of messages in different partitions as observed by the consumer does not necessarily reflect the order in which they were sent.

If you want a strict ordering of messages from one topic, use one partition per consumer.

12.4.2.4. Optimizing throughput and latency

Control the number of messages returned when your client application calls **KafkaConsumer.poll()**.

Use the **fetch.max.wait.ms** and **fetch.min.bytes** properties to increase the minimum amount of data fetched by the consumer from the Kafka broker. Time-based batching is configured using **fetch.max.wait.ms**, and size-based batching is configured using **fetch.min.bytes**.

If CPU utilization in the consumer or broker is high, it might be because there are too many requests from the consumer. You can adjust **fetch.max.wait.ms** and **fetch.min.bytes** properties higher so that there are fewer requests and messages are delivered in bigger batches. By adjusting higher, throughput is improved with some cost to latency. You can also adjust higher if the amount of data being produced is low.

For example, if you set **fetch.max.wait.ms** to 500ms and **fetch.min.bytes** to 16384 bytes, when Kafka receives a fetch request from the consumer it will respond when the first of either threshold is reached.

Conversely, you can adjust the **fetch.max.wait.ms** and **fetch.min.bytes** properties lower to improve end-to-end latency.

```
# ...
fetch.max.wait.ms=500 1
fetch.min.bytes=16384 2
# ...
```

1. The maximum time in milliseconds the broker will wait before completing fetch requests. The default is **500** milliseconds.

2. If a minimum batch size in bytes is used, a request is sent when the minimum is reached, or messages have been queued for longer than **fetch.max.wait.ms** (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.

Lowering latency by increasing the fetch request size

Use the **fetch.max.bytes** and **max.partition.fetch.bytes** properties to increase the maximum amount of data fetched by the consumer from the Kafka broker.

The **fetch.max.bytes** property sets a maximum limit in bytes on the amount of data fetched from the broker at one time.

The **max.partition.fetch.bytes** sets a maximum limit in bytes on how much data is returned for each partition, which must always be larger than the number of bytes set in the broker or topic configuration for **max.message.bytes**.

The maximum amount of memory a client can consume is calculated approximately as:

> *NUMBER-OF-BROKERS* * fetch.max.bytes and *NUMBER-OF-PARTITIONS* * max.partition.fetch.bytes

If memory usage can accommodate it, you can increase the values of these two properties. By allowing more data in each request, latency is improved as there are fewer fetch requests.

```
# ...
fetch.max.bytes=52428800 (1)
max.partition.fetch.bytes=1048576 (2)
# ...
```

1. The maximum amount of data in bytes returned for a fetch request.

2. The maximum amount of data in bytes returned for each partition.

12.4.2.5. Avoiding data loss or duplication when committing offsets

The Kafka *auto-commit mechanism* allows a consumer to commit the offsets of messages automatically. If enabled, the consumer will commit offsets received from polling the broker at 5000ms intervals.

The auto-commit mechanism is convenient, but it introduces a risk of data loss and duplication. If a consumer has fetched and transformed a number of messages, but the system crashes with processed messages in the consumer buffer when performing an auto-commit, that data is lost. If the system crashes after processing the messages, but before performing the auto-commit, the data is duplicated on another consumer instance after rebalancing.

Auto-committing can avoid data loss only when all messages are processed before the next poll to the broker, or the consumer closes.

To minimize the likelihood of data loss or duplication, you can set **enable.auto.commit** to **false** and develop your client application to have more control over committing offsets. Or you can use **auto.commit.interval.ms** to decrease the intervals between commits.

```
# ...
enable.auto.commit=false (1)
# ...
```

(1) Auto commit is set to false to provide more control over committing offsets.

By setting to **enable.auto.commit** to **false**, you can commit offsets after **all** processing has been performed and the message has been consumed. For example, you can set up your application to call the Kafka **commitSync** and **commitAsync** commit APIs.

The **commitSync** API commits the offsets in a message batch returned from polling. You call the API when you are finished processing all the messages in the batch. If you use the **commitSync** API, the application will not poll for new messages until the last offset in the batch is committed. If this negatively affects throughput, you can commit less frequently, or you can use the **commitAsync** API. The **commitAsync** API does not wait for the broker to respond to a commit request, but risks creating more duplicates when rebalancing. A common approach is to combine both commit APIs in an application, with the **commitSync** API used just before shutting the consumer down or rebalancing to make sure the final commit is successful.

12.4.2.5.1. Controlling transactional messages

Consider using transactional ids and enabling idempotence (**enable.idempotence=true**) on the producer side to guarantee exactly-once delivery. On the consumer side, you can then use the **isolation.level** property to control how transactional messages are read by the consumer.

The **isolation.level** property has two valid values:

- **read_committed**
- **read_uncommitted** (default)

Use **read_committed** to ensure that only transactional messages that have been committed are read by the consumer. However, this will cause an increase in end-to-end latency, because the consumer will not be able to return a message until the brokers have written the transaction markers that record the result of the transaction (*committed* or *aborted*).

```
# ...
enable.auto.commit=false
isolation.level=read_committed (1)
# ...
```

(1) Set to **read_committed** so that only committed messages are read by the consumer.

12.4.2.6. Recovering from failure to avoid data loss

Use the **session.timeout.ms** and **heartbeat.interval.ms** properties to configure the time taken to check and recover from consumer failure within a consumer group.

The **session.timeout.ms** property specifies the maximum amount of time in milliseconds a consumer within a consumer group can be out of contact with a broker before being considered inactive and a *rebalancing* is triggered between the active consumers in the group. When the group rebalances, the partitions are reassigned to the members of the group.

The **heartbeat.interval.ms** property specifies the interval in milliseconds between *heartbeat* checks to the consumer group coordinator to indicate that the consumer is active and connected. The heartbeat interval must be lower, usually by a third, than the session timeout interval.

If you set the **session.timeout.ms** property lower, failing consumers are detected earlier, and rebalancing can take place quicker. However, take care not to set the timeout so low that the broker fails to receive a heartbeat in time and triggers an unnecessary rebalance.

Decreasing the heartbeat interval reduces the chance of accidental rebalancing, but more frequent heartbeats increases the overhead on broker resources.

12.4.2.7. Managing offset policy

Use the **auto.offset.reset** property to control how a consumer behaves when no offsets have been committed, or a committed offset is no longer valid or deleted.

Suppose you deploy a consumer application for the first time, and it reads messages from an existing topic. Because this is the first time the **group.id** is used, the **__consumer_offsets** topic does not contain any offset information for this application. The new application can start processing all existing messages from the start of the log or only new messages. The default reset value is **latest**, which starts at the end of the partition, and consequently means some messages are missed. To avoid data loss, but increase the amount of processing, set **auto.offset.reset** to **earliest** to start at the beginning of the partition.

Also consider using the **earliest** option to avoid messages being lost when the offsets retention period (**offsets.retention.minutes**) configured for a broker has ended. If a consumer group or standalone consumer is inactive and commits no offsets during the retention period, previously committed offsets are deleted from **__consumer_offsets**.

```
# ...
heartbeat.interval.ms=3000 1
session.timeout.ms=10000 2
auto.offset.reset=earliest 3
# ...
```

1. Adjust the heartbeat interval lower according to anticipated rebalances.
2. If no heartbeats are received by the Kafka broker before the timeout duration expires, the consumer is removed from the consumer group and a rebalance is initiated. If the broker configuration has a **group.min.session.timeout.ms** and **group.max.session.timeout.ms**, the session timeout value must be within that range.
3. Set to **earliest** to return to the start of a partition and avoid data loss if offsets were not committed.

If the amount of data returned in a single fetch request is large, a timeout might occur before the consumer has processed it. In this case, you can lower **max.partition.fetch.bytes** or increase **session.timeout.ms**.

12.4.2.8. Minimizing the impact of rebalances

The rebalancing of a partition between active consumers in a group is the time it takes for:

- Consumers to commit their offsets

- The new consumer group to be formed
- The group leader to assign partitions to group members
- The consumers in the group to receive their assignments and start fetching

Clearly, the process increases the downtime of a service, particularly when it happens repeatedly during a rolling restart of a consumer group cluster.

In this situation, you can use the concept of *static membership* to reduce the number of rebalances. Rebalancing assigns topic partitions evenly among consumer group members. Static membership uses persistence so that a consumer instance is recognized during a restart after a session timeout.

The consumer group coordinator can identify a new consumer instance using a unique id that is specified using the **group.instance.id** property. During a restart, the consumer is assigned a new member id, but as a static member it continues with the same instance id, and the same assignment of topic partitions is made.

If the consumer application does not make a call to poll at least every **max.poll.interval.ms** milliseconds, the consumer is considered to be failed, causing a rebalance. If the application cannot process all the records returned from poll in time, you can avoid a rebalance by using the **max.poll.interval.ms** property to specify the interval in milliseconds between polls for new messages from a consumer. Or you can use the **max.poll.records** property to set a maximum limit on the number of records returned from the consumer buffer, allowing your application to process fewer records within the **max.poll.interval.ms** limit.

```
# ...
group.instance.id=UNIQUE-ID 1
max.poll.interval.ms=300000 2
max.poll.records=500 3
# ...
```

1. The unique instance id ensures that a new consumer instance receives the same assignment of topic partitions.
2. Set the interval to check the consumer is continuing to process messages.
3. Sets the number of processed records returned from the consumer.

12.5. 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 Designating AMQ Streams administrators in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

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. COMMON CONFIGURATION PROPERTIES

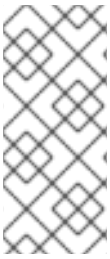
Common configuration properties apply to more than one resource.

B.1.1. replicas

Use the **replicas** property to configure replicas.

The type of replication depends on the resource.

- **KafkaTopic** uses a replication factor to configure the number of replicas of each partition within a Kafka cluster.
- Kafka components use replicas to configure the number of pods in a deployment to provide better availability and scalability.



NOTE

When running a Kafka component on OpenShift it may not be necessary to run multiple replicas for high availability. When the node where the component is deployed crashes, OpenShift will automatically reschedule the Kafka component pod to a different node. However, running Kafka components with multiple replicas can provide faster failover times as the other nodes will be up and running.

B.1.2. bootstrapServers

Use the **bootstrapServers** property to configure a list of bootstrap servers.

The bootstrap server lists can refer to Kafka clusters that are not deployed in the same OpenShift cluster. They can also refer to a Kafka cluster not deployed by AMQ Streams.

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 number. 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 with a Kafka cluster not managed by AMQ Streams, you can specify the bootstrap servers list according to the configuration of the given cluster.

B.1.3. ssl

Use the three allowed **ssl** configuration options for client connection using a specific *cipher suite* for a TLS version. A cipher suite combines algorithms for secure connection and data transfer.

You can also configure the **ssl.endpoint.identification.algorithm** property to enable or disable hostname verification.

Example SSL configuration

```
# ...
spec:
```

```
  config:
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (1)
    ssl.enabled.protocols: "TLSv1.2" (2)
    ssl.protocol: "TLSv1.2" (3)
    ssl.endpoint.identification.algorithm: HTTPS (4)
  # ...
```

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**.
4. Hostname verification is enabled by setting to **HTTPS**. An empty string disables the verification.

B.1.4. trustedCertificates

Having set **tls** to configure TLS encryption, use the **trustedCertificates** property to provide a list of secrets with key names under which the certificates are stored in X.509 format.

You can use the secrets created by the Cluster Operator for the Kafka cluster, or you can create your own TLS certificate file, then create a **Secret** from the file:

```
oc create secret generic MY-SECRET \
--from-file=MY-TLS-CERTIFICATE-FILE.crt
```

Example TLS encryption configuration

```
tls:
  trustedCertificates:
    - secretName: my-cluster-cluster-cert
      certificate: ca.crt
    - secretName: my-cluster-cluster-cert
      certificate: ca2.crt
```

If certificates are stored in the same secret, it can be listed multiple times.

If you want to enable TLS, but use the default set of public certification authorities shipped with Java, you can specify **trustedCertificates** as an empty array:

Example of enabling TLS with the default Java certificates

```
tls:
  trustedCertificates: []
```

For information on configuring TLS client authentication, see **KafkaClientAuthenticationTls** schema reference.

B.1.5. resources

You request CPU and memory resources for components. Limits specify the maximum resources that can be consumed by a given container.

Resource requests and limits for the Topic Operator and User Operator are set in the **Kafka** resource.

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.

B.1.6. image

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.

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

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaMirrorMaker2.spec**
- **KafkaBridge.spec**

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.

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 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:1.6.7** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration

environment variable from the Cluster Operator configuration.

2. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.6.7** 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-26-rhel7:1.6.7** 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-26-rhel7:1.6.7** 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-rhel7:1.6.7** 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:1.6.7** container image.

- For Kafka broker initializer:

1. **registry.redhat.io/amq7/amq-streams-rhel7-operator:1.6.7** container image.

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:
    # ...
```

B.1.7. livenessProbe and readinessProbe 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**

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.

B.1.8. 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 in the *Deploying and Upgrading AMQ Streams on OpenShift* guide.

B.1.9. jvmOptions

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

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

- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaMirrorMaker.spec**
- **KafkaMirrorMaker2.spec**
- **KafkaBridge.spec**

Only the following JVM options are supported:

-Xms

Configures the minimum initial allocation heap size when the JVM starts.

-Xmx

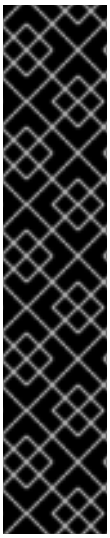
Configures the maximum heap size.

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



IMPORTANT

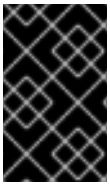
Setting **-Xmx** explicitly requires some care:

- 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 an 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.

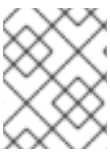
Setting the same value for initial (**-Xms**) and maximum (**-Xmx**) heap sizes avoids the JVM having to allocate memory after startup, at the cost of possibly allocating more heap than is really needed. For Kafka and ZooKeeper pods such allocation could cause unwanted latency. For Kafka Connect avoiding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

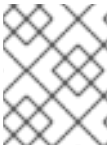
Example showing the use of the -XX object

```
jvmOptions:
  "-XX":
    "UseG1GC": true
```

```
    "MaxGCPauseMillis": 20
    "InitiatingHeapOccupancyPercent": 35
    "ExplicitGCInvokesConcurrent": true
```

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**) are specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** is used.

B.1.10. Garbage collector logging

The **jvmOptions** property 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
# ...
```

B.2. 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.3. KAFKASPEC SCHEMA REFERENCE

Used in: **Kafka**

Property	Description
kafka	Configuration of the Kafka cluster.
KafkaClusterSpec	

Property	Description
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	
cruiseControl	Configuration for Cruise Control deployment. Deploys a Cruise Control instance when specified.
CruiseControlSpec	
kafkaExporter	Configuration of the Kafka Exporter. Kafka Exporter can provide additional metrics, for example lag of consumer group at topic/partition.
KafkaExporterSpec	
maintenanceTimeWindows	A list of time windows for maintenance tasks (that is, certificates renewal). Each time window is defined by a cron expression.
string array	

B.4. KAFKACLUSTERSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Configures a Kafka cluster.

B.4.1. listeners

Use the **listeners** property to configure listeners to provide access to Kafka brokers.

Example configuration of a plain (unencrypted) listener without authentication

-

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
    # ...
  zookeeper:
    # ...
```

B.4.2. config

Use the **config** properties to configure Kafka brokers 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, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **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**

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 supported options are passed to Kafka.

There are exceptions to the forbidden options. For client connection using a specific *cipher suite* for a TLS version, you can configure allowed **ssl** properties. You can also configure the **zookeeper.connection.timeout.ms** property to set the maximum time allowed for establishing a ZooKeeper connection.

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"
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
      zookeeper.connection.timeout.ms: 6000
    # ...
```

Property	Description
replicas	The number of pods in the cluster.
integer	
image	The docker image for the pods. The default value depends on the configured **Kafka.spec.kafka.version**.

Property	Description
string	
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 **GenericKafkaListener** array or **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].
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, zookeeper.ssl, zookeeper.clientCnxnSocket, 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,cruise.control.metrics.topic.auto.create.retries, cruise.control.metrics.topic.auto.create.timeout.ms).
rack **Rack**	Configuration of the **broker.rack** broker config.
brokerRackInitImage string	The image of the init container used for initializing the **broker.rack**.

Property	Description
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 **KafkaJmxOptions**	JMX Options for Kafka brokers.
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.
logging **InlineLogging**, **ExternalLogging**	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].
tlsSidecar **TlsSidecar**	**The property tlsSidecar has been deprecated.** TLS sidecar configuration.

Property	Description
template **KafkaClusterTemplate**	Template for Kafka cluster resources. The template allows users to specify how are the **StatefulSet**, **Pods** and **Services** generated.
version string	The kafka broker version. Defaults to 2.6.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.

B.5. 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 integer	Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'.
sizeLimit string	When type=ephemeral, defines the total amount of local storage required for this EmptyDir volume (for example 1Gi).
type string	Must be **ephemeral**.

B.6. 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 string	Must be **persistent-claim**.

Property	Description
size string	When type=persistent-claim, defines the size of the persistent volume claim (i.e 1Gi). Mandatory when type=persistent-claim.
selector map	Specifies a specific persistent volume to use. It contains key:value pairs representing labels for selecting such a volume.
deleteClaim boolean	Specifies if the persistent volume claim has to be deleted when the cluster is un-deployed.
class string	The storage class to use for dynamic volume allocation.
id integer	Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'.
overrides **PersistentClaimStorageOverride** array	Overrides for individual brokers. The **overrides** field allows to specify a different configuration for different brokers.

B.7. 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.8. 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	Must be **jbod**.
string	
volumes	List of volumes as Storage objects representing the JBOD disks array.
EphemeralStorage, **PersistentClaimStorage** array	

B.9. GENERICKAFKALISTENER SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Configures listeners to connect to Kafka brokers within and outside OpenShift.

You configure the listeners in the **Kafka** resource.

Example Kafka resource showing listener configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    #...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external1
        port: 9094
        type: route
        tls: true
      - name: external2
        port: 9095
        type: ingress
        tls: false
        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
  #...
```

B.9.1. listeners

You configure Kafka broker listeners using the **listeners** property in the **Kafka** resource. Listeners are defined as an array.

Example listener configuration

```
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
```

The name and port must be unique within the Kafka cluster. The name can be up to 25 characters long, comprising lower-case letters and numbers. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX.

By specifying a unique name and port for each listener, you can configure multiple listeners.

B.9.2. type

The type is set as **internal**, or for external listeners, as **route**, **loadbalancer**, **nodeport** or **ingress**.

internal

You can configure internal listeners with or without encryption using the **tls** property.

Example internal listener configuration

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
```

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

route

Configures an external listener to expose 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. The client connects on port 443, the default router port, but traffic is then routed to the port you configure, which is **9094** in this example.

Example route listener configuration

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external1
        port: 9094
        type: route
        tls: true
    #...
```

ingress

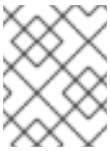
Configures an external listener to expose Kafka using Kubernetes **Ingress** and the NGINX Ingress Controller for Kubernetes.
A dedicated **Ingress** resource is created for every Kafka broker pod. An additional **Ingress** resource is created to serve as a Kafka bootstrap address. Kafka clients can use these **Ingress** resources to connect to Kafka on port 443. The client connects on port 443, the default controller port, but traffic is then routed to the port you configure, which is **9095** in the following example.

You must specify the hostnames used by the bootstrap and per-broker services using **GenericKafkaListenerConfigurationBootstrap** and **GenericKafkaListenerConfigurationBroker** properties.

Example ingress listener configuration

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external2
        port: 9095
        type: ingress
        tls: false
        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
#...
```



NOTE

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

loadbalancer

Configures an external listener to expose Kafka **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 the specified port number, which is port **9094** in the following example.

You can use the **loadBalancerSourceRanges** property to configure source ranges to restrict access to the specified IP addresses.

Example loadbalancer listener configuration

```
#...
spec:
  kafka:
    #...
    listeners:
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        configuration:
          loadBalancerSourceRanges:
            - 10.0.0.0/8
            - 88.208.76.87/32
    #...
```

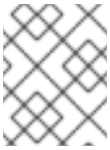
nodeport

Configures an external listener to expose Kafka using **NodePort** type **Services**.
Kafka clients connect directly to the nodes of OpenShift. 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. You can use **preferredNodePortAddressType** property to configure the first address type checked as the node address.

Example nodeport listener configuration

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external4
        port: 9095
        type: nodeport
        tls: false
        configuration:
          preferredNodePortAddressType: InternalDNS
    #...
```



NOTE

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

B.9.3. port

The port number is the port used in the Kafka cluster, which might not be the same port used for access by a client.

- **loadbalancer** listeners use the specified port number, as do **internal** listeners
- **ingress** and **route** listeners use port 443 for access
- **nodeport** listeners use the port number assigned by OpenShift

For client connection, use the address and port for the bootstrap service of the listener. You can retrieve this from the status of the **Kafka** resource.

Example command to retrieve the address and port for client connection

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?
(@.type=="external")].bootstrapServers}{"\n"}'
```

NOTE

Listeners cannot be configured to use the ports set aside for interbroker communication (9091) and metrics (9404).

B.9.4. tls

The TLS property is required.

By default, TLS encryption is not enabled. To enable it, set the **tls** property to **true**.

TLS encryption is always used with **route** listeners.

B.9.5. authentication

Authentication for the listener can be specified as:

- Mutual TLS (**tls**)
- SCRAM-SHA-512 (**scram-sha-512**)
- Token-based OAuth 2.0 (**oauth**).

B.9.6. networkPolicyPeers

Use **networkPolicyPeers** to configure network policies that restrict access to a listener at the network level. The following example shows a **networkPolicyPeers** configuration for a **plain** and a **tls** listener.

```
listeners:
  #...
  - name: plain
    port: 9092
    type: internal
    tls: true
    authentication:
      type: scram-sha-512
    networkPolicyPeers:
      - podSelector:
          matchLabels:
            app: kafka-sasl-consumer
      - podSelector:
          matchLabels:
            app: kafka-sasl-producer
  - name: tls
    port: 9093
    type: internal
    tls: true
    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.

Backwards compatibility with KafkaListeners

GenericKafkaListener replaces the **KafkaListeners** schema, which is now deprecated.

To convert the listeners configured using the **KafkaListeners** schema into the format of the **GenericKafkaListener** schema, with backwards compatibility, use the following names, ports and types:

```
listeners:
  #...
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: tls
    port: 9093
    type: internal
    tls: true
  - name: external
    port: 9094
    type: EXTERNAL-LISTENER-TYPE (1)
    tls: true
# ...
```

(1) Options: **ingress**, **loadbalancer**, **nodeport**, **route**

Property	Description
name string	Name of the listener. The name will be used to identify the listener and the related OpenShift objects. The name has to be unique within given a Kafka cluster. The name can consist of lowercase characters and numbers and be up to 11 characters long.
port integer	Port number used by the listener inside Kafka. The port number has to be unique within a given Kafka cluster. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX. Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.
type string (one of [ingress, internal, route, loadbalancer, nodeport])	Type of the listener. Currently the supported types are **internal**, **route**, **loadbalancer**, **nodeport** and **ingress**. * **internal** type exposes Kafka internally only within the OpenShift cluster. * **route** type uses OpenShift Routes to expose Kafka. * **loadbalancer** type uses LoadBalancer type services to expose Kafka. * **nodeport** type uses NodePort type services to expose Kafka. * **ingress** type uses OpenShift Nginx Ingress to expose Kafka. .

Property	Description
tls	Enables TLS encryption on the listener. This is a required property.
boolean	
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	Additional listener configuration.
GenericKafkaListenerConfiguration	
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: **GenericKafkaListener**, **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: **GenericKafkaListener**, **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 string	Must be **scram-sha-512**.

B.12. KAFKALISTENERAUTHENTICATIONOAUTH SCHEMA REFERENCE

Used in: GenericKafkaListener, 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 boolean	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**.
checkAccessTokenType boolean	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**.
checkIssuer boolean	Enable or disable issuer checking. By default issuer is checked using the value configured by **validIssuerUri**. Default value is **true**.
clientId string	OAuth Client ID which the Kafka broker can use to authenticate against the authorization server and use the introspect endpoint URI.
clientSecret GenericSecretSource	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.
disableTlsHostnameVerification boolean	Enable or disable TLS hostname verification. Default value is **false**.

Property	Description
enableECDSA boolean	Enable or disable ECDSA support by installing BouncyCastle crypto provider. Default value is **false**.
fallbackUserNameClaim string	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.
fallbackUserNamePrefix string	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.
introspectionEndpointUri string	URI of the token introspection endpoint which can be used to validate opaque non-JWT tokens.
jwksEndpointUri string	URI of the JWKS certificate endpoint, which can be used for local JWT validation.
jwksExpirySeconds integer	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.
jwksMinRefreshPauseSeconds integer	The minimum pause between two consecutive refreshes. When an unknown signing key is encountered the refresh is scheduled immediately, but will always wait for this minimum pause. Defaults to 1 second.
jwksRefreshSeconds integer	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.

Property	Description
maxSecondsWithoutReauthentication integer	Maximum number of seconds the authenticated session remains valid without re-authentication. This enables Apache Kafka re-authentication feature, and causes sessions to expire when the access token expires. If the access token expires before max time or if max time is reached, the client has to re-authenticate, otherwise the server will drop the connection. Not set by default - the authenticated session does not expire when the access token expires.
tlsTrustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection to the OAuth server.
type string	Must be **oauth**.
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.

Property	Description
secretName	The name of the OpenShift Secret containing the secret value.
string	

B.14. CERTSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaAuthorizationKeycloak**, **KafkaBridgeTls**, **KafkaClientAuthenticationOAuth**, **KafkaConnectTls**, **KafkaListenerAuthenticationOAuth**, **KafkaMirrorMaker2Tls**, **KafkaMirrorMakerTls**

Property	Description
certificate	The name of the file certificate in the Secret.
string	
secretName	The name of the Secret containing the certificate.
string	

B.15. GENERICKAFKALISTENERCONFIGURATION SCHEMA REFERENCE

Used in: **GenericKafkaListener**

Configuration for Kafka listeners.

B.15.1. brokerCertChainAndKey

The **brokerCertChainAndKey** property is only used with listeners that have TLS encryption enabled. You can use the property to providing your own Kafka listener certificates.

Example configuration for a loadbalancer external listener with TLS encryption enabled

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

B.15.2. externalTrafficPolicy

The **externalTrafficPolicy** property is used with **loadbalancer** and **nodeport** listeners. When exposing Kafka outside of OpenShift you can choose **Local** or **Cluster**. **Local** avoids hops to other nodes and preserves the client IP, whereas **Cluster** does neither. The default is **Cluster**.

B.15.3. loadBalancerSourceRanges

The **loadBalancerSourceRanges** property is only used with **loadbalancer** listeners. When exposing Kafka outside of OpenShift use source ranges, in addition to labels and annotations, to customize how a service is created.

Example source ranges configured for a loadbalancer listener

```
listeners:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: false
    configuration:
      externalTrafficPolicy: Local
      loadBalancerSourceRanges:
        - 10.0.0.0/8
        - 88.208.76.87/32
      # ...
# ...
```

B.15.4. class

The **class** property is only used with **ingress** listeners.

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:
  #...
  - name: external
    port: 9094
    type: ingress
    tls: false
    configuration:
      class: nginx-internal
    # ...
# ...
```

B.15.5. preferredNodePortAddressType

The **preferredNodePortAddressType** property is only used with **nodeport** listeners.

Use the **preferredNodePortAddressType** 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:

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

Example of an external listener configured with a preferred node port address type

```
listeners:
  #...
  - name: external
    port: 9094
    type: nodeport
    tls: false
    configuration:
      preferredNodePortAddressType: InternalDNS
      # ...
# ...
```

B.15.6. useServiceDnsDomain

The **useServiceDnsDomain** property is only used with **internal** listeners. It defines whether the fully-qualified DNS names that include the cluster service suffix (usually **.cluster.local**) are used. With **useServiceDnsDomain** set as **false**, the advertised addresses are generated without the service suffix; for example, **my-cluster-kafka-0.my-cluster-kafka-brokers.myproject.svc**. With **useServiceDnsDomain** set as **true**, the advertised addresses are generated with the service suffix; for example, **my-cluster-kafka-0.my-cluster-kafka-brokers.myproject.svc.cluster.local**. Default is **false**.

Example of an internal listener configured to use the Service DNS domain

```
listeners:
  #...
  - name: plain
    port: 9092
    type: internal
    tls: false
    configuration:
      useServiceDnsDomain: true
      # ...
# ...
```

If your OpenShift cluster uses a different service suffix than **.cluster.local**, you can configure the suffix using the **KUBERNETES_SERVICE_DNS_DOMAIN** environment variable in the Cluster Operator configuration. See Section 5.1.1, “Cluster Operator configuration” for more details.

Property	Description
brokerCertChainAndKey **CertAndKeySecretSource**	Reference to the **Secret** which holds the certificate and private key pair which will be used for this listener. The certificate can optionally contain the whole chain. This field can be used only with listeners with enabled TLS encryption.
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.This field can be used only with **loadbalancer** or **nodeport** type listener.
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://v1-17.docs.kubernetes.io/docs/tasks/access-application-cluster/configure-cloud-provider-firewall/. This field can be used only with **loadbalancer** type listener.
bootstrap **GenericKafkaListenerConfigurationBootstrap**	Bootstrap configuration.
brokers **GenericKafkaListenerConfigurationBroker** array	Per-broker configurations.
class string	Configures the **Ingress** class that defines which **Ingress** controller will be used. If not set, the **Ingress** class is set to **nginx**. This field can be used only with **ingress** type listener.

Property	Description
preferredNodePortAddressType string (one of [ExternalDNS, ExternalIP, Hostname, InternalIP, InternalDNS])	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.This field can be used only with **nodeport** type listener..
useServiceDnsDomain boolean	Configures whether the OpenShift service DNS domain should be used or not. If set to **true**, the generated addresses with contain the service DNS domain suffix (by default **.cluster.local**, can be configured using environment variable **KUBERNETES_SERVICE_DNS_DOMAIN**). Defaults to **false**.This field can be used only with **internal** type listener.

B.16. CERTANDKEYSECRETSOURCE SCHEMA REFERENCE

Used in: **GenericKafkaListenerConfiguration**, **IngressListenerConfiguration**, **KafkaClientAuthenticationTls**, **KafkaListenerExternalConfiguration**, **NodePortListenerConfiguration**, **TlsListenerConfiguration**

Property	Description
certificate string	The name of the file certificate in the Secret.
key string	The name of the private key in the Secret.
secretName string	The name of the Secret containing the certificate.

B.17. GENERICKAFKALISTENERCONFIGURATIONBOOTSTRAP SCHEMA REFERENCE

Used in: **GenericKafkaListenerConfiguration**

Configures bootstrap service overrides for external listeners.

Broker service equivalents of **nodePort**, **host**, **loadBalancerIP** and **annotations** properties are configured in the **GenericKafkaListenerConfigurationBroker** schema.

B.17.1. alternativeNames

You can specify alternative names for the bootstrap service. The names are added to the broker certificates and can be used for TLS hostname verification. The **alternativeNames** property is applicable to all types of external listeners.

Example of an external route listener configured with an additional bootstrap address

```
listeners:
  #...
  - name: external
    port: 9094
    type: route
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        alternativeNames:
          - example.hostname1
          - example.hostname2
# ...
```

B.17.2. host

The **host** property is used with **route** and **ingress** listeners to specify the hostnames used by the bootstrap and per-broker services.

A **host** property value is mandatory for **ingress** listener configuration, as the Ingress controller does not assign any hostnames automatically. 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 host configuration for an ingress listener

```
listeners:
  #...
  - name: external
    port: 9094
    type: ingress
    tls: true
    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
# ...
```

By default, **route** listener hosts are automatically assigned by OpenShift. However, you can override the assigned route hosts by specifying hosts.

AMQ Streams does not perform any validation that the requested hosts are available. You must ensure that they are free and can be used.

Example of host configuration for a route listener

```
# ...
listeners:
  #...
  - name: external
    port: 9094
    type: route
    tls: true
    authentication:
      type: tls
    configuration:
      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
# ...
```

B.17.3. nodePort

By default, the port numbers used for the bootstrap and broker services are automatically assigned by OpenShift. You can override the assigned node ports for **nodeport** listeners by specifying the requested port numbers.

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:
  #...
  - name: external
    port: 9094
```

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

B.17.4. loadBalancerIP

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:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    configuration:
      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
# ...
```

B.17.5. annotations

Use the **annotations** property to add annotations to **loadbalancer**, **nodeport** or **ingress** listeners. 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 annotations

```
# ...
listeners:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-bootstrap.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      brokers:
      - broker: 0
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-0.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 1
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-1.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 2
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-2.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
# ...
```

Property	Description
alternativeNames string array	Additional alternative names for the bootstrap service. The alternative names will be added to the list of subject alternative names of the TLS certificates.
host string	The bootstrap host. This field will be used in the Ingress resource or in the Route resource to specify the desired hostname. This field can be used only with **route** (optional) or **ingress** (required) type listeners.
nodePort integer	Node port for the bootstrap service. This field can be used only with **nodeport** type listener.
loadBalancerIP	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.This field can be used only with **loadbalancer** type listener.

Property	Description
string	
annotations map	Annotations that will be added to the **Ingress** or **Service** resource. You can use this field to configure DNS providers such as External DNS. This field can be used only with **loadbalancer**, **nodeport**, or **ingress** type listeners.

B.18. GENERICKAFKALISTENERCONFIGURATIONBROKER SCHEMA REFERENCE

Used in: **GenericKafkaListenerConfiguration**

Configures broker service overrides for external listeners.

You can see example configuration for the **nodePort**, **host**, **loadBalancerIP** and **annotations** properties in the **GenericKafkaListenerConfigurationBootstrap** schema, which configures bootstrap service overrides for external listeners.

Advertised addresses for brokers

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 specify a broker ID and customize the advertised hostname and port in the **configuration** 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 route listener configured with overrides for advertised addresses

```
listeners:
  #...
  - name: external
    port: 9094
    type: route
    tls: true
    authentication:
      type: tls
    configuration:
      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
# ...
```

Property	Description
broker integer	ID of the kafka broker (broker identifier). Broker IDs start from 0 and correspond to the number of broker replicas.
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	The broker host. This field will be used in the Ingress resource or in the Route resource to specify the desired hostname. This field can be used only with **route** (optional) or **ingress** (required) type listeners.
nodePort integer	Node port for the per-broker service. This field can be used only with **nodeport** type listener.
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.This field can be used only with **loadbalancer** type listener.
annotations map	Annotations that will be added to the **Ingress** or **Service** resource. You can use this field to configure DNS providers such as External DNS. This field can be used only with **loadbalancer**, **nodeport**, or **ingress** type listeners.

B.19. KAFKALISTENERS SCHEMA REFERENCE

The type KafkaListeners has been deprecated. Please use **GenericKafkaListener** instead.

Used in: **KafkaClusterSpec**

Refer to previous documentation for example configuration.

Property	Description
plain	Configures plain listener on port 9092.
KafkaListenerPlain	
tls	Configures TLS listener on port 9093.
KafkaListenerTls	
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].
KafkaListenerExternalRoute, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalIngress**	

B.20. 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.21. KAFKALISTENERTLS SCHEMA REFERENCE

Used in: **KafkaListeners**

Property	Description
authentication	Authentication configuration for this listener. The

Property	Description
	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.22. 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.23. 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	

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 **RouteListenerOverride**	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.

B.24. ROUTELISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalRoute**

Property	Description
bootstrap **RouteListenerBootstrapOverride**	External bootstrap service configuration.
brokers **RouteListenerBrokerOverride** array	External broker services configuration.

B.25. 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.26. 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.27. 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.28. 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**.
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.29. LOADBALANCERLISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalLoadBalancer**

Property	Description
bootstrap	External bootstrap service configuration.

Property	Description
LoadBalancerListenerBootstrapOverride	
brokers	External broker services configuration.
LoadBalancerListenerBrokerOverride array	

B.30. LOADBALANCERLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **LoadBalancerListenerOverride**

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	
loadBalancerIP	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.
string	

B.31. LOADBALANCERLISTENERBROKEROVERRIDE SCHEMA REFERENCE

Used in: **LoadBalancerListenerOverride**

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
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.32. 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.

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.
tls boolean	Enables TLS encryption on the listener. By default set to **true** for enabled TLS encryption.

B.33. NODEPORTLISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalNodePort**

Property	Description
bootstrap **NodePortListenerBootstrapOverride**	External bootstrap service configuration.
brokers **NodePortListenerBrokerOverride** array	External broker services configuration.

B.34. NODEPORTLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **NodePortListenerOverride**

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.
nodePort	Node port for the bootstrap service.

Property	Description
integer	

B.35. 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	
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.36. 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	

Property	Description
preferredAddressType string (one of [ExternalDNS, ExternalIP, Hostname, InternalIP, InternalDNS])	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..

B.37. 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 string	Must be **ingress**.
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.

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.38. 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.39. 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.40. INGRESSLISTENERBROKERCONFIGURATION SCHEMA REFERENCE

Used in: **IngressListenerConfiguration**

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	
host	Host for the broker ingress. This field will be used in the Ingress resource.
string	
dnsAnnotations	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.
map	

B.41. KAFKAAUTHORIZATIONSIMPLE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Simple authorization in AMQ Streams uses the **AclAuthorizer** 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.

Configure the **Kafka** custom resource to use simple authorization. Set the **type** property in the **authorization** section to the value **simple**, and configure a list of super users.

Access rules are configured for the **KafkaUser**, as described in the ACLRule schema reference.

B.41.1. superUsers

A list of user principals treated as super users, so that they are always allowed without querying ACL rules. For more information see Kafka authorization.

An example of simple authorization configuration

```
authorization:
  type: simple
  superUsers:
```

```
- CN=client_1
- user_2
- CN=client_3
```

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.

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.42. KAFKAAUTHORIZATIONOPA SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

To use Open Policy Agent authorization, set the **type** property in the **authorization** section to the value **opa**, and configure OPA properties as required.

B.42.1. 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.**

B.42.2. 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.

B.42.3. initialCacheCapacity

Initial capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request. Defaults to **5000**.

B.42.4. maximumCacheSize

Maximum capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request. Defaults to **50000**.

B.42.5. 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).

B.42.6. 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 Kafka authorization.

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	Must be **opa**.
string	
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. This option is required.
string	
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.
boolean	
initialCacheCapacity	Initial capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request Defaults to **5000**.
integer	

Property	Description
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.43. 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 string	Must be **keycloak**.
clientId string	OAuth Client ID which the Kafka client can use to authenticate against the OAuth server and use the token endpoint URI.
tokenEndpointUri string	Authorization server token endpoint URI.
tlsTrustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection to the OAuth server.
disableTlsHostnameVerification boolean	Enable or disable TLS hostname verification. Default value is **false**.

Property	Description
delegateToKafkaAcls boolean	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**.
grantsRefreshPeriodSeconds integer	The time between two consecutive grants refresh runs in seconds. The default value is 60.
grantsRefreshPoolSize integer	The number of threads to use to refresh grants for active sessions. The more threads, the more parallelism, so the sooner the job completes. However, using more threads places a heavier load on the authorization server. The default value is 5.
superUsers string array	List of super users. Should contain list of user principals which should get unlimited access rights.

B.44. RACK SCHEMA REFERENCE

Used in: **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**

Property	Description
topologyKey string	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.

B.45. PROBE SCHEMA REFERENCE

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaExporterSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **TlsSidecar**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Property	Description
failureThreshold integer	Minimum consecutive failures for the probe to be considered failed after having succeeded. Defaults to 3. Minimum value is 1.
initialDelaySeconds	

Property	Description
	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.46. 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	
gcLoggingEnabled	Specifies whether the Garbage Collection logging is enabled. The default is false.
boolean	
javaSystemProperties	A map of additional system properties which will be passed using the **-D** option to the JVM.
SystemProperty array	

B.47. SYSTEMPROPERTY SCHEMA REFERENCE

Used in: **JvmOptions**

Property	Description
name	The system property name.
string	
value	The system property value.
string	

B.48. KAFKAJMXOPTIONS SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
authentication	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].
KafkaJmxAuthenticationPassword	

B.49. 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.50. 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.51. 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.52. 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**.

Property	Description
string (one of [emerg, debug, crit, err, alert, warning, notice, info])	
readinessProbe **Probe**	Pod readiness checking.
resources ResourceRequirements	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.

B.53. KAFKACLUSTERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
statefulset **StatefulSetTemplate**	Template for Kafka **StatefulSet**.
pod **PodTemplate**	Template for Kafka **Pods**.
bootstrapService **ResourceTemplate**	Template for Kafka bootstrap **Service**.
brokersService **ResourceTemplate**	Template for Kafka broker **Service**.
externalBootstrapService **ExternalServiceTemplate**	Template for Kafka external bootstrap **Service**.
perPodService **ExternalServiceTemplate**	Template for Kafka per-pod **Services** used for access from outside of OpenShift.
externalBootstrapRoute	Template for Kafka external bootstrap **Route**.

Property	Description
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	**The property tlsSidecarContainer has been deprecated.** Template for the Kafka broker TLS sidecar container.
ContainerTemplate	
initContainer	Template for the Kafka init container.
ContainerTemplate	

B.54. STATEFULSETTEMPLATE SCHEMA REFERENCE

Used in: **KafkaClusterTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata	Metadata applied to the resource.
MetadataTemplate	

Property	Description
podManagementPolicy	PodManagementPolicy which will be used for this StatefulSet. Valid values are **Parallel** and **OrderedReady**. Defaults to **Parallel**.
string (one of [OrderedReady, Parallel])	

B.55. METADATATEMPLATE SCHEMA REFERENCE

Used in: **ExternalServiceTemplate**, **PodDisruptionBudgetTemplate**, **PodTemplate**, **ResourceTemplate**, **StatefulSetTemplate**

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.

Property	Description
labels	Labels added to the resource template. Can be applied to different resources such as **StatefulSets**, **Deployments**, **Pods**, and **Services**.
map	
annotations	Annotations added to the resource template. Can be applied to different resources such as **StatefulSets**, **Deployments**, **Pods**, and **Services**.
map	

B.56. PODTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Example PodTemplate configuration

-

```
# ...
template:
  pod:
    metadata:
      labels:
        label1: value1
      annotations:
        anno1: value1
    imagePullSecrets:
      - name: my-docker-credentials
    securityContext:
      runAsUser: 1000001
      fsGroup: 0
    terminationGracePeriodSeconds: 120
# ...
```

B.56.1. hostAliases

Use the **hostAliases** property to a specify a list of hosts and IP addresses, which are injected into the **/etc/hosts** file of the pod.

This configuration is especially useful for Kafka Connect or MirrorMaker when a connection outside of the cluster is also requested by users.

Example hostAliases configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
#...
spec:
  # ...
  template:
    pod:
      hostAliases:
      - ip: "192.168.1.86"
        hostnames:
        - "my-host-1"
        - "my-host-2"
      #...
```

Property	Description
metadata	Metadata applied to the resource.
MetadataTemplate	

Property	Description
imagePullSecrets LocalObjectReference array	List of references to secrets in the same namespace to use for pulling any of the images used by this Pod. When the **STRIMZI_IMAGE_PULL_SECRETS** environment variable in Cluster Operator and the **imagePullSecrets** option are specified, only the **imagePullSecrets** variable is used and the **STRIMZI_IMAGE_PULL_SECRETS** variable is ignored. See external documentation of core/v1 localobjectreference.
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 to longer than the expected cleanup time for your process. Value must be a non-negative integer. A zero value indicates delete immediately. 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. Defaults to 30 seconds.
affinity Affinity	The pod's affinity rules. See external documentation of core/v1 affinity.
tolerations Toleration array	The pod's tolerations. See external documentation of core/v1 toleration.
priorityClassName string	The name of the priority class used to assign priority to the pods. For more information about priority classes, see Pod Priority and Preemption.
schedulerName string	The name of the scheduler used to dispatch this **Pod**. If not specified, the default scheduler will be used.
hostAliases HostAlias array	The pod's HostAliases. HostAliases is an optional list of hosts and IPs that will be injected into the pod's hosts file if specified. See external documentation of core/v1 HostAlias.

B.57. RESOURCETEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **KafkaUserTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata	Metadata applied to the resource.
MetadataTemplate	

B.58. EXTERNALSERVICETEMPLATE SCHEMA REFERENCE

Used in: **KafkaClusterTemplate**

When exposing Kafka outside of OpenShift using loadbalancers or node ports, you can use properties, in addition to labels and annotations, to customize how a Service is created.

An example showing customized external services

```
# ...
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
# ...
```

Property	Description
metadata	Metadata applied to the resource.
MetadataTemplate	

Property	Description
externalTrafficPolicy string (one of [Local, Cluster])	**The property externalTrafficPolicy has been deprecated.** 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	**The property loadBalancerSourceRanges has been deprecated.** 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://v1-17.docs.kubernetes.io/docs/tasks/access-application-cluster/configure-cloud-provider-firewall/.

B.59. PODDISRUPTIONBUDGETTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

AMQ Streams creates a **PodDisruptionBudget** for every new **StatefulSet** or **Deployment**. By default, pod disruption budgets only allow a single pod to be unavailable at a given time. You can increase the amount of unavailable pods allowed by changing the default value of the **maxUnavailable** property in the **PodDisruptionBudget.spec** resource.

An example of PodDisruptionBudget template

```
# ...
template:
    podDisruptionBudget:
        metadata:
            labels:
                key1: label1
                key2: label2
            annotations:
                key1: label1
                key2: label2
        maxUnavailable: 1
# ...
```

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.60. CONTAINERTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

You can set custom security context and environment variables for a container.

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 and a custom security context set for the Kafka broker containers:

```
# ...
template:
  kafkaContainer:
    env:
    - name: EXAMPLE_ENV_1
      value: example.env.one
    - name: EXAMPLE_ENV_2
      value: example.env.two
    securityContext:
      runAsUser: 2000
# ...
```

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.

Property	Description
env **ContainerEnvVar** array	Environment variables which should be applied to the container.
securityContext SecurityContext	Security context for the container. See external documentation of core/v1 securitycontext.

B.61. CONTAINERENVVAR SCHEMA REFERENCE

Used in: **ContainerTemplate**

Property	Description
name	The environment variable key.
string	
value	The environment variable value.
string	

B.62. 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**	
config	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).
map	

Property	Description
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.
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.63. ZOOKEEPERCLUSTERTEMPLATE SCHEMA REFERENCE

Used in: **ZookeeperClusterSpec**

Property	Description
statefulset	Template for ZooKeeper **StatefulSet**.
StatefulSetTemplate	
pod	Template for ZooKeeper **Pods**.
PodTemplate	
clientService	Template for ZooKeeper client **Service**.
ResourceTemplate	
nodesService	Template for ZooKeeper nodes **Service**.
ResourceTemplate	
persistentVolumeClaim	Template for all ZooKeeper **PersistentVolumeClaims**.
ResourceTemplate	
podDisruptionBudget	Template for ZooKeeper **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	
zookeeperContainer	Template for the ZooKeeper container.
ContainerTemplate	
tlsSidecarContainer	**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.
ContainerTemplate	

B.64. TOPICOPERATORSPEC SCHEMA REFERENCE

The type TopicOperatorSpec has been deprecated. Please use **EntityTopicOperatorSpec** instead.

Used in: **KafkaSpec**

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	
affinity	Pod affinity rules. See external documentation of core/v1 affinity.
Affinity	
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	
tlsSidecar	TLS sidecar configuration.
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.

Property	Description
Probe	

B.65. ENTITYOPERATORSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
topicOperator **EntityTopicOperatorSpec**	Configuration of the Topic Operator.
userOperator **EntityUserOperatorSpec**	Configuration of the User Operator.
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.
tlsSidecar **TlsSidecar**	TLS sidecar configuration.
template **EntityOperatorTemplate**	Template for Entity Operator resources. The template allows users to specify how is the **Deployment** and **Pods** generated.

B.66. ENTITYTOPICOPERATORSPEC SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
watchedNamespace string	The namespace the Topic Operator should watch.

Property	Description
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	
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	

B.67. ENTITYUSEROPERATORSPEC SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
watchedNamespace	The namespace the User Operator should watch.

Property	Description
string	
image	The image to use for the User 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	
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	

B.68. ENTITYOPERATORTEMPLATE SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
deployment	Template for Entity Operator **Deployment**.
ResourceTemplate	

Property	Description
pod	Template for Entity Operator **Pods**.
PodTemplate	
tlsSidecarContainer	Template for the Entity Operator TLS sidecar container.
ContainerTemplate	
topicOperatorContainer	Template for the Entity Topic Operator container.
ContainerTemplate	
userOperatorContainer	Template for the Entity User Operator container.
ContainerTemplate	

B.69. 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	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.
boolean	
validityDays	The number of days generated certificates should be valid for. The default is 365.
integer	
renewalDays	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.
integer	

Property	Description
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.70. CRUISECONTROLSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
image string	The docker image for the pods.
tlsSidecar **TlsSidecar**	TLS sidecar configuration.
resources ResourceRequirements	CPU and memory resources to reserve for the Cruise Control container. See external documentation of core/v1 resourcerequirements.
livenessProbe **Probe**	Pod liveness checking for the Cruise Control container.
readinessProbe **Probe**	Pod readiness checking for the Cruise Control container.
jvmOptions **JvmOptions**	JVM Options for the Cruise Control container.
logging **InlineLogging**, **ExternalLogging**	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].
template **CruiseControlTemplate**	Template to specify how Cruise Control resources, **Deployments** and **Pods**, are generated.

Property	Description
brokerCapacity	The Cruise Control **brokerCapacity** configuration.
BrokerCapacity	
config	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.
map	
metrics	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
map	

B.71. CRUISECONTROLTEMPLATE SCHEMA REFERENCE

Used in: **CruiseControlSpec**

Property	Description
deployment	Template for Cruise Control **Deployment**.
ResourceTemplate	
pod	Template for Cruise Control **Pods**.
PodTemplate	
apiService	Template for Cruise Control API **Service**.
ResourceTemplate	
podDisruptionBudget	Template for Cruise Control **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	
cruiseControlContainer	Template for the Cruise Control container.

Property	Description
ContainerTemplate	
tlsSidecarContainer	Template for the Cruise Control TLS sidecar container.
ContainerTemplate	

B.72. 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.73. KAFKAEXPORTERSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
image	The docker image for the pods.
string	
groupRegex	Regular expression to specify which consumer groups to collect. Default value is **.***.
string	

Property	Description
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.74. KAFKAEXPORTERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaExporterSpec**

Property	Description
deployment **ResourceTemplate**	Template for Kafka Exporter **Deployment**.
pod **PodTemplate**	Template for Kafka Exporter **Pods**.
service	Template for Kafka Exporter **Service**.

Property	Description
ResourceTemplate	
container	Template for the Kafka Exporter container.
ContainerTemplate	

B.75. 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.76. CONDITION SCHEMA REFERENCE

Used in: **KafkaBridgeStatus**, **KafkaConnectorStatus**, **KafkaConnectS2IStatus**, **KafkaConnectStatus**, **KafkaMirrorMaker2Status**, **KafkaMirrorMakerStatus**, **KafkaRebalanceStatus**, **KafkaStatus**, **KafkaTopicStatus**, **KafkaUserStatus**

Property	Description
type	The unique identifier of a condition, used to distinguish between other conditions in the resource.
string	
status	The status of the condition, either True, False or Unknown.
string	
lastTransitionTime	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.

Property	Description
string	
reason	The reason for the condition's last transition (a single word in CamelCase).
string	
message	Human-readable message indicating details about the condition's last transition.
string	

B.77. LISTENERSTATUS SCHEMA REFERENCE

Used in: **KafkaStatus**

Property	Description
type	The type of the listener. Can be one of the following three types: **plain**, **tls**, and **external**.
string	
addresses	A list of the addresses for this listener.
ListenerAddress array	
bootstrapServers	A comma-separated list of **host:port** pairs for connecting to the Kafka cluster using this listener.
string	
certificates	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.
string array	

B.78. LISTENERADDRESS SCHEMA REFERENCE

Used in: **ListenerStatus**

Property	Description
host	The DNS name or IP address of the Kafka bootstrap service.
string	
port	The port of the Kafka bootstrap service.

Property	Description
integer	

B.79. KAFKACONNECT SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Connect cluster.
KafkaConnectSpec	
status	The status of the Kafka Connect cluster.
KafkaConnectStatus	

B.80. KAFKACONNECTSPEC SCHEMA REFERENCE

Used in: **KafkaConnect**

Configures a Kafka Connect cluster.

B.80.1. config

Use the **config** properties to configure Kafka options as keys.

Standard Apache Kafka Connect configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

Configuration options that cannot be configured relate to:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Listener / REST interface configuration
- Plugin path configuration

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.

There are exceptions to the forbidden options. You can use three allowed **ssl** configuration options for client connection using a specific *cipher suite* for a TLS version. A cipher suite combines algorithms for secure connection and data transfer. You can also configure the **ssl.endpoint.identification.algorithm** property to enable or disable hostname verification.

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"
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    ssl.endpoint.identification.algorithm: HTTPS
  # ...
```

For client connection using a specific *cipher suite* for a TLS version, you can configure allowed **ssl** properties. You can also configure the **ssl.endpoint.identification.algorithm** property to enable or disable hostname verification.

B.80.2. logging

Kafka Connect (and Kafka Connect with Source2Image support) has its own configurable loggers:

- **connect.root.logger.level**
- **log4j.logger.org.reflections**

Further loggers are added depending on the Kafka Connect plugins running.

Use a curl request to get a complete list of Kafka Connect loggers running from any Kafka broker pod:

```
curl -s http://<connect-cluster-name>-connect-api:8083/admin/loggers/
```

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**. For more information about log levels, see Apache logging services.

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
  # ...
```

Any available loggers that are not configured have their level set to **OFF**.

If Kafka Connect was deployed using the Cluster Operator, changes to Kafka Connect logging levels are applied dynamically.

If you use external logging, a rolling update is triggered when logging appenders are changed.

Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

Property	Description
replicas	The number of pods in the Kafka Connect group.
integer	
version	The Kafka Connect version. Defaults to 2.6.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.
string	
image	The docker image for the pods.
string	
bootstrapServers	Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>:<port>* pairs.
string	
tls	TLS configuration.
KafkaConnectTls	

Property	Description
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].
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).
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].

Property	Description
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.
clientRackInitImage string	The image of the init container used for initializing the **client.rack**.
rack **Rack**	Configuration of the node label which will be used as the client.rack consumer configuration.

B.81. KAFKACONNECTTLS SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

Configures TLS trusted certificates for connecting Kafka Connect to the cluster.

B.81.1. trustedCertificates

Provide a list of secrets using the **trustedCertificates** property.

Property	Description
trustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection.

B.82. 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.

B.82.1. certificateAndKey

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.

You can use the secrets created by the User Operator, or you can create your own TLS certificate file, with the keys used for authentication, then create a **Secret** from the file:

```
oc create secret generic MY-SECRET \
--from-file=MY-PUBLIC-TLS-CERTIFICATE-FILE.crt \
--from-file=MY-PRIVATE.key
```

NOTE

TLS client authentication can only be used with TLS connections.

Example TLS client authentication configuration

```
authentication:
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: my-public-tls-certificate-file.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.83. 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.

B.83.1. username

Specify the username in the **username** property.

B.83.2. passwordSecret

In the **passwordSecret** property, specify a link to a **Secret** containing the password.

You can use the secrets created by the User Operator.

If required, you can create a text file that contains the password, in cleartext, to use for authentication:

```
echo -n PASSWORD > MY-PASSWORD.txt
```

You can then create a **Secret** from the text file, setting your own field name (key) for the password:

```
oc create secret generic MY-CONNECT-SECRET-NAME --from-file=MY-PASSWORD-FIELD-NAME=./MY-PASSWORD.txt
```

Example Secret for SCRAM-SHA-512 client authentication for Kafka Connect

```
apiVersion: v1
kind: Secret
metadata:
  name: my-connect-secret-name
type: Opaque
data:
  my-connect-password-field: LFTIyFRFlMmU2N2Tm
```

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** property.

Example SASL-based SCRAM-SHA-512 client authentication configuration for Kafka Connect

```
authentication:
  type: scram-sha-512
  username: my-connect-username
  passwordSecret:
    secretName: my-connect-secret-name
    password: my-connect-password-field
```

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.84. 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.85. 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.

B.85.1. username

Specify the username in the **username** property.

B.85.2. passwordSecret

In the **passwordSecret** property, specify a link to a **Secret** containing the password.

You can use the secrets created by the User Operator.

If required, create a text file that contains the password, in cleartext, to use for authentication:

```
echo -n PASSWORD > MY-PASSWORD.txt
```

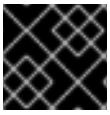
You can then create a **Secret** from the text file, setting your own field name (key) for the password:

```
oc create secret generic MY-CONNECT-SECRET-NAME --from-file=MY-PASSWORD-FIELD-NAME=./MY-PASSWORD.txt
```

Example Secret for PLAIN client authentication for Kafka Connect

```
apiVersion: v1
kind: Secret
metadata:
  name: my-connect-secret-name
type: Opaque
data:
  my-password-field-name: LFTIyFRFlMmU2N2Tm
```

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** property.

An example SASL based PLAIN client authentication configuration

```
authentication:
  type: plain
  username: my-connect-username
```

```
passwordSecret:
  secretName: my-connect-secret-name
  password: my-password-field-name
```

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.86. 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 one of the following options:

- 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**.

Property	Description
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 string	Must be **oauth**.

B.87. 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 string	Must be **jaeger**.

B.88. 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	
initContainer	Template for the Kafka init container.
ContainerTemplate	
podDisruptionBudget	Template for Kafka Connect **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

B.89. EXTERNALCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**

Configures external storage properties that define configuration options for Kafka Connect connectors.

You can mount ConfigMaps or Secrets into a Kafka Connect pod as environment variables or volumes. Volumes and environment variables are configured in the **externalConfiguration** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**.

When applied, the environment variables and volumes are available for use when developing your connectors.

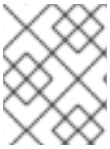
B.89.1. env

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.

Example Secret containing values for environment variables

```
apiVersion: v1
kind: Secret
metadata:
```

```
  name: aws-creds
type: Opaque
data:
  awsAccessKey: QUtJQVhYWFhYWFhYWFhYWFg=
  awsSecretAccessKey: Ylhsd1lYTnpkMjl5WkE=
```



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**.

Example environment variables set to values from a Secret

```
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
```

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 environment variables set to values 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
```

B.89.2. 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

Example Secret with properties

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |- 1
    dbUsername: my-user 2
    dbPassword: my-password
```

1 The connector configuration in properties file format.

2 Database username and password properties used in the configuration.

In this example, a Secret named **mysecret** is mounted to a volume named **connector-config**. In the **config** property, a configuration provider (**FileConfigProvider**) is specified, which will load configuration values from external sources. The Kafka **FileConfigProvider** is given the alias **file**, and will read and extract database *username* and *password* property values from the file to use in the connector configuration.

Example external volumes set to values from a Secret

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: file 1
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider 2
  #...
  externalConfiguration:
    volumes:
      - name: connector-config 3
        secret:
          secretName: mysecret 4
```

1. The alias for the configuration provider, which is used to define other configuration parameters. Use a comma-separated list if you want to add more than one provider.
2. The **FileConfigProvider** is the configuration provider that provides values from properties files. The parameter uses the alias from **config.providers**, taking the form **config.providers.${alias}.class**.
3. The name of the volume containing the Secret. Each volume must specify a name in the **name** property and a reference to ConfigMap or Secret.
4. The name of the Secret.

The volumes are mounted inside the Kafka Connect containers in the path **/opt/kafka/external-configuration/*<volume-name>***. For example, the files from a volume named **connector-config** would appear in the directory **/opt/kafka/external-configuration/connector-config**.

The **FileConfigProvider** is used to read the values from the mounted properties files in connector configurations.

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.90. 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.91. 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.92. 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.93. 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	The URL of the REST API endpoint for managing and monitoring Kafka Connect connectors.

Property	Description
string	
connectorPlugins **ConnectorPlugin** array	The list of connector plugins available in this Kafka Connect deployment.
labelSelector string	Label selector for pods providing this resource.
replicas integer	The current number of pods being used to provide this resource.

B.94. CONNECTORPLUGIN SCHEMA REFERENCE

Used in: **KafkaConnectS2IStatus**, **KafkaConnectStatus**, **KafkaMirrorMaker2Status**

Property	Description
type string	The type of the connector plugin. The available types are **sink** and **source**.
version string	The version of the connector plugin.
class string	The class of the connector plugin.

B.95. KAFKACONNECTS2I SCHEMA REFERENCE

Property	Description
spec **KafkaConnectS2ISpec**	The specification of the Kafka Connect Source-to-Image (S2I) cluster.
status **KafkaConnectS2IStatus**	The status of the Kafka Connect Source-to-Image (S2I) cluster.

B.96. KAFKACONNECTS2ISPEC SCHEMA REFERENCE

Used in: **KafkaConnectS2I**

Configures a Kafka Connect cluster with Source-to-Image (S2I) support.

When extending Kafka Connect with connector plugins on OpenShift (only), you can use OpenShift builds and S2I to create a container image that is used by the Kafka Connect deployment.

The configuration options are similar to Kafka Connect configuration using the **KafkaConnectSpec** schema.

Property	Description
replicas integer	The number of pods in the Kafka Connect group.
image string	The docker image for the pods.
buildResources ResourceRequirements	CPU and memory resources to reserve. 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.
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].

Property	Description
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.
clientRackInitImage string	The image of the init container used for initializing the **client.rack**.
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.
rack **Rack**	Configuration of the node label which will be used as the client.rack consumer configuration.

Property	Description
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 2.6.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.

B.97. KAFKACONNECTS2ISTATUS SCHEMA REFERENCE

Used in: **KafkaConnectS2I**

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.

Property	Description
buildConfigName	The name of the build configuration.
string	
labelSelector	Label selector for pods providing this resource.
string	
replicas	The current number of pods being used to provide this resource.
integer	

B.98. KAFKATOPIC SCHEMA REFERENCE

Property	Description
spec	The specification of the topic.
KafkaTopicSpec	
status	The status of the topic.
KafkaTopicStatus	

B.99. KAFKATOPICSPEC SCHEMA REFERENCE

Used in: **KafkaTopic**

Property	Description
partitions	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.
integer	
replicas	The number of replicas the topic should have.
integer	
config	The topic configuration.
map	

Property	Description
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.100. 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.101. KAFKAUSER SCHEMA REFERENCE

Property	Description
spec **KafkaUserSpec**	The specification of the user.
status **KafkaUserStatus**	The status of the Kafka User.

B.102. 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].

Property	Description
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.
template **KafkaUserTemplate**	Template to specify how Kafka User **Secrets** are generated.

B.103. 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.104. 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 string	Must be **scram-sha-512**.

B.105. 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.106. ACLRULE SCHEMA REFERENCE

Used in: **KafkaUserAuthorizationSimple**

Configures access control rule for a **KafkaUser** when brokers are using the **AclAuthorizer**.

Example KafkaUser configuration 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
```

B.106.1. resource

Use the **resource** property to specify 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.

B.106.2. type

The **type** of rule, which is to **allow** or **deny** (not currently supported) an operation.

The **type** field is optional. If **type** is unspecified, the ACL rule is treated as an **allow** rule.

B.106.3. operation

Specify an **operation** for the rule 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 **AclAuthorizer**, ACLs and supported combinations of resources and operations, see Authorization and ACLs.

B.106.4. host

Use the **host** property to specify 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.

Property	Description
host string	The host from which the action described in the ACL rule is allowed or denied.
operation string (one of [Read, Write, Delete, Alter, Describe, All, IdempotentWrite, ClusterAction, Create, AlterConfigs, DescribeConfigs])	Operation which will be allowed or denied. Supported operations are: Read, Write, Create, Delete, Alter, Describe, ClusterAction, AlterConfigs, DescribeConfigs, IdempotentWrite and All.
resource **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource**	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].
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.107. 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	Must be **topic**.

Property	Description
string	
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.108. 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**.
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.109. 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.110. 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	
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 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.111. KAFKAUSERQUOTAS SCHEMA REFERENCE

Used in: **KafkaUserSpec**

Kafka allows a user to set **quotas** to control the use of resources by clients.

B.111.1. quotas

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, refer to the 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.
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.112. KAFKAUSERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaUserSpec**

Specify additional labels and annotations for the secret created by the User Operator.

An example showing the KafkaUserTemplate

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  template:
    secret:
      metadata:
```

```
      labels:
        label1: value1
      annotations:
        anno1: value1
  # ...
```

Property	Description
secret	Template for KafkaUser resources. The template allows users to specify how the **Secret** with password or TLS certificates is generated.
ResourceTemplate	

B.113. KAFKAUSERSTATUS SCHEMA REFERENCE

Used in: **KafkaUser**

Property	Description
conditions	List of status conditions.
Condition array	
observedGeneration	The generation of the CRD that was last reconciled by the operator.
integer	
username	Username.
string	
secret	The name of **Secret** where the credentials are stored.
string	

B.114. KAFKAMIRRORMAKER SCHEMA REFERENCE

Property	Description
spec	The specification of Kafka MirrorMaker.
KafkaMirrorMakerSpec	
status	The status of Kafka MirrorMaker.
KafkaMirrorMakerStatus	

B.115. KAFKAMIRRORMAKERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker**

Configures Kafka MirrorMaker.

B.115.1. 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.

B.115.2. KafkaMirrorMakerConsumerSpec and KafkaMirrorMakerProducerSpec

Use the **KafkaMirrorMakerConsumerSpec** and **KafkaMirrorMakerProducerSpec** to configure source (consumer) and target (producer) clusters.

Kafka MirrorMaker always works together with two Kafka clusters (source and target). To establish a connection, the bootstrap servers for the source and the target Kafka clusters are specified as comma-separated lists 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*** pair.

B.115.3. logging

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**. For more information about log levels, see Apache logging services.

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
  # ...
```

Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

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	

Property	Description
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.
jvmOptions **JvmOptions**	JVM Options for pods.
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 2.6.0. Consult the documentation to understand the process required to upgrade or downgrade the version.

B.116. KAFKAMIRRORMAKERCONSUMERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Configures a MirrorMaker consumer.

B.116.1. numStreams

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.

B.116.2. offsetCommitInterval

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.

B.116.3. config

Use the **consumer.config** properties to configure Kafka options for the consumer.

The **config** property contains the Kafka MirrorMaker consumer configuration options as keys, with values set in one of the following JSON types:

- String
- Number
- Boolean

For client connection using a specific *cipher suite* for a TLS version, you can configure allowed **ssl** properties. You can also configure the **ssl.endpoint.identification.algorithm** property to enable or disable hostname verification.

Exceptions

You can specify and configure the options listed in the 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
- Interceptors

Specifically, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **bootstrap.servers**
- **group.id**

- **interceptor.classes**
- **ssl.** (not including specific exceptions)
- **sasl.**
- **security.**

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** object should be fixed and the Cluster Operator will roll out the new configuration for Kafka MirrorMaker.

B.116.4. groupId

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.

Property	Description
numStreams	Specifies the number of consumer stream threads to create.
integer	
offsetCommitInterval	Specifies the offset auto-commit interval in ms. Default value is 60000.
integer	
groupId	A unique string that identifies the consumer group this consumer belongs to.
string	
bootstrapServers	A list of host:port pairs for establishing the initial connection to the Kafka cluster.
string	
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].

Property	Description
KafkaClientAuthenticationTls, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	
config	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).
map	
tls	TLS configuration for connecting MirrorMaker to the cluster.
KafkaMirrorMakerTls	

B.117. KAFKAMIRRORMAKERTLS SCHEMA REFERENCE

Used in: **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Configures TLS trusted certificates for connecting MirrorMaker to the cluster.

B.117.1. trustedCertificates

Provide a list of secrets using the **trustedCertificates** property.

Property	Description
trustedCertificates	Trusted certificates for TLS connection.
CertSecretSource array	

B.118. KAFKAMIRRORMAKERPRODUCERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Configures a MirrorMaker producer.

B.118.1. abortOnSendFailure

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.

B.118.2. config

Use the **producer.config** properties to configure Kafka options for the producer.

The **config** property contains the Kafka MirrorMaker producer configuration options as keys, with values set in one of the following JSON types:

- String
- Number
- Boolean

For client connection using a specific *cipher suite* for a TLS version, you can configure allowed **ssl** properties. You can also configure the **ssl.endpoint.identification.algorithm** property to enable or disable hostname verification.

Exceptions

You can specify and configure the options listed in the Apache Kafka configuration documentation for producers.

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)
- Interceptors

Specifically, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **bootstrap.servers**
- **interceptor.classes**
- **ssl.** (not including specific exceptions)
- **sasl.**
- **security.**

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.producer.config** object should be fixed and the Cluster Operator will roll out the new configuration for Kafka MirrorMaker.

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	
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.119. KAFKAMIRRORMAKERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Property	Description
deployment	Template for Kafka MirrorMaker **Deployment**.
ResourceTemplate	
pod	Template for Kafka MirrorMaker **Pods**.

Property	Description
PodTemplate	
mirrorMakerContainer	Template for Kafka MirrorMaker container.
ContainerTemplate	
podDisruptionBudget	Template for Kafka MirrorMaker **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

B.120. 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	
labelSelector	Label selector for pods providing this resource.
string	
replicas	The current number of pods being used to provide this resource.
integer	

B.121. KAFKABRIDGE SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Bridge.
KafkaBridgeSpec	
status	The status of the Kafka Bridge.
KafkaBridgeStatus	

B.122. KAFKABRIDGESPEC SCHEMA REFERENCE

Used in: **KafkaBridge**

Configures a Kafka Bridge cluster.

Configuration options relate to:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Consumer configuration
- Producer configuration
- HTTP configuration

B.122.1. logging

Kafka Bridge has its own configurable loggers:

- **logger.bridge**
- **logger.*<operation-id>***

You can replace ***<operation-id>*** in the **logger.*<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.

Each logger has to be configured assigning it a **name** as **http.openapi.operation.*<operation-id>***. For example, configuring the logging level for the **send** operation logger means defining the following:

```
logger.send.name = http.openapi.operation.send
logger.send.level = DEBUG
```

Kafka Bridge uses the Apache **log4j2** logger implementation. Loggers are defined in the **log4j2.properties** file, which has the following default configuration for **healthy** and **ready** endpoints:

```
logger.healthy.name = http.openapi.operation.healthy
logger.healthy.level = WARN
logger.ready.name = http.openapi.operation.ready
logger.ready.level = WARN
```

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**. For more information about log levels, see Apache logging services.

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

Inline logging

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaBridge
spec:
  # ...
  logging:
    type: inline
    loggers:
      logger.bridge.level: "INFO"
      # enabling DEBUG just for send operation
      logger.send.name: "http.openapi.operation.send"
      logger.send.level: "DEBUG"
  # ...
```

External logging

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

```
    type: external
    name: customConfigMap
  # ...
```

Any available loggers that are not configured have their level set to **OFF**.

If the Kafka Bridge was deployed using the Cluster Operator, changes to Kafka Bridge logging levels are applied dynamically.

If you use external logging, a rolling update is triggered when logging appenders are changed.

Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

Property	Description
replicas	The number of pods in the **Deployment**.
integer	
image	The docker image for the pods.
string	
bootstrapServers	A list of host:port pairs for establishing the initial connection to the Kafka cluster.
string	
tls	TLS configuration for connecting Kafka Bridge to the cluster.
KafkaBridgeTls	
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**	
http	The HTTP related configuration.
KafkaBridgeHttpConfig	
consumer	Kafka consumer related configuration.
KafkaBridgeConsumerSpec	
producer	Kafka producer related configuration.

Property	Description
KafkaBridgeProducerSpec	
resources	CPU and memory resources to reserve. See external documentation of core/v1 resourcerequirements.
ResourceRequirements	
jvmOptions	**Currently not supported** JVM Options for pods.
JvmOptions	
logging	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].
InlineLogging, **ExternalLogging**	
enableMetrics	Enable the metrics for the Kafka Bridge. Default is false.
boolean	
livenessProbe	Pod liveness checking.
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.123. KAFKABRIDGETLS SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
trustedCertificates	Trusted certificates for TLS connection.
CertSecretSource array	

B.124. KAFKABRIDGEHTTPCONFIG SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Configures HTTP access to a Kafka cluster for the Kafka Bridge.

The default HTTP configuration is for the Kafka Bridge to listen on port 8080.

B.124.1. cors

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 access methods. For the origins, you can use a URL or a Java regular expression.

Example Kafka Bridge HTTP configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  http:
    port: 8080
    cors:
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  # ...
```

Property	Description
port	The port which is the server listening on.
integer	
cors	CORS configuration for the HTTP Bridge.
KafkaBridgeHttpCors	

B.125. KAFKABRIDGEHTTPCORS SCHEMA REFERENCE

Used in: **KafkaBridgeHttpConfig**

Property	Description
allowedOrigins	List of allowed origins. Java regular expressions can be used.

Property	Description
string array	
allowedMethods	List of allowed HTTP methods.
string array	

B.126. KAFKABRIDGECONSUMERSPEC SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Configures consumer options for the Kafka Bridge 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 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 is 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. If an invalid configuration is provided, the Kafka Bridge cluster might not start or might become unstable. Fix the configuration so that the Cluster Operator can roll out the new configuration to all Kafka Bridge nodes.

There are exceptions to the forbidden options. For client connection using a specific *cipher suite* for a TLS version, you can configure allowed **ssl** properties.

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"
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
      ssl.endpoint.identification.algorithm: HTTPS
    # ...
```

Property	Description
config	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).
map	

B.127. KAFKABRIDGEPRODUCERSPEC SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Configures producer options for the Kafka Bridge as keys.

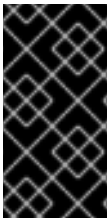
The values can be one of the following JSON types:

- String
- Number
- Boolean

You can specify and configure the options listed in the Apache Kafka 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**

When one of the forbidden options is present in the **config** property, it is 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. If an invalid configuration is provided, the Kafka Bridge cluster might not start or might become unstable. Fix the configuration so that the Cluster Operator can roll out the new configuration to all Kafka Bridge nodes.

There are exceptions to the forbidden options. For client connection using a specific *cipher suite* for a TLS version, you can configure allowed **ssl** properties.

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"
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
      ssl.endpoint.identification.algorithm: HTTPS
    # ...
```

Property	Description
config	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).
map	

B.128. KAFKABRIDGETEMPLATE SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
deployment	Template for Kafka Bridge **Deployment**.
ResourceTemplate	
pod	Template for Kafka Bridge **Pods**.
PodTemplate	

Property	Description
apiService	Template for Kafka Bridge API **Service**.
ResourceTemplate	
bridgeContainer	Template for the Kafka Bridge container.
ContainerTemplate	
podDisruptionBudget	Template for Kafka Bridge **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

B.129. 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	
labelSelector	Label selector for pods providing this resource.
string	
replicas	The current number of pods being used to provide this resource.
integer	

B.130. KAFKACONNECTOR SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Connector.

Property	Description
KafkaConnectorSpec	
status	The status of the Kafka Connector.
KafkaConnectorStatus	

B.131. 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.132. KAFKACONNECTORSTATUS SCHEMA REFERENCE

Used in: **KafkaConnector**

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
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.133. 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.134. 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 2.6.0. Consult the user documentation to understand the process required to upgrade or downgrade the version.
string	
image	The docker image for the pods.
string	
connectCluster	The cluster alias used for Kafka Connect. The alias must match a cluster in the list at **spec.clusters**.
string	

Property	Description
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 map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.

Property	Description
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.135. KAFKAMIRRORMAKER2CLUSTERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2Spec**

Configures Kafka clusters for mirroring.

B.135.1. config

Use the **config** properties to configure Kafka options.

Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

For client connection using a specific *cipher suite* for a TLS version, you can configure allowed **ssl** properties. You can also configure the **ssl.endpoint.identification.algorithm** property to enable or disable hostname verification.

Property	Description
alias string	Alias used to reference the Kafka cluster.
bootstrapServers string	A comma-separated list of **host:port** pairs for establishing the connection to the Kafka cluster.

Property	Description
config map	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).
tls **KafkaMirrorMaker2Tls**	TLS configuration for connecting MirrorMaker 2.0 connectors to a 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].

B.136. KAFKAMIRRORMAKER2TLS SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2ClusterSpec**

Property	Description
trustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection.

B.137. KAFKAMIRRORMAKER2MIRRORSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2Spec**

Property	Description
sourceCluster string	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**.
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**.

Property	Description
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.138. KAFKAMIRRORMAKER2CONNECTORSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2MirrorSpec**

Property	Description
tasksMax integer	The maximum number of tasks for the Kafka Connector.
config map	The Kafka Connector configuration. The following properties cannot be set: connector.class, tasks.max.
pause	Whether the connector should be paused. Defaults to false.

Property	Description
boolean	

B.139. KAFKAMIRRORMAKER2STATUS SCHEMA REFERENCE

Used in: **KafkaMirrorMaker2**

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	
connectors	List of MirrorMaker 2.0 connector statuses, as reported by the Kafka Connect REST API.
map array	
labelSelector	Label selector for pods providing this resource.
string	
replicas	The current number of pods being used to provide this resource.
integer	

B.140. KAFKAREBALANCE SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka rebalance.
KafkaRebalanceSpec	

Property	Description
status	The status of the Kafka rebalance.
KafkaRebalanceStatus	

B.141. KAFKAREBALANCESPEC SCHEMA REFERENCE

Used in: **KafkaRebalance**

Property	Description
goals string array	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.
skipHardGoalCheck boolean	Whether to allow the hard goals specified in the Kafka CR to be skipped in optimization proposal generation. This can be useful when some of those hard goals are preventing a balance solution being found. Default is false.
excludedTopics string	A regular expression where any matching topics will be excluded from the calculation of optimization proposals. This expression will be parsed by the java.util.regex.Pattern class; for more information on the supported formar consult the documentation for that class.
concurrentPartitionMovementsPerBroker integer	The upper bound of ongoing partition replica movements going into/out of each broker. Default is 5.
concurrentIntraBrokerPartitionMovements integer	The upper bound of ongoing partition replica movements between disks within each broker. Default is 2.
concurrentLeaderMovements integer	The upper bound of ongoing partition leadership movements. Default is 1000.
replicationThrottle integer	The upper bound, in bytes per second, on the bandwidth used to move replicas. There is no limit by default.

Property	Description
replicaMovementStrategies string array	A list of strategy class names used to determine the execution order for the replica movements in the generated optimization proposal. By default BaseReplicaMovementStrategy is used, which will execute the replica movements in the order that they were generated.

B.142. KAFKAREBALANCESTATUS SCHEMA REFERENCE

Used in: **KafkaRebalance**

Property	Description
conditions **Condition** array	List of status conditions.
observedGeneration integer	The generation of the CRD that was last reconciled by the operator.
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 2022-02-01 16:35:35 UTC