# Red Hat AMQ Streams 2.3

# Configuring AMQ Streams on OpenShift

Configure and manage a deployment of AMQ Streams 2.3 on OpenShift Container Platform

# Red Hat AMQ Streams 2.3 Configuring AMQ Streams on OpenShift

Configure and manage a deployment of AMQ Streams 2.3 on OpenShift Container Platform

## Legal Notice

## Abstract

Configure the operators and Kafka components deployed with AMQ Streams to build a large-scale messaging network.

# Table of Contents

# MAKING OPEN SOURCE MORE INCLUSIVE

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

# CHAPTER 1. CONFIGURATION OVERVIEW

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

This guide describes how to configure and manage an AMQ Streams deployment.

## 1.1. CONFIGURING CUSTOM RESOURCES

Use custom resources to configure your AMQ Streams deployment.

You can use custom resources to configure and create instances of the following components:

- Kafka clusters

- Kafka Connect clusters

- Kafka MirrorMaker

- Kafka Bridge

- Cruise Control

You can also use custom resource configuration to manage your instances or modify your deployment to introduce additional features. This might include configuration that supports the following:

- Securing client access to Kafka brokers

- Accessing Kafka brokers from outside the cluster

- Creating topics

- Creating users (clients)

- Controlling feature gates

- Changing logging frequency

- Allocating resource limits and requests

- Introducing features, such as AMQ Streams Drain Cleaner, Cruise Control, or distributed tracing.

The Custom resource API reference describes the properties you can use in your configuration.

## 1.2. USING CONFIGMAPS TO ADD CONFIGURATION

Use **ConfigMap** resources to add specific configuration to your AMQ Streams deployment. ConfigMaps use key-value pairs to store non-confidential data. Configuration data added to ConfigMaps is maintained in one place and can be reused amongst components.

ConfigMaps can only store configuration data related to the following:

- Logging configuration

- Metrics configuration

- External configuration for Kafka Connect connectors

You can't use ConfigMaps for other areas of configuration.

When you configure a component, you can add a reference to a ConfigMap using the **configMapKeyRef** property.

For example, you can use **configMapKeyRef** to reference a ConfigMap that provides configuration for logging. You might use a ConfigMap to pass a Log4j configuration file. You add the reference to the **logging** configuration.

### Example ConfigMap for logging

```
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: my-config-map
        key: my-config-map-key
```

To use a ConfigMap for metrics configuration, you add a reference to the **metricsConfig** configuration of the component in the same way.

**ExternalConfiguration** properties make data from a ConfigMap (or Secret) mounted to a pod available as environment variables or volumes. You can use external configuration data for the connectors used by Kafka Connect. The data might be related to an external data source, providing the values needed for the connector to communicate with that data source.

For example, you can use the **configMapKeyRef** property to pass configuration data from a ConfigMap as an environment variable.

### Example ConfigMap providing environment variable values

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

If you are using ConfigMaps that are managed externally, use configuration providers to load the data in the ConfigMaps. For more information on using configuration providers, see Chapter 3, *Loading configuration values from external sources*.

## 1.2.1. Naming custom ConfigMaps

AMQ Streams creates its own ConfigMaps and other resources when it is deployed to OpenShift. The ConfigMaps contain data necessary for running components. The ConfigMaps created by AMQ Streams must not be edited.

Make sure that any custom ConfigMaps you create do not have the same name as these default ConfigMaps. If they have the same name, they will be overwritten. For example, if your ConfigMap has the same name as the ConfigMap for the Kafka cluster, it will be overwritten when there is an update to the Kafka cluster.

**Additional resources**

- Section 2.2.9, "List of Kafka cluster resources" (including ConfigMaps)

- Section 2.10, "Logging configuration"

- Section 12.1.8, "**metricsConfig**"

- Section 12.2.74, "**ExternalConfiguration** schema reference"

- Chapter 3, *Loading configuration values from external sources*

## 1.3. CONFIGURING LISTENERS TO CONNECT TO KAFKA BROKERS

Listeners are used for client connection to Kafka brokers. AMQ Streams provides a generic **GenericKafkaListener** schema with properties to configure listeners through the **Kafka** resource.

The **GenericKafkaListener** provides a flexible approach to listener configuration. You can specify properties to configure *internal* listeners for connecting within the OpenShift cluster or *external* listeners for connecting outside the OpenShift cluster.

Each listener is defined as an array in the **Kafka** resource. You can configure as many listeners as required, as long as their names and ports are unique. You can configure listeners for secure connection using authentication.

### 1.3.1. Configuring internal listeners

Internal listeners connect clients to Kafka brokers within the OpenShift cluster. An **internal** type listener configuration uses a headless service and the DNS names given to the broker pods.

You might need to join your OpenShift network to an outside network. In which case, you can configure an **internal** type listener (using the **useServiceDnsDomain** property) so that the OpenShift service DNS domain (typically **.cluster.local**) is not used.

You can also configure a **cluster-ip** type of listener that exposes a Kafka cluster based on per-broker **ClusterIP** services. This is a useful option when you can't route through the headless service or you wish to incorporate a custom access mechanism. For example, you might use this listener when building your own type of external listener for a specific Ingress controller or the OpenShift Gateway API.

### 1.3.2. Configuring external listeners

Configure external listeners to handle access to a Kafka cluster from networks that require different authentication mechanisms.

You can configure external listeners for client access outside an OpenShift environment using a specified connection mechanism, such as a loadbalancer or route.

### 1.3.3. Providing listener certificates

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

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

**Additional resources**

- Section 12.2.4, "**GenericKafkaListener** schema reference"

- Chapter 5, *Accessing Kafka outside of the OpenShift cluster*

- Section 6.3, "Securing access to Kafka brokers"

## 1.4. DOCUMENT CONVENTIONS

**User-replaced values**

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

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

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

## 1.5. ADDITIONAL RESOURCES

- AMQ Streams Overview

- Deploying and Upgrading AMQ Streams

- Using the AMQ Streams Kafka Bridge

# CHAPTER 2. CONFIGURING AN AMQ STREAMS ON OPENSHIFT DEPLOYMENT

Configure your AMQ Streams deployment using custom resources. AMQ Streams provides example configuration files, which can serve as a starting point when building your own Kafka component configuration for deployment.

> **NOTE**
>
> Labels applied to a custom resource are also applied to the OpenShift resources making up its cluster. This provides a convenient mechanism for resources to be labeled as required.

**Monitoring an AMQ Streams deployment**

You can use Prometheus and Grafana to monitor your AMQ Streams deployment. For more information, see Introducing metrics to Kafka.

## 2.1. USING STANDARD KAFKA CONFIGURATION PROPERTIES

Use standard Kafka configuration properties to configure Kafka components.

The properties provide options to control and tune the configuration of the following Kafka components:

- Brokers

- Topics

- Clients (producers and consumers)

- Admin client

- Kafka Connect

- Kafka Streams

Broker and client parameters include options to configure authorization, authentication and encryption.

> **NOTE**
>
> For AMQ Streams on OpenShift, some configuration properties are managed entirely by AMQ Streams and cannot be changed.

For further information on Kafka configuration properties and how to use the properties to tune your deployment, see the following guides:

- Kafka configuration properties

- Kafka configuration tuning

## 2.2. KAFKA CLUSTER CONFIGURATION

Configure a Kafka deployment using the **Kafka** resource. A Kafka cluster is deployed with a ZooKeeper cluster, so configuration options are also available for ZooKeeper within the **Kafka** resource. The Entity

Operator comprises the Topic Operator and User Operator. You can also configure **entityOperator** properties in the **Kafka** resource to include the Topic Operator and User Operator in the deployment.

Section 12.2.1, "**Kafka** schema reference" describes the full schema of the **Kafka** resource.

For more information about Apache Kafka, see the Apache Kafka documentation.

### Listener configuration

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

### Authorizing access to Kafka

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

### Managing TLS certificates

When deploying Kafka, the Cluster Operator automatically sets up and renews TLS certificates to enable encryption and authentication within your cluster. If required, you can manually renew the cluster and clients CA certificates before their renewal period starts. You can also replace the keys used by the cluster and clients CA certificates. For more information, see Renewing CA certificates manually and Replacing private keys.

## 2.2.1. Configuring Kafka

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

As well as configuring Kafka, you can add configuration for ZooKeeper and the AMQ Streams Operators. Common configuration properties, such as logging and healthchecks, are configured independently for each component.

This procedure 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 of clients)

- Authentication

- Storage

- Rack awareness

- Metrics

- Cruise Control for cluster rebalancing

### Kafka versions

The **inter.broker.protocol.version** property for the Kafka **config** must be the version supported by the specified Kafka version (**spec.kafka.version**). The property represents the version of Kafka protocol used in a Kafka cluster.

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

An update to the **inter.broker.protocol.version** is required when upgrading your Kafka version. For more information, see Upgrading Kafka .

**Prerequisites**

- An OpenShift cluster

- A running Cluster Operator

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

- Cluster Operator

- Kafka cluster

**Procedure**

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

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       replicas: 3 1
       version: 3.3.1 2
       logging: 3
         type: inline
         loggers:
           kafka.root.logger.level: "INFO"
       resources: 4
         requests:
           memory: 64Gi
           cpu: "8"
         limits:
           memory: 64Gi
           cpu: "12"
       readinessProbe: 5
         initialDelaySeconds: 15
         timeoutSeconds: 5
       livenessProbe:
         initialDelaySeconds: 15
         timeoutSeconds: 5
       jvmOptions: 6
         -Xms: 8192m
         -Xmx: 8192m
       image: my-org/my-image:latest 7
       listeners: 8
         - name: plain 9
   ```

```
        port: 9092 10
        type: internal 11
        tls: false 12
        configuration:
          useServiceDnsDomain: true 13
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication: 14
          type: tls
      - name: external 15
        port: 9094
        type: route
        tls: true
        configuration:
          brokerCertChainAndKey: 16
            secretName: my-secret
            certificate: my-certificate.crt
            key: my-key.key
    authorization: 17
      type: simple
    config: 18
      auto.create.topics.enable: "false"
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      default.replication.factor: 3
      min.insync.replicas: 2
      inter.broker.protocol.version: "3.3"
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 19
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
    storage: 20
      type: persistent-claim 21
      size: 10000Gi 22
    rack: 23
      topologyKey: topology.kubernetes.io/zone
    metricsConfig: 24
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef: 25
          name: my-config-map
          key: my-key
    # ...
  zookeeper: 26
    replicas: 3 27
    logging: 28
      type: inline
      loggers:
        zookeeper.root.logger: "INFO"
    resources:
      requests:
```

```
        memory: 8Gi
        cpu: "2"
      limits:
        memory: 8Gi
        cpu: "2"
    jvmOptions:
      -Xms: 4096m
      -Xmx: 4096m
    storage:
      type: persistent-claim
      size: 1000Gi
    metricsConfig:
      # ...
  entityOperator:
    tlsSidecar:
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: "INFO"
      resources:
        requests:
          memory: 512Mi
          cpu: "1"
        limits:
          memory: 512Mi
          cpu: "1"
    userOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
      resources:
        requests:
          memory: 512Mi
          cpu: "1"
        limits:
          memory: 512Mi
          cpu: "1"
  kafkaExporter:
    # ...
  cruiseControl:
    # ...
```

entityOperator: **29**

tlsSidecar: **30**

logging: **31**

logging: **32**

kafkaExporter: **33**

cruiseControl: **34**

**1** The number of replica nodes . If your cluster already has topics defined, you can scale clusters.

**2** Kafka version, which can be changed to a supported version by following the upgrade procedure.

**3** Kafka loggers and log levels added directly (**inline**) or indirectly (**external**) through a ConfigMap. A custom ConfigMap must be placed under the **log4j.properties** key. For the Kafka **kafka.root.logger.level** logger, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

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

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

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

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

**8** Listeners configure how clients connect to the Kafka cluster via bootstrap addresses. Listeners are configured as *internal* or *external* listeners for connection from inside or outside the OpenShift cluster.

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

**10** 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.

**11** Listener type specified as **internal** or **cluster-ip** (to expose Kafka using per-broker **ClusterIP** services), or for external listeners, as **route**, **loadbalancer**, **nodeport** or **ingress**.

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

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

**14** Listener authentication mechanism specified as mTLS, SCRAM-SHA-512, or token-based OAuth 2.0.

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

**16** Optional configuration for a Kafka listener certificate managed by an external CA (certificate authority). The **brokerCertChainAndKey** specifies a **Secret** that contains a server certificate and a private key. You can configure Kafka listener certificates on any listener with enabled TLS encryption.

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

**18** Broker configuration. Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

**19** SSL properties for listeners with TLS encryption enabled to enable a specific *cipher suite* or TLS version.

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

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

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

**23** Rack awareness configuration to spread replicas across different racks, data centers, or availability zones. The **topologyKey** must match a node label containing the rack ID. The example used in this configuration specifies a zone using the standard **topology.kubernetes.io/zone** label.

**24** Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter (the default metrics exporter).

**25** Prometheus rules for exporting metrics to a Grafana dashboard through the Prometheus JMX Exporter, which are enabled by referencing a ConfigMap containing configuration for the Prometheus JMX exporter. You can enable metrics without further configuration using a reference to a ConfigMap containing an empty file under **metricsConfig.valueFrom.configMapKeyRef.key**.

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

**27** The number of ZooKeeper nodes. 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.

**28** Specified ZooKeeper loggers and log levels.

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

**30** Entity Operator TLS sidecar configuration. Entity Operator uses the TLS sidecar for secure communication with ZooKeeper.

**31** Specified Topic Operator loggers and log levels. This example uses **inline** logging.

**32** Specified User Operator loggers and log levels.

**33** Kafka Exporter configuration. Kafka Exporter is an optional component for extracting metrics data from Kafka brokers, in particular consumer lag data. For Kafka Exporter to be able to work properly, consumer groups need to be in use.

**34** Optional configuration for Cruise Control, which is used to rebalance the Kafka cluster.

2. Create or update the resource:

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

## 2.2.2. Configuring the 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.

The operators are automatically configured to manage the topics and users of the Kafka cluster. The Topic Operator and User Operator can only watch a single namespace. For more information, see Section 7.1, "Watching namespaces with AMQ Streams operators" .
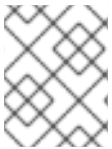
> **NOTE**
>
> When deployed, the Entity Operator pod contains the operators according to the deployment configuration.

### 2.2.2.1. 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.

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 2.8, "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 used to configure the Entity Operator, see the **EntityUserOperatorSpec** schema reference.

### Example of basic configuration enabling both operators

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

### 2.2.2.2. 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 **KafkaTopic** resources. 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 **18**.

**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 might 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 12.1.6, "**image**".

**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 12.1.5, "**resources**".

**logging**

The **logging** property configures the logging of the Topic Operator. For more details, see Section 12.2.45.1, "**logging**".

### Example Topic Operator configuration

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

```
  spec:
    kafka:
      # ...
    zookeeper:
      # ...
    entityOperator:
      # ...
      topicOperator:
        watchedNamespace: my-topic-namespace
        reconciliationIntervalSeconds: 60
      # ...
```

### 2.2.2.3. 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 **KafkaUser** resources. Default is the namespace where the Kafka cluster is deployed.

**reconciliationIntervalSeconds**

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

**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 12.1.6, "**image**".

**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 12.1.5, "**resources**".

**logging**

> The **logging** property configures the logging of the User Operator. For more details, see Section 12.2.45.1, "**logging**".

**secretPrefix**

> The **secretPrefix** property adds a prefix to the name of all Secrets created from the KafkaUser resource. For example, **secretPrefix: kafka-** would prefix all Secret names with **kafka-**. So a KafkaUser named **my-user** would create a Secret named **kafka-my-user**.

### Example User Operator configuration

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

```
watchedNamespace: my-user-namespace
reconciliationIntervalSeconds: 60
# ...
```

## 2.2.3. Configuring Kafka and ZooKeeper storage

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

- Ephemeral (Recommended for development only)

- Persistent

- JBOD (**Kafka only** not 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.

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

- **EphemeralStorage** schema reference

- **PersistentClaimStorage** schema reference

- **JbodStorage** schema reference

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

### 2.2.3.1. Data storage considerations

For AMQ Streams to work well, an efficient data storage infrastructure is essential. Block storage is required. File storage, such as NFS, does not work with Kafka.

Choose one of the following options for your block storage:

- A cloud-based block storage solution, such as Amazon Elastic Block Store (EBS)

- Persistent storage using 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.

### 2.2.3.1.1. File systems

Kafka uses a file system for storing messages. AMQ Streams is compatible with the XFS and ext4 file systems, which are commonly used with Kafka. Consider the underlying architecture and requirements of your deployment when choosing and setting up your file system.

For more information, refer to Filesystem Selection in the Kafka documentation.

### 2.2.3.1.2. Disk usage

Use separate disks for Apache Kafka and ZooKeeper.

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.

## 2.2.3.2. Ephemeral storage

Ephemeral data storage is transient. All pods on a node share a local ephemeral storage space. Data is retained for as long as the pod that uses it is running. The data is lost when a pod is deleted. Although a pod can recover data in a highly available environment.

Because of its transient nature, ephemeral storage is only recommended for development and testing.

Ephemeral storage uses **emptyDir** volumes to store data. An **emptyDir** volume is created when a pod is assigned to a node. You can set the total amount of storage for the **emptyDir** using the **sizeLimit** property .

> **IMPORTANT**
>
> Ephemeral storage is not suitable for single-node ZooKeeper clusters or Kafka topics with a replication factor of 1.

To use ephemeral storage, you set the storage type configuration in the **Kafka** or **ZooKeeper** resource to **ephemeral**.

### Example ephemeral storage configuration

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

```
  storage:
    type: ephemeral
  # ...
zookeeper:
  # ...
  storage:
    type: ephemeral
  # ...
```

### 2.2.3.2.1. Mount path of Kafka log directories

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

```
/var/lib/kafka/data/kafka-logIDX
```

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

### 2.2.3.3. Persistent storage

Persistent data storage retains data in the event of system disruption. For pods that use persistent data storage, data is persisted across pod failures and restarts.

A dynamic provisioning framework enables clusters to be created with persistent storage. Pod configuration uses Persistent Volume Claims (PVCs) to make storage requests on persistent volumes (PVs). PVs are storage resources that represent a storage volume. PVs are independent of the pods that use them. The PVC requests the amount of storage required when a pod is being created. The underlying storage infrastructure of the PV does not need to be understood. If a PV matches the storage criteria, the PVC is bound to the PV.

Because of its permanent nature, persistent storage is recommended for production.

PVCs can request different types of persistent storage by specifying a StorageClass. Storage classes define storage profiles and dynamically provision PVs. If a storage class is not specified, the default storage class is used. Persistent storage options might include SAN storage types or local persistent volumes.

To use persistent storage, you set the storage type configuration in the **Kafka** or **ZooKeeper** resource to **persistent-claim**.

In the production environment, the following configuration is recommended:

- For Kafka, configure **type: jbod** with one or more **type: persistent-claim** volumes

- For ZooKeeper, configure **type: persistent-claim**

Persistent storage also has the following configuration options:

**id** (optional)

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

**size** (required)

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

**class** (optional)

The OpenShift StorageClass to use for dynamic volume provisioning. Storage **class** configuration includes parameters that describe the profile of a volume in detail.

**selector** (optional)

Configuration to specify a specific PV. Provides key:value pairs representing the labels of the volume selected.

**deleteClaim** (optional)

Boolean value to specify whether the PVC is deleted when the cluster is uninstalled. 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 that 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 persistent storage configuration for Kafka and ZooKeeper**

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

If you do not specify a storage class, the default is used. The following example specifies a storage class.

**Example persistent storage configuration with specific storage class**

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

Use a **selector** to specify a labeled persistent volume that provides certain features, such as an SSD.

## Example persistent storage configuration with selector

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

### 2.2.3.3.1. Storage class overrides

Instead of using the default storage class, you can specify a different storage class for one or more Kafka brokers or ZooKeeper nodes. This is useful, for example, when 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/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  # ...
  kafka:
    replicas: 3
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
        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 use **my-storage-class-zone-1a**.

- The persistent volumes of ZooKeeper node 1 use **my-storage-class-zone-1b**.

- The persistent volumes of ZooKeeepr node 2 use **my-storage-class-zone-1c**.

- The persistent volumes of Kafka broker 0 use **my-storage-class-zone-1a**.

- The persistent volumes of Kafka broker 1 use **my-storage-class-zone-1b**.

- The persistent volumes of Kafka broker 2 use **my-storage-class-zone-1c**.

The **overrides** property is currently used only to override storage class configurations. Overrides for other storage configuration properties is not currently supported. Other storage configuration properties are currently not supported.

### 2.2.3.3.2. PVC resources for persistent storage

When persistent storage is used, it creates PVCs with the following names:

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

  PVC for the volume used for storing data for the Kafka broker pod **idx**.

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

  PVC for the volume used for storing data for the ZooKeeper node pod **idx**.

### 2.2.3.3.3. Mount path of Kafka log directories

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

```
/var/lib/kafka/data/kafka-logIDX
```

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

## 2.2.3.4. 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. Edit the **Kafka** resource for your cluster.
   Change the **size** property to increase the size of the persistent volume allocated to a Kafka cluster, a ZooKeeper cluster, or both.

   - For Kafka clusters, update the **size** property under **spec.kafka.storage**.

   - For ZooKeeper clusters, update the **size** property under **spec.zookeeper.storage**.

   **Kafka configuration to increase the volume size to 2000Gi**

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

   ```
   oc apply -f <kafka_configuration_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.

3. Verify that the storage capacity has increased for the relevant pods on the cluster:

```
oc get pv
```

### Kafka broker pods with increased storage

```
NAME             CAPACITY   CLAIM
pvc-0ca459ce-...  2000Gi    my-project/data-my-cluster-kafka-2
pvc-6e1810be-...  2000Gi    my-project/data-my-cluster-kafka-0
pvc-82dc78c9-...  2000Gi    my-project/data-my-cluster-kafka-1
```

The output shows the names of each PVC associated with a broker pod.

**Additional resources**

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

## 2.2.3.5. JBOD storage

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.

> **NOTE**
>
> JBOD storage is supported for **Kafka only** not ZooKeeper.

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 decrease the size of a persistent storage volume after it has been provisioned, or you cannot change the value of **sizeLimit** when the type is **ephemeral**.

To use JBOD storage, you set the storage type configuration in the **Kafka** resource to **jbod**. The **volumes** property allows you to describe the disks that make up your JBOD storage array or configuration.

### Example JBOD storage 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. You can add or remove volumes from the JBOD configuration.

### 2.2.3.5.1. PVC resource for JBOD storage

When persistent storage is used to declare JBOD volumes, it creates a PVC with the following name:

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

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

### 2.2.3.5.2. Mount path of Kafka log directories

The JBOD volumes are used by 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**.

### 2.2.3.6. 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.
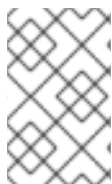
> **NOTE**
>
> When adding a new volume under an **id** which was already used in the past and removed, you have to make sure that the previously used **PersistentVolumeClaims** have been deleted.

**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/v1beta2
   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:

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

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

### Additional resources

For more information about reassigning topics, see Section 2.2.4.2, "Partition reassignment tool" .

### 2.2.3.7. 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/v1beta2
   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:

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

**Additional resources**

For more information about reassigning topics, see Section 2.2.4.2, "Partition reassignment tool" .

## 2.2.4. Scaling clusters

Scale Kafka clusters by adding or removing brokers. If a cluster already has topics defined, you also have to reassign partitions.

Use the **kafka-reassign-partitions.sh** tool to reassign partitions. The tool uses a reassignment JSON file that specifies the topics to reassign.

You can generate a reassignment JSON file or create a file manually if you want to move specific partitions.

### 2.2.4.1. Broker scaling configuration

You configure the **Kafka.spec.kafka.replicas** configuration to add or reduce the number of brokers.

**Broker addition**
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 reassign from the existing brokers to the new broker.

Once the partitions have been redistributed between all the brokers, the resource utilization of each broker is reduced.

**Broker removal**
If you are using **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.

### 2.2.4.2. Partition reassignment tool

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** tool 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 only want to reassign some partitions of some topics.

**--execute**

Takes a *reassignment JSON file* and applies it to the partitions and brokers in the cluster. Brokers that gain partitions as a result become followers of the partition leader. For a given partition, once the new broker has caught up and joined the ISR (in-sync replicas) the old broker will stop being a follower and will delete its replica.

**--verify**

Using the same *reassignment JSON file* as the **--execute** step, **--verify** checks whether all the partitions in the file have been moved to their intended brokers. If the reassignment is complete, **--verify** also removes any traffic throttles ( **--throttle**) that are in effect. Unless removed, throttles will continue to affect the cluster even after the reassignment has finished.

It is only possible to have one reassignment running in a cluster at any given time, and it is not possible to cancel a running reassignment. If you 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.

### 2.2.4.2.1. Partition 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 partition **4** of topic **topic-a** to brokers **2**, **4** and **7**, and partition **2** of topic **topic-b** to brokers **1**, **5** and **7**:

**Example partition reassignment file**

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "topic-a",
      "partition": 4,
      "replicas": [2,4,7]
    },
    {
      "topic": "topic-b",
      "partition": 2,
      "replicas": [1,5,7]
    }
  ]
}
```

Partitions not included in the JSON are not changed.

### 2.2.4.2.2. Partition reassignment between JBOD volumes

When using JBOD storage in your Kafka cluster, you can choose to reassign the partitions between specific volumes and their log directories (each volume has a single log directory). To reassign a partition to a specific volume, add the **log_dirs** option to *<PartitionObjects>* in the reassignment JSON file.

```
{
  "topic": <TopicName>,
  "partition": <Partition>,
  "replicas": [ <AssignedBrokerIds> ],
  "log_dirs": [ <AssignedLogDirs> ]
}
```

The **log_dirs** object should contain the same number of log directories as the number of replicas specified in the **replicas** object. The value should be either an absolute path to the log directory, or the **any** keyword.

**Example partition reassignment file specifying log directories**

```
{
    "topic": "topic-a",
    "partition": 4,
    "replicas": [2,4,7].
    "log_dirs": [ "/var/lib/kafka/data-0/kafka-log2", "/var/lib/kafka/data-0/kafka-log4",
"/var/lib/kafka/data-0/kafka-log7" ]
}
```

**Partition 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. Use the **--throttle** parameter with the **kafka-reassign-partitions.sh** tool to throttle a reassignment. You specify a maximum threshold in bytes per second for the movement of partitions between brokers. For example, **--throttle 5000000** sets a maximum threshold for moving partitions of 50 MBps.

Throttling might cause the reassignment to take longer to complete.

- If the throttle is too low, the newly assigned brokers will not be able to keep up with records being published and the reassignment will never complete.

- If the throttle is too high, clients will be impacted.

For example, for producers, this could manifest as higher than normal latency waiting for acknowledgment. For consumers, this could manifest as a drop in throughput caused by higher latency between polls.

## 2.2.4.3. Generating reassignment JSON files

This procedure describes how to generate a reassignment JSON file. Use the reassignment file with the **kafka-reassign-partitions.sh** tool to reassign partitions after scaling a Kafka cluster.

You run the tool from an interactive pod container connected to the Kafka cluster.

The steps describe a secure reassignment process that uses mTLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

You'll need the following to establish a connection:

- The cluster CA certificate and password generated by the Cluster Operator when the Kafka cluster is created

- The user CA certificate and password generated by the User Operator when a user is created for client access to the Kafka cluster

In this procedure, the CA certificates and corresponding passwords are extracted from the cluster and user secrets that contain them in PKCS #12 (**.p12** and **.password**) format. The passwords allow access to the **.p12** stores that contain the certificates. You use the **.p12** stores to specify a truststore and keystore to authenticate connection to the Kafka cluster.

**Prerequisites**

- You have a running Cluster Operator.

- You have a running Kafka cluster based on a **Kafka** resource configured with internal TLS encryption and mTLS authentication.

**Kafka configuration with TLS encryption and mTLS authentication**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      # ...
      - name: tls
        port: 9093
        type: internal
        tls: true 1
        authentication:
          type: tls 2
    # ...
```

**1**    Enables TLS encryption for the internal listener.

**2**    Listener authentication mechanism specified as mutual **tls**.

- The running Kafka cluster contains a set of topics and partitions to reassign.

**Example topic configuration for my-topic**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 3
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
    # ...
```

- You have a **KafkaUser** configured with ACL rules that specify permission to produce and consume topics from the Kafka brokers.

**Example Kafka user configuration with ACL rules to allow operations on my-topic and my-cluster**

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

```
spec:
  authentication:    1
    type: tls
  authorization:
    type: simple    2
    acls:
      # access to the topic
      - resource:
          type: topic
          name: my-topic
        operations:
          - Create
          - Describe
          - Read
          - AlterConfigs
        host: "*"
      # access to the cluster
      - resource:
          type: cluster
        operations:
          - Alter
          - AlterConfigs
        host: "*"
      # ...
    # ...
```

**1**  User authentication mechanism defined as mutual **tls**.

**2**  Simple authorization and accompanying list of ACL rules.

**Procedure**

1. Extract the cluster CA certificate and password from the ***&lt;cluster_name&gt;*-cluster-ca-cert** secret of the Kafka cluster.

   ```
   oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' | base64 -d >
   ca.p12
   ```

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

   Replace *&lt;cluster_name&gt;* with the name of the Kafka cluster. When you deploy Kafka using the **Kafka** resource, a secret with the cluster CA certificate is created with the Kafka cluster name (***&lt;cluster_name&gt;*-cluster-ca-cert**). For example, **my-cluster-cluster-ca-cert**.

2. Run a new interactive pod container using the AMQ Streams Kafka image to connect to a running Kafka broker.

   ```
   oc run --restart=Never --image=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0
   <interactive_pod_name> -- /bin/sh -c "sleep 3600"
   ```

   Replace *&lt;interactive_pod_name&gt;* with the name of the pod.

3. Copy the cluster CA certificate to the interactive pod container.

```
oc cp ca.p12 <interactive_pod_name>:/tmp
```

4. Extract the user CA certificate and password from the secret of the Kafka user that has permission to access the Kafka brokers.

```
oc get secret <kafka_user> -o jsonpath='{.data.user\.p12}' | base64 -d > user.p12
```

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

Replace *<kafka_user>* with the name of the Kafka user. When you create a Kafka user using the **KafkaUser** resource, a secret with the user CA certificate is created with the Kafka user name. For example, **my-user**.

5. Copy the user CA certificate to the interactive pod container.

```
oc cp user.p12 <interactive_pod_name>:/tmp
```

The CA certificates allow the interactive pod container to connect to the Kafka broker using TLS.

6. Create a **config.properties** file to specify the truststore and keystore used to authenticate connection to the Kafka cluster.
Use the certificates and passwords you extracted in the previous steps.

```
bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093  ❶
security.protocol=SSL  ❷
ssl.truststore.location=/tmp/ca.p12  ❸
ssl.truststore.password=<truststore_password>  ❹
ssl.keystore.location=/tmp/user.p12  ❺
ssl.keystore.password=<keystore_password>  ❻
```

❶ The bootstrap server address to connect to the Kafka cluster. Use your own Kafka cluster name to replace *<kafka_cluster_name>*.

❷ The security protocol option when using TLS for encryption.

❸ The truststore location contains the public key certificate (**ca.p12**) for the Kafka cluster.

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

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

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

7. Copy the **config.properties** file to the interactive pod container.

```
oc cp config.properties <interactive_pod_name>:/tmp/config.properties
```

8. Prepare a JSON file named **topics.json** that specifies the topics to move.
Specify topic names as a comma-separated list.

### Example JSON file to reassign all the partitions of  topic-a and topic-b

```
{
  "version": 1,
  "topics": [
    { "topic": "topic-a"},
    { "topic": "topic-b"}
  ]
}
```

9. Copy the *topics.json* file to the interactive pod container.

```
oc cp topics.json <interactive_pod_name>:/tmp/topics.json
```

10. Start a shell process in the interactive pod container.

```
oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

Replace *<namespace>* with the OpenShift namespace where the pod is running.

11. Use the **kafka-reassign-partitions.sh** command to generate the reassignment JSON.

**Example command to move all the partitions of topic-a and topic-b to brokers 0, 1 and 2**

```
bin/kafka-reassign-partitions.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --topics-to-move-json-file /tmp/topics.json \
  --broker-list 0,1,2 \
  --generate
```

**Additional resources**

- Section 2.2.1, "Configuring Kafka"

- Section 7.3.3, "Configuring Kafka topics"

- Section 7.4.1, "Configuring Kafka users"

## 2.2.4.4. Scaling up a Kafka cluster

Use a reassignment file to increase the number of brokers in a Kafka cluster.

The reassignment file should describe how partitions are reassigned to brokers in the enlarged Kafka cluster.

This procedure describes a secure scaling process that uses TLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

**Prerequisites**

- You have a running Kafka cluster based on a **Kafka** resource configured with internal TLS encryption and mTLS authentication.

- You have generated a reassignment JSON file named **reassignment.json**.

- You are running an interactive pod container that is connected to the running Kafka broker.

- You are connected as a **KafkaUser** configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

See Generating reassignment JSON files.

**Procedure**

1. Add as many new brokers as you need by increasing the **Kafka.spec.kafka.replicas** configuration option.

2. Verify that the new broker pods have started.

3. If you haven't done so, run an interactive pod container to generate a reassignment JSON file named **reassignment.json**.

4. Copy the ***reassignment.json*** file to the interactive pod container.

   oc cp reassignment.json *<interactive_pod_name>*:/tmp/reassignment.json

   Replace *<interactive_pod_name>* with the name of the pod.

5. Start a shell process in the interactive pod container.

   oc exec -n *<namespace>* -ti *<interactive_pod_name>* /bin/bash

   Replace *<namespace>* with the OpenShift namespace where the pod is running.

6. Run the partition reassignment using the **kafka-reassign-partitions.sh** script from the interactive pod container.

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --execute
   ```

   Replace *<cluster_name>* with the name of your Kafka cluster. For example, **my-cluster-kafka-bootstrap:9093**

   If you are going to throttle replication, you can also pass the **--throttle** option with an inter-broker throttled rate in bytes per second. For example:

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --throttle 5000000 \
    --execute
   ```

   This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

If you need to change the throttle during reassignment, you can use the same command with a different throttled rate. For example:

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

7. Verify that the reassignment has completed using the **kafka-reassign-partitions.sh** command line tool from any of the broker pods. This is the same command as the previous step, but with the **--verify** option instead of the **--execute** option.

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

The reassignment has finished when the **--verify** command reports that each of the partitions being moved has completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles.

8. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

### 2.2.4.5. Scaling down a Kafka cluster

Use a reassignment file to decrease the number of brokers in a Kafka cluster.

The reassignment file must describe how partitions are reassigned to the remaining brokers in the Kafka cluster. Brokers in the highest numbered pods are removed first.

This procedure describes a secure scaling process that uses TLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

**Prerequisites**

- You have a running Kafka cluster based on a **Kafka** resource configured with internal TLS encryption and mTLS authentication.

- You have generated a reassignment JSON file named **reassignment.json**.

- You are running an interactive pod container that is connected to the running Kafka broker.

- You are connected as a **KafkaUser** configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

See Generating reassignment JSON files.

**Procedure**

1. If you haven't done so, run an interactive pod container to generate a reassignment JSON file named **reassignment.json**.

2. Copy the ***reassignment.json*** file to the interactive pod container.

   ```
   oc cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
   ```

   Replace *<interactive_pod_name>* with the name of the pod.

3. Start a shell process in the interactive pod container.

   ```
   oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
   ```

   Replace *<namespace>* with the OpenShift namespace where the pod is running.

4. Run the partition reassignment using the **kafka-reassign-partitions.sh** script from the interactive pod container.

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --execute
   ```

   Replace *<cluster_name>* with the name of your Kafka cluster. For example,  **my-cluster-kafka-bootstrap:9093**

   If you are going to throttle replication, you can also pass the **--throttle** option with an inter–broker throttled rate in bytes per second. For example:

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --throttle 5000000 \
    --execute
   ```

   This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

   If you need to change the throttle during reassignment, you can use the same command with a different throttled rate. For example:

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --throttle 10000000 \
    --execute
   ```

5. Verify that the reassignment has completed using the **kafka-reassign-partitions.sh** command line tool from any of the broker pods. This is the same command as the previous step, but with the **--verify** option instead of the **--execute** option.

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
   ```

```
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--reassignment-json-file /tmp/reassignment.json \
--verify
```

The reassignment has finished when the **--verify** command reports that each of the partitions being moved has completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles.

6. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

7. When all the partition reassignments have finished, the brokers being removed should not have responsibility for any of the partitions in the cluster. You can verify this by checking that the broker's data log directory does not contain any live partition logs. If the log directory on the broker contains a directory that does not match the extended regular expression **\.[a-z0-9]-delete$**, the broker still has live partitions and should not be stopped.
You can check this by executing the command:

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  /bin/bash -c \
  "ls -l /var/lib/kafka/kafka-log_<n>_ | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'"
```

where *n* is the number of the pods being deleted.

If the above command prints any output then the broker still has live partitions. In this case, either the reassignment has not finished or the reassignment JSON file was incorrect.

8. When you have confirmed that the broker has no live partitions, you can edit the **Kafka.spec.kafka.replicas** property of your **Kafka** resource to reduce the number of brokers.

## 2.2.5. 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.

### 2.2.5.1. 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.

### 2.2.5.2. 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.
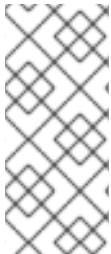
### NOTE

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 7.2.3, "Configuring the Cluster Operator with environment variables".

## 2.2.5.3. Configuring a maintenance time window

You can configure a maintenance time window for rolling updates triggered by supported processes.

### Prerequisites

- An OpenShift cluster.

- The Cluster Operator is running.

### Procedure

1. Add or edit the **maintenanceTimeWindows** property in the **Kafka** resource. For example to allow maintenance between 0800 and 1059 and between 1400 and 1559 you would set the **maintenanceTimeWindows** as shown below:

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

```
zookeeper:
  # ...
maintenanceTimeWindows:
  - "* * 8-10 * * ?"
  - "* * 14-15 * * ?"
```

2. Create or update the resource:

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

**Additional resources**

Performing rolling updates:

- Section 11.4.2, "Performing a rolling update using a pod management annotation"

- Section 11.4.3, "Performing a rolling update using a Pod annotation"

## 2.2.6. Connecting to ZooKeeper from a terminal

Most Kafka CLI tools can connect directly to Kafka, so under normal circumstances you should not need to connect to ZooKeeper. 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.

**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 -ti 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.

## 2.2.7. Deleting Kafka nodes manually

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.
   Kafka broker pods are named *<cluster-name>*-kafka-*<index>*, where *<index>* starts at zero and ends at the total number of replicas minus one. For example, **my-cluster-kafka-0**.

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.

## 2.2.8. 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.
   ZooKeeper pods are named *<cluster-name>*-zookeeper-*<index>*, where *<index>* starts at zero and ends at the total number of replicas minus one. For example, **my-cluster-zookeeper-0**.

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.

## 2.2.9. List of Kafka cluster resources

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

**Shared resources**

***cluster-name*-cluster-ca**

Secret with the Cluster CA private key 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 certificates

***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**

Name given to the following ZooKeeper resources:

- StatefulSet or StrimziPodSet (if the UseStrimziPodSets feature gate is enabled) for managing the ZooKeeper node pods.

- Service account used by the ZooKeeper nodes.

- PodDisruptionBudget configured for the ZooKeeper nodes.

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

Pods created by the ZooKeeper StatefulSet or StrimziPodSet.

***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*-**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**

Name given to the following Kafka resources:

- StatefulSet or StrimziPodSet (if the UseStrimziPodSets feature gate is enabled) for managing the Kafka broker pods.

- Service account used by the Kafka pods.

- PodDisruptionBudget configured for the Kafka brokers.

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

Name given to the following Kafka resources:

- Pods created by the Kafka StatefulSet or StrimziPodSet.

- ConfigMap with Kafka broker configuration (if the UseStrimziPodSets feature gate is enabled).

*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 connecting from within the OpenShift cluster.

*cluster-name*-**kafka-external-bootstrap**

Bootstrap service for clients connecting from outside the OpenShift cluster. This resource is created only when an external listener is enabled. The old service name will be used for backwards compatibility when the listener name is **external** and port is **9094**.

*cluster-name*-**kafka-***pod-id*

Service used to route traffic from outside the OpenShift cluster to individual pods. This resource is created only when an external listener is enabled. The old service name will be used for backwards compatibility when the listener name is **external** and port is **9094**.

*cluster-name*-**kafka-external-bootstrap**

Bootstrap route for clients connecting from outside the OpenShift cluster. This resource is created only when an external listener is enabled and set to type **route**. The old route name will be used for backwards compatibility when the listener name is **external** and port is **9094**.

*cluster-name*-kafka-*pod-id*

Route for traffic from outside the OpenShift cluster to individual pods. This resource is created only when an external listener is enabled and set to type **route**. The old route name will be used for backwards compatibility when the listener name is **external** and port is **9094**.

*cluster-name*-kafka-*listener-name*-bootstrap

Bootstrap service for clients connecting from outside the OpenShift cluster. This resource is created only when an external listener is enabled. The new service name will be used for all other external listeners.

*cluster-name*-kafka-*listener-name*-*pod-id*

Service used to route traffic from outside the OpenShift cluster to individual pods. This resource is created only when an external listener is enabled. The new service name will be used for all other external listeners.

*cluster-name*-kafka-*listener-name*-bootstrap

Bootstrap route for clients connecting from outside the OpenShift cluster. This resource is created only when an external listener is enabled and set to type **route**. The new route name will be used for all other external listeners.

*cluster-name*-kafka-*listener-name*-*pod-id*

Route for traffic from outside the OpenShift cluster to individual pods. This resource is created only when an external listener is enabled and set to type **route**. The new route name will be used for all other external listeners.

*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*-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 is 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 is 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 created only 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

Name given to the following Entity Operator resources:

- Deployment with Topic and User Operators.

- Service account used by the Entity Operator.

***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-topic-operator-certs**

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

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

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

**strimzi-*cluster-name*-entity-topic-operator**

Role binding used by the Entity Topic Operator.

**strimzi-*cluster-name*-entity-user-operator**

Role binding used by the Entity User Operator.

## Kafka Exporter

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

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

Name given to the following Kafka Exporter resources:

- Deployment with Kafka Exporter.

- Service used to collect consumer lag metrics.

- Service account used by the Kafka Exporter.

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

Pod created by the Kafka Exporter deployment.

## Cruise Control

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

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

Name given to the following Cruise Control resources:

- Deployment with Cruise Control.

- Service used to communicate with Cruise Control.

- Service account used by the 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*-network-policy-cruise-control**

Network policy managing access to the Cruise Control service.

## 2.3. KAFKA CONNECT CLUSTER CONFIGURATION

Configure a Kafka Connect deployment using the **KafkaConnect** resource. 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.

Section 12.2.60, "**KafkaConnect** schema reference" describes the full schema of the **KafkaConnect** resource.

For more information on deploying connector plugins, see Extending Kafka Connect with connector plugins.

### 2.3.1. Configuring Kafka Connect

Use Kafka Connect to set up external data connections to your Kafka cluster. Use the properties of the **KafkaConnect** resource to configure your Kafka Connect deployment.

### KafkaConnector configuration

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

In your Kafka Connect configuration, you enable KafkaConnectors for a Kafka Connect cluster by adding the **strimzi.io/use-connector-resources** annotation. You can also add a **build** configuration so that AMQ Streams automatically builds a container image with the connector plugins you require for your data connections. External configuration for Kafka Connect connectors is specified through the **externalConfiguration** property.

To manage connectors, you can use the Kafka Connect REST API, or use **KafkaConnector** custom resources. **KafkaConnector** resources must be deployed to the same namespace as the Kafka Connect cluster they link to. For more information on using these methods to create, reconfigure, or delete connectors, see Creating and managing connectors.

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, which keeps the configuration separate and more secure, if needed. This method applies especially to confidential data, such as usernames, passwords, or certificates.

### Handling high volumes of messages

You can tune the configuration to handle high volumes of messages. For more information, see Section 2.7, "Handling high volumes of messages" .

### 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 of the **KafkaConnect** resource.
   The properties you can configure are shown in this example configuration:

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   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
     build: 8
       output: 9
         type: docker
         image: my-registry.io/my-org/my-connect-cluster:latest
         pushSecret: my-registry-credentials
       plugins: 10
   ```

```
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb54804
5e115e33a9e69b1b2a352bee24df035a0447cb820077af00c03
      - name: camel-telegram
        artifacts:
          - type: tgz
            url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-
telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
            sha512sum:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e
8020bf4187870699819f54ef5859c7846ee4081507f48873479
  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
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  logging:
    type: inline
    loggers:
      log4j.rootLogger: "INFO"
  readinessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  metricsConfig:
    type: jmxPrometheusExporter
    valueFrom:
      configMapKeyRef:
        name: my-config-map
        key: my-key
  jvmOptions:
    "-Xmx": "1g"
    "-Xms": "1g"
  image: my-org/my-image:latest
```

externalConfiguration: **11**

resources: **12**

logging: **13**

readinessProbe: **14**

metricsConfig: **15**

jvmOptions: **16**

image: my-org/my-image:latest **17**

```
    rack:
      topologyKey: topology.kubernetes.io/zone 18
    template: 19
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                  topologyKey: "kubernetes.io/hostname"
      connectContainer: 20
        env:
          - name: JAEGER_SERVICE_NAME
            value: my-jaeger-service
          - name: JAEGER_AGENT_HOST
            value: jaeger-agent-name
          - name: JAEGER_AGENT_PORT
            value: "6831"
```

**1**  Use **KafkaConnect**.

**2**  Enables KafkaConnectors for the Kafka Connect cluster.

**3**  The number of replica nodes for the workers that run tasks.

**4**  Authentication for the Kafka Connect cluster, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN. 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**  Build configuration properties for building a container image with connector plugins automatically.

**9**  (Required) Configuration of the container registry where new images are pushed.

**10**  (Required) List of connector plugins and their artifacts to add to the new container image. Each plugin must be configured with at least one **artifact**.

**11**  External configuration for Kafka connectors using environment variables, as shown here, or volumes. You can also use *configuration provider plugins* to load configuration values from external sources.

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

**13** 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.

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

**15** Prometheus metrics, which are enabled by referencing a ConfigMap containing configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using a reference to a ConfigMap containing an empty file under **metricsConfig.valueFrom.configMapKeyRef.key**.

**16** JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka Connect.

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

**18** SPECIALIZED OPTION: Rack awareness configuration for the deployment. This is a specialized option intended for a deployment within the same location, not across regions. Use this option if you want connectors to consume from the closest replica rather than the leader replica. In certain cases, consuming from the closest replica can improve network utilization or reduce costs . The **topologyKey** must match a node label containing the rack ID. The example used in this configuration specifies a zone using the standard **topology.kubernetes.io/zone** label. To consume from the closest replica, enable the **RackAwareReplicaSelector** in the Kafka broker configuration.

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

**20** Environment variables are set for distributed tracing.

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.

**Additional resources**

- Introducing distributed tracing

## 2.3.2. 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/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
```

```
spec:
  # ...
  config:
    group.id: connect-cluster 1
    offset.storage.topic: connect-cluster-offsets 2
    config.storage.topic: connect-cluster-configs 3
    status.storage.topic: connect-cluster-status 4
    # ...
  # ...
```

**1** The Kafka Connect cluster ID within Kafka.

**2** Kafka topic that stores connector offsets.

**3** Kafka topic that stores connector and task status configurations.

**4** Kafka topic that stores connector and task status updates.

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

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

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

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

## 2.3.3. 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** resource.

**Example configuration properties in the KafkaConnect resource**

```
apiVersion: kafka.strimzi.io/v1beta2
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] The Kafka Connect cluster ID within Kafka.

[2] Kafka topic that stores connector offsets.

[3] Kafka topic that stores connector and task status configurations.

[4] Kafka topic that stores connector and task status updates.

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.
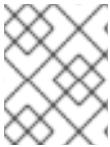
> **NOTE**
>
> The default values for the consumer group and topics will differ when running multiple instances.

**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** |
| **config.storage.topic** | **connect-cluster-configs** |
| **group** | **connect-cluster** |

```
apiVersion: kafka.strimzi.io/v1beta2
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
          operations:
            - Create
            - Describe
            - Read
            - Write
          host: "*"
        # access to status.storage.topic
        - resource:
            type: topic
            name: connect-cluster-status
            patternType: literal
          operations:
            - Create
            - Describe
            - Read
            - Write
          host: "*"
        # access to config.storage.topic
        - resource:
            type: topic
            name: connect-cluster-configs
            patternType: literal
          operations:
            - Create
            - Describe
            - Read
            - Write
          host: "*"
        # consumer group
        - resource:
            type: group
            name: connect-cluster
            patternType: literal
          operations:
            - Read
          host: "*"
```

2. Create or update the resource.

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

## 2.3.4. 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.

### 2.3.5. Integrating with the Red Hat build of Debezium for change data capture

The Red Hat build of 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 the Red Hat build of 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 the Red Hat build of Debezium with AMQ Streams, refer to the product documentation. The 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.

## 2.4. KAFKA MIRRORMAKER 2.0 CLUSTER CONFIGURATION

Configure a Kafka MirrorMaker 2.0 deployment using the **KafkaMirrorMaker2** resource. MirrorMaker 2.0 replicates data between two or more Kafka clusters, within or across data centers.

Section 12.2.126, "**KafkaMirrorMaker2** schema reference" describes the full schema of the **KafkaMirrorMaker2** resource.

MirrorMaker 2.0 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.

## 2.4.1. MirrorMaker 2.0 data replication

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

### 2.4.1.1. MirrorMaker 2.0 configuration

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.

You configure MirrorMaker 2.0 to define the Kafka Connect deployment, including the connection details of the source and target clusters, and then run a set of MirrorMaker 2.0 connectors to make the connection.

MirrorMaker 2.0 consists of the following connectors:

**MirrorSourceConnector**

The source connector replicates topics from a source cluster to a target cluster. It also replicates ACLs and is necessary for the **MirrorCheckpointConnector** to run.

**MirrorCheckpointConnector**

The checkpoint connector periodically tracks offsets. If enabled, it also synchronizes consumer group offsets between the source and target cluster.

**MirrorHeartbeatConnector**

The heartbeat connector periodically checks connectivity between the source and target cluster.

> **NOTE**
>
> If you are using the User Operator to manage ACLs, ACL replication through the connector is not possible.

The process of *mirroring* data from a source cluster to a target cluster is asynchronous. Each MirrorMaker 2.0 instance mirrors data from one source cluster to one target cluster. You can use more than one MirrorMaker 2.0 instance to mirror data between any number of clusters.

Figure 2.1. Replication across two clusters
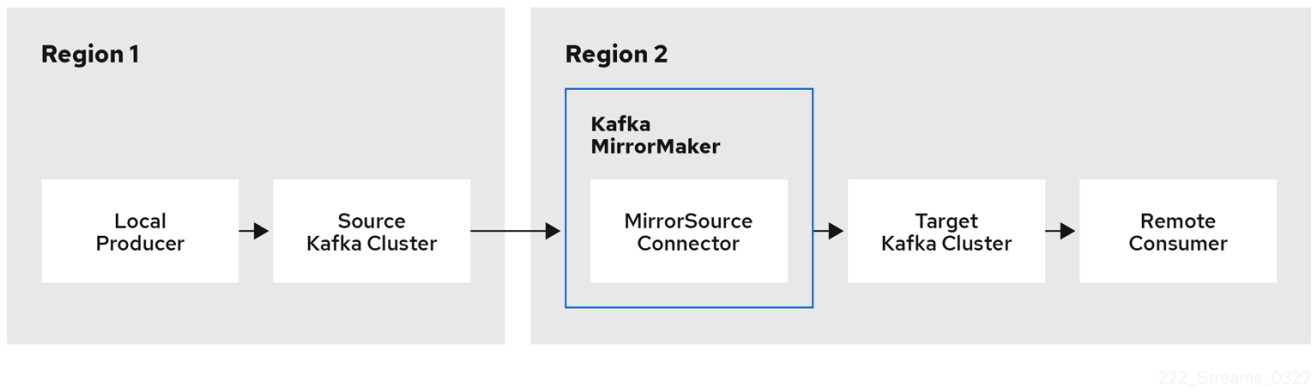


By default, a check for new topics in the source cluster is made every 10 minutes. You can change the frequency by adding **refresh.topics.interval.seconds** to the source connector configuration.

### 2.4.1.1.1. Cluster configuration

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

**active/active cluster configuration**

An active/active configuration has two active clusters replicating data bidirectionally. Applications can use either cluster. Each cluster can provide the same data. In this way, you can make the same data available in different geographical locations. As consumer groups are active in both clusters, consumer offsets for replicated topics are not synchronized back to the source cluster.

**active/passive cluster configuration**

An active/passive configuration has an active cluster replicating data to a passive cluster. The passive cluster remains on standby. You might use the passive cluster 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.

### 2.4.1.1.2. 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
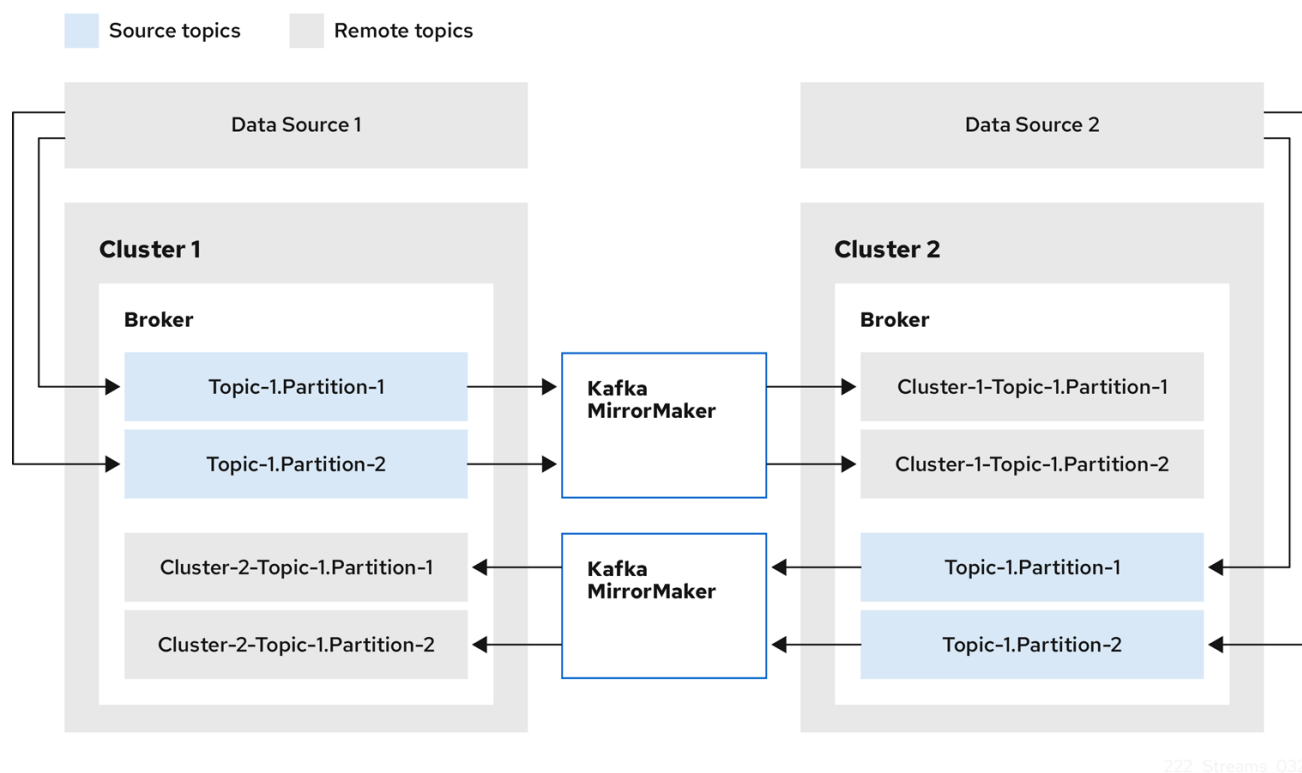


By flagging the originating cluster, topics are not replicated back to that cluster.

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.

### 2.4.1.1.3. 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. With this configuration applied, topics retain their original names.

### 2.4.1.2. Topic configuration synchronization

MirrorMaker 2.0 supports topic configuration synchronization between source and target clusters. You specify source topics in the MirrorMaker 2.0 configuration. MirrorMaker 2.0 monitors the source topics. MirrorMaker 2.0 detects and propagates changes to the source topics to the remote topics. Changes might include automatically creating missing topics and partitions.

> **NOTE**
>
> In most cases you write to local topics and read from remote topics. Though write operations are not prevented on remote topics, they should be avoided.

### 2.4.1.3. Offset tracking

MirrorMaker 2.0 tracks offsets for consumer groups using internal topics.

**offset-syncs** topic

The **offset-syncs** topic maps the source and target offsets for replicated topic partitions from record metadata.

**checkpoints** topic

The **checkpoints** topic maps the last committed offset in the source and target cluster for replicated topic partitions in each consumer group.

As they used internally by MirrorMaker 2.0, you do not interact directly with these topics.

**MirrorCheckpointConnector** emits *checkpoints* for offset tracking. Offsets for the **checkpoints** topic are tracked at predetermined intervals through configuration. Both topics enable replication to be fully restored from the correct offset position on failover.

The location of the **offset-syncs** topic is the **source** cluster by default. You can use the **offset-syncs.topic.location** connector configuration to change this to the **target** cluster. You need read/write access to the cluster that contains the topic. Using the target cluster as the location of the **offset-syncs** topic allows you to use MirrorMaker 2.0 even if you have only read access to the source cluster.

## 2.4.1.4. Synchronizing consumer group offsets

The **__consumer_offsets** topic stores information on committed offsets for each consumer group. Offset synchronization periodically transfers the consumer offsets for the consumer groups of a source cluster into the consumer offsets topic of a target cluster.

Offset synchronization is particularly useful in an *active/passive* configuration. If the active cluster goes down, consumer applications can switch to the passive (standby) cluster and pick up from the last transferred offset position.

To use topic offset synchronization, enable the synchronization by adding **sync.group.offsets.enabled** to the checkpoint connector configuration, and setting the property to **true**. Synchronization is disabled by default.

When using the **IdentityReplicationPolicy** in the source connector, it also has to be configured in the checkpoint connector configuration. This ensures that the mirrored consumer offsets will be applied for the correct topics.

Consumer offsets are only synchronized for consumer groups that are not active in the target cluster. If the consumer groups are in the target cluster, the synchronization cannot be performed and an **UNKNOWN_MEMBER_ID** error is returned.

If enabled, the synchronization of offsets from the source cluster is made periodically. You can change the frequency by adding **sync.group.offsets.interval.seconds** and **emit.checkpoints.interval.seconds** to the checkpoint connector configuration. The properties specify the frequency in seconds that the consumer group offsets are synchronized, and the frequency of checkpoints emitted for offset tracking. The default for both properties is 60 seconds. You can also change the frequency of checks for new consumer groups using the **refresh.groups.interval.seconds** property, which is performed every 10 minutes by default.

Because the synchronization is time-based, any switchover by consumers to a passive cluster will likely result in some duplication of messages.

NOTE

If you have an application written in Java, you can use the **RemoteClusterUtils.java** utility to synchronize offsets through the application. The utility fetches remote offsets for a consumer group from the **checkpoints** topic.

### 2.4.1.5. Connectivity checks

**MirrorHeartbeatConnector** emits *heartbeats* to check connectivity between clusters.

An internal **heartbeat** topic is replicated from the source cluster. Target clusters use the **heartbeat** topic to check the following:

- The connector managing connectivity between clusters is running

- The source cluster is available

### 2.4.2. Connector configuration

Use Mirrormaker 2.0 connector configuration for the internal connectors that orchestrate the synchronization of data between Kafka clusters.

The following table describes connector properties and the connectors you configure to use them.

Table 2.1. MirrorMaker 2.0 connector configuration properties

| Property | sourceConnector | checkpointConnector | heartbeatConnector |
|---|---|---|---|
| **admin.timeout.ms**<br>Timeout for admin tasks, such as detecting new topics. Default is **60000** (1 minute). | ✓ | ✓ | ✓ |
| **replication.policy.class**<br>Policy to define the remote topic naming convention. Default is **org.apache.kafka.connect.mirror .DefaultReplicationPolicy**. | ✓ | ✓ | ✓ |
| **replication.policy.separator**<br>The separator used for topic naming in the target cluster. Default is **.** (dot). It is only used when the **replication.policy.class** is the **DefaultReplicationPolicy**. | ✓ | ✓ | ✓ |
| **consumer.poll.timeout.ms**<br>Timeout when polling the source cluster. Default is **1000** (1 second). | ✓ | ✓ | |

| Property | sourceConnector | checkpointConnector | heartbeatConnector |
|---|---|---|---|
| **offset-syncs.topic.location**<br><br>The location of the **offset-syncs** topic, which can be the **source** (default) or **target** cluster. | ✓ | ✓ | |
| **topic.filter.class**<br><br>Topic filter to select the topics to replicate. Default is **org.apache.kafka.connect.mirror .DefaultTopicFilter**. | ✓ | ✓ | |
| **config.property.filter.class**<br><br>Topic filter to select the topic configuration properties to replicate. Default is **org.apache.kafka.connect.mirror .DefaultConfigPropertyFilter**. | ✓ | | |
| **config.properties.exclude**<br><br>Topic configuration properties that should not be replicated. Supports comma-separated property names and regular expressions. | ✓ | | |
| **offset.lag.max**<br><br>Maximum allowable (out-of-sync) offset lag before a remote partition is synchronized. Default is **100**. | ✓ | | |
| **offset-syncs.topic.replication.factor**<br><br>Replication factor for the internal **offset-syncs** topic. Default is **3**. | ✓ | | |
| **refresh.topics.enabled**<br><br>Enables check for new topics and partitions. Default is **true**. | ✓ | | |
| **refresh.topics.interval.seconds**<br><br>Frequency of topic refresh. Default is **600** (10 minutes). | ✓ | | |

| Property | sourceConnector | checkpointConnector | heartbeatConnector |
|---|---|---|---|
| **replication.factor**<br><br>The replication factor for new topics. Default is **2**. | ✓ | | |
| **sync.topic.acls.enabled**<br><br>Enables synchronization of ACLs from the source cluster. Default is **true**. Not compatible with the User Operator. | ✓ | | |
| **sync.topic.acls.interval.seconds**<br><br>Frequency of ACL synchronization. Default is **600** (10 minutes). | ✓ | | |
| **sync.topic.configs.enabled**<br><br>Enables synchronization of topic configuration from the source cluster. Default is **true**. | ✓ | | |
| **sync.topic.configs.interval.seconds**<br><br>Frequency of topic configuration synchronization. Default **600** (10 minutes). | ✓ | | |
| **checkpoints.topic.replication.factor**<br><br>Replication factor for the internal **checkpoints** topic. Default is **3**. | | ✓ | |
| **emit.checkpoints.enabled**<br><br>Enables synchronization of consumer offsets to the target cluster. Default is **true**. | | ✓ | |
| **emit.checkpoints.interval.seconds**<br><br>Frequency of consumer offset synchronization. Default is **60** (1 minute). | | ✓ | |

| Property | sourceConnector | checkpointConnector | heartbeatConnector |
|---|---|---|---|
| **group.filter.class**<br>Group filter to select the consumer groups to replicate. Default is **org.apache.kafka.connect.mirror.DefaultGroupFilter**. | | ✓ | |
| **refresh.groups.enabled**<br>Enables check for new consumer groups. Default is **true**. | | ✓ | |
| **refresh.groups.interval.seconds**<br>Frequency of consumer group refresh. Default is **600** (10 minutes). | | ✓ | |
| **sync.group.offsets.enabled**<br>Enables synchronization of consumer group offsets to the target cluster **__consumer_offsets** topic. Default is **false**. | | ✓ | |
| **sync.group.offsets.interval.seconds**<br>Frequency of consumer group offset synchronization. Default is **60** (1 minute). | | ✓ | |
| **emit.heartbeats.enabled**<br>Enables connectivity checks on the target cluster. Default is **true**. | | | ✓ |
| **emit.heartbeats.interval.seconds**<br>Frequency of connectivity checks. Default is **1** (1 second). | | | ✓ |
| **heartbeats.topic.replication.factor**<br>Replication factor for the internal **heartbeats** topic. Default is **3**. | | | ✓ |

## 2.4.3. Connector producer and consumer configuration

MirrorMaker 2.0 connectors use internal producers and consumers. If needed, you can configure these producers and consumers to override the default settings.

For example, you can increase the **batch.size** for the source producer that sends topics to the target Kafka cluster to better accommodate large volumes of messages.

**IMPORTANT**

Producer and consumer configuration options depend on the MirrorMaker 2.0 implementation, and may be subject to change.

The following tables describe the producers and consumers for each of the connectors and where you can add configuration.

Table 2.2. Source connector producers and consumers

| Type | Description | Configuration |
| --- | --- | --- |
| Producer | Sends topic messages to the target Kafka cluster. Consider tuning the configuration of this producer when it is handling large volumes of data. | **mirrors.sourceConnector.config: producer.override.*** |
| Producer | Writes to the **offset-syncs** topic, which maps the source and target offsets for replicated topic partitions. | **mirrors.sourceConnector.config: producer.*** |
| Consumer | Retrieves topic messages from the source Kafka cluster. | **mirrors.sourceConnector.config: consumer.*** |

Table 2.3. Checkpoint connector producers and consumers

| Type | Description | Configuration |
| --- | --- | --- |
| Producer | Emits consumer offset checkpoints. | **mirrors.checkpointConnector.config: producer.override.*** |
| Consumer | Loads the **offset-syncs** topic. | **mirrors.checkpointConnector.config: consumer.*** |
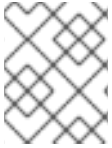
**NOTE**

You can set **offset-syncs.topic.location** to **target** to use the target Kafka cluster as the location of the **offset-syncs** topic.

Table 2.4. Heartbeat connector producer

| Type | Description | Configuration |
|------|-------------|---------------|
| Producer | Emits heartbeats. | **mirrors.heartbeatConnector.config: producer.override.*** |

The following example shows how you configure the producers and consumers.

**Example configuration for connector producers and consumers**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.3.1
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 5
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100
        producer.request.timeout.ms: 30000
        consumer.fetch.max.bytes: 52428800
        # ...
    checkpointConnector:
      config:
        producer.override.request.timeout.ms: 30000
        consumer.max.poll.interval.ms: 300000
        # ...
    heartbeatConnector:
      config:
        producer.override.request.timeout.ms: 30000
        # ...
```

**Additional resources**

- Section 12.2.130, "**KafkaMirrorMaker2ConnectorSpec** schema reference"

- Section 12.2.129, "**KafkaMirrorMaker2MirrorSpec** schema reference"

## 2.4.4. Specifying a maximum number of tasks

Connectors create the tasks that are responsible for moving data in and out of Kafka. Each connector

comprises one or more tasks that are distributed across a group of worker pods that run the tasks. Increasing the number of tasks can help with performance issues when replicating a large number of partitions or synchronizing the offsets of a large number of consumer groups.

Tasks run in parallel. Workers are assigned one or more tasks. A single task is handled by one worker pod, so you don't need more worker pods than tasks. If there are more tasks than workers, workers handle multiple tasks.

You can specify the maximum number of connector tasks in your MirrorMaker configuration using the **tasksMax** property. Without specifying a maximum number of tasks, the default setting is a single task.

The heartbeat connector always uses a single task.

The number of tasks that are started for the source and checkpoint connectors is the lower value between the maximum number of possible tasks and the value for **tasksMax**. For the source connector, the maximum number of tasks possible is one for each partition being replicated from the source cluster. For the checkpoint connector, the maximum number of tasks possible is one for each consumer group being replicated from the source cluster. When setting a maximum number of tasks, consider the number of partitions and the hardware resources that support the process.

If the infrastructure supports the processing overhead, increasing the number of tasks can improve throughput and latency. For example, adding more tasks reduces the time taken to poll the source cluster when there is a high number of partitions or consumer groups.

Increasing the number of tasks for the checkpoint connector is useful when you have a large number of partitions.

**Increasing the number of tasks for the source connector**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 10
  # ...
```

Increasing the number of tasks for the checkpoint connector is useful when you have a large number of consumer groups.

**Increasing the number of tasks for the checkpoint connector**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
```

```
  checkpointConnector:
    tasksMax: 10
# ...
```

By default, MirrorMaker 2.0 checks for new consumer groups every 10 minutes. You can adjust the **refresh.groups.interval.seconds** configuration to change the frequency. Take care when adjusting lower. More frequent checks can have a negative impact on performance.

### 2.4.4.1. Checking connector task operations

If you are using Prometheus and Grafana to monitor your deployment, you can check MirrorMaker 2.0 performance. The example MirrorMaker 2.0 Grafana dashboard provided with AMQ Streams shows the following metrics related to tasks and latency.

- The number of tasks

- Replication latency

- Offset synchronization latency

**Additional resources**

- [Grafana dashboards](#)

### 2.4.5. 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.

### 2.4.6. Configuring Kafka MirrorMaker 2.0

Use the properties of the **KafkaMirrorMaker2** resource to configure your Kafka MirrorMaker 2.0 deployment. Use MirrorMaker 2.0 to synchronize data between Kafka clusters.

The configuration must specify:

- Each Kafka cluster

- Connection information for each cluster, including authentication

- The replication flow and direction

  - Cluster to cluster

  - Topic to topic

**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:

**Minimal configuration for MirrorMaker 2.0**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.3.1
  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 mTLS or SASL authentication. This procedure shows a configuration that uses TLS encryption and mTLS authentication for the source and target cluster.

You can specify the topics and consumer groups you wish to replicate from a source cluster in the **KafkaMirrorMaker2** resource. You use the **topicsPattern** and **groupsPattern** properties to do this. You can provide a list of names or use a regular expression. By default, all topics and consumer groups are replicated if you do not set the **topicsPattern** and **groupsPattern** properties. You can also replicate all topics and consumer groups by using **".*"** as a regular expression. However, try to specify only the topics and consumer groups you need to avoid causing any unnecessary extra load on the cluster.

### Handling high volumes of messages

You can tune the configuration to handle high volumes of messages. For more information, see Section 2.7, "Handling high volumes of messages".

### Prerequisites

- AMQ Streams is running

- Source and target Kafka clusters are available

### Procedure

1. Edit the **spec** properties for the **KafkaMirrorMaker2** resource.
   The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.3.1 ❶
  replicas: 3 ❷
  connectCluster: "my-cluster-target" ❸
  clusters: ❹
  - alias: "my-cluster-source" ❺
    authentication: ❻
      certificateAndKey:
        certificate: source.crt
        key: source.key
        secretName: my-user-source
      type: tls
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092 ❼
    tls: ❽
      trustedCertificates:
      - certificate: ca.crt
        secretName: my-cluster-source-cluster-ca-cert
  - alias: "my-cluster-target" ❾
    authentication: ❿
      certificateAndKey:
        certificate: target.crt
        key: target.key
        secretName: my-user-target
      type: tls
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092 ⓫
    config: ⓬
      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" ⓭
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
      ssl.endpoint.identification.algorithm: HTTPS ⓮
    tls: ⓯
      trustedCertificates:
      - certificate: ca.crt
        secretName: my-cluster-target-cluster-ca-cert
  mirrors: ⓰
  - sourceCluster: "my-cluster-source" ⓱
    targetCluster: "my-cluster-target" ⓲
    sourceConnector: ⓳
      tasksMax: 10 ⓴
      config:
        replication.factor: 1 ㉑
        offset-syncs.topic.replication.factor: 1 ㉒
        sync.topic.acls.enabled: "false" ㉓
        refresh.topics.interval.seconds: 60 ㉔
        replication.policy.separator: "" ㉕
```

```yaml
      replication.policy.class: "org.apache.kafka.connect.mirror.IdentityReplicationPolicy" 26
    heartbeatConnector: 27
      config:
        heartbeats.topic.replication.factor: 1 28
    checkpointConnector: 29
      config:
        checkpoints.topic.replication.factor: 1 30
        refresh.groups.interval.seconds: 600 31
        sync.group.offsets.enabled: true 32
        sync.group.offsets.interval.seconds: 60 33
        emit.checkpoints.interval.seconds: 60 34
        replication.policy.class: "org.apache.kafka.connect.mirror.IdentityReplicationPolicy"
    topicsPattern: "topic1|topic2|topic3" 35
    groupsPattern: "group1|group2|group3" 36
  resources: 37
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  logging: 38
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  readinessProbe: 39
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  jvmOptions: 40
    "-Xmx": "1g"
    "-Xms": "1g"
  image: my-org/my-image:latest 41
  rack:
    topologyKey: topology.kubernetes.io/zone 42
  template: 43
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
    connectContainer: 44
      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 45
  externalConfiguration: 46
    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 and Mirror Maker 2.0 version, which will always be the same.

**2** The number of replica nodes for the workers that run tasks.

**3** Kafka cluster alias for Kafka Connect, which must specify the **target** Kafka cluster. The Kafka cluster is used by Kafka Connect for its internal topics.

**4** Specification for the Kafka clusters being synchronized.

**5** Cluster alias for the source Kafka cluster.

**6** Authentication for the source cluster, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN.

**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** The maximum number of tasks that the connector may create. Tasks handle the data replication and run in parallel. If the infrastructure supports the processing overhead, increasing this value can improve throughput. Kafka Connect distributes the tasks between members of the cluster. If there are more tasks than workers, workers are assigned multiple tasks. For sink connectors, aim to have one task for each topic partition consumed. For source connectors, the number of tasks that can run in parallel may also depend on the external system. The connector creates fewer than the maximum number of tasks if it cannot achieve the parallelism.

**21** Replication factor for mirrored topics created at the target cluster.

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

**23** When ACL rules synchronization is enabled, ACLs are applied to synchronized topics. The default is **true**. This feature is not compatible with the User Operator. If you are using the User Operator, set this property to **false**.

**24** Optional setting to change the frequency of checks for new topics. The default is for a check every 10 minutes.

**25** Defines the separator used for the renaming of remote topics.

**26** 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. To configure topic offset synchronization, this property must also be set for the **checkpointConnector.config**.

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

**28** Replication factor for the heartbeat topic created at the target cluster.

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

**30** Replication factor for the checkpoints topic created at the target cluster.

**31** Optional setting to change the frequency of checks for new consumer groups. The default is for a check every 10 minutes.

**32** Optional setting to synchronize consumer group offsets, which is useful for recovery in an active/passive configuration. Synchronization is not enabled by default.

**33** If the synchronization of consumer group offsets is enabled, you can adjust the frequency of the synchronization.

**34** Adjusts the frequency of checks for offset tracking. If you change the frequency of offset synchronization, you might also need to adjust the frequency of these checks.

**35** Topic replication from the source cluster defined as a comma-separated list or regular expression pattern. The source connector replicates the specified topics. The checkpoint connector tracks offsets for the specified topics. Here we request three topics by name.

**36** Consumer group replication from the source cluster defined as a comma-separated list or regular expression pattern. The checkpoint connector replicates the specified consumer groups. Here we request three consumer groups by name.

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

**38** 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.

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

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

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

**42** SPECIALIZED OPTION: Rack awareness configuration for the deployment. This is a specialized option intended for a deployment within the same location, not across regions. Use this option if you want connectors to consume from the closest replica rather than the leader replica. In certain cases, consuming from the closest replica can improve network utilization or reduce costs . The **topologyKey** must match a node label containing the rack ID. The example used in this configuration specifies a zone using the standard **topology.kubernetes.io/zone** label. To consume from the closest replica, enable the **RackAwareReplicaSelector** in the Kafka broker configuration.

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

**44** Environment variables are set for distributed tracing.

**45** Distributed tracing is enabled for Jaeger.

**46** External configuration for an OpenShift Secret mounted to Kafka MirrorMaker as an environment variable. You can also use *configuration provider plugins* to load configuration values from external sources.

2. Create or update the resource:

```
oc apply -f MIRRORMAKER-CONFIGURATION-FILE
```

**Additional resources**

- Introducing distributed tracing

## 2.4.7. Securing a Kafka MirrorMaker 2.0 deployment

This procedure describes in outline the configuration required to secure a MirrorMaker 2.0 deployment.

You need separate configuration for the source Kafka cluster and the target Kafka cluster. You also need separate user configuration to provide the credentials required for MirrorMaker to connect to the source and target Kafka clusters.

For the Kafka clusters, you specify internal listeners for secure connections within an OpenShift cluster and external listeners for connections outside the OpenShift cluster.

You can configure authentication and authorization mechanisms. The security options implemented for the source and target Kafka clusters must be compatible with the security options implemented for MirrorMaker 2.0.

After you have created the cluster and user authentication credentials, you specify them in your MirrorMaker configuration for secure connections.

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

### Before you start

Before starting this procedure, take a look at the example configuration files provided by AMQ Streams. They include examples for securing a deployment of MirrorMaker 2.0 using mTLS or SCRAM-SHA-512 authentication. The examples specify internal listeners for connecting within an OpenShift cluster.

The examples provide the configuration for full authorization, including all the ACLs needed by MirrorMaker 2.0 to allow operations on the source and target Kafka clusters.

### Prerequisites

- AMQ Streams is running

- Separate namespaces for source and target clusters

The procedure assumes that the source and target Kafka clusters are installed to separate namespaces If you want to use the Topic Operator, you'll need to do this. The Topic Operator only watches a single cluster in a specified namespace.

By separating the clusters into namespaces, you will need to copy the cluster secrets so they can be accessed outside the namespace. You need to reference the secrets in the MirrorMaker configuration.

### Procedure

1. Configure two **Kafka** resources, one to secure the source Kafka cluster and one to secure the target Kafka cluster.
   You can add listener configuration for authentication and enable authorization.

   In this example, an internal listener is configured for a Kafka cluster with TLS encryption and mTLS authentication. Kafka **simple** authorization is enabled.

Example source Kafka cluster configuration with TLS encryption and mTLS authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-source-cluster
spec:
  kafka:
    version: 3.3.1
    replicas: 1
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      default.replication.factor: 1
      min.insync.replicas: 1
      inter.broker.protocol.version: "3.3"
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
  zookeeper:
    replicas: 1
    storage:
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

Example target Kafka cluster configuration with TLS encryption and mTLS authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-target-cluster
spec:
  kafka:
    version: 3.3.1
    replicas: 1
```

```
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      default.replication.factor: 1
      min.insync.replicas: 1
      inter.broker.protocol.version: "3.3"
    storage:
      type: jbod
      volumes:
        - id: 0
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
  zookeeper:
    replicas: 1
    storage:
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Create or update the **Kafka** resources in separate namespaces.

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

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

The certificates are created in the secret ***&lt;cluster_name&gt;*-cluster-ca-cert**.

3. Configure two **KafkaUser** resources, one for a user of the source Kafka cluster and one for a user of the target Kafka cluster.

   a. Configure the same authentication and authorization types as the corresponding source and target Kafka cluster. For example, if you used **tls** authentication and the **simple** authorization type in the **Kafka** configuration for the source Kafka cluster, use the same in the **KafkaUser** configuration.

   b. Configure the ACLs needed by MirrorMaker 2.0 to allow operations on the source and target Kafka clusters.
   The ACLs are used by the internal MirrorMaker connectors, and by the underlying Kafka Connect framework.

### Example source user configuration for mTLS authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-source-user
  labels:
    strimzi.io/cluster: my-source-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
    acls:
      # MirrorSourceConnector
      - resource: # Not needed if offset-syncs.topic.location=target
          type: topic
          name: mm2-offset-syncs.my-target-cluster.internal
        operations:
          - Create
          - DescribeConfigs
          - Read
          - Write
      - resource: # Needed for every topic which is mirrored
          type: topic
          name: "*"
        operations:
          - DescribeConfigs
          - Read
      # MirrorCheckpointConnector
      - resource:
          type: cluster
        operations:
          - Describe
      - resource: # Needed for every group for which offsets are synced
          type: group
          name: "*"
        operations:
          - Describe
      - resource: # Not needed if offset-syncs.topic.location=target
          type: topic
          name: mm2-offset-syncs.my-target-cluster.internal
        operations:
          - Read
```

**Example target user configuration for mTLS authentication**

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

```yaml
type: simple
acls:
  # Underlying Kafka Connect internal topics to store configuration, offsets, or status
  - resource:
      type: group
      name: mirrormaker2-cluster
    operations:
      - Read
  - resource:
      type: topic
      name: mirrormaker2-cluster-configs
    operations:
      - Create
      - Describe
      - DescribeConfigs
      - Read
      - Write
  - resource:
      type: topic
      name: mirrormaker2-cluster-status
    operations:
      - Create
      - Describe
      - DescribeConfigs
      - Read
      - Write
  - resource:
      type: topic
      name: mirrormaker2-cluster-offsets
    operations:
      - Create
      - Describe
      - DescribeConfigs
      - Read
      - Write
  # MirrorSourceConnector
  - resource: # Needed for every topic which is mirrored
      type: topic
      name: "*"
    operations:
      - Create
      - Alter
      - AlterConfigs
      - Write
  # MirrorCheckpointConnector
  - resource:
      type: cluster
    operations:
      - Describe
  - resource:
      type: topic
      name: my-source-cluster.checkpoints.internal
    operations:
      - Create
      - Describe
      - Read
```

```
        - Write
    - resource: # Needed for every group for which the offset is synced
        type: group
        name: "*"
      operations:
        - Read
        - Describe
    # MirrorHeartbeatConnector
    - resource:
        type: topic
        name: heartbeats
      operations:
        - Create
        - Describe
        - Write
```
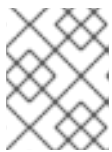
> **NOTE**
>
> You can use a certificate issued outside the User Operator by setting **type** to **tls-external**. For more information, see User authentication.

4. Create or update a **KafkaUser** resource in each of the namespaces you created for the source and target Kafka clusters.

   ```
   oc apply -f <kafka_user_configuration_file> -n <namespace>
   ```

   The User Operator creates the users representing the client (MirrorMaker), and the security credentials used for client authentication, based on the chosen authentication type.

   The User Operator creates a new secret with the same name as the **KafkaUser** resource. The secret contains a private and public key for mTLS authentication. The public key is contained in a user certificate, which is signed by the clients CA.

5. Configure a **KafkaMirrorMaker2** resource with the authentication details to connect to the source and target Kafka clusters.

   **Example MirrorMaker 2.0 configuration with TLS encryption and mTLS authentication**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaMirrorMaker2
   metadata:
     name: my-mirror-maker-2
   spec:
     version: 3.3.1
     replicas: 1
     connectCluster: "my-target-cluster"
     clusters:
       - alias: "my-source-cluster"
         bootstrapServers: my-source-cluster-kafka-bootstrap:9093
         tls: 1
           trustedCertificates:
             - secretName: my-source-cluster-cluster-ca-cert
               certificate: ca.crt
   ```

```
        authentication: 2
          type: tls
          certificateAndKey:
            secretName: my-source-user
            certificate: user.crt
            key: user.key
      - alias: "my-target-cluster"
        bootstrapServers: my-target-cluster-kafka-bootstrap:9093
        tls: 3
          trustedCertificates:
            - secretName: my-target-cluster-cluster-ca-cert
              certificate: ca.crt
        authentication: 4
          type: tls
          certificateAndKey:
            secretName: my-target-user
            certificate: user.crt
            key: user.key
        config:
          # -1 means it will use the default replication factor configured in the broker
          config.storage.replication.factor: -1
          offset.storage.replication.factor: -1
          status.storage.replication.factor: -1
    mirrors:
      - sourceCluster: "my-source-cluster"
        targetCluster: "my-target-cluster"
        sourceConnector:
          config:
            replication.factor: 1
            offset-syncs.topic.replication.factor: 1
            sync.topic.acls.enabled: "false"
        heartbeatConnector:
          config:
            heartbeats.topic.replication.factor: 1
        checkpointConnector:
          config:
            checkpoints.topic.replication.factor: 1
            sync.group.offsets.enabled: "true"
        topicsPattern: "topic1|topic2|topic3"
        groupsPattern: "group1|group2|group3"
```

**1** The TLS certificates for the source Kafka cluster. If they are in a separate namespace, copy the cluster secrets from the namespace of the Kafka cluster.

**2** The user authentication for accessing the source Kafka cluster using the TLS mechanism.

**3** The TLS certificates for the target Kafka cluster.

**4** The user authentication for accessing the target Kafka cluster.

6. Create or update the **KafkaMirrorMaker2** resource in the same namespace as the target Kafka cluster.

```
oc apply -f <mirrormaker2_configuration_file> -n <namespace_of_target_cluster>
```

**Additional resources**

- [Supported listener authentication options](#)

- [Supported authorization options for a Kafka cluster](#)

- [Securing Kafka brokers](#)

- [Securing user access to Kafka](#)

- [Managing TLS certificates](#)

## 2.4.8. Performing a restart of a Kafka MirrorMaker 2.0 connector

This procedure describes how to manually trigger a restart of a Kafka MirrorMaker 2.0 connector by using an OpenShift annotation.

**Prerequisites**

- The Cluster Operator is running.

**Procedure**

1. Find the name of the **KafkaMirrorMaker2** custom resource that controls the Kafka MirrorMaker 2.0 connector you want to restart:

   ```
   oc get KafkaMirrorMaker2
   ```

2. Find the name of the Kafka MirrorMaker 2.0 connector to be restarted from the **KafkaMirrorMaker2** custom resource.

   ```
   oc describe KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME
   ```

3. To restart the connector, annotate the **KafkaMirrorMaker2** resource in OpenShift. In this example, **oc annotate** restarts a connector named **my-source->my-target.MirrorSourceConnector**:

   ```
   oc annotate KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME "strimzi.io/restart-connector=my-source->my-target.MirrorSourceConnector"
   ```

4. Wait for the next reconciliation to occur (every two minutes by default).
   The Kafka MirrorMaker 2.0 connector is restarted, as long as the annotation was detected by the reconciliation process. When the restart request is accepted, the annotation is removed from the **KafkaMirrorMaker2** custom resource.

**Additional resources**

- [Kafka MirrorMaker 2.0 cluster configuration](#) .

## 2.4.9. Performing a restart of a Kafka MirrorMaker 2.0 connector task

This procedure describes how to manually trigger a restart of a Kafka MirrorMaker 2.0 connector task by using an OpenShift annotation.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Find the name of the **KafkaMirrorMaker2** custom resource that controls the Kafka MirrorMaker 2.0 connector you want to restart:

   ```
   oc get KafkaMirrorMaker2
   ```

2. Find the name of the Kafka MirrorMaker 2.0 connector and the ID of the task to be restarted from the **KafkaMirrorMaker2** custom resource. Task IDs are non-negative integers, starting from 0.

   ```
   oc describe KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME
   ```

3. To restart the connector task, annotate the **KafkaMirrorMaker2** resource in OpenShift. In this example, **oc annotate** restarts task *0* of a connector named **my-source->my-target.MirrorSourceConnector**:

   ```
   oc annotate KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME "strimzi.io/restart-connector-task=my-source->my-target.MirrorSourceConnector:0"
   ```

4. Wait for the next reconciliation to occur (every two minutes by default).
   The Kafka MirrorMaker 2.0 connector task is restarted, as long as the annotation was detected by the reconciliation process. When the restart task request is accepted, the annotation is removed from the **KafkaMirrorMaker2** custom resource.

Additional resources

- Kafka MirrorMaker 2.0 cluster configuration .

## 2.5. KAFKA MIRRORMAKER CLUSTER CONFIGURATION

Configure a Kafka MirrorMaker deployment using the **KafkaMirrorMaker** resource. KafkaMirrorMaker replicates data between Kafka clusters.

Section 12.2.108, "**KafkaMirrorMaker** schema reference" describes the full schema of the **KafkaMirrorMaker** resource.

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.

IMPORTANT

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

## 2.5.1. 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 mTLS 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/v1beta2
   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
include: "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
metricsConfig: 18
 type: jmxPrometheusExporter
 valueFrom:
  configMapKeyRef:
    name: my-config-map
    key: my-key
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 times.

**7** Authentication for consumer or producer, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN.

**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 included 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 by referencing a ConfigMap containing configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using a reference to a ConfigMap containing an empty file under **metricsConfig.valueFrom.configMapKeyRef.key**.

**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 set for distributed tracing.

**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>
   ```

**Additional resources**

- **Introducing distributed tracing**

## 2.5.2. 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.

## 2.6. KAFKA BRIDGE CLUSTER CONFIGURATION

Configure a Kafka Bridge deployment using the **KafkaBridge** resource. Kafka Bridge provides an API for integrating HTTP-based clients with a Kafka cluster.

Section 12.2.114, "**KafkaBridge** schema reference" describes the full schema of the **KafkaBridge** resource.

### 2.6.1. 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/v1beta2
   kind: KafkaBridge
   metadata:
     name: my-bridge
   spec:
     replicas: 3 ❶
     bootstrapServers: <cluster_name>-cluster-kafka-bootstrap:9092 ❷
     tls: ❸
       trustedCertificates:
         - secretName: my-cluster-cluster-cert
           certificate: ca.crt
         - secretName: my-cluster-cluster-cert
           certificate: ca2.crt
     authentication: ❹
       type: tls
       certificateAndKey:
         secretName: my-secret
         certificate: public.crt
         key: private.key
   ```

```yaml
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. Use the name of the Kafka cluster as the *<cluster_name>*.

**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, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN. 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** Optional: 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 set for distributed tracing.

2. Create or update the resource:

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

**Additional resources**

- Using the AMQ Streams Kafka Bridge

- Introducing distributed tracing

## 2.6.2. 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.

# 2.7. HANDLING HIGH VOLUMES OF MESSAGES

If your AMQ Streams deployment needs to handle a high volume of messages, you can use configuration options to optimize for throughput and latency.

Producer and consumer configuration can help control the size and frequency of requests to Kafka brokers. For more information on the configuration options, see the following:

- Apache Kafka configuration documentation for producers

- Apache Kafka configuration documentation for consumers

You can also use the same configuration options with the producers and consumers used by the Kafka Connect runtime source connectors (including MirrorMaker 2.0) and sink connectors.

**Source connectors**

- Producers from the Kafka Connect runtime send messages to the Kafka cluster.

- For MirrorMaker 2.0, since the source system is Kafka, consumers retrieve messages from a source Kafka cluster.

**Sink connectors**

- Consumers from the Kafka Connect runtime retrieve messages from the Kafka cluster.

For consumers, you might increase the amount of data fetched in a single fetch request to reduce latency. You increase the fetch request size using the **fetch.max.bytes** and **max.partition.fetch.bytes** properties. You can also set a maximum limit on the number of messages returned from the consumer buffer using the **max.poll.records** property.

For MirrorMaker 2.0, configure the **fetch.max.bytes**, **max.partition.fetch.bytes**, and **max.poll.records** values at the source connector level (**consumer.***), as they relate to the specific consumer that fetches messages from the source.

For producers, you might increase the size of the message batches sent in a single produce request. You increase the batch size using the **batch.size** property. A larger batch size reduces the number of outstanding messages ready to be sent and the size of the backlog in the message queue. Messages being sent to the same partition are batched together. A produce request is sent to the target cluster

when the batch size is reached. By increasing the batch size, produce requests are delayed and more messages are added to the batch and sent to brokers at the same time. This can improve throughput when you have just a few topic partitions that handle large numbers of messages.

Consider the number and size of the records that the producer handles for a suitable producer batch size.

Use **linger.ms** to add a wait time in milliseconds to delay produce requests when producer load decreases. The delay means that more records can be added to batches if they are under the maximum batch size.

Configure the **batch.size** and **linger.ms** values at the source connector level ( **producer.override.\***), as they relate to the specific producer that sends messages to the target Kafka cluster.

For Kafka Connect source connectors, the data streaming pipeline to the target Kafka cluster is as follows:

### Data streaming pipeline for Kafka Connect source connector

external data source → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

For Kafka Connect sink connectors, the data streaming pipeline to the target external data source is as follows:

### Data streaming pipeline for Kafka Connect sink connector

source Kafka topic → (Kafka Connect tasks) sink message queue → consumer buffer → external data source

For MirrorMaker 2.0, the data mirroring pipeline to the target Kafka cluster is as follows:

### Data mirroring pipeline for MirrorMaker 2.0

source Kafka topic → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

The producer sends messages in its buffer to topics in the target Kafka cluster. While this is happening, Kafka Connect tasks continue to poll the data source to add messages to the source message queue.

The size of the producer buffer for the source connector is set using the **producer.override.buffer.memory** property. Tasks wait for a specified timeout period (**offset.flush.timeout.ms**) before the buffer is flushed. This should be enough time for the sent messages to be acknowledged by the brokers and offset data committed. The source task does not wait for the producer to empty the message queue before committing offsets, except during shutdown.

If the producer is unable to keep up with the throughput of messages in the source message queue, buffering is blocked until there is space available in the buffer within a time period bounded by **max.block.ms**. Any unacknowledged messages still in the buffer are sent during this period. New messages are not added to the buffer until these messages are acknowledged and flushed.

You can try the following configuration changes to keep the underlying source message queue of outstanding messages at a manageable size:

- Increasing the default value in milliseconds of the **offset.flush.timeout.ms**

- Ensuring that there are enough CPU and memory resources

- Increasing the number of tasks that run in parallel by doing the following:

  - Increasing the number of tasks that run in parallel using the **tasksMax** property

  - Increasing the number of worker nodes that run tasks using the **replicas** property

Consider the number of tasks that can run in parallel according to the available CPU and memory resources and number of worker nodes. You might need to keep adjusting the configuration values until they have the desired effect.

### 2.7.1. Configuring Kafka Connect for high-volume messages

Kafka Connect fetches data from the source external data system and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows configuration for Kafka Connect using the **KafkaConnect** custom resource.

**Example Kafka Connect configuration for handling high volumes of messages**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  replicas: 3
  config:
    offset.flush.timeout.ms: 10000
    # ...
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  # ...
```

Producer configuration is added for the source connector, which is managed using the **KafkaConnector** custom resource.

**Example source connector configuration for handling high volumes of messages**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
```

```
producer.override.batch.size: 327680
producer.override.linger.ms: 100
# ...
```
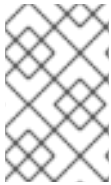
**NOTE**

**FileStreamSourceConnector** and **FileStreamSinkConnector** are provided as example connectors. For information on deploying them as **KafkaConnector** resources, see Deploying example KafkaConnector resources .

Consumer configuration is added for the sink connector.

**Example sink connector configuration for handling high volumes of messages**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector
  tasksMax: 2
  config:
    consumer.fetch.max.bytes: 52428800
    consumer.max.partition.fetch.bytes: 1048576
    consumer.max.poll.records: 500
    # ...
```

If you are using the Kafka Connect API instead of the **KafkaConnector** custom resource to manage your connectors, you can add the connector configuration as a JSON object.

**Example curl request to add source connector configuration for handling high volumes of messages**

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
      "producer.override.batch.size": 327680
      "producer.override.linger.ms": 100
    }
}'
```

## 2.7.2. Configuring MirrorMaker 2.0 for high-volume messages

MirrorMaker 2.0 fetches data from the source cluster and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows the configuration for MirrorMaker 2.0 using the **KafkaMirrorMaker2** custom resource.

**Example MirrorMaker 2.0 configuration for handling high volumes of messages**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.3.1
  replicas: 1
  connectCluster: "my-cluster-target"
  clusters:
  - alias: "my-cluster-source"
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
  - alias: "my-cluster-target"
    config:
      offset.flush.timeout.ms: 10000
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 2
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100
        consumer.fetch.max.bytes: 52428800
        consumer.max.partition.fetch.bytes: 1048576
        consumer.max.poll.records: 500
    # ...
  resources:
    requests:
      cpu: "1"
      memory: Gi
    limits:
      cpu: "2"
      memory: 4Gi
```

## 2.7.3. Checking the MirrorMaker 2.0 message flow

If you are using Prometheus and Grafana to monitor your deployment, you can check the MirrorMaker 2.0 message flow.

The example MirrorMaker 2.0 Grafana dashboards provided with AMQ Streams show the following metrics related to the flush pipeline.

- The number of messages in Kafka Connect's outstanding messages queue

- The available bytes of the producer buffer

- The offset commit timeout in milliseconds

You can use these metrics to gauge whether or not you need to tune your configuration based on the volume of messages.

**Additional resources**

- Grafana dashboards

- Creating and managing connectors

## 2.8. CUSTOMIZING OPENSHIFT RESOURCES

An AMQ Streams deployment creates OpenShift resources, such as **Deployments**, **StatefulSets**, **Pods**, and **Services**. These resources 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.

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 the 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 12.2.33, "**KafkaClusterTemplate** schema reference"

**Kafka.spec.zookeeper**

See Section 12.2.43, "**ZookeeperClusterTemplate** schema reference"

**Kafka.spec.entityOperator**

See Section 12.2.48, "**EntityOperatorTemplate** schema reference"

**Kafka.spec.kafkaExporter**

See Section 12.2.55, "**KafkaExporterTemplate** schema reference"

**Kafka.spec.cruiseControl**

See Section 12.2.51, "**CruiseControlTemplate** schema reference"

**KafkaConnect.spec**

See Section 12.2.71, "**KafkaConnectTemplate** schema reference"

**KafkaMirrorMaker.spec**

See Section 12.2.112, "**KafkaMirrorMakerTemplate** schema reference"

**KafkaMirrorMaker2.spec**

See Section 12.2.71, "**KafkaConnectTemplate** schema reference"

**KafkaBridge.spec**

See Section 12.2.121, "**KafkaBridgeTemplate** schema reference"

**KafkaUser.spec**

See Section 12.2.106, "**KafkaUserTemplate** schema reference"

In the following example, the **template** property is used to modify the labels in a Kafka broker's pod.

**Example template customization**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  labels:
    app: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        metadata:
          labels:
            mylabel: myvalue
    # ...
```

## 2.8.1. 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 7.2, "Using the Cluster Operator".

- For more information about Image Pull Policies, see Disruptions.

## 2.8.2. Applying a termination grace period

Apply a termination grace period to give a Kafka cluster enough time to shut down cleanly.

Specify the time using the **terminationGracePeriodSeconds** property. Add the property to the **template.pod** configuration of the **Kafka** custom resource.

The time you add will depend on the size of your Kafka cluster. The OpenShift default for the termination grace period is 30 seconds. If you observe that your clusters are not shutting down cleanly, you can increase the termination grace period.

A termination grace period is applied every time a pod is restarted. The period begins when OpenShift sends a *term* (termination) signal to the processes running in the pod. The period should reflect the amount of time required to transfer the processes of the terminating pod to another pod before they are stopped. After the period ends, a *kill* signal stops any processes still running in the pod.

The following example adds a termination grace period of 120 seconds to the **Kafka** custom resource. You can also specify the configuration in the custom resources of other Kafka components.

**Example termination grace period configuration**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        terminationGracePeriodSeconds: 120
        # ...
    # ...
```

# 2.9. CONFIGURING POD SCHEDULING

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.

## 2.9.1. Specifying affinity, tolerations, and topology spread constraints

Use affinity, tolerations and topology spread constraints to schedule the pods of kafka resources onto nodes. Affinity, tolerations and topology spread constraints are configured using the **affinity**, **tolerations**, and **topologySpreadConstraint** properties in following resources:

- **Kafka.spec.kafka.template.pod**

- **Kafka.spec.zookeeper.template.pod**

- **Kafka.spec.entityOperator.template.pod**

- **KafkaConnect.spec.template.pod**

- **KafkaBridge.spec.template.pod**

- **KafkaMirrorMaker.spec.template.pod**

- **KafkaMirrorMaker2.spec.template.pod**

The format of the **affinity**, **tolerations**, and **topologySpreadConstraint** properties follows the OpenShift specification. The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

**Additional resources**

- Kubernetes node and pod affinity documentation
- Kubernetes taints and tolerations
- Controlling pod placement by using pod topology spread constraints

### 2.9.1.1. Use pod anti-affinity to avoid critical applications sharing nodes

Use pod anti-affinity to ensure that critical applications are never scheduled on the same disk. When running a Kafka cluster, it is recommended to use pod anti-affinity to ensure that the Kafka brokers do not share nodes with other workloads, such as databases.

### 2.9.1.2. Use node affinity to schedule workloads onto specific nodes

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.

### 2.9.1.3. Use node affinity and tolerations for dedicated nodes

Use taints to create dedicated nodes, then schedule Kafka pods on the dedicated nodes by configuring node affinity and tolerations.

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.

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.

### 2.9.2. Configuring pod anti-affinity to schedule each Kafka broker on a different worker node

Many Kafka brokers or ZooKeeper nodes can run on the same OpenShift worker node. If the worker node fails, they will all become unavailable at the same time. To improve reliability, you can use **podAntiAffinity** configuration to schedule each Kafka broker or ZooKeeper node on a different OpenShift worker node.

**Prerequisites**

- An OpenShift cluster

- A running Cluster Operator

**Procedure**

1. Edit the **affinity** property in the resource specifying the cluster deployment. To make sure that no worker nodes are shared by Kafka brokers or ZooKeeper nodes, use the **strimzi.io/name** label. Set the **topologyKey** to **kubernetes.io/hostname** to specify that the selected pods are not scheduled on nodes with the same hostname. This will still allow the same worker node to be shared by a single Kafka broker and a single ZooKeeper node. For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/name
                    operator: In
                    values:
                      - CLUSTER-NAME-kafka
              topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/name
                    operator: In
                    values:
                      - CLUSTER-NAME-zookeeper
              topologyKey: "kubernetes.io/hostname"
    # ...
```

   Where **CLUSTER-NAME** is the name of your Kafka custom resource.

2. If you even want to make sure that a Kafka broker and ZooKeeper node do not share the same worker node, use the **strimzi.io/cluster** label. For example:

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

```
kafka:
  # ...
  template:
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/cluster
                    operator: In
                    values:
                      - CLUSTER-NAME
              topologyKey: "kubernetes.io/hostname"
  # ...
zookeeper:
  # ...
  template:
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/cluster
                    operator: In
                    values:
                      - CLUSTER-NAME
              topologyKey: "kubernetes.io/hostname"
  # ...
```

Where **CLUSTER-NAME** is the name of your Kafka custom resource.

3. Create or update the resource.

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

## 2.9.3. Configuring pod anti-affinity in Kafka components

Pod anti-affinity configuration helps with the stability and performance of Kafka brokers. By using **podAntiAffinity**, OpenShift will not schedule Kafka brokers on the same nodes as other workloads. Typically, you want to avoid Kafka running on the same worker node as other network or storage intensive applications such as databases, storage or other messaging platforms.

**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/v1beta2
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 <kafka_configuration_file>
   ```

## 2.9.4. 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 NAME-OF-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/v1beta2
   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 <kafka_configuration_file>
```

## 2.9.5. 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 NAME-OF-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 NAME-OF-NODE dedicated=Kafka
```

5. Edit the **affinity** and **tolerations** properties in the resource specifying the cluster deployment.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta2
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 <kafka_configuration_file>
   ```

## 2.10. LOGGING CONFIGURATION

Configure logging levels in the custom resources of Kafka components and AMQ Streams Operators.
You can specify the logging levels directly in the **spec.logging** property of the custom resource. Or you
can define the logging properties in a ConfigMap that's referenced in the custom resource using the
**configMapKeyRef** 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. You can also reuse the ConfigMap for more than one
resource. If you are using a ConfigMap to specify loggers for AMQ Streams Operators, you can also
append the logging specification to add filters.

You specify a logging **type** in your logging specification:

- **inline** when specifying logging levels directly

- **external** when referencing a ConfigMap

**Example inline logging configuration**

```
spec:
  # ...
  logging:
    type: inline
    loggers:
      kafka.root.logger.level: "INFO"
```

**Example external logging configuration**

```
spec:
```

```
# ...
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: my-config-map-key
```

Values for the **name** and **key** of the ConfigMap are mandatory. Default logging is used if the   **name** or **key** is not set.

## 2.10.1. Logging options for Kafka components and operators

For more information on configuring logging for specific Kafka components or operators, see the following sections.

**Kafka component logging**

- Kafka logging

- ZooKeeper logging

- Kafka Connect and Mirror Maker 2.0 logging

- MirrorMaker logging

- Kafka Bridge logging

- Cruise Control logging

**Operator logging**

- Cluster Operator logging

- Topic Operator logging

- User Operator logging

## 2.10.2. 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.

In this procedure 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.

   ConfigMap example with a root logger definition for Kafka:

   ```
   kind: ConfigMap
   apiVersion: v1
   metadata:
     name: logging-configmap
   data:
     log4j.properties:
       kafka.root.logger.level="INFO"
   ```

   If you are using a properties file, specify the file at the command line:

   ```
   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.valueFrom.configMapKeyRef.name** to the name of the ConfigMap and **logging.valueFrom.configMapKeyRef.key** to the key in this ConfigMap.

   ```
   spec:
     # ...
     logging:
       type: external
       valueFrom:
         configMapKeyRef:
           name: logging-configmap
           key: log4j.properties
   ```

3. Create or update the resource.

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

## 2.10.3. Adding logging filters to Operators

If you are using a ConfigMap to configure the (log4j2) logging levels for AMQ Streams Operators, you can also define logging filters to limit what's returned in the log.

Logging filters are useful when you have a large number of logging messages. Suppose you set the log level for the logger as DEBUG (**rootLogger.level="DEBUG"**). Logging filters reduce the number of logs returned for the logger at that level, so you can focus on a specific resource. When the filter is set, only log messages matching the filter are logged.

Filters use *markers* to specify what to include in the log. You specify a kind, namespace and name for the marker. For example, if a Kafka cluster is failing, you can isolate the logs by specifying the kind as **Kafka**, and use the namespace and name of the failing cluster.

This example shows a marker filter for a Kafka cluster named **my-kafka-cluster**.

## Basic logging filter configuration

```
rootLogger.level="INFO"
appender.console.filter.filter1.type=MarkerFilter          1
appender.console.filter.filter1.onMatch=ACCEPT             2
appender.console.filter.filter1.onMismatch=DENY            3
appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster)   4
```

**1**      The **MarkerFilter** type compares a specified marker for filtering.

**2**      The **onMatch** property accepts the log if the marker matches.

**3**      The **onMismatch** property rejects the log if the marker does not match.

**4**      The marker used for filtering is in the format *KIND(NAMESPACE/NAME-OF-RESOURCE)*.

You can create one or more filters. Here, the log is filtered for two Kafka clusters.

## Multiple logging filter configuration

```
appender.console.filter.filter1.type=MarkerFilter
appender.console.filter.filter1.onMatch=ACCEPT
appender.console.filter.filter1.onMismatch=DENY
appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster-1)
appender.console.filter.filter2.type=MarkerFilter
appender.console.filter.filter2.onMatch=ACCEPT
appender.console.filter.filter2.onMismatch=DENY
appender.console.filter.filter2.marker=Kafka(my-namespace/my-kafka-cluster-2)
```

## Adding filters to the Cluster Operator

To add filters to the Cluster Operator, update its logging ConfigMap YAML file (**install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml**).

### Procedure

1. Update the **050-ConfigMap-strimzi-cluster-operator.yaml** file to add the filter properties to the ConfigMap.
   In this example, the filter properties return logs only for the **my-kafka-cluster** Kafka cluster:

   ```
   kind: ConfigMap
   apiVersion: v1
   metadata:
     name: strimzi-cluster-operator
   data:
     log4j2.properties:
       #...
       appender.console.filter.filter1.type=MarkerFilter
       appender.console.filter.filter1.onMatch=ACCEPT
       appender.console.filter.filter1.onMismatch=DENY
       appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster)
   ```

Alternatively, edit the **ConfigMap** directly:

```
oc edit configmap strimzi-cluster-operator
```

2. If you updated the YAML file instead of editing the **ConfigMap** directly, apply the changes by deploying the ConfigMap:

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

## Adding filters to the Topic Operator or User Operator

To add filters to the Topic Operator or User Operator, create or edit a logging ConfigMap.

In this procedure a logging ConfigMap is created with filters for the Topic Operator. The same approach is used for the User Operator.

### Procedure

1. Create the ConfigMap.
   You can create the ConfigMap as a YAML file or from a properties file.

   In this example, the filter properties return logs only for the **my-topic** topic:

   ```
   kind: ConfigMap
   apiVersion: v1
   metadata:
     name: logging-configmap
   data:
     log4j2.properties:
       rootLogger.level="INFO"
       appender.console.filter.filter1.type=MarkerFilter
       appender.console.filter.filter1.onMatch=ACCEPT
       appender.console.filter.filter1.onMismatch=DENY
       appender.console.filter.filter1.marker=KafkaTopic(my-namespace/my-topic)
   ```

   If you are using a properties file, specify the file at the command line:

   ```
   oc create configmap logging-configmap --from-file=log4j2.properties
   ```

   The properties file defines the logging configuration:

   ```
   # Define the logger
   rootLogger.level="INFO"
   # Set the filters
   appender.console.filter.filter1.type=MarkerFilter
   appender.console.filter.filter1.onMatch=ACCEPT
   appender.console.filter.filter1.onMismatch=DENY
   appender.console.filter.filter1.marker=KafkaTopic(my-namespace/my-topic)
   # ...
   ```

2. Define *external* logging in the **spec** of the resource, setting the **logging.valueFrom.configMapKeyRef.name** to the name of the ConfigMap and **logging.valueFrom.configMapKeyRef.key** to the key in this ConfigMap.

For the Topic Operator, logging is specified in the **topicOperator** configuration of the **Kafka** resource.

```
spec:
  # ...
  entityOperator:
    topicOperator:
      logging:
        type: external
        valueFrom:
          configMapKeyRef:
            name: logging-configmap
            key: log4j2.properties
```

3. Apply the changes by deploying the Cluster Operator:

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

**Additional resources**

- Configuring Kafka

- Cluster Operator logging

- Topic Operator logging

- User Operator logging

# CHAPTER 3. LOADING CONFIGURATION VALUES FROM EXTERNAL SOURCES

Use configuration provider plugins to load configuration data from external sources. The providers operate independently of AMQ Streams. You can use them to load configuration data for all Kafka components, including producers and consumers. Use them, for example, to provide the credentials for Kafka Connect connector configuration.

**OpenShift Configuration Provider**

The OpenShift Configuration Provider plugin loads configuration data from OpenShift secrets or ConfigMaps.

Suppose you have a **Secret** object that's managed outside the Kafka namespace, or outside the Kafka cluster. The OpenShift Configuration Provider allows you to reference the values of the secret in your configuration without extracting the files. You just need to tell the provider what secret to use and provide access rights. The provider loads the data without needing to restart the Kafka component, even when using a new **Secret** or **ConfigMap** object. This capability avoids disruption when a Kafka Connect instance hosts multiple connectors.

**Environment Variables Configuration Provider**

The Environment Variables Configuration Provider plugin loads configuration data from environment variables.

The values for the environment variables can be mapped from secrets or ConfigMaps. You can use the Environment Variables Configuration Provider, for example, to load certificates or JAAS configuration from environment variables mapped from OpenShift secrets.

> **NOTE**
>
> OpenShift Configuration Provider can't use mounted files. For example, it can't load values that need the location of a truststore or keystore. Instead, you can mount ConfigMaps or secrets into a Kafka Connect pod as environment variables or volumes. You can use the Environment Variables Configuration Provider to load values for environment variables. You add configuration using the **externalConfiguration** property in **KafkaConnect.spec**. You don't need to set up access rights with this approach. However, Kafka Connect will need a restart when using a new **Secret** or **ConfigMap** for a connector. This will cause disruption to all the Kafka Connect instance's connectors.

## 3.1. LOADING CONFIGURATION VALUES FROM A CONFIGMAP

This procedure shows how to use the OpenShift Configuration Provider plugin.

In the procedure, an external **ConfigMap** object provides configuration properties for a connector.

**Prerequisites**

- An OpenShift cluster is available.

- A Kafka cluster is running.

- The Cluster Operator is running.

**Procedure**

1. Create a **ConfigMap** or **Secret** that contains the configuration properties.
   In this example, a **ConfigMap** object named **my-connector-configuration** contains connector properties:

   **Example ConfigMap with connector properties**

   ```
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: my-connector-configuration
   data:
     option1: value1
     option2: value2
   ```

2. Specify the OpenShift Configuration Provider in the Kafka Connect configuration.
   The specification shown here can support loading values from secrets and ConfigMaps.

   **Example Kafka Connect configuration to enable the OpenShift Configuration Provider**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnect
   metadata:
     name: my-connect
     annotations:
       strimzi.io/use-connector-resources: "true"
   spec:
     # ...
     config:
       # ...
       config.providers: secrets,configmaps        1
       config.providers.secrets.class: io.strimzi.kafka.KubernetesSecretConfigProvider        2
       config.providers.configmaps.class: io.strimzi.kafka.KubernetesConfigMapConfigProvider
   3
     # ...
   ```

   **1**    The alias for the configuration provider is used to define other configuration parameters. The provider parameters use the alias from **config.providers**, taking the form **config.providers.${alias}.class**.

   **2**    **KubernetesSecretConfigProvider** provides values from secrets.

   **3**    **KubernetesConfigMapConfigProvider** provides values from config maps.

3. Create or update the resource to enable the provider.

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

4. Create a role that permits access to the values in the external config map.

   **Example role to access values from a config map**

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   ```

```
kind: Role
metadata:
  name: connector-configuration-role
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["my-connector-configuration"]
  verbs: ["get"]
  # ...
```

The rule gives the role permission to access the **my-connector-configuration** config map.

5. Create a role binding to permit access to the namespace that contains the config map.

   **Example role binding to access the namespace that contains the config map**

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: RoleBinding
   metadata:
     name: connector-configuration-role-binding
   subjects:
   - kind: ServiceAccount
     name: my-connect-connect
     namespace: my-project
   roleRef:
     kind: Role
     name: connector-configuration-role
     apiGroup: rbac.authorization.k8s.io
     # ...
   ```

   The role binding gives the role permission to access the **my-project** namespace.

   The service account must be the same one used by the Kafka Connect deployment. The service account name format is *<cluster_name>*–connect, where *<cluster_name>* is the name of the **KafkaConnect** custom resource.

6. Reference the config map in the connector configuration.

   **Example connector configuration referencing the config map**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnector
   metadata:
     name: my-connector
     labels:
       strimzi.io/cluster: my-connect
   spec:
     # ...
     config:
       option: ${configmaps:my-project/my-connector-configuration:option1}
       # ...
     # ...
   ```

   Placeholders for the property values in the config map are referenced in the connector configuration. The placeholder structure is **configmaps:*<path_and_file_name>*:*<property>***.

**KubernetesConfigMapConfigProvider** reads and extracts the *option1* property value from the external config map.

## 3.2. LOADING CONFIGURATION VALUES FROM ENVIRONMENT VARIABLES

This procedure shows how to use the Environment Variables Configuration Provider plugin.

In the procedure, environment variables provide configuration properties for a connector. A database password is specified as an environment variable.

### Prerequisites

- An OpenShift cluster is available.

- A Kafka cluster is running.

- The Cluster Operator is running.

### Procedure

1. Specify the Environment Variables Configuration Provider in the Kafka Connect configuration. Define environment variables using the **externalConfiguration** property.

   **Example Kafka Connect configuration to enable the Environment Variables Configuration Provider**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnect
   metadata:
     name: my-connect
     annotations:
       strimzi.io/use-connector-resources: "true"
   spec:
     # ...
     config:
       # ...
       config.providers: env        1
       config.providers.env.class: io.strimzi.kafka.EnvVarConfigProvider   2
     # ...
     externalConfiguration:
       env:
         - name: DB_PASSWORD        3
           valueFrom:
             secretKeyRef:
               name: db-creds       4
               key: dbPassword      5
     # ...
   ```

   **1** The alias for the configuration provider is used to define other configuration parameters. The provider parameters use the alias from **config.providers**, taking the form **config.providers.${alias}.class**.

**2**     **EnvVarConfigProvider** provides values from environment variables.

**3**     The **DB_PASSWORD** environment variable takes a password value from a secret.

**4**     The name of the secret containing the predefined password.

**5**     The key for the password stored inside the secret.

2. Create or update the resource to enable the provider.

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

3. Reference the environment variable in the connector configuration.

   **Example connector configuration referencing the environment variable**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnector
   metadata:
     name: my-connector
     labels:
       strimzi.io/cluster: my-connect
   spec:
     # ...
     config:
       option: ${env:DB_PASSWORD}
       # ...
   # ...
   ```

# CHAPTER 4. APPLYING SECURITY CONTEXT TO AMQ STREAMS PODS AND CONTAINERS

Security context defines constraints on pods and containers. By specifying a security context, pods and containers only have the permissions they need. For example, permissions can control runtime operations or access to resources.

## 4.1. HANDLING OF SECURITY CONTEXT BY OPENSHIFT PLATFORM

Handling of security context depends on the tooling of the OpenShift platform you are using.

For example, OpenShift uses built-in security context constraints (SCCs) to control permissions. SCCs are the settings and strategies that control the security features a pod has access to.

By default, OpenShift injects security context configuration automatically. In most cases, this means you don't need to configure security context for the pods and containers created by the Cluster Operator. Although you can still create and manage your own SCCs.

For more information, see the OpenShift documentation.

# CHAPTER 5. ACCESSING KAFKA OUTSIDE OF THE OPENSHIFT CLUSTER

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 a **NodePort** type **Service**

- **loadbalancer** uses a **Loadbalancer** type **Service**

- **ingress** uses Kubernetes **Ingress** and the Ingress NGINX Controller for Kubernetes

- **route** uses OpenShift **Routes** and the HAProxy router

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

If you want to know more about the pros and cons of each connection type, refer to Accessing Apache Kafka in Strimzi.
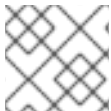
> **NOTE**
>
> **route** is only supported on OpenShift

## 5.1. ACCESSING KAFKA USING NODE PORTS

This procedure describes how to access an 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/v1beta2
   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_configuration_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 in the secret ***<cluster_name>*-cluster-ca-cert**.

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

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

   For example:

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

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 it in your client.

## 5.2. ACCESSING KAFKA USING LOADBALANCERS

This procedure describes how to access an 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/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: loadbalancer
        tls: true
        # ...
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

```
oc apply -f <kafka_configuration_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 in the secret ***<cluster_name>*-cluster-ca-cert**.

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[?
(@.name=="<listener_name>")].bootstrapServers}{"\n"}'
```

For example:

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="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 the TLS connection. If you enabled any authentication, you will also need to configure it in your client.

## 5.3. ACCESSING KAFKA USING AN INGRESS NGINX CONTROLLER FOR OPENSHIFT

Use an Ingress NGINX Controller for Kubernetes to access an AMQ Streams Kafka cluster from clients outside the OpenShift cluster.

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

To connect to a broker, you specify a hostname for the ingress bootstrap address, as well as the TLS certificate. Authentication is optional.

For access using an ingress, the port used in the Kafka client is typically 443.

## TLS passthrough

Make sure that you enable TLS passthrough in your Ingress NGINX Controller for OpenShift deployment. Kafka uses a binary protocol over TCP, but the Ingress NGINX Controller for OpenShift is designed to work with a HTTP protocol. To be able to route TCP traffic through ingresses, AMQ Streams uses TLS passthrough with Server Name Indication (SNI).

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

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

### Prerequisites

- An Ingress NGINX Controller for OpenShift is running with TLS passthrough enabled

- A running Cluster Operator

In this procedure, the Kafka cluster name is **my-cluster**.

### Procedure

1. Configure a **Kafka** resource with an external listener set to the **ingress** type.
   Specify an ingress hostname for the bootstrap service and each of the Kafka brokers in the Kafka cluster. Add any hostname to the **bootstrap** and **broker-<index>** prefixes that identify the bootstrap and brokers.

   For example:

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

```
        type: ingress
        tls: true ❶
        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
          class: nginx ❷
    # ...
  zookeeper:
    # ...
```

❶ For **ingress** type listeners, TLS encryption must be enabled ( **true**).

❷ (Optional) Class that specifies the ingress controller to use. You might need to add a class if you have not set up a default and a class name is missing in the ingresses created.

2. Create or update the resource.

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

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

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

   An **ingress** is also created for each service, with a DNS address to expose them using the Ingress NGINX Controller for OpenShift.

   **Ingresses created for the bootstrap and brokers**

   ```
   NAME                     CLASS HOSTS                ADDRESS               PORTS
   my-cluster-kafka-0          nginx broker-0.myingress.com   external.ingress.com  80,443
   my-cluster-kafka-1          nginx broker-1.myingress.com   external.ingress.com  80,443
   my-cluster-kafka-2          nginx broker-2.myingress.com   external.ingress.com  80,443
   my-cluster-kafka-bootstrap  nginx bootstrap.myingress.com  external.ingress.com  80,443
   ```

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

   **Status for the bootstrap ingress**

   ```
   status:
     loadBalancer:
       ingress:
         - hostname: external.ingress.com
     # ...
   ```

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

   ```
   openssl s_client -connect broker-0.myingress.com:443 -servername broker-0.myingress.com
   -showcerts
   ```

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

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

   **Certificates for the broker**

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

4. Extract the cluster CA certificate.

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

5. Configure your client to connect to the brokers.

   a. Specify the bootstrap host (from the listener **configuration**) and port 443 in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, **bootstrap.myingress.com:443**.

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

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

## 5.4. ACCESSING KAFKA USING OPENSHIFT ROUTES

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

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

To connect to a broker, you specify a hostname for the route bootstrap address, as well as the certificate used for authentication.

For access using routes, the port is always 443.

> **WARNING**
>
> An OpenShift route address comprises the name of the Kafka cluster, the name of the listener, and the name of the project it is created in. For example, **my-cluster-kafka-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.

## TLS passthrough

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

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

### Prerequisites

- A running Cluster Operator

In this procedure, the Kafka cluster name is **my-cluster**.

### Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
     labels:
       app: my-cluster
     name: my-cluster
     namespace: myproject
   spec:
     kafka:
       # ...
       listeners:
         - name: listener1
           port: 9094
           type: route
           tls: true    1
           # ...
       # ...
     zookeeper:
       # ...
   ```

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

2. Create or update the resource.

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

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

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

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

The routes are preconfigured with TLS passthrough.

### Routes created for the bootstraps and brokers

```
NAME                          HOST/PORT                               SERVICES
PORT  TERMINATION
my-cluster-kafka-listener1-0      my-cluster-kafka-listener1-0-my-project.router.com
my-cluster-kafka-listener1-0      9094  passthrough
my-cluster-kafka-listener1-1      my-cluster-kafka-listener1-1-my-project.router.com
my-cluster-kafka-listener1-1      9094  passthrough
my-cluster-kafka-listener1-2      my-cluster-kafka-listener1-2-my-project.router.com
my-cluster-kafka-listener1-2      9094  passthrough
my-cluster-kafka-listener1-bootstrap  my-cluster-kafka-listener1-bootstrap-my-
project.router.com  my-cluster-kafka-listener1-bootstrap 9094  passthrough
```

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

### Example status for the bootstrap route

```
status:
  ingress:
    - host: >-
        my-cluster-kafka-listener1-bootstrap-my-project.router.com
    # ...
```

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

```
openssl s_client -connect my-cluster-kafka-listener1-0-my-project.router.com:443 -
servername my-cluster-kafka-listener1-0-my-project.router.com -showcerts
```

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

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

### Certificates for the broker

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

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

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="listener1")].bootstrapServers}{"\n"}'

my-cluster-kafka-listener1-bootstrap-my-project.router.com:443
```

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

5. Extract the cluster CA certificate.

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

6. Configure your client to connect to the brokers.

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

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

**NOTE**

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

# CHAPTER 6. 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

## 6.1. SECURITY OPTIONS FOR KAFKA

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

### 6.1.1. Listener authentication

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

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

For more information on the configuration options for connecting an external client, see Accessing Kafka outside of the OpenShift cluster.

Supported authentication options:

1. mTLS authentication (only on the listeners with TLS enabled encryption)

2. SCRAM-SHA-512 authentication

3. OAuth 2.0 token-based authentication

4. Custom authentication

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

**NOTE**

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

Figure 6.1. Kafka listener authentication options
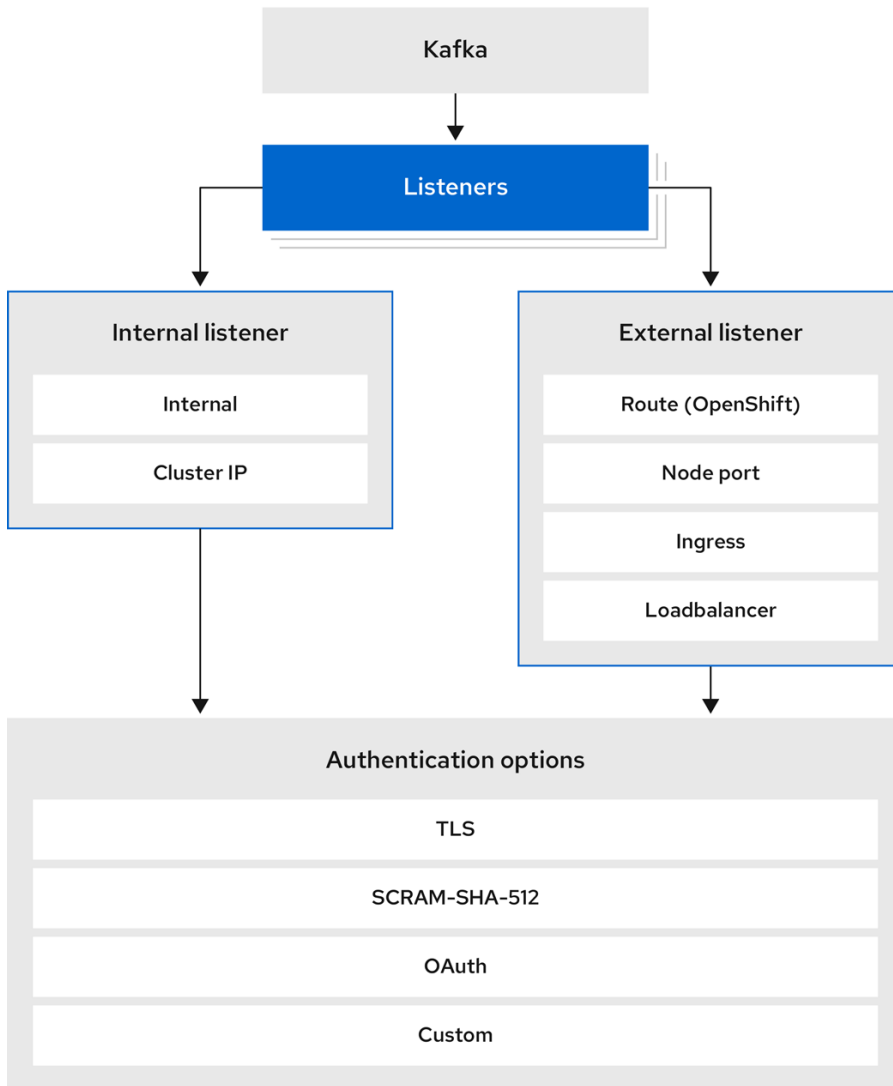


The listener **authentication** property is used to specify an authentication mechanism specific to that listener.

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

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

The following example shows:

- A **plain** listener configured for SCRAM-SHA-512 authentication

- A **tls** listener with mTLS authentication

- An **external** listener with mTLS authentication

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

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

**Example listener authentication configuration**

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

### 6.1.1.1. mTLS authentication

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

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

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

- **tls: true** to specify TLS encryption and server authentication

- **authentication.type: tls** to specify the client authentication

When a Kafka cluster is created by the Cluster Operator, it creates a new secret with the name **<cluster_name>-cluster-ca-cert**. The secret contains a CA certificate. The CA certificate is in PEM and PKCS #12 format. To verify a Kafka cluster, add the CA certificate to the truststore in your client configuration. To verify a client, add a user certificate and key to the keystore in your client configuration. For more information on configuring a client for mTLS, see Section 6.2.2, "User authentication".

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

### 6.1.1.2. SCRAM-SHA-512 authentication

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

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

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

- The passwords are not sent in the clear over the communication channel. Instead the client and the server are each challenged by the other to offer proof that they know the password of the authenticating user.

- The server and client each generate a new challenge for each authentication exchange. This means that the exchange is resilient against replay attacks.

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

### 6.1.1.3. Network policies

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

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

If you want to use custom network policies, you can set the **STRIMZI_NETWORK_POLICY_GENERATION** environment variable to **false** in the Cluster Operator configuration. For more information, see Cluster Operator configuration.
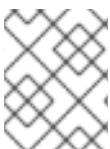
> **NOTE**
>
> Your configuration of OpenShift must support ingress **NetworkPolicies** in order to use network policies in AMQ Streams.

### 6.1.1.4. Additional listener configuration options

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

## 6.1.2. Kafka authorization

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

Supported authorization options:

- Simple authorization

- OAuth 2.0 authorization (if you are using OAuth 2.0 token based authentication)

- Open Policy Agent (OPA) authorization

- Custom authorization

Figure 6.2. Kafka cluster authorization options
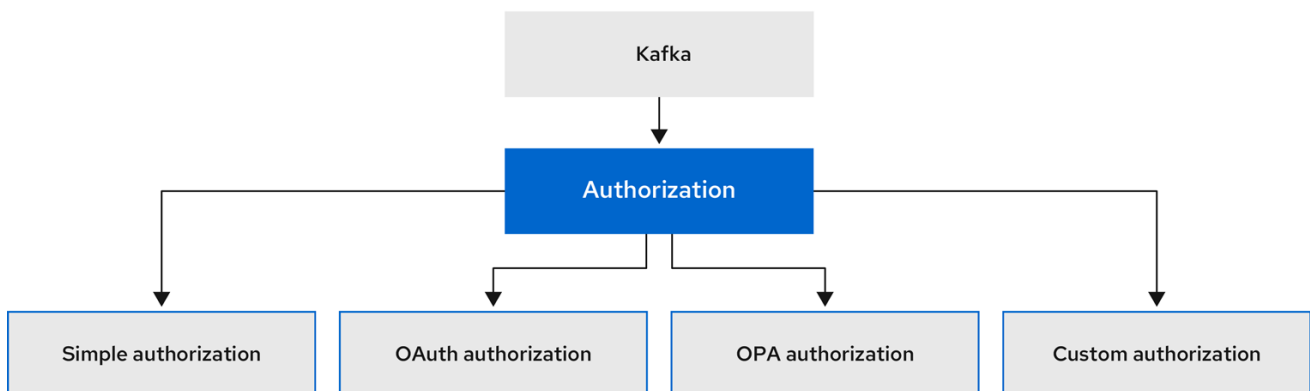


### 6.1.2.1. Super users

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

To designate super users for a Kafka cluster, add a list of user principals to the **superUsers** property. If a user uses mTLS authentication, the username is the common name from the TLS certificate subject prefixed with **CN=**. If you are not using the User Operator and using your own certificates for mTLS, the username is the full certificate subject. A full certificate subject can have the following fields: **CN=user,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=my_country_code**. Omit any fields that are not present.

### An example configuration with super users

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

```
    namespace: myproject
  spec:
    kafka:
      # ...
      authorization:
        type: simple
        superUsers:
          - CN=client_1
          - user_2
          - CN=client_3
          - CN=client_4,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=US
          - CN=client_5,OU=my_ou,O=my_org,C=GB
          - CN=client_6,O=my_org
      # ...
```

# 6.2. 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.

### Configuring users for secure access to Kafka brokers

For more information on configuring a **KafkaUser** resource to access Kafka brokers securely, see the following sections:

- Securing user access to Kafka

- Setting up client access to a Kafka cluster using listeners

## 6.2.1. Identifying a Kafka cluster for user handling

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

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

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

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

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

## 6.2.2. User authentication

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

Supported authentication types:

- **tls** for mTLS authentication

- **tls-external** for mTLS authentication using external certificates

- **scram-sha-512** for SCRAM-SHA-512 authentication

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

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

### 6.2.2.1. mTLS authentication

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

**Example user with mTLS authentication enabled**

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

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

When the user is created by the User Operator, it creates a new secret with the same name as the **KafkaUser** resource. The secret contains a private and public key for mTLS. The public key is contained in a user certificate, which is signed by a clients CA (certificate authority) when it is created. All keys are in X.509 format.

NOTE

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

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

## Example secret with user credentials

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

When you configure a client, you specify the following:

- **Truststore** properties for the public cluster CA certificate to verify the identity of the Kafka cluster

- **Keystore** properties for the user authentication credentials to verify the client

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

## Example client configuration using mTLS in PKCS #12 format

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

**1**     The bootstrap server address to connect to the Kafka cluster.

**2**     The security protocol option when using TLS for encryption.

**3**     The truststore location contains the public key certificate (**ca.p12**) for the Kafka cluster. A cluster CA certificate and password is generated by the Cluster Operator in the **<cluster_name>-cluster-ca-cert** secret when the Kafka cluster is created.

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

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

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

### 6.2.2.2. mTLS authentication using a certificate issued outside the User Operator

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

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

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

### 6.2.2.3. SCRAM-SHA-512 authentication

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

**Example user with SCRAM-SHA-512 authentication enabled**

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

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

**Example secret with user credentials**

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
```

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

1     The generated password, base64 encoded.

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

Decoding the generated password:

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

### 6.2.2.3.1. Custom password configuration

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

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

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

1     The name of the secret containing the predefined password.

2     The key for the password stored inside the secret.

## 6.2.3. User authorization

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

To use simple authorization, you set the **type** property to **simple** in **KafkaUser.spec.authorization**. The simple authorization uses the Kafka Admin API to manage the ACL rules inside your Kafka cluster.

Whether ACL management in the User Operator is enabled or not depends on your authorization configuration in the Kafka cluster.

- For simple authorization, ACL management is always enabled.

- For OPA authorization, ACL management is always disabled. Authorization rules are configured in the OPA server.

- For Red Hat Single Sign-On authorization, you can manage the ACL rules directly in Red Hat Single Sign-On. You can also delegate authorization to the simple authorizer as a fallback option in the configuration. When delegation to the simple authorizer is enabled, the User Operator will enable management of ACL rules as well.

- For custom authorization using a custom authorization plugin, use the **supportsAdminApi** property in the **.spec.kafka.authorization** configuration of the **Kafka** custom resource to enable or disable the support.

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

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

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

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

### 6.2.3.1. 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.

### 6.2.3.2. 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.

### 6.2.3.3. User quotas

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

### An example **KafkaUser** with user quotas

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
```

```
    name: my-user
    labels:
      strimzi.io/cluster: my-cluster
  spec:
    # ...
    quotas:
      producerByteRate: 1048576 ❶
      consumerByteRate: 2097152 ❷
      requestPercentage: 55 ❸
      controllerMutationRate: 10 ❹
```

❶ Byte-per-second quota on the amount of data the user can push to a Kafka broker

❷ Byte-per-second quota on the amount of data the user can fetch from a Kafka broker

❸ CPU utilization limit as a percentage of time for a client group

❹ Number of concurrent partition creation and deletion operations (mutations) allowed per second

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

## 6.3. 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 CA (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.

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

- **Kafka** schema reference

- **KafkaUser** schema reference

- **GenericKafkaListener** schema reference

## 6.3.1. Securing Kafka brokers

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

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

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

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

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

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

**Procedure**

1. Configure the **Kafka** resource.

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

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

      ```
      apiVersion: kafka.strimzi.io/v1beta2
      kind: Kafka
      spec:
        kafka:
          # ...
          authorization: 1
            type: simple
            superUsers: 2
              - CN=client_1
              - user_2
              - CN=client_3
      ```

```
      listeners:
        - name: tls
          port: 9093
          type: internal
          tls: true
          authentication:
            type: tls  3
      # ...
    zookeeper:
      # ...
```

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

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

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

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

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

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

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

A service is created for each Kafka broker pod.

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

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

## 6.3.2. Securing user access to Kafka

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

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

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

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

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

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

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

**Prerequisites**

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.

- A running User Operator (typically deployed with the Entity Operator).

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

**Procedure**

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

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

   **1**    User authentication mechanism, defined as mutual **tls** or **scram-sha-512**.

   **2**    Simple authorization, which requires an accompanying list of ACL rules.

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

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

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

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

### 6.3.3. Restricting access to Kafka listeners using network policies

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

**Prerequisites**

- An OpenShift cluster with support for Ingress NetworkPolicies.

- The Cluster Operator is running.

**Procedure**

1. Open the **Kafka** resource.

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

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

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

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

**Additional resources**

- **networkPolicyPeers** configuration

- NetworkPolicyPeer API reference

## 6.4. USING OAUTH 2.0 TOKEN-BASED AUTHENTICATION

AMQ Streams supports the use of OAuth 2.0 authentication using the *OAUTHBEARER* and *PLAIN* mechanisms.

OAuth 2.0 enables standardized token-based authentication and authorization between applications, using a central authorization server to issue tokens that grant limited access to resources.

You can configure OAuth 2.0 authentication, then OAuth 2.0 authorization.

Kafka brokers and clients both need to be configured to use OAuth 2.0. OAuth 2.0 authentication can also be used in conjunction with **simple** or OPA-based Kafka authorization.

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

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

In the context of AMQ Streams:

- Kafka brokers act as OAuth 2.0 resource servers

- Kafka clients act as OAuth 2.0 application clients

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

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

- Server-side OAuth 2.0 support for Kafka brokers

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

### 6.4.1. OAuth 2.0 authentication mechanisms

AMQ Streams supports the OAUTHBEARER and PLAIN mechanisms for OAuth 2.0 authentication. Both mechanisms allow Kafka clients to establish authenticated sessions with Kafka brokers. The authentication flow between clients, the authorization server, and Kafka brokers is different for each mechanism.

We recommend that you configure clients to use OAUTHBEARER whenever possible. OAUTHBEARER provides a higher level of security than PLAIN because client credentials are *never* shared with Kafka brokers. Consider using PLAIN only with Kafka clients that do not support OAUTHBEARER.

You configure Kafka broker listeners to use OAuth 2.0 authentication for connecting clients. If necessary, you can use the OAUTHBEARER and PLAIN mechanisms on the same **oauth** listener. The properties to support each mechanism must be explicitly specified in the **oauth** listener configuration.

#### OAUTHBEARER overview

OAUTHBEARER is automatically enabled in the **oauth** listener configuration for the Kafka broker. You can set the **enableOauthBearer** property to **true**, though this is not required.

```
# ...
authentication:
  type: oauth
  # ...
  enableOauthBearer: true
```

Many Kafka client tools use libraries that provide basic support for OAUTHBEARER at the protocol level. To support application development, AMQ Streams provides an *OAuth callback handler* for the upstream Kafka Client Java libraries (but not for other libraries). Therefore, you do not need to write your own callback handlers. An application client can use the callback handler to provide the access token. Clients written in other languages, such as Go, must use custom code to connect to the authorization server and obtain the access token.

With OAUTHBEARER, the client initiates a session with the Kafka broker for credentials exchange, where credentials take the form of a bearer token provided by the callback handler. Using the callbacks, you can configure token provision in one of three ways:

- Client ID and Secret (by using the *OAuth 2.0 client credentials* mechanism)

- A long-lived access token, obtained manually at configuration time

- A long-lived refresh token, obtained manually at configuration time

**NOTE**

OAUTHBEARER authentication can only be used by Kafka clients that support the OAUTHBEARER mechanism at the protocol level.

### PLAIN overview

To use PLAIN, you must enable it in the **oauth** listener configuration for the Kafka broker.

In the following example, PLAIN is enabled in addition to OAUTHBEARER, which is enabled by default. If you want to use PLAIN only, you can disable OAUTHBEARER by setting **enableOauthBearer** to **false**.

```
# ...
authentication:
  type: oauth
  # ...
  enablePlain: true
  tokenEndpointUri: https://OAUTH-SERVER-ADDRESS/auth/realms/external/protocol/openid-connect/token
```

PLAIN is a simple authentication mechanism used by all Kafka client tools. To enable PLAIN to be used with OAuth 2.0 authentication, AMQ Streams provides *OAuth 2.0 over PLAIN* server-side callbacks.

With the AMQ Streams implementation of PLAIN, the client credentials are not stored in ZooKeeper. Instead, client credentials are handled centrally behind a compliant authorization server, similar to when OAUTHBEARER authentication is used.

When used with the OAuth 2.0 over PLAIN callbacks, Kafka clients authenticate with Kafka brokers using either of the following methods:

- Client ID and secret (by using the OAuth 2.0 client credentials mechanism)

- A long-lived access token, obtained manually at configuration time

For both methods, the client must provide the PLAIN **username** and **password** properties to pass credentials to the Kafka broker. The client uses these properties to pass a client ID and secret or username and access token.

Client IDs and secrets are used to obtain access tokens.

Access tokens are passed as **password** property values. You pass the access token with or without an **$accessToken:** prefix.

- If you configure a token endpoint (**tokenEndpointUri**) in the listener configuration, you need the prefix.

- If you don't configure a token endpoint (**tokenEndpointUri**) in the listener configuration, you don't need the prefix. The Kafka broker interprets the password as a raw access token.

If the **password** is set as the access token, the **username** must be set to the same principal name that the Kafka broker obtains from the access token. You can specify username extraction options in your listener using the **userNameClaim**, **fallbackUserNameClaim**, **fallbackUsernamePrefix**, and **userInfoEndpointUri** properties. The username extraction process also depends on your authorization server; in particular, how it maps client IDs to account names.

> **NOTE**
>
> OAuth over PLAIN does not support **password grant** mechanism. You can only 'proxy' through SASL PLAIN mechanism the **client credentials** (clientId + secret) or the access token as described above.

**Additional resources**

- [Section 6.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers"](#)

## 6.4.2. 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.

### 6.4.2.1. 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
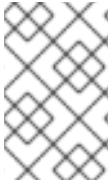
**NOTE**

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.

## 6.4.2.2. OAuth 2.0 authentication configuration in the Kafka cluster

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

### Assigining the authentication method type for OAuth 2.0

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
      #...
```

You can configure OAuth 2.0 authentication in your listeners. We recommend using OAuth 2.0 authentication together with TLS encryption (**tls: true**). Without encryption, the connection is vulnerable to network eavesdropping and unauthorized access through token theft.

You configure an **external** listener with **type: oauth** for a secure transport layer to communicate with the client.

### Using OAuth 2.0 with an external listener

```
# ...
listeners:
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: oauth
  #...
```

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

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

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

## 6.4.2.3. Fast local JWT token validation configuration

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

The local check ensures that a token:

- Conforms to type by containing a (*typ*) claim value of **Bearer** for an access token

- Is valid (not expired)

- Has an issuer that matches a **validIssuerURI**

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

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

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

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

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

**Example configuration for fast local JWT token validation**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    #...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          jwksEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-connect/certs>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
    #...
```

### 6.4.2.4. OAuth 2.0 introspection endpoint configuration

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

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

### Example configuration for an introspection endpoint

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          clientId: kafka-broker
          clientSecret:
            secretName: my-cluster-oauth
            key: clientSecret
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          introspectionEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-connect/token/introspect>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
```

## 6.4.3. Session re-authentication for Kafka brokers

You can configure **oauth** listeners to use Kafka *session re-authentication* for OAuth 2.0 sessions between Kafka clients and Kafka brokers. This mechanism enforces the expiry of an authenticated session between the client and the broker after a defined period of time. When a session expires, the client immediately starts a new session by reusing the existing connection rather than dropping it.

Session re-authentication is disabled by default. To enable it, you set a time value for **maxSecondsWithoutReauthentication** in the **oauth** listener configuration. The same property is used to configure session re-authentication for OAUTHBEARER and PLAIN authentication. For an example configuration, see Section 6.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers".

Session re-authentication must be supported by the Kafka client libraries used by the client.

Session re-authentication can be used with *fast local JWT* or *introspection endpoint* token validation.

### Client re-authentication

When the broker's authenticated session expires, the client must re-authenticate to the existing session by sending a new, valid access token to the broker, without dropping the connection.

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

Session re-authentication also applies to refresh tokens, if used. When the session expires, the client refreshes the access token by using its refresh token. The client then uses the new access token to re-authenticate to the existing session.

## Session expiry for OAUTHBEARER and PLAIN

When session re-authentication is configured, session expiry works differently for OAUTHBEARER and PLAIN authentication.

For OAUTHBEARER and PLAIN, using the client ID and secret method:

- The broker's authenticated session will expire at the configured **maxSecondsWithoutReauthentication**.

- The session will expire earlier if the access token expires before the configured time.

For PLAIN using the long-lived access token method:

- The broker's authenticated session will expire at the configured **maxSecondsWithoutReauthentication**.

- Re-authentication will fail if the access token expires before the configured time. Although session re-authentication is attempted, PLAIN has no mechanism for refreshing tokens.

If **maxSecondsWithoutReauthentication** is *not* configured, OAUTHBEARER and PLAIN clients can remain connected to brokers indefinitely, without needing to re-authenticate. Authenticated sessions do not end with access token expiry. However, this can be considered when configuring authorization, for example, by using **keycloak** authorization or installing a custom authorizer.

### Additional resources

- Section 6.4.2, "OAuth 2.0 Kafka broker configuration"

- Section 6.4.6.2, "Configuring OAuth 2.0 support for Kafka brokers"

- KafkaListenerAuthenticationOAuth schema reference

- KIP-368

## 6.4.4. OAuth 2.0 Kafka client configuration

A Kafka client is configured with either:

- The credentials required to obtain a valid access token from an authorization server (client ID and Secret)

- A valid long-lived access token or refresh token, obtained using tools provided by an authorization server

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

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

The simplest mechanism is authentication with a client ID and Secret. Using a long-lived access token, or a long-lived refresh token, adds more complexity because there is an additional dependency on authorization server tools.
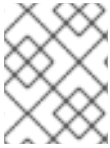
> **NOTE**
>
> If you are using long-lived access tokens, you may need to configure the client in the authorization server to increase the maximum lifetime of the token.

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

- Client ID and Secret

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

- Username and password, with client ID and (optionally) a secret

## 6.4.5. OAuth 2.0 client authentication flows

OAuth 2.0 authentication flows depend on the underlying Kafka client and Kafka broker configuration. The flows must also be supported by the authorization server used.

The Kafka broker listener configuration determines how clients authenticate using an access token. The client can pass a client ID and secret to request an access token.

If a listener is configured to use PLAIN authentication, the client can authenticate with a client ID and secret or username and access token. These values are passed as the **username** and **password** properties of the PLAIN mechanism.

Listener configuration supports the following token validation options:

- You can use fast local token validation based on JWT signature checking and local token introspection, without contacting an authorization server. The authorization server provides a JWKS endpoint with public certificates that are used to validate signatures on the tokens.

- You can use a call to a token introspection endpoint provided by an authorization server. Each time a new Kafka broker connection is established, the broker passes the access token received from the client to the authorization server. The Kafka broker checks the response to confirm whether or not the token is valid.

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

Kafka client credentials can also be configured for the following types of authentication:

- Direct local access using a previously generated long-lived access token

- Contact with the authorization server for a new access token to be issued (using a client ID and a secret, or a refresh token, or a username and a password)

### 6.4.5.1. Example client authentication flows using the SASL OAUTHBEARER mechanism

You can use the following communication flows for Kafka authentication using the SASL OAUTHBEARER mechanism.

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

- Client using client ID and secret, with broker performing fast local token validation

- Client using long-lived access token, with broker delegating validation to authorization server

- Client using long-lived access token, with broker performing fast local validation

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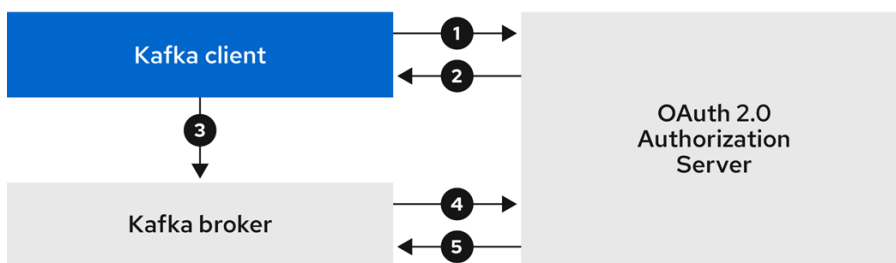


1. The Kafka client requests an access token from the authorization server using a client ID and secret, and optionally a refresh token. Alternatively, the client may authenticate using a username and a password.

2. The authorization server generates a new access token.

3. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the access token.

4. The Kafka broker validates the access token by calling a token introspection endpoint on the authorization server using its own client ID and secret.

5. A Kafka client session is established if the token is valid.

## Client using client ID and secret, with broker performing fast local token validation
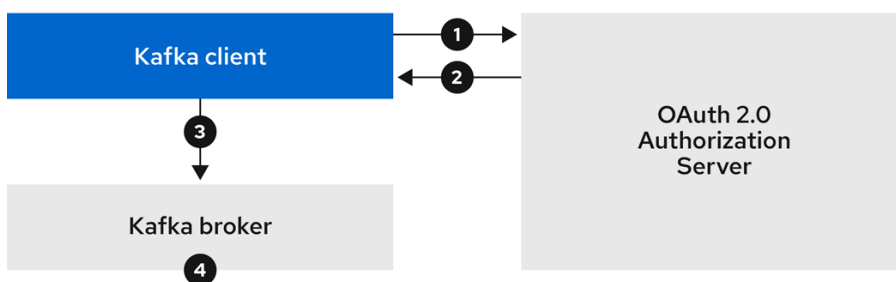


1. The Kafka client authenticates with the authorization server from the token endpoint, using a client ID and secret, and optionally a refresh token. Alternatively, the client may authenticate using a username and a password.

2. The authorization server generates a new access token.

3. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the access token.

4. The Kafka broker validates the access token locally using a JWT token signature check, and local token introspection.

## Client using long-lived access token, with broker delegating validation to authorization server
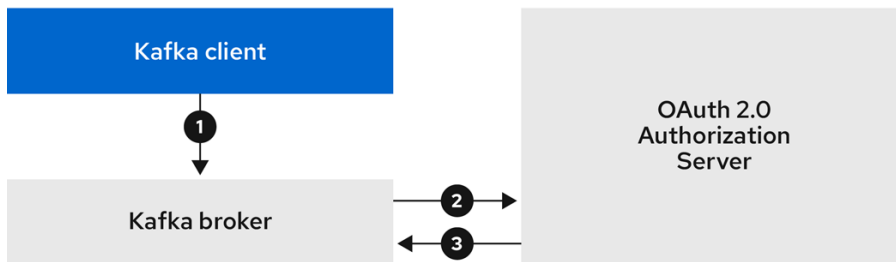


1. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the long-lived access token.

2. The Kafka broker validates the access token by calling a token introspection endpoint on the authorization server, using its own client ID and secret.

3. A Kafka client session is established if the token is valid.

## Client using long-lived access token, with broker performing fast local validation



1. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the long-lived access token.

2. The Kafka broker validates the access token locally using a JWT token signature check and local token introspection.

> **WARNING**
>
> Fast local JWT token signature validation is suitable only for short-lived tokens as there is no check with the authorization server if a token has been revoked. Token expiration is written into the token, but revocation can happen at any time, so cannot be accounted for without contacting the authorization server. Any issued token would be considered valid until it expires.

### 6.4.5.2. Example client authentication flows using the SASL PLAIN mechanism

You can use the following communication flows for Kafka authentication using the OAuth PLAIN mechanism.

- Client using a client ID and secret, with the broker obtaining the access token for the client

- Client using a long-lived access token without a client ID and secret

## Client using a client ID and secret, with the broker obtaining the access token for the client
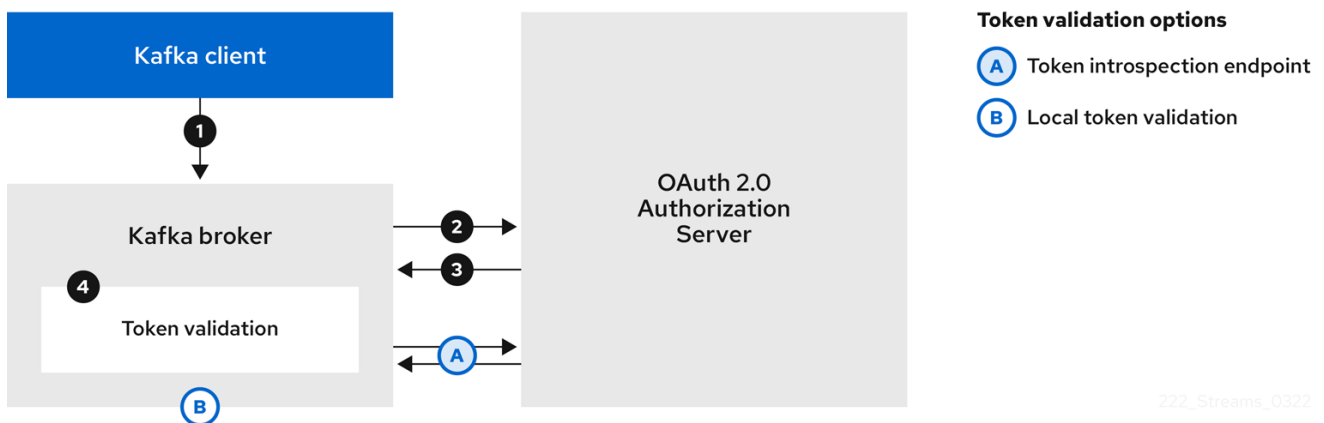


1. The Kafka client passes a **clientId** as a username and a **secret** as a password.

2. The Kafka broker uses a token endpoint to pass the **clientId** and **secret** to the authorization server.

3. The authorization server returns a fresh access token or an error if the client credentials are not valid.

4. The Kafka broker validates the token in one of the following ways:

   a. If a token introspection endpoint is specified, the Kafka broker validates the access token by calling the endpoint on the authorization server. A session is established if the token validation is successful.

   b. If local token introspection is used, a request is not made to the authorization server. The Kafka broker validates the access token locally using a JWT token signature check.

## Client using a long-lived access token without a client ID and secret
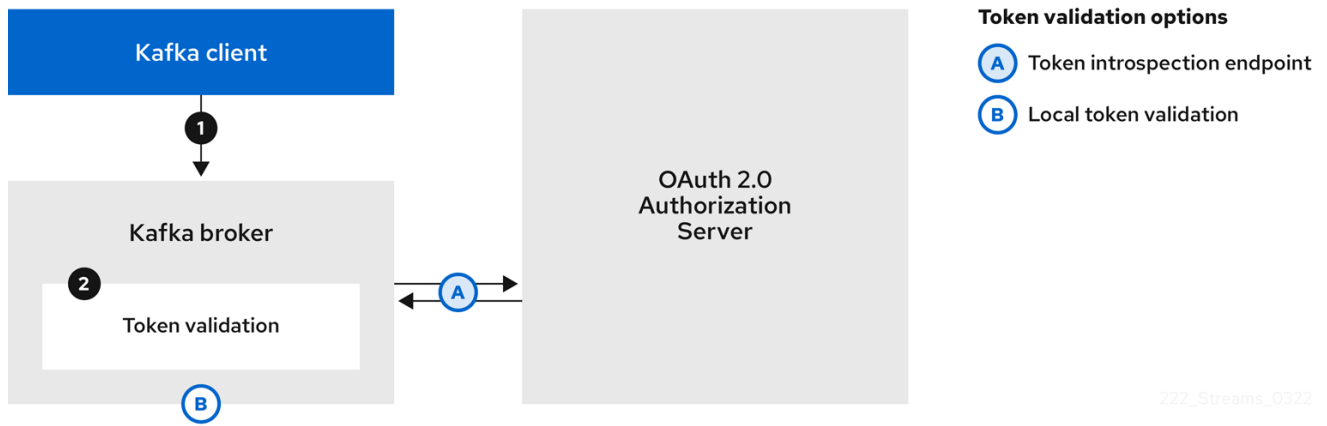
1. The Kafka client passes a username and password. The password provides the value of an access token that was obtained manually and configured before running the client.

2. The password is passed with or without an **$accessToken:** string prefix depending on whether or not the Kafka broker listener is configured with a token endpoint for authentication.

   a. If the token endpoint is configured, the password should be prefixed by **$accessToken:** to let the broker know that the password parameter contains an access token rather than a client secret. The Kafka broker interprets the username as the account username.

   b. If the token endpoint is not configured on the Kafka broker listener (enforcing a **no-client-credentials mode**), the password should provide the access token without the prefix. The Kafka broker interprets the username as the account username. In this mode, the client doesn't use a client ID and secret, and the **password** parameter is always interpreted as a raw access token.

3. The Kafka broker validates the token in one of the following ways:

   a. If a token introspection endpoint is specified, the Kafka broker validates the access token by calling the endpoint on the authorization server. A session is established if token validation is successful.

   b. If local token introspection is used, there is no request made to the authorization server. Kafka broker validates the access token locally using a JWT token signature check.

## 6.4.6. 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

### 6.4.6.1. Configuring an OAuth 2.0 authorization server

This procedure describes in general what you need to do to configure an authorization server for integration with AMQ Streams.

These instructions are not product specific.

The steps are dependent on the chosen authorization server. Consult the product documentation for the authorization server for information on how to set up OAuth 2.0 access.

> **NOTE**
>
> If you already have an authorization server deployed, you can skip the deployment step and use your current deployment.

**Procedure**

1. Deploy the authorization server to your cluster.

2. Access the CLI or admin console for the authorization server to configure OAuth 2.0 for AMQ Streams.
   Now prepare the authorization server to work with AMQ Streams.

3. Configure a **kafka-broker** client.

4. Configure clients for each Kafka client component of your application.

**What to do next**

After deploying and configuring the authorization server, configure the Kafka brokers to use OAuth 2.0 .

### 6.4.6.2. Configuring OAuth 2.0 support for Kafka brokers

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

We advise use of OAuth 2.0 over an encrypted interface through through a listener with **tls: true**. Plain listeners are not recommended.

If the authorization server is using certificates signed by the trusted CA and matching the OAuth 2.0 server hostname, TLS connection works using the default settings. Otherwise, you may need to configure the truststore with proper certificates or disable the certificate hostname validation.

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

- Configuring fast local JWT token validation

- Configuring token validation using an introspection endpoint

**Before you start**

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

- KafkaListenerAuthenticationOAuth schema reference

- OAuth 2.0 authentication mechanisms

**Prerequisites**

- AMQ Streams and Kafka are running

- An OAuth 2.0 authorization server is deployed

**Procedure**

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

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

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

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

   **Example 1: Configuring fast local JWT token validation**

   ```
   #...
   - name: external
     port: 9094
     type: loadbalancer
     tls: true
     authentication:
       type: oauth 1
       validIssuerUri: <https://<auth-server-address>/auth/realms/external> 2
       jwksEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-connect/certs> 3
       userNameClaim: preferred_username 4
       maxSecondsWithoutReauthentication: 3600 5
       tlsTrustedCertificates: 6
       - secretName: oauth-server-cert
         certificate: ca.crt
       disableTlsHostnameVerification: true 7
       jwksExpirySeconds: 360 8
       jwksRefreshSeconds: 300 9
       jwksMinRefreshPauseSeconds: 1 10
   ```

   **1** Listener type set to **oauth**.

   **2** URI of the token issuer used for authentication.

   **3** URI of the JWKS certificate endpoint used for local JWT validation.

   **4** The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The **userNameClaim** value will depend on the authentication flow and the authorization server used.

   **5** (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.

   **6** (Optional) Trusted certificates for TLS connection to the authorization server.

**7** (Optional) Disable TLS hostname verification. Default is **false**.

**8** The duration the JWKS certificates are considered valid before they expire. Default is **360** seconds. If you specify a longer time, consider the risk of allowing access to revoked certificates.

**9** The period between refreshes of JWKS certificates. The interval must be at least 60 seconds shorter than the expiry interval. Default is **300** seconds.

**10** The minimum pause in seconds between consecutive attempts to refresh JWKS public keys. When an unknown signing key is encountered, the JWKS keys refresh is scheduled outside the regular periodic schedule with at least the specified pause since the last refresh attempt. The refreshing of keys follows the rule of exponential backoff, retrying on unsuccessful refreshes with ever increasing pause, until it reaches **jwksRefreshSeconds**. The default value is 1.

### Example 2: Configuring token validation using an introspection endpoint

```
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth
    validIssuerUri: <https://<auth-server-address>/auth/realms/external>
    introspectionEndpointUri: <https://<auth-server-
address>/auth/realms/external/protocol/openid-connect/token/introspect> 1
    clientId: kafka-broker 2
    clientSecret: 3
      secretName: my-cluster-oauth
      key: clientSecret
    userNameClaim: preferred_username 4
    maxSecondsWithoutReauthentication: 3600 5
```

**1** URI of the token introspection endpoint.

**2** Client ID to identify the client.

**3** Client Secret and client ID is used for authentication.

**4** The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The **userNameClaim** value will depend on the authorization server used.

**5** (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.

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

```
# ...
authentication:
  type: oauth
  # ...
  checkIssuer: false 1
  checkAudience: true 2
  fallbackUserNameClaim: client_id 3
  fallbackUserNamePrefix: client-account- 4
  validTokenType: bearer 5
  userInfoEndpointUri: https://OAUTH-SERVER-
ADDRESS/auth/realms/external/protocol/openid-connect/userinfo 6
  enableOauthBearer: false 7
  enablePlain: true 8
  tokenEndpointUri: https://OAUTH-SERVER-
ADDRESS/auth/realms/external/protocol/openid-connect/token 9
  customClaimCheck: "@.custom == 'custom-value'" 10
  clientAudience: AUDIENCE 11
  clientScope: SCOPE 12
  connectTimeoutSeconds: 60 13
  readTimeoutSeconds: 60 14
  groupsClaim: "$.groups" 15
  groupsClaimDelimiter: "," 16
```

**1** If your authorization server does not provide an **iss** claim, it is not possible to perform an issuer check. In this situation, set **checkIssuer** to **false** and do not specify a **validIssuerUri**. Default is **true**.

**2** If your authorization server provides an **aud** (audience) claim, and you want to enforce an audience check, set **checkAudience** to **true**. Audience checks identify the intended recipients of tokens. As a result, the Kafka broker will reject tokens that do not have its **clientId** in their **aud** claim. Default is **false**.

**3** An authorization server may not provide a single attribute to identify both regular users and clients. When a client authenticates in its own name, the server might provide a *client ID*. When a user authenticates using a username and password, to obtain a refresh token or an access token, the server might provide a *username* attribute in addition to a client ID. Use this fallback option to specify the username claim (attribute) to use if a primary user ID attribute is not available.

**4** In situations where **fallbackUserNameClaim** is applicable, it may also be necessary to prevent name collisions between the values of the username claim, and those of the fallback username claim. Consider a situation where a client called **producer** exists, but also a regular user called **producer** exists. In order to differentiate between the two, you can use this property to add a prefix to the user ID of the client.

**5** (Only applicable when using **introspectionEndpointUri**) Depending on the authorization server you are using, the introspection endpoint may or may not return the *token type* attribute, or it may contain different values. You can specify a valid token type value that the response from the introspection endpoint has to contain.

**6** (Only applicable when using **introspectionEndpointUri**) The authorization server may be configured or implemented in such a way to not provide any identifiable information in an Introspection Endpoint response. In order to obtain the user ID, you can configure the URI of the **userinfo** endpoint as a fallback. The **userNameClaim**, **fallbackUserNameClaim**,

and **fallbackUserNamePrefix** settings are applied to the response of **userinfo** endpoint.

**7** Set this to **false** to disable the OAUTHBEARER mechanism on the listener. At least one of PLAIN or OAUTHBEARER has to be enabled. Default is **true**.

**8** Set to **true** to enable PLAIN authentication on the listener, which is supported for clients on all platforms.

**9** Additional configuration for the PLAIN mechanism. If specified, clients can authenticate over PLAIN by passing an access token as the **password** using an **$accessToken:** prefix. For production, always use **https://** urls.

**10** Additional custom rules can be imposed on the JWT access token during validation by setting this to a JsonPath filter query. If the access token does not contain the necessary data, it is rejected. When using the **introspectionEndpointUri**, the custom check is applied to the introspection endpoint response JSON.

**11** An **audience** parameter passed to the token endpoint. An *audience* is used when obtaining an access token for inter-broker authentication. It is also used in the name of a client for OAuth 2.0 over PLAIN client authentication using a **clientId** and **secret**. This only affects the ability to obtain the token, and the content of the token, depending on the authorization server. It does not affect token validation rules by the listener.

**12** A **scope** parameter passed to the token endpoint. A *scope* is used when obtaining an access token for inter-broker authentication. It is also used in the name of a client for OAuth 2.0 over PLAIN client authentication using a **clientId** and **secret**. This only affects the ability to obtain the token, and the content of the token, depending on the authorization server. It does not affect token validation rules by the listener.

**13** The connect timeout in seconds when connecting to the authorization server. The default value is 60.

**14** The read timeout in seconds when connecting to the authorization server. The default value is 60.

**15** A JsonPath query used to extract groups information from JWT token or introspection endpoint response. Not set by default. This can be used by a custom authorizer to make authorization decisions based on user groups.

**16** A delimiter used to parse groups information when returned as a single delimited string. The default value is ',' (comma).

3. Save and exit the editor, then wait for rolling updates to complete.

4. Check the update in the logs or by watching the pod state transitions:

```
oc logs -f ${POD_NAME} -c ${CONTAINER_NAME}
oc get pod -w
```

The rolling update configures the brokers to use OAuth 2.0 authentication.

**What to do next**

- Configure your Kafka clients to use OAuth 2.0

### 6.4.6.3. Configuring Kafka Java clients to use OAuth 2.0

Configure Kafka producer and consumer APIs to use OAuth 2.0 for interaction with Kafka brokers. Add a callback plugin to your client **pom.xml** file, then configure your client for OAuth 2.0.

Specify the following in your client configuration:

- A SASL (Simple Authentication and Security Layer) security protocol:

  - **SASL_SSL** for authentication over TLS encrypted connections

  - **SASL_PLAINTEXT** for authentication over unencrypted connections
    Use **SASL_SSL** for production and **SASL_PLAINTEXT** for local development only. When using **SASL_SSL**, additional **ssl.truststore** configuration is needed. The truststore configuration is required for secure connection (**https://**) to the OAuth 2.0 authorization server. To verify the OAuth 2.0 authorization server, add the CA certificate for the authorization server to the truststore in your client configuration. You can configure a truststore in PEM or PKCS #12 format.

- A Kafka SASL mechanism:

  - **OAUTHBEARER** for credentials exchange using a bearer token

  - **PLAIN** to pass client credentials (clientId + secret) or an access token

- A JAAS (Java Authentication and Authorization Service) module that implements the SASL mechanism:

  - **org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule** implements the OAUTHBEARER mechanism

  - **org.apache.kafka.common.security.plain.PlainLoginModule** implements the PLAIN mechanism

- SASL authentication properties, which support the following authentication methods:

  - OAuth 2.0 client credentials

  - OAuth 2.0 password grant (deprecated)

  - Access token

  - Refresh token

Add the SASL authentication properties as JAAS configuration (**sasl.jaas.config**). How you configure the authentication properties depends on the authentication method you are using to access the OAuth 2.0 authorization server. In this procedure, the properties are specified in a properties file, then loaded into the client configuration.

> **NOTE**
>
> You can also specify authentication properties as environment variables, or as Java system properties. For Java system properties, you can set them using **setProperty** and pass them on the command line using the **-D** option.

**Prerequisites**

- AMQ Streams and Kafka are running

- An OAuth 2.0 authorization server is deployed and configured for OAuth access to Kafka brokers

- Kafka brokers are configured for OAuth 2.0

**Procedure**

1. Add the client library with OAuth 2.0 support to the **pom.xml** file for the Kafka client:

   ```
   <dependency>
    <groupId>io.strimzi</groupId>
    <artifactId>kafka-oauth-client</artifactId>
    <version>0.11.0.redhat-00003</version>
   </dependency>
   ```

2. Configure the client properties by specifying the following configuration in a properties file:

   - The security protocol

   - The SASL mechanism

   - The JAAS module and authentication properties according to the method being used
     For example, we can add the following to a **client.properties** file:

     **Client credentials mechanism properties**

     ```
     security.protocol=SASL_SSL 1
     sasl.mechanism=OAUTHBEARER 2
     ssl.truststore.location=/tmp/truststore.p12 3
     ssl.truststore.password=$STOREPASS
     ssl.truststore.type=PKCS12
     sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
     required \
       oauth.token.endpoint.uri="<token_endpoint_url>" \ 4
       oauth.client.id="<client_id>" \ 5
       oauth.client.secret="<client_secret>" \ 6
       oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \ 7
       oauth.ssl.truststore.password="$STOREPASS" \ 8
       oauth.ssl.truststore.type="PKCS12" \ 9
       oauth.scope="<scope>" \ 10
       oauth.audience="<audience>" ; 11
     ```

     **1** **SASL_SSL** security protocol for TLS-encrypted connections. Use **SASL_PLAINTEXT** over unencrypted connections for local development only.

     **2** The SASL mechanism specified as **OAUTHBEARER** or **PLAIN**.

     **3** The truststore configuration for secure access to the Kafka cluster.

     **4** URI of the authorization server token endpoint.

     **5** Client ID, which is the name used when creating the *client* in the authorization server.

**6** Client secret created when creating the *client* in the authorization server.

**7** The location contains the public key certificate (**truststore.p12**) for the authorization server.

**8** The password for accessing the truststore.

**9** The truststore type.

**10** (Optional) The **scope** for requesting the token from the token endpoint. An authorization server may require a client to specify the scope.

**11** (Optional) The **audience** for requesting the token from the token endpoint. An authorization server may require a client to specify the audience.

### Password grants mechanism properties

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.password.grant.username="<username>" \ 3
  oauth.password.grant.password="<password>" \ 4
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.scope="<scope>" \
  oauth.audience="<audience>" ;
```

**1** Client ID, which is the name used when creating the *client* in the authorization server.

**2** (Optional) Client secret created when creating the *client* in the authorization server.

**3** Username for password grant authentication. OAuth password grant configuration (username and password) uses the OAuth 2.0 password grant method. To use password grants, create a user account for a client on your authorization server with limited permissions. The account should act like a service account. Use in environments where user accounts are required for authentication, but consider using a refresh token first.

**4** Password for password grant authentication.

> **NOTE**
>
> SASL PLAIN does not support passing a username and password (password grants) using the OAuth 2.0 password grant method.

### Access token properties

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.access.token="<access_token>" ; 1
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
```

**1**     Long-lived access token for Kafka clients.

### Refresh token properties

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.refresh.token="<refresh_token>" ; 3
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
```

**1**     Client ID, which is the name used when creating the *client* in the authorization server.

**2**     (Optional) Client secret created when creating the *client* in the authorization server.

**3**     Long-lived refresh token for Kafka clients.

3. Input the client properties for OAUTH 2.0 authentication into the Java client code.

### Example showing input of client properties

```java
Properties props = new Properties();
try (FileReader reader = new FileReader("client.properties", StandardCharsets.UTF_8)) {
  props.load(reader);
}
```

4. Verify that the Kafka client can access the Kafka brokers.

## 6.4.6.4. 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/v1beta2
   kind: Secret
   metadata:
    name: my-bridge-oauth
   type: Opaque
   data:
    clientSecret: MGQ1OTRmMzYtZTlIZS00MDY2LWI5OGEtMTM5MzM2NjdlZjQw 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

   - Username and password

- 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/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: oauth 1
    tokenEndpointUri: https://<auth-server-address>/auth/realms/master/protocol/openid-connect/token 2
    clientId: kafka-bridge
    clientSecret:
      secretName: my-bridge-oauth
      key: clientSecret
    tlsTrustedCertificates: 3
    - secretName: oauth-server-cert
      certificate: tls.crt
```

**1**    Authentication type set to **oauth**.

**2**    URI of the token endpoint for authentication.

**3**    Trusted certificates for TLS connection to the authorization server.

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

```
# ...
spec:
  # ...
  authentication:
    # ...
    disableTlsHostnameVerification: true 1
    checkAccessTokenType: false 2
    accessTokenIsJwt: false 3
    scope: any 4
    audience: kafka 5
    connectTimeoutSeconds: 60 6
    readTimeoutSeconds: 60 7
```

**1**    (Optional) Disable TLS hostname verification. Default is **false**.

**2**    If the authorization server does not return a **typ** (type) claim inside the JWT token, you can apply **checkAccessTokenType: false** to skip the token type check. Default is **true**.

**3**    If you are using opaque tokens, you can apply **accessTokenIsJwt: false** so that access tokens are not treated as JWT tokens.

**4** (Optional) The **scope** for requesting the token from the token endpoint. An authorization server may require a client to specify the scope. In this case it is **any**.

**5** (Optional) The **audience** for requesting the token from the token endpoint. An authorization server may require a client to specify the audience. In this case it is **kafka**.

**6** (Optional) The connect timeout in seconds when connecting to the authorization server. The default value is 60.

**7** (Optional) The read timeout in seconds when connecting to the authorization server. The default value is 60.

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.

# 6.5. 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

## 6.5.1. 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.

### 6.5.1.1. 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.

## 6.5.2. Configuring OAuth 2.0 authorization support

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

### Before you begin

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

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

With Red Hat Single Sign-On, you can store users and groups in LDAP, whereas clients and roles cannot be stored this way. Storage and access to user data may be a factor in how you choose to configure authorization policies.
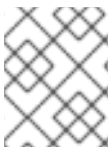
> **NOTE**
>
> Super users always have unconstrained access to a Kafka broker regardless of the authorization implemented on the Kafka broker.

### Prerequisites

- AMQ Streams must be configured to use OAuth 2.0 with Red Hat Single Sign-On for token-based authentication. You use the same Red Hat Single Sign-On server endpoint when you set up authorization.

- OAuth 2.0 authentication must be configured with the **maxSecondsWithoutReauthentication** option to enable re-authentication.

### Procedure

1. Access the Red Hat Single Sign-On Admin Console or use the Red Hat Single Sign-On Admin CLI to enable Authorization Services for the Kafka broker client you created when setting up OAuth 2.0 authentication.

2. Use Authorization Services to define resources, authorization scopes, policies, and permissions for the client.

3. Bind the permissions to users and clients by assigning them roles and groups.

4. Configure the Kafka brokers to use Red Hat Single Sign-On authorization by updating the Kafka broker configuration (**Kafka.spec.kafka**) of your **Kafka** resource in an editor.

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

5. Configure the Kafka broker **kafka** configuration to use **keycloak** authorization, and to be able to access the authorization server and Authorization Services.
   For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       authorization:
         type: keycloak 1
         tokenEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-connect/token> 2
         clientId: kafka 3
         delegateToKafkaAcls: false 4
         disableTlsHostnameVerification: false 5
         superUsers: 6
         - CN=fred
         - sam
         - CN=edward
         tlsTrustedCertificates: 7
         - secretName: oauth-server-cert
           certificate: ca.crt
         grantsRefreshPeriodSeconds: 60 8
         grantsRefreshPoolSize: 5 9
         connectTimeoutSeconds: 60 10
         readTimeoutSeconds: 60 11
       #...
   ```

   **1** Type **keycloak** enables Red Hat Single Sign-On authorization.

   **2** URI of the Red Hat Single Sign-On token endpoint. For production, always use **https://** urls. When you configure token-based **oauth** authentication, you specify a **jwksEndpointUri** as the URI for local JWT validation. The hostname for the **tokenEndpointUri** URI must be the same.

   **3** The client ID of the OAuth 2.0 client definition in Red Hat Single Sign-On that has Authorization Services enabled. Typically, **kafka** is used as the ID.

   **4** (Optional) Delegate authorization to Kafka **AclAuthorizer** if access is denied by Red Hat Single Sign-On Authorization Services policies. Default is **false**.

**5** (Optional) Disable TLS hostname verification. Default is **false**.

**6** (Optional) Designated super users.

**7** (Optional) Trusted certificates for TLS connection to the authorization server.

**8** (Optional) The time between two consecutive grants refresh runs. That is the maximum time for active sessions to detect any permissions changes for the user on Red Hat Single Sign-On. The default value is 60.

**9** (Optional) The number of threads to use to refresh (in parallel) the grants for the active sessions. The default value is 5.

**10** (Optional) The connect timeout in seconds when connecting to the Red Hat Single Sign-On token endpoint. The default value is 60.

**11** (Optional) The read timeout in seconds when connecting to the Red Hat Single Sign-On token endpoint. The default value is 60.

6. Save and exit the editor, then wait for rolling updates to complete.

7. Check the update in the logs or by watching the pod state transitions:

```
oc logs -f ${POD_NAME} -c kafka
oc get pod -w
```

The rolling update configures the brokers to use OAuth 2.0 authorization.

8. Verify the configured permissions by accessing Kafka brokers as clients or users with specific roles, making sure they have the necessary access, or do not have the access they are not supposed to have.

## 6.5.3. Managing policies and permissions in Red Hat Single Sign-On Authorization Services

This section describes the authorization models used by Red Hat Single Sign-On Authorization Services and Kafka, and defines the important concepts in each model.

To grant permissions to access Kafka, you can map Red Hat Single Sign-On Authorization Services objects to Kafka resources by creating an *OAuth client specification* in Red Hat Single Sign-On. Kafka permissions are granted to user accounts or service accounts using Red Hat Single Sign-On Authorization Services rules.

Examples are shown of the different user permissions required for common Kafka operations, such as creating and listing topics.

### 6.5.3.1. Kafka and Red Hat Single Sign-On authorization models overview

Kafka and Red Hat Single Sign-On Authorization Services use different authorization models.

**Kafka authorization model**
Kafka's authorization model uses *resource types*. When a Kafka client performs an action on a broker, the broker uses the configured **KeycloakRBACAuthorizer** to check the client's permissions, based on the action and resource type.

Kafka uses five resource types to control access: **Topic**, **Group**, **Cluster**, **TransactionalId**, and **DelegationToken**. Each resource type has a set of available permissions.

Topic

- **Create**

- **Write**

- **Read**

- **Delete**

- **Describe**

- **DescribeConfigs**

- **Alter**

- **AlterConfigs**

Group

- **Read**

- **Describe**

- **Delete**

Cluster

- **Create**

- **Describe**

- **Alter**

- **DescribeConfigs**

- **AlterConfigs**

- **IdempotentWrite**

- **ClusterAction**

TransactionalId

- **Describe**

- **Write**

DelegationToken

- **Describe**

Red Hat Single Sign-On Authorization Services model
The Red Hat Single Sign-On Authorization Services model has four concepts for defining and granting permissions: *resources*, *authorization scopes*, *policies*, and *permissions*.

Resources

A resource is a set of resource definitions that are used to match resources with permitted actions. A resource might be an individual topic, for example, or all topics with names starting with the same prefix. A resource definition is associated with a set of available authorization scopes, which represent a set of all actions available on the resource. Often, only a subset of these actions is actually permitted.

Authorization scopes

An authorization scope is a set of all the available actions on a specific resource definition. When you define a new resource, you add scopes from the set of all scopes.

Policies

A policy is an authorization rule that uses criteria to match against a list of accounts. Policies can match:

- *Service accounts* based on client ID or roles

- *User accounts* based on username, groups, or roles.

Permissions

A permission grants a subset of authorization scopes on a specific resource definition to a set of users.

Additional resources

- [Kafka authorization model](#)

### 6.5.3.2. Map Red Hat Single Sign-On Authorization Services to the Kafka authorization model

The Kafka authorization model is used as a basis for defining the Red Hat Single Sign-On roles and resources that will control access to Kafka.

To grant Kafka permissions to user accounts or service accounts, you first create an *OAuth client specification* in Red Hat Single Sign-On for the Kafka broker. You then specify Red Hat Single Sign-On Authorization Services rules on the client. Typically, the client id of the OAuth client that represents the broker is **kafka**. The [example configuration files](#) provided with AMQ Streams use **kafka** as the OAuth client id.

> **NOTE**
>
> If you have multiple Kafka clusters, you can use a single OAuth client (**kafka**) for all of them. This gives you a single, unified space in which to define and manage authorization rules. However, you can also use different OAuth client ids (for example, **my-cluster-kafka** or **cluster-dev-kafka**) and define authorization rules for each cluster within each client configuration.

The **kafka** client definition must have the **Authorization Enabled** option enabled in the Red Hat Single Sign-On Admin Console.

All permissions exist within the scope of the **kafka** client. If you have different Kafka clusters configured with different OAuth client IDs, they each need a separate set of permissions even though they're part of the same Red Hat Single Sign-On realm.

When the Kafka client uses OAUTHBEARER authentication, the Red Hat Single Sign-On authorizer

(**KeycloakRBACAuthorizer**) uses the access token of the current session to retrieve a list of grants from the Red Hat Single Sign-On server. To retrieve the grants, the authorizer evaluates the Red Hat Single Sign-On Authorization Services policies and permissions.

## Authorization scopes for Kafka permissions

An initial Red Hat Single Sign-On configuration usually involves uploading authorization scopes to create a list of all possible actions that can be performed on each Kafka resource type. This step is performed once only, before defining any permissions. You can add authorization scopes manually instead of uploading them.

Authorization scopes must contain all the possible Kafka permissions regardless of the resource type:

- **Create**

- **Write**

- **Read**

- **Delete**

- **Describe**

- **Alter**

- **DescribeConfig**

- **AlterConfig**

- **ClusterAction**

- **IdempotentWrite**

> **NOTE**
>
> If you're certain you won't need a permission (for example, **IdempotentWrite**), you can omit it from the list of authorization scopes. However, that permission won't be available to target on Kafka resources.

## Resource patterns for permissions checks

Resource patterns are used for pattern matching against the targeted resources when performing permission checks. The general pattern format is *RESOURCE-TYPE:PATTERN-NAME*.

The resource types mirror the Kafka authorization model. The pattern allows for two matching options:

- Exact matching (when the pattern does not end with **\***)

- Prefix matching (when the pattern ends with **\***)

## Example patterns for resources

```
Topic:my-topic
Topic:orders-*
Group:orders-*
Cluster:*
```

Additionally, the general pattern format can be prefixed by **kafka-cluster:*CLUSTER-NAME*** followed by a comma, where *CLUSTER-NAME* refers to the **metadata.name** in the Kafka custom resource.

### Example patterns for resources with cluster prefix

```
kafka-cluster:my-cluster,Topic:*
kafka-cluster:*,Group:b_*
```

When the **kafka-cluster** prefix is missing, it is assumed to be **kafka-cluster:***.

When defining a resource, you can associate it with a list of possible authorization scopes which are relevant to the resource. Set whatever actions make sense for the targeted resource type.

Though you may add any authorization scope to any resource, only the scopes supported by the resource type are considered for access control.

### Policies for applying access permission

Policies are used to target permissions to one or more user accounts or service accounts. Targeting can refer to:

- Specific user or service accounts

- Realm roles or client roles

- User groups

- JavaScript rules to match a client IP address

A policy is given a unique name and can be reused to target multiple permissions to multiple resources.

### Permissions to grant access

Use fine-grained permissions to pull together the policies, resources, and authorization scopes that grant access to users.

The name of each permission should clearly define which permissions it grants to which users. For example, **Dev Team B can read from topics starting with x**.

### Additional resources

- For more information about how to configure permissions through Red Hat Single Sign-On Authorization Services, see Section 6.5.4, "Trying Red Hat Single Sign-On Authorization Services".

### 6.5.3.3. Example permissions required for Kafka operations

The following examples demonstrate the user permissions required for performing common operations on Kafka.

### Create a topic

To create a topic, the **Create** permission is required for the specific topic, or for **Cluster:kafka-cluster**.

```
bin/kafka-topics.sh --create --topic my-topic \
    --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## List topics

If a user has the **Describe** permission on a specified topic, the topic is listed.

```
bin/kafka-topics.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Display topic details

To display a topic's details, **Describe** and **DescribeConfigs** permissions are required on the topic.

```
bin/kafka-topics.sh --describe --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Produce messages to a topic

To produce messages to a topic, **Describe** and **Write** permissions are required on the topic.

If the topic hasn't been created yet, and topic auto–creation is enabled, the permissions to create a topic are required.

```
bin/kafka-console-producer.sh  --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --producer.config=/tmp/config.properties
```

## Consume messages from a topic

To consume messages from a topic, **Describe** and **Read** permissions are required on the topic. Consuming from the topic normally relies on storing the consumer offsets in a consumer group, which requires additional **Describe** and **Read** permissions on the consumer group.

Two **resources** are needed for matching. For example:

```
Topic:my-topic
Group:my-group-*
```

```
bin/kafka-console-consumer.sh --topic my-topic --group my-group-1 --from-beginning \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --consumer.config /tmp/config.properties
```

## Produce messages to a topic using an idempotent producer

As well as the permissions for producing to a topic, an additional **IdempotentWrite** permission is required on the **Cluster:kafka-cluster** resource.

Two **resources** are needed for matching. For example:

```
Topic:my-topic
Cluster:kafka-cluster
```

```
bin/kafka-console-producer.sh  --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --producer.config=/tmp/config.properties --
producer-property enable.idempotence=true --request-required-acks -1
```

## List consumer groups

When listing consumer groups, only the groups on which the user has the **Describe** permissions are returned. Alternatively, if the user has the **Describe** permission on the **Cluster:kafka-cluster**, all the consumer groups are returned.

```
bin/kafka-consumer-groups.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Display consumer group details

To display a consumer group's details, the **Describe** permission is required on the group and the topics associated with the group.

```
bin/kafka-consumer-groups.sh --describe --group my-group-1 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Change topic configuration

To change a topic's configuration, the **Describe** and **Alter** permissions are required on the topic.

```
bin/kafka-topics.sh --alter --topic my-topic --partitions 2 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Display Kafka broker configuration

In order to use **kafka-configs.sh** to get a broker's configuration, the **DescribeConfigs** permission is required on the **Cluster:kafka-cluster**.

```
bin/kafka-configs.sh --entity-type brokers --entity-name 0 --describe --all \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Change Kafka broker configuration

To change a Kafka broker's configuration, **DescribeConfigs** and **AlterConfigs** permissions are required on **Cluster:kafka-cluster**.

```
bin/kafka-configs --entity-type brokers --entity-name 0 --alter --add-config log.cleaner.threads=2 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Delete a topic

To delete a topic, the **Describe** and **Delete** permissions are required on the topic.

```
bin/kafka-topics.sh --delete --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

## Select a lead partition

To run leader selection for topic partitions, the **Alter** permission is required on the **Cluster:kafka-cluster**.

```
bin/kafka-leader-election.sh --topic my-topic --partition 0 --election-type PREFERRED  /
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --admin.config /tmp/config.properties
```

## Reassign partitions

To generate a partition reassignment file, **Describe** permissions are required on the topics involved.

```
bin/kafka-reassign-partitions.sh --topics-to-move-json-file /tmp/topics-to-move.json --broker-list "0,1" --generate \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties > /tmp/partition-reassignment.json
```

To execute the partition reassignment, **Describe** and **Alter** permissions are required on **Cluster:kafka-cluster**. Also, **Describe** permissions are required on the topics involved.

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --execute \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties
```

To verify partition reassignment, **Describe**, and **AlterConfigs** permissions are required on **Cluster:kafka-cluster**, and on each of the topics involved.

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --verify \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties
```

## 6.5.4. Trying Red Hat Single Sign-On Authorization Services

This example explains how to use Red Hat Single Sign-On Authorization Services with **keycloak** authorization. Use Red Hat Single Sign-On Authorization Services to enforce access restrictions on Kafka clients. Red Hat Single Sign-On Authorization Services use authorization scopes, policies and permissions to define and apply access control to resources.

Red Hat Single Sign-On Authorization Services REST endpoints provide a list of granted permissions on resources for authenticated users. The list of grants (permissions) is fetched from the Red Hat Single Sign-On server as the first action after an authenticated session is established by the Kafka client. The list is refreshed in the background so that changes to the grants are detected. Grants are cached and enforced locally on the Kafka broker for each user session to provide fast authorization decisions.

AMQ Streams provides example configuration files. These include the following example files for setting up Red Hat Single Sign-On:

**kafka-ephemeral-oauth-single-keycloak-authz.yaml**

An example **Kafka** custom resource configured for OAuth 2.0 token-based authorization using Red Hat Single Sign-On. You can use the custom resource to deploy a Kafka cluster that uses **keycloak** authorization and token-based **oauth** authentication.

**kafka-authz-realm.json**

An example Red Hat Single Sign-On realm configured with sample groups, users, roles and clients. You can import the realm into a Red Hat Single Sign-On instance to set up fine-grained permissions to access Kafka.

If you want to try the example with Red Hat Single Sign-On, use these files to perform the tasks outlined in this section in the order shown.

1. Accessing the Red Hat Single Sign-On Admin Console

2. Deploying a Kafka cluster with Red Hat Single Sign-On authorization

3. Preparing TLS connectivity for a CLI Kafka client session

4. Checking authorized access to Kafka using a CLI Kafka client session

## Authentication

When you configure token-based **oauth** authentication, you specify a **jwksEndpointUri** as the URI for local JWT validation. When you configure **keycloak** authorization, you specify a **tokenEndpointUri** as the URI of the Red Hat Single Sign-On token endpoint. The hostname for both URIs must be the same.

## Targeted permissions with group or role policies

In Red Hat Single Sign-On, confidential clients with service accounts enabled can authenticate to the server in their own name using a client ID and a secret. This is convenient for microservices that typically act in their own name, and not as agents of a particular user (like a web site). Service accounts can have roles assigned like regular users. They cannot, however, have groups assigned. As a consequence, if you want to target permissions to microservices using service accounts, you cannot use group policies, and should instead use role policies. Conversely, if you want to limit certain permissions only to regular user accounts where authentication with a username and password is required, you can achieve that as a side effect of using the group policies rather than the role policies. This is what is used in this example for permissions that start with **ClusterManager**. Performing cluster management is usually done interactively using CLI tools. It makes sense to require the user to log in before using the resulting access token to authenticate to the Kafka broker. In this case, the access token represents the specific user, rather than the client application.

### 6.5.4.1. Accessing the Red Hat Single Sign-On Admin Console

Set up Red Hat Single Sign-On, then connect to its Admin Console and add the preconfigured realm. Use the example **kafka-authz-realm.json** file to import the realm. You can check the authorization rules defined for the realm in the Admin Console. The rules grant access to the resources on the Kafka cluster configured to use the example Red Hat Single Sign-On realm.

### Prerequisites

- A running OpenShift cluster.

- The AMQ Streams **examples/security/keycloak-authorization/kafka-authz-realm.json** file that contains the preconfigured realm.

### Procedure

1. Install the Red Hat Single Sign-On server using the Red Hat Single Sign-On Operator as described in Server Installation and Configuration in the Red Hat Single Sign-On documentation.

2. Wait until the Red Hat Single Sign-On instance is running.

3. Get the external hostname to be able to access the Admin Console.

   ```
   NS=sso
   oc get ingress keycloak -n $NS
   ```

   In this example, we assume the Red Hat Single Sign-On server is running in the **sso** namespace.

4. Get the password for the **admin** user.

   ```
   oc get -n $NS pod keycloak-0 -o yaml | less
   ```

   The password is stored as a secret, so get the configuration YAML file for the Red Hat Single Sign-On instance to identify the name of the secret (**secretKeyRef.name**).

5. Use the name of the secret to obtain the clear text password.

> SECRET_NAME=credential-keycloak
> oc get -n $NS secret $SECRET_NAME -o yaml | grep PASSWORD | awk '{print $2}' |
> base64 -D

In this example, we assume the name of the secret is **credential-keycloak**.

6. Log in to the Admin Console with the username **admin** and the password you obtained.
Use **https://***HOSTNAME* to access the OpenShift ingress.

You can now upload the example realm to Red Hat Single Sign-On using the Admin Console.

7. Click **Add Realm** to import the example realm.

8. Add the **examples/security/keycloak-authorization/kafka-authz-realm.json** file, and then
click **Create**.
You now have **kafka-authz** as your current realm in the Admin Console.

The default view displays the **Master** realm.

9. In the Red Hat Single Sign-On Admin Console, go to **Clients** > **kafka** > **Authorization** > **Settings**
and check that **Decision Strategy** is set to **Affirmative**.
An affirmative policy means that at least one policy must be satisfied for a client to access the
Kafka cluster.

10. In the Red Hat Single Sign-On Admin Console, go to **Groups**, **Users**, **Roles** and **Clients** to view
the realm configuration.

   Groups

   **Groups** are used to create user groups and set user permissions. Groups are sets of users
   with a name assigned. They are used to compartmentalize users into geographical,
   organizational or departmental units. Groups can be linked to an LDAP identity provider. You
   can make a user a member of a group through a custom LDAP server admin user interface,
   for example, to grant permissions on Kafka resources.

   Users

   **Users** are used to create users. For this example,   **alice** and **bob** are defined.  **alice** is a
   member of the **ClusterManager** group and **bob** is a member of  **ClusterManager-my-
   cluster** group. Users can be stored in an LDAP identity provider.

   Roles

   **Roles** mark users or clients as having certain permissions. Roles are a concept analogous to
   groups. They are usually used to *tag* users with organizational roles and have the requisite
   permissions. Roles cannot be stored in an LDAP identity provider. If LDAP is a requirement,
   you can use groups instead, and add Red Hat Single Sign-On roles to the groups so that
   when users are assigned a group they also get a corresponding role.

   Clients

   **Clients** can have specific configurations. For this example,  **kafka**, **kafka-cli**, **team-a-client**,
   and **team-b-client** clients are configured.

   - The **kafka** client is used by Kafka brokers to perform the necessary OAuth 2.0
   communication for access token validation. This client also contains the authorization
   services resource definitions, policies, and authorization scopes used to perform

authorization on the Kafka brokers. The authorization configuration is defined in the **kafka** client from the **Authorization** tab, which becomes visible when **Authorization Enabled** is switched on from the **Settings** tab.

- The **kafka-cli** client is a public client that is used by the Kafka command line tools when authenticating with username and password to obtain an access token or a refresh token.

- The **team-a-client** and **team-b-client** clients are confidential clients representing services with partial access to certain Kafka topics.

11. In the Red Hat Single Sign-On Admin Console, go to **Authorization** > **Permissions** to see the granted permissions that use the resources and policies defined for the realm.
    For example, the **kafka** client has the following permissions:

> Dev Team A can write to topics that start with x_ on any cluster
> Dev Team B can read from topics that start with x_ on any cluster
> Dev Team B can update consumer group offsets that start with x_ on any cluster
> ClusterManager of my-cluster Group has full access to cluster config on my-cluster
> ClusterManager of my-cluster Group has full access to consumer groups on my-cluster
> ClusterManager of my-cluster Group has full access to topics on my-cluster

**Dev Team A**

The Dev Team A realm role can write to topics that start with **x_** on any cluster. This combines a resource called **Topic:x_\***, **Describe** and **Write** scopes, and the **Dev Team A** policy. The **Dev Team A** policy matches all users that have a realm role called **Dev Team A**.

**Dev Team B**

The Dev Team B realm role can read from topics that start with **x_** on any cluster. This combines **Topic:x_\***, **Group:x_\*** resources, **Describe** and **Read** scopes, and the **Dev Team B** policy. The **Dev Team B** policy matches all users that have a realm role called **Dev Team B**. Matching users and clients have the ability to read from topics, and update the consumed offsets for topics and consumer groups that have names starting with **x_**.

### 6.5.4.2. Deploying a Kafka cluster with Red Hat Single Sign-On authorization

Deploy a Kafka cluster configured to connect to the Red Hat Single Sign-On server. Use the example **kafka-ephemeral-oauth-single-keycloak-authz.yaml** file to deploy the Kafka cluster as a **Kafka** custom resource. The example deploys a single-node Kafka cluster with **keycloak** authorization and **oauth** authentication.

#### Prerequisites

- The Red Hat Single Sign-On authorization server is deployed to your OpenShift cluster and loaded with the example realm.

- The Cluster Operator is deployed to your OpenShift cluster.

- The AMQ Streams **examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml** custom resource.

#### Procedure

1. Use the hostname of the Red Hat Single Sign-On instance you deployed to prepare a truststore certificate for Kafka brokers to communicate with the Red Hat Single Sign-On server.

   ```
   SSO_HOST=SSO-HOSTNAME
   SSO_HOST_PORT=$SSO_HOST:443
   STOREPASS=storepass

   echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
   /BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.crt
   ```

   The certificate is required as OpenShift ingress is used to make a secure (HTTPS) connection.

2. Deploy the certificate to OpenShift as a secret.

   ```
   oc create secret generic oauth-server-cert --from-file=/tmp/sso.crt -n $NS
   ```

3. Set the hostname as an environment variable

   ```
   SSO_HOST=SSO-HOSTNAME
   ```

4. Create and deploy the example Kafka cluster.

   ```
   cat examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml | sed -E 's#\${SSO_HOST}'"#$SSO_HOST#" | oc create -n $NS -f -
   ```

### 6.5.4.3. Preparing TLS connectivity for a CLI Kafka client session

Create a new pod for an interactive CLI session. Set up a truststore with a Red Hat Single Sign-On certificate for TLS connectivity. The truststore is to connect to Red Hat Single Sign-On and the Kafka broker.

**Prerequisites**

- The Red Hat Single Sign-On authorization server is deployed to your OpenShift cluster and loaded with the example realm.
  In the Red Hat Single Sign-On Admin Console, check the roles assigned to the clients are displayed in **Clients** > **Service Account Roles**.

- The Kafka cluster configured to connect with Red Hat Single Sign-On is deployed to your OpenShift cluster.

**Procedure**

1. Run a new interactive pod container using the AMQ Streams Kafka image to connect to a running Kafka broker.

   ```
   NS=sso
   oc run -ti --restart=Never --image=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0 kafka-cli -n $NS -- /bin/sh
   ```

> **NOTE**
>
> If **oc** times out waiting on the image download, subsequent attempts may result in an *AlreadyExists* error.

2. Attach to the pod container.

   ```
   oc attach -ti kafka-cli -n $NS
   ```

3. Use the hostname of the Red Hat Single Sign-On instance to prepare a certificate for client connection using TLS.

   ```
   SSO_HOST=SSO-HOSTNAME
   SSO_HOST_PORT=$SSO_HOST:443
   STOREPASS=storepass

   echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
   /BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.crt
   ```

4. Create a truststore for TLS connection to the Kafka brokers.

   ```
   keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias sso -storepass $STOREPASS
   -import -file /tmp/sso.crt -noprompt
   ```

5. Use the Kafka bootstrap address as the hostname of the Kafka broker and the **tls** listener port (9093) to prepare a certificate for the Kafka broker.

   ```
   KAFKA_HOST_PORT=my-cluster-kafka-bootstrap:9093
   STOREPASS=storepass

   echo "Q" | openssl s_client -showcerts -connect $KAFKA_HOST_PORT 2>/dev/null | awk '
   /BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/my-cluster-kafka.crt
   ```

6. Add the certificate for the Kafka broker to the truststore.

   ```
   keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias my-cluster-kafka -storepass
   $STOREPASS -import -file /tmp/my-cluster-kafka.crt -noprompt
   ```

   Keep the session open to check authorized access.

### 6.5.4.4. Checking authorized access to Kafka using a CLI Kafka client session

Check the authorization rules applied through the Red Hat Single Sign-On realm using an interactive CLI session. Apply the checks using Kafka's example producer and consumer clients to create topics with user and service accounts that have different levels of access.

Use the **team-a-client** and **team-b-client** clients to check the authorization rules. Use the **alice** admin user to perform additional administrative tasks on Kafka.

The AMQ Streams Kafka image used in this example contains Kafka producer and consumer binaries.

**Prerequisites**

- ZooKeeper and Kafka are running in the OpenShift cluster to be able to send and receive messages.

- The interactive CLI Kafka client session is started.
  Apache Kafka download .

**Setting up client and admin user configuration**

1. Prepare a Kafka configuration file with authentication properties for the **team-a-client** client.

   ```
   SSO_HOST=SSO-HOSTNAME

   cat > /tmp/team-a-client.properties << EOF
   security.protocol=SASL_SSL
   ssl.truststore.location=/tmp/truststore.p12
   ssl.truststore.password=$STOREPASS
   ssl.truststore.type=PKCS12
   sasl.mechanism=OAUTHBEARER
   sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
     oauth.client.id="team-a-client" \
     oauth.client.secret="team-a-client-secret" \
     oauth.ssl.truststore.location="/tmp/truststore.p12" \
     oauth.ssl.truststore.password="$STOREPASS" \
     oauth.ssl.truststore.type="PKCS12" \
     oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-connect/token" ;
   sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler
   EOF
   ```

   The SASL OAUTHBEARER mechanism is used. This mechanism requires a client ID and client secret, which means the client first connects to the Red Hat Single Sign-On server to obtain an access token. The client then connects to the Kafka broker and uses the access token to authenticate.

2. Prepare a Kafka configuration file with authentication properties for the **team-b-client** client.

   ```
   cat > /tmp/team-b-client.properties << EOF
   security.protocol=SASL_SSL
   ssl.truststore.location=/tmp/truststore.p12
   ssl.truststore.password=$STOREPASS
   ssl.truststore.type=PKCS12
   sasl.mechanism=OAUTHBEARER
   sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
     oauth.client.id="team-b-client" \
     oauth.client.secret="team-b-client-secret" \
     oauth.ssl.truststore.location="/tmp/truststore.p12" \
     oauth.ssl.truststore.password="$STOREPASS" \
     oauth.ssl.truststore.type="PKCS12" \
     oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-connect/token" ;
   sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler
   EOF
   ```

3. Authenticate admin user **alice** by using  **curl** and performing a password grant authentication to obtain a refresh token.

```
USERNAME=alice
PASSWORD=alice-password

GRANT_RESPONSE=$(curl -X POST "https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-connect/token" -H 'Content-Type: application/x-www-form-urlencoded' -d
"grant_type=password&username=$USERNAME&password=$PASSWORD&client_id=kafka-cli&scope=offline_access" -s -k)

REFRESH_TOKEN=$(echo $GRANT_RESPONSE | awk -F "refresh_token\":\"" '{printf $2}' | awk -F "\"" '{printf $1}')
```

The refresh token is an offline token that is long-lived and does not expire.

4. Prepare a Kafka configuration file with authentication properties for the admin user **alice**.

```
cat > /tmp/alice.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.refresh.token="$REFRESH_TOKEN" \
  oauth.client.id="kafka-cli" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler
EOF
```

The **kafka-cli** public client is used for the  **oauth.client.id** in the  **sasl.jaas.config**. Since it's a public client it does not require a secret. The client authenticates with the refresh token that was authenticated in the previous step. The refresh token requests an access token behind the scenes, which is then sent to the Kafka broker for authentication.

## Producing messages with authorized access

Use the **team-a-client** configuration to check that you can produce messages to topics that start with **a_** or **x_**.

1. Write to topic **my-topic**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic my-topic \
  --producer.config=/tmp/team-a-client.properties
First message
```

This request returns a **Not authorized to access topics: [my-topic]** error.

**team-a-client** has a **Dev Team A** role that gives it permission to perform any supported actions on topics that start with **a_**, but can only write to topics that start with **x_**. The topic named **my-topic** matches neither of those rules.

2. Write to topic **a_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic a_messages \
  --producer.config /tmp/team-a-client.properties
First message
Second message
```

Messages are produced to Kafka successfully.

3. Press CTRL+C to exit the CLI application.

4. Check the Kafka container log for a debug log of **Authorization GRANTED** for the request.

```
oc logs my-cluster-kafka-0 -f -n $NS
```

### Consuming messages with authorized access

Use the **team-a-client** configuration to consume messages from topic **a_messages**.

1. Fetch messages from topic **a_messages**.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties
```

The request returns an error because the **Dev Team A** role for **team-a-client** only has access to consumer groups that have names starting with **a_**.

2. Update the **team-a-client** properties to specify the custom consumer group it is permitted to use.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group a_consumer_group_1
```

The consumer receives all the messages from the **a_messages** topic.

### Administering Kafka with authorized access

The **team-a-client** is an account without any cluster-level access, but it can be used with some administrative operations.

1. List topics.

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties --list
```

The **a_messages** topic is returned.

2. List consumer groups.

```
bin/kafka-consumer-groups.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --
command-config /tmp/team-a-client.properties --list
```

The **a_consumer_group_1** consumer group is returned.

Fetch details on the cluster configuration.

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-a-client.properties \
    --entity-type brokers --describe --entity-default
```

The request returns an error because the operation requires cluster level permissions that **team-a-client** does not have.

## Using clients with different permissions

Use the **team-b-client** configuration to produce messages to topics that start with    **b_**.

1. Write to topic **a_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
a_messages \
    --producer.config /tmp/team-b-client.properties
Message 1
```

This request returns a **Not authorized to access topics: [a_messages]** error.

2. Write to topic **b_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
b_messages \
    --producer.config /tmp/team-b-client.properties
Message 1
Message 2
Message 3
```

Messages are produced to Kafka successfully.

3. Write to topic **x_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
    --producer.config /tmp/team-b-client.properties
Message 1
```

A **Not authorized to access topics: [x_messages]** error is returned, The  **team-b-client** can only read from topic **x_messages**.

4. Write to topic **x_messages** using **team-a-client**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
    --producer.config /tmp/team-a-client.properties
```

> Message 1

This request returns a **Not authorized to access topics: [x_messages]** error. The **team-a-client** can write to the **x_messages** topic, but it does not have a permission to create a topic if it does not yet exist. Before **team-a-client** can write to the **x_messages** topic, an admin *power user* must create it with the correct configuration, such as the number of partitions and replicas.

## Managing Kafka with an authorized admin user

Use admin user **alice** to manage Kafka. **alice** has full access to manage everything on any Kafka cluster.

1. Create the **x_messages** topic as **alice**.

   ```
   bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
   /tmp/alice.properties \
     --topic x_messages --create --replication-factor 1 --partitions 1
   ```

   The topic is created successfully.

2. List all topics as **alice**.

   ```
   bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
   /tmp/alice.properties --list
   bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
   /tmp/team-a-client.properties --list
   bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
   /tmp/team-b-client.properties --list
   ```

   Admin user **alice** can list all the topics, whereas **team-a-client** and **team-b-client** can only list the topics they have access to.

   The **Dev Team A** and **Dev Team B** roles both have **Describe** permission on topics that start with **x_**, but they cannot see the other team's topics because they do not have **Describe** permissions on them.

3. Use the **team-a-client** to produce messages to the **x_messages** topic:

   ```
   bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
   x_messages \
     --producer.config /tmp/team-a-client.properties
   Message 1
   Message 2
   Message 3
   ```

   As **alice** created the **x_messages** topic, messages are produced to Kafka successfully.

4. Use the **team-b-client** to produce messages to the **x_messages** topic.

   ```
   bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
   x_messages \
     --producer.config /tmp/team-b-client.properties
   Message 4
   Message 5
   ```

   This request returns a **Not authorized to access topics: [x_messages]** error.

5. Use the **team-b-client** to consume messages from the **x_messages** topic:

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-b-client.properties --group
x_consumer_group_b
```

The consumer receives all the messages from the **x_messages** topic.

6. Use the **team-a-client** to consume messages from the **x_messages** topic.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
x_consumer_group_a
```

This request returns a **Not authorized to access topics: [x_messages]** error.

7. Use the **team-a-client** to consume messages from a consumer group that begins with **a_**.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
a_consumer_group_a
```

This request returns a **Not authorized to access topics: [x_messages]** error.

**Dev Team A** has no **Read** access on topics that start with a **x_**.

8. Use **alice** to produce messages to the **x_messages** topic.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/alice.properties
```

Messages are produced to Kafka successfully.

**alice** can read from or write to any topic.

9. Use **alice** to read the cluster configuration.

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/alice.properties \
  --entity-type brokers --describe --entity-default
```

The cluster configuration for this example is empty.

## Additional resources

- Server Installation and Configuration

- Map Red Hat Single Sign-On Authorization Services to the Kafka authorization model

# CHAPTER 7. USING STRIMZI OPERATORS

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

## 7.1. WATCHING NAMESPACES WITH AMQ STREAMS OPERATORS

Operators watch and manage AMQ Streams resources in namespaces. The Cluster Operator can watch a single namespace, multiple namespaces, or all namespaces in an OpenShift cluster. The Topic Operator and User Operator can watch a single namespace.

- The Cluster Operator watches for **Kafka** resources

- The Topic Operator watches for **KafkaTopic** resources

- The User Operator watches for **KafkaUser** resources

The Topic Operator and the User Operator can only watch a single Kafka cluster in a namespace. And they can only be connected to a single Kafka cluster.

If multiple Topic Operators watch the same namespace, name collisions and topic deletion can occur. This is because each Kafka cluster uses Kafka topics that have the same name (such as **__consumer_offsets**). Make sure that only one Topic Operator watches a given namespace.

When using multiple User Operators with a single namespace, a user with a given username can exist in more than one Kafka cluster.

If you deploy the Topic Operator and User Operator using the Cluster Operator, they watch the Kafka cluster deployed by the Cluster Operator by default. You can also specify a namespace using **watchedNamespace** in the operator configuration.

For a standalone deployment of each operator, you specify a namespace and connection to the Kafka cluster to watch in the configuration.

## 7.2. USING THE CLUSTER OPERATOR

The Cluster Operator is used to deploy a Kafka cluster and other Kafka components.

For information on deploying the Cluster Operator, see Deploying the Cluster Operator .

### 7.2.1. Role-Based Access Control (RBAC) resources

The Cluster Operator creates and manages RBAC resources for AMQ Streams components that need access to OpenShift resources.

For the Cluster Operator to function, it needs permission within the OpenShift cluster to interact with Kafka resources, such as **Kafka** and **KafkaConnect**, as well as managed resources like **ConfigMap**, **Pod**, **Deployment**, **StatefulSet**, and **Service**.

Permission is specified through OpenShift role-based access control (RBAC) resources:

- **ServiceAccount**

- **Role** and **ClusterRole**

- **RoleBinding** and **ClusterRoleBinding**

## 7.2.1.1. Delegating privileges to AMQ Streams components

The Cluster Operator runs under a service account called **strimzi-cluster-operator**. It is assigned cluster roles that give it permission to create the RBAC resources for AMQ Streams components. Role bindings associate the cluster roles with the service account.

OpenShift prevents components operating under one **ServiceAccount** from granting another **ServiceAccount** privileges that the granting **ServiceAccount** does not have. Because the Cluster Operator creates the **RoleBinding** and **ClusterRoleBinding** RBAC resources needed by the resources it manages, it requires a role that gives it the same privileges.

The following tables describe the RBAC resources created by the Cluster Operator.

Table 7.1. **ServiceAccount** resources

| Name | Used by |
|---|---|
| *<cluster_name>*-kafka | Kafka broker pods |
| *<cluster_name>*-zookeeper | ZooKeeper pods |
| *<cluster_name>*-cluster-connect | Kafka Connect pods |
| *<cluster_name>*-mirror-maker | MirrorMaker pods |
| *<cluster_name>*-mirrormaker2 | MirrorMaker 2.0 pods |
| *<cluster_name>*-bridge | Kafka Bridge pods |
| *<cluster_name>*-entity-operator | Entity Operator |

Table 7.2. **ClusterRole** resources

| Name | Used by |
|---|---|
| **strimzi-cluster-operator-namespaced** | Cluster Operator |
| **strimzi-cluster-operator-global** | Cluster Operator |
| **strimzi-cluster-operator-leader-election** | Cluster Operator |
| **strimzi-kafka-broker** | Cluster Operator, rack feature (when used) |
| **strimzi-entity-operator** | Cluster Operator, Topic Operator, User Operator |
| **strimzi-kafka-client** | Cluster Operator, Kafka clients for rack awareness |

Table 7.3. **ClusterRoleBinding** resources

| Name | Used by |
|------|---------|
| **strimzi-cluster-operator** | Cluster Operator |
| **strimzi-cluster-operator-kafka-broker-delegation** | Cluster Operator, Kafka brokers for rack awareness |
| **strimzi-cluster-operator-kafka-client-delegation** | Cluster Operator, Kafka clients for rack awareness |

Table 7.4. **RoleBinding** resources

| Name | Used by |
|------|---------|
| **strimzi-cluster-operator** | Cluster Operator |
| **strimzi-cluster-operator-kafka-broker-delegation** | Cluster Operator, Kafka brokers for rack awareness |

### 7.2.1.2. Running the Cluster Operator using a **ServiceAccount**

The Cluster Operator is best run using a **ServiceAccount**:

Example **ServiceAccount** for the Cluster Operator

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
```

The **Deployment** of the operator then needs to specify this in its
**spec.template.spec.serviceAccountName**:

Partial example of **Deployment** for the Cluster Operator

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 1
  selector:
    matchLabels:
      name: strimzi-cluster-operator
```

```
    strimzi.io/kind: cluster-operator
  template:
    # ...
```

Note line 12, where **strimzi-cluster-operator** is specified as the **serviceAccountName**.

### 7.2.1.3. ClusterRole resources

The Cluster Operator uses **ClusterRole** resources to provide the necessary access to resources. Depending on the OpenShift cluster setup, a cluster administrator might be needed to create the cluster roles.

> **NOTE**
>
> Cluster administrator rights are only needed for the creation of **ClusterRole** resources. The Cluster Operator will not run under a cluster admin account.

**ClusterRole** resources follow the *principle of least privilege* and contain only those privileges needed by the Cluster Operator to operate the cluster of the Kafka component. The first set of assigned privileges allow the Cluster Operator to manage OpenShift resources such as **StatefulSet**, **Deployment**, **Pod**, and **ConfigMap**.

All cluster roles are required by the Cluster Operator in order to delegate privileges.

The Cluster Operator uses the **strimzi-cluster-operator-namespaced** and **strimzi-cluster-operator-global** cluster roles to grant permission at the namespace-scoped resources level and cluster-scoped resources level.

**ClusterRole with namespaced resources for the Cluster Operator**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-namespaced
  labels:
    app: strimzi
rules:
  # Resources in this role are used by the operator based on an operand being deployed in some namespace. When needed, you
  # can deploy the operator as a cluster-wide operator. But grant the rights listed in this role only on the namespaces
  # where the operands will be deployed. That way, you can limit the access the operator has to other namespaces where it
  # does not manage any clusters.
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to access and manage rolebindings to grant Strimzi components cluster permissions
      - rolebindings
    verbs:
      - get
      - list
      - watch
      - create
```

```
      - delete
      - patch
      - update
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to access and manage roles to grant the entity operator permissions
      - roles
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - ""
    resources:
      # The cluster operator needs to access and delete pods, this is to allow it to monitor pod health
      and coordinate rolling updates
      - pods
      # The cluster operator needs to access and manage service accounts to grant Strimzi
      components cluster permissions
      - serviceaccounts
      # The cluster operator needs to access and manage config maps for Strimzi components
      configuration
      - configmaps
      # The cluster operator needs to access and manage services and endpoints to expose Strimzi
      components to network traffic
      - services
      - endpoints
      # The cluster operator needs to access and manage secrets to handle credentials
      - secrets
      # The cluster operator needs to access and manage persistent volume claims to bind them to
      Strimzi components for persistent data
      - persistentvolumeclaims
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      # 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:
    - "" # legacy core events api, used by topic operator
    - "events.k8s.io" # new events api, used by cluster operator
  resources:
    # The cluster operator needs to be able to create events and delegate permissions to do so
    - events
  verbs:
    - create
 - apiGroups:
    # Kafka Connect Build on OpenShift requirement
    - build.openshift.io
  resources:
    - buildconfigs
    - buildconfigs/instantiate
    - builds
  verbs:
```

```
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - networking.k8s.io
    resources:
      # The cluster operator needs to access and manage network policies to lock down
communication between Strimzi components
      - networkpolicies
      # The cluster operator needs to access and manage ingresses which allow external access to the
services in a cluster
      - ingresses
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - route.openshift.io
    resources:
      # The cluster operator needs to access and manage routes to expose Strimzi components for
external access
      - routes
      - routes/custom-host
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - policy
    resources:
      # The cluster operator needs to access and manage pod disruption budgets this limits the number
of concurrent disruptions
      # that a Strimzi component experiences, allowing for higher availability
      - poddisruptionbudgets
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
```

**ClusterRole with cluster-scoped resources for the Cluster Operator**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-global
  labels:
    app: strimzi
rules:
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to create and manage cluster role bindings in the case of an install where a user
      # has specified they want their cluster role bindings generated
      - clusterrolebindings
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - storage.k8s.io
    resources:
      # The cluster operator requires "get" permissions to view storage class details
      # This is because only a persistent volume of a supported storage class type can be resized
      - storageclasses
    verbs:
      - get
  - apiGroups:
      - ""
    resources:
      # The cluster operator requires "list" permissions to view all nodes in a cluster
      # The listing is used to determine the node addresses when NodePort access is configured
      # These addresses are then exposed in the custom resource states
      - nodes
    verbs:
      - list
```

The **strimzi-cluster-operator-leader-election** cluster role represents the permissions needed for the leader election.

**ClusterRole with leader election permissions**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
rules:
  - apiGroups:
      - coordination.k8s.io
    resources:
```

```
    # The cluster operator needs to access and manage leases for leader election
    # The "create" verb cannot be used with "resourceNames"
    - leases
  verbs:
    - create
- apiGroups:
    - coordination.k8s.io
  resources:
    # The cluster operator needs to access and manage leases for leader election
    - leases
  resourceNames:
    # The default RBAC files give the operator only access to the Lease resource names strimzi-
cluster-operator
    # If you want to use another resource name or resource namespace, you have to configure the
RBAC resources accordingly
    - strimzi-cluster-operator
  verbs:
    - get
    - list
    - watch
    - delete
    - patch
    - update
```

The **strimzi-kafka-broker** cluster role represents the access needed by the init container in Kafka pods that use rack awareness.

A role binding named **strimzi-<*cluster_name*>-kafka-init** grants the **<*cluster_name*>-kafka** service account access to nodes within a cluster using the **strimzi-kafka-broker** role. If the rack feature is not used and the cluster is not exposed through **nodeport**, no binding is created.

### ClusterRole for the Cluster Operator allowing it to delegate access to OpenShift nodes to the Kafka broker pods

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-kafka-broker
  labels:
    app: strimzi
rules:
  - apiGroups:
      - ""
    resources:
      # The Kafka Brokers require "get" permissions to view the node they are on
      # This information is used to generate a Rack ID that is used for High Availability configurations
      - nodes
    verbs:
      - get
```

The **strimzi-entity-operator** cluster role represents the access needed by the Topic Operator and User Operator.

The Topic Operator produces OpenShift events with status information, so the ***<cluster_name>-*** **entity-operator** service account is bound to the **strimzi-entity-operator** role, which grants this access via the **strimzi-entity-operator** role binding.

**ClusterRole** for the Cluster Operator allowing it to delegate access to events to the Topic and User Operators

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-entity-operator
  labels:
    app: strimzi
rules:
  - apiGroups:
    - "kafka.strimzi.io"
    resources:
      # The entity operator runs the KafkaTopic assembly operator, which needs to access and manage
KafkaTopic resources
      - kafkatopics
      - kafkatopics/status
      # The entity operator runs the KafkaUser assembly operator, which needs to access and manage
KafkaUser resources
      - kafkausers
      - kafkausers/status
    verbs:
      - get
      - list
      - watch
      - create
      - patch
      - update
      - delete
  - apiGroups:
    - ""
    resources:
      - events
    verbs:
      # The entity operator needs to be able to create events
      - create
  - apiGroups:
    - ""
    resources:
      # The entity operator user-operator needs to access and manage secrets to store generated
credentials
      - secrets
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
```

The **strimzi-kafka-client** cluster role represents the access needed by Kafka clients that use rack awareness.

### ClusterRole for the Cluster Operator allowing it to delegate access to OpenShift nodes to the Kafka client-based pods

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

### 7.2.1.4. ClusterRoleBinding resources

The Cluster Operator uses **ClusterRoleBinding** and **RoleBinding** resources to associate its **ClusterRole** with its **ServiceAccount**: Cluster role bindings are required by cluster roles containing cluster-scoped resources.

### Example **ClusterRoleBinding** for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
 - kind: ServiceAccount
   name: strimzi-cluster-operator
   namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-cluster-operator-global
  apiGroup: rbac.authorization.k8s.io
```

Cluster role bindings are also needed for the cluster roles used in delegating privileges:

### Example **ClusterRoleBinding** for the Cluster Operator and Kafka broker rack awareness

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
  name: strimzi-cluster-operator-kafka-broker-delegation
  labels:
    app: strimzi
# The Kafka broker cluster role must be bound to the cluster operator service account so that it can
delegate the cluster role to the Kafka brokers.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-broker
  apiGroup: rbac.authorization.k8s.io
```

### Example **ClusterRoleBinding** for the Cluster Operator and Kafka client rack awareness

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

Cluster roles containing only namespaced resources are bound using role bindings only.

### Example **RoleBinding** for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-cluster-operator-namespaced
  apiGroup: rbac.authorization.k8s.io
```

**Example RoleBinding for the Cluster Operator and Kafka broker rack awareness**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-entity-operator-delegation
  labels:
    app: strimzi
# The Entity Operator cluster role must be bound to the cluster operator service account so that it can
delegate the cluster role to the Entity Operator.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-entity-operator
  apiGroup: rbac.authorization.k8s.io
```

## 7.2.2. ConfigMap for Cluster Operator logging

Cluster Operator logging is configured through a **ConfigMap** named **strimzi-cluster-operator**.

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

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

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

Alternatively, edit the **ConfigMap** directly:

```
oc edit configmap strimzi-cluster-operator
```

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

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

If the **ConfigMap** is accidentally deleted after the Cluster Operator is deployed, the most recently loaded logging configuration is used. Create a new **ConfigMap** to load a new logging configuration.
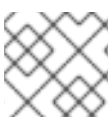
> **NOTE**
>
> Do not remove the **monitorInterval** option from the **ConfigMap**.

## 7.2.3. Configuring the Cluster Operator with environment variables

You can configure the Cluster Operator using environment variables. The supported environment variables are listed here.
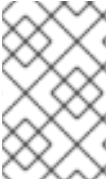
> **NOTE**
>
> The environment variables relate to the container configuration for the deployment of the Cluster Operator image. For more information on **image** configuration, see, Section 12.1.6, "**image**".

**STRIMZI_NAMESPACE**

A comma-separated list of namespaces that the operator operates in. When not set, set to empty string, or set to **\***, the Cluster Operator operates in all namespaces.
The Cluster Operator deployment might use the downward API to set this automatically to the namespace the Cluster Operator is deployed in.

**Example configuration for Cluster Operator namespaces**

```
env:
  - name: STRIMZI_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

**STRIMZI_FULL_RECONCILIATION_INTERVAL_MS**

Optional, default is 120000 ms. The interval between periodic reconciliations, in milliseconds.

**STRIMZI_OPERATION_TIMEOUT_MS**

Optional, default 300000 ms. The timeout for internal operations, in milliseconds. Increase this value when using AMQ Streams on clusters where regular OpenShift operations take longer than usual (because of slow downloading of Docker images, for example).

**STRIMZI_ZOOKEEPER_ADMIN_SESSION_TIMEOUT_MS**

Optional, default 10000 ms. The session timeout for the Cluster Operator's ZooKeeper admin client, in milliseconds. Increase the value if ZooKeeper requests from the Cluster Operator are regularly failing due to timeout issues. There is a maximum allowed session time set on the ZooKeeper server side via the **maxSessionTimeout** config. By default, the maximum session timeout value is 20 times the default **tickTime** (whose default is 2000) at 40000 ms. If you require a higher timeout, change the **maxSessionTimeout** ZooKeeper server configuration value.

**STRIMZI_OPERATIONS_THREAD_POOL_SIZE**

Optional, default 10. The worker thread pool size, which is used for various asynchronous and blocking operations that are run by the Cluster Operator.

**STRIMZI_OPERATOR_NAME**

Optional, defaults to the pod's hostname. The operator name identifies the AMQ Streams instance when emitting OpenShift events.

**STRIMZI_OPERATOR_NAMESPACE**

The name of the namespace where the Cluster Operator is running. Do not configure this variable manually. Use the downward API.

```
env:
  - name: STRIMZI_OPERATOR_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

**STRIMZI_OPERATOR_NAMESPACE_LABELS**

Optional. The labels of the namespace where the AMQ Streams Cluster Operator is running. Use namespace labels to configure the namespace selector in network policies. Network policies allow the AMQ Streams Cluster Operator access only to the operands from the namespace with these labels. When not set, the namespace selector in network policies is configured to allow access to the Cluster Operator from any namespace in the OpenShift cluster.

```
env:
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS
    value: label1=value1,label2=value2
```

### STRIMZI_LABELS_EXCLUSION_PATTERN

Optional, default regex pattern is **^app.kubernetes.io/(?!part-of).***. The regex exclusion pattern used to filter labels propagation from the main custom resource to its subresources. The labels exclusion filter is not applied to labels in template sections such as **spec.kafka.template.pod.metadata.labels**.

```
env:
  - name: STRIMZI_LABELS_EXCLUSION_PATTERN
    value: "^key1.*"
```

### STRIMZI_CUSTOM_{COMPONENT_NAME}_LABELS

Optional. One or more custom labels to apply to all the pods created by the **{COMPONENT_NAME}** custom resource. The Cluster Operator labels the pods when the custom resource is created or is next reconciled.
Labels can be applied to the following components:

- **KAFKA**
- **KAFKA_CONNECT**
- **KAFKA_CONNECT_BUILD**
- **ZOOKEEPER**
- **ENTITY_OPERATOR**
- **KAFKA_MIRROR_MAKER2**
- **KAFKA_MIRROR_MAKER**
- **CRUISE_CONTROL**
- **KAFKA_BRIDGE**
- **KAFKA_EXPORTER**

### STRIMZI_CUSTOM_RESOURCE_SELECTOR

Optional. The label selector to filter the custom resources handled by the Cluster Operator. The operator will operate only on those custom resources that have the specified labels set. Resources without these labels will not be seen by the operator. The label selector applies to **Kafka**, **KafkaConnect**, **KafkaBridge**, **KafkaMirrorMaker**, and **KafkaMirrorMaker2** resources. **KafkaRebalance** and **KafkaConnector** resources are operated only when their corresponding Kafka and Kafka Connect clusters have the matching labels.

```
env:
  - name: STRIMZI_CUSTOM_RESOURCE_SELECTOR
    value: label1=value1,label2=value2
```

**STRIMZI_KAFKA_IMAGES**

Required. The mapping from the Kafka version to the corresponding Docker image containing a Kafka broker for that version. The required syntax is whitespace or comma-separated *<version>=<image>* pairs. For example **3.2.3=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.3.0, 3.3.1=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0**. This is used when a **Kafka.spec.kafka.version** property is specified but not the **Kafka.spec.kafka.image** in the **Kafka** resource.

**STRIMZI_DEFAULT_KAFKA_INIT_IMAGE**

Optional, default **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0**. The image name to use as default for the init container if no image is specified as the **kafka-init-image** in the **Kafka** resource. The init container is started before the broker for initial configuration work, such as rack support.

**STRIMZI_KAFKA_CONNECT_IMAGES**

Required. The mapping from the Kafka version to the corresponding Docker image of Kafka Connect for that version. The required syntax is whitespace or comma-separated *<version>=<image>* pairs. For example **3.2.3=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.3.0, 3.3.1=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0**. This is used when a **KafkaConnect.spec.version** property is specified but not the **KafkaConnect.spec.image**.

**STRIMZI_KAFKA_MIRROR_MAKER_IMAGES**

Required. The mapping from the Kafka version to the corresponding Docker image of MirrorMaker for that version. The required syntax is whitespace or comma-separated *<version>=<image>* pairs. For example **3.2.3=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.3.0, 3.3.1=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0**. This is used when a **KafkaMirrorMaker.spec.version** property is specified but not the **KafkaMirrorMaker.spec.image**.

**STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE**

Optional, default **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0**. The image name to use as the default when deploying the Topic Operator if no image is specified as the **Kafka.spec.entityOperator.topicOperator.image** in the **Kafka** resource.

**STRIMZI_DEFAULT_USER_OPERATOR_IMAGE**

Optional, default **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0**. The image name to use as the default when deploying the User Operator if no image is specified as the **Kafka.spec.entityOperator.userOperator.image** in the **Kafka** resource.

**STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE**

Optional, default **registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0**. The image name to use as the default when deploying the sidecar container for the Entity Operator if no image is specified as the **Kafka.spec.entityOperator.tlsSidecar.image** in the **Kafka** resource. The sidecar provides TLS support.

**STRIMZI_IMAGE_PULL_POLICY**

Optional. The **ImagePullPolicy** that is applied to containers in all pods managed by the Cluster Operator. The valid values are **Always**, **IfNotPresent**, and **Never**. If not specified, the OpenShift defaults are used. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

**STRIMZI_IMAGE_PULL_SECRETS**

Optional. A comma-separated list of **Secret** names. The secrets referenced here contain the credentials to the container registries where the container images are pulled from. The secrets are

specified in the **imagePullSecrets** property for all pods created by the Cluster Operator. Changing this list results in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

### STRIMZI_KUBERNETES_VERSION

Optional. Overrides the OpenShift version information detected from the API server.

**Example configuration for OpenShift version override**

```
env:
  - name: STRIMZI_KUBERNETES_VERSION
    value: |
          major=1
          minor=16
          gitVersion=v1.16.2
          gitCommit=c97fe5036ef3df2967d086711e6c0c405941e14b
          gitTreeState=clean
          buildDate=2019-10-15T19:09:08Z
          goVersion=go1.12.10
          compiler=gc
          platform=linux/amd64
```

### KUBERNETES_SERVICE_DNS_DOMAIN

Optional. Overrides the default OpenShift DNS domain name suffix.
By default, services assigned in the OpenShift cluster have a DNS domain name that uses the default suffix **cluster.local**.

For example, for broker *kafka-0*:

> *<cluster-name>*-kafka-0.*<cluster-name>*-kafka-brokers.*<namespace>*.svc.*cluster.local*

The DNS domain name is added to the Kafka broker certificates used for hostname verification.

If you are using a different DNS domain name suffix in your cluster, change the **KUBERNETES_SERVICE_DNS_DOMAIN** environment variable from the default to the one you are using in order to establish a connection with the Kafka brokers.

### STRIMZI_CONNECT_BUILD_TIMEOUT_MS

Optional, default 300000 ms. The timeout for building new Kafka Connect images with additional connectors, in milliseconds. Consider increasing this value when using AMQ Streams to build container images containing many connectors or using a slow container registry.

### STRIMZI_NETWORK_POLICY_GENERATION

Optional, default **true**. Network policy for resources. Network policies allow connections between Kafka components.
Set this environment variable to **false** to disable network policy generation. You might do this, for example, if you want to use custom network policies. Custom network policies allow more control over maintaining the connections between components.

### STRIMZI_DNS_CACHE_TTL

Optional, default **30**. Number of seconds to cache successful name lookups in local DNS resolver. Any negative value means cache forever. Zero means do not cache, which can be useful for avoiding connection errors due to long caching policies being applied.

### STRIMZI_POD_SET_RECONCILIATION_ONLY

Optional, default **false**. When set to **true**, the Cluster Operator reconciles only the **StrimziPodSet** resources and any changes to the other custom resources (**Kafka**, **KafkaConnect**, and so on) are ignored. This mode is useful for ensuring that your pods are recreated if needed, but no other changes happen to the clusters.

**STRIMZI_FEATURE_GATES**

Optional. Enables or disables the features and functionality controlled by feature gates.

**STRIMZI_POD_SECURITY_PROVIDER_CLASS**

Optional. Configuration for the pluggable **PodSecurityProvider** class, which can be used to provide the security context configuration for Pods and containers.

### 7.2.3.1. Leader election environment variables

Use leader election environment variables when running additional Cluster Operator replicas . You might run additional replicas to safeguard against disruption caused by major failure.

**STRIMZI_LEADER_ELECTION_ENABLED**

Optional, disabled (**false**) by default. Enables or disables leader election, which allows additional Cluster Operator replicas to run on standby.

> **NOTE**
>
> Leader election is disabled by default. It is only enabled when applying this environment variable on installation.

**STRIMZI_LEADER_ELECTION_LEASE_NAME**

Required when leader election is enabled. The name of the OpenShift **Lease** resource that is used for the leader election.

**STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE**

Required when leader election is enabled. The namespace where the OpenShift **Lease** resource used for leader election is created. You can use the downward API to configure it to the namespace where the Cluster Operator is deployed.

```
env:
  - name: STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

**STRIMZI_LEADER_ELECTION_IDENTITY**

Required when leader election is enabled. Configures the identity of a given Cluster Operator instance used during the leader election. The identity must be unique for each operator instance. You can use the downward API to configure it to the name of the pod where the Cluster Operator is deployed.

```
env:
  - name: STRIMZI_LEADER_ELECTION_IDENTITY
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
```

**STRIMZI_LEADER_ELECTION_LEASE_DURATION_MS**

Optional, default 15000 ms. Specifies the duration the acquired lease is valid.

**STRIMZI_LEADER_ELECTION_RENEW_DEADLINE_MS**

Optional, default 10000 ms. Specifies the period the leader should try to maintain leadership.

**STRIMZI_LEADER_ELECTION_RETRY_PERIOD_MS**

Optional, default 2000 ms. Specifies the frequency of updates to the lease lock by the leader.

### 7.2.3.2. Restricting Cluster Operator access with network policy

Use the **STRIMZI_OPERATOR_NAMESPACE_LABELS** environment variable to establish network policy for the Cluster Operator using namespace labels.

The Cluster Operator can run in the same namespace as the resources it manages, or in a separate namespace. By default, the **STRIMZI_OPERATOR_NAMESPACE** environment variable is configured to use the downward API to find the namespace the Cluster Operator is running in. If the Cluster Operator is running in the same namespace as the resources, only local access is required and allowed by AMQ Streams.

If the Cluster Operator is running in a separate namespace to the resources it manages, any namespace in the OpenShift cluster is allowed access to the Cluster Operator unless network policy is configured. By adding namespace labels, access to the Cluster Operator is restricted to the namespaces specified.

**Network policy configured for the Cluster Operator deployment**

```
#...
env:
  # ...
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS
    value: label1=value1,label2=value2
  #...
```

### 7.2.3.3. Setting the time interval for periodic reconciliation

Use the **STRIMZI_FULL_RECONCILIATION_INTERVAL_MS** variable to set the time interval for periodic reconciliations.

The Cluster Operator reacts to all notifications about applicable cluster resources received from the OpenShift cluster. If the operator is not running, or if a notification is not received for any reason, resources will get out of sync with the state of the running OpenShift cluster. In order to handle failovers properly, a periodic reconciliation process is executed by the Cluster Operator so that it can compare the state of the resources with the current cluster deployments in order to have a consistent state across all of them.

**Additional resources**

- Downward API

### 7.2.4. Configuring the Cluster Operator with default proxy settings

If you are running a Kafka cluster behind a HTTP proxy, you can still pass data in and out of the cluster. For example, you can run Kafka Connect with connectors that push and pull data from outside the proxy. Or you can use a proxy to connect with an authorization server.

Configure the Cluster Operator deployment to specify the proxy environment variables. The Cluster Operator accepts standard proxy configuration (**HTTP_PROXY**, **HTTPS_PROXY** and **NO_PROXY**) as environment variables. The proxy settings are applied to all AMQ Streams containers.

The format for a proxy address is *http://IP-ADDRESS:PORT-NUMBER*. To set up a proxy with a name and password, the format is *http://USERNAME:PASSWORD@IP-ADDRESS:PORT-NUMBER*.

### Prerequisites

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

### Procedure

1. To add proxy environment variables to the Cluster Operator, update its **Deployment** configuration (**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml**).

   **Example proxy configuration for the Cluster Operator**

   ```
   apiVersion: apps/v1
   kind: Deployment
   spec:
     # ...
     template:
       spec:
         serviceAccountName: strimzi-cluster-operator
         containers:
           # ...
           env:
           # ...
           - name: "HTTP_PROXY"
             value: "http://proxy.com"          1
           - name: "HTTPS_PROXY"
             value: "https://proxy.com"         2
           - name: "NO_PROXY"
             value: "internal.com, other.domain.com"   3
       # ...
   ```

   **1**      Address of the proxy server.

   **2**      Secure address of the proxy server.

   **3**      Addresses for servers that are accessed directly as exceptions to the proxy server. The URLs are comma-separated.

   Alternatively, edit the **Deployment** directly:

   ```
   oc edit deployment strimzi-cluster-operator
   ```

2. If you updated the YAML file instead of editing the **Deployment** directly, apply the changes:

   ```
   oc create -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
   ```

**Additional resources**

- [Host aliases](#)

- [Designating AMQ Streams administrators](#)

## 7.2.5. Running multiple Cluster Operator replicas with leader election

The default Cluster Operator configuration enables leader election. Use leader election to run multiple parallel replicas of the Cluster Operator. One replica is elected as the active leader and operates the deployed resources. The other replicas run in standby mode. When the leader stops or fails, one of the standby replicas is elected as the new leader and starts operating the deployed resources.

By default, AMQ Streams runs with a single Cluster Operator replica that is always the leader replica. When a single Cluster Operator replica stops or fails, OpenShift starts a new replica.

Running the Cluster Operator with multiple replicas is not essential. But it's useful to have replicas on standby in case of large-scale disruptions. For example, suppose multiple worker nodes or an entire availability zone fails. This failure might cause the Cluster Operator pod and many Kafka pods to go down at the same time. If subsequent pod scheduling causes congestion through lack of resources, this can delay operations when running a single Cluster Operator.

### 7.2.5.1. Configuring Cluster Operator replicas

To run additional Cluster Operator replicas in standby mode, you will need to increase the number of replicas and enable leader election. To configure leader election, use the leader election environment variables.

To make the required changes, configure the following Cluster Operator installation files located in **install/cluster-operator/**:

- 060-Deployment-strimzi-cluster-operator.yaml

- 022-ClusterRole-strimzi-cluster-operator-role.yaml

- 022-RoleBinding-strimzi-cluster-operator.yaml

Leader election has its own **ClusterRole** and **RoleBinding** RBAC resources that target the namespace where the Cluster Operator is running, rather than the namespace it is watching.

The default deployment configuration creates a **Lease** resource called **strimzi-cluster-operator** in the same namespace as the Cluster Operator. The Cluster Operator uses leases to manage leader election. The RBAC resources provide the permissions to use the **Lease** resource. If you use a different **Lease** name or namespace, update the **ClusterRole** and **RoleBinding** files accordingly.

**Prerequisites**

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

**Procedure**

Edit the **Deployment** resource that is used to deploy the Cluster Operator, which is defined in the **060-Deployment-strimzi-cluster-operator.yaml** file.

1. Change the **replicas** property from the default (1) to a value that matches the required number of replicas.

### Increasing the number of Cluster Operator replicas

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 3
```

2. Check that the leader election **env** properties are set.
   If they are not set, configure them.

   To enable leader election, **STRIMZI_LEADER_ELECTION_ENABLED** must be set to **true** (default).

   In this example, the name of the lease is changed to **my-strimzi-cluster-operator**.

### Configuring leader election environment variables for the Cluster Operator

```
# ...
spec
  containers:
    - name: strimzi-cluster-operator
      # ...
      env:
        - name: STRIMZI_LEADER_ELECTION_ENABLED
          value: "true"
        - name: STRIMZI_LEADER_ELECTION_LEASE_NAME
          value: "my-strimzi-cluster-operator"
        - name: STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: STRIMZI_LEADER_ELECTION_IDENTITY
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
```

For a description of the available environment variables, see Section 7.2.3.1, "Leader election environment variables".

If you specified a different name or namespace for the **Lease** resource used in leader election, update the RBAC resources.

3. (optional) Edit the **ClusterRole** resource in the **022-ClusterRole-strimzi-cluster-operator-role.yaml** file.
   Update **resourceNames** with the name of the **Lease** resource.

### Updating the ClusterRole references to the lease

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
    name: strimzi-cluster-operator-leader-election
    labels:
      app: strimzi
  rules:
    - apiGroups:
        - coordination.k8s.io
      resourceNames:
        - my-strimzi-cluster-operator
  # ...
```

4. (optional) Edit the **RoleBinding** resource in the **022-RoleBinding-strimzi-cluster-operator.yaml** file.
   Update **subjects.name** and **subjects.namespace** with the name of the **Lease** resource and the namespace where it was created.

   **Updating the RoleBinding references to the lease**

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: RoleBinding
   metadata:
     name: strimzi-cluster-operator-leader-election
     labels:
       app: strimzi
   subjects:
     - kind: ServiceAccount
       name: my-strimzi-cluster-operator
       namespace: myproject
   # ...
   ```

5. Deploy the Cluster Operator:

   ```
   oc create -f install/cluster-operator -n myproject
   ```

6. Check the status of the deployment:

   ```
   oc get deployments -n myproject
   ```

   **Output shows the deployment name and readiness**

   ```
   NAME                    READY  UP-TO-DATE  AVAILABLE
   strimzi-cluster-operator  3/3    3           3
   ```

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

## 7.2.6. Configuring FIPS mode in the Cluster Operator

Federal Information Processing Standards (FIPS) are standards for computer security and interoperability. When running AMQ Streams on a FIPS-enabled OpenShift cluster, the OpenJDK used in AMQ Streams container images will automatically switch to FIPS mode. This prevents AMQ Streams from running on the cluster. When you deploy AMQ Streams to the cluster, you will see errors similar to the following:

Exception in thread "main" io.fabric8.kubernetes.client.KubernetesClientException: An error has occurred.
 ...
Caused by: java.security.KeyStoreException: sun.security.pkcs11.wrapper.PKCS11Exception: CKR_SESSION_READ_ONLY
 ...
Caused by: sun.security.pkcs11.wrapper.PKCS11Exception: CKR_SESSION_READ_ONLY
 ...

If you want to run AMQ Streams on your FIPS-enabled cluster, you can disable the OpenJDK FIPS mode by setting the **FIPS_MODE** environment variable to **disabled** in the deployment configuration for the Cluster Operator. The AMQ Streams deployment won't be FIPS compliant, but the AMQ Streams operators as well as all of its operands will be able to run on the FIPS-enabled OpenShift cluster.

**Procedure**

1. To disable the FIPS mode in the Cluster Operator, update its **Deployment** configuration (**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml**) and add the **FIPS_MODE** environment variable.

   **Example FIPS configuration for the Cluster Operator**

   ```
   apiVersion: apps/v1
   kind: Deployment
   spec:
     # ...
     template:
       spec:
         serviceAccountName: strimzi-cluster-operator
         containers:
           # ...
           env:
           # ...
           - name: "FIPS_MODE"
             value: "disabled" 1
       # ...
   ```

   **1**    Disables the FIPS mode.

   Alternatively, edit the **Deployment** directly:

   ```
   oc edit deployment strimzi-cluster-operator
   ```

2. If you updated the YAML file instead of editing the **Deployment** directly, apply the changes:

   ```
   oc apply -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
   ```

**Additional resources**

- What are Federal Information Processing Standards (FIPS)

# 7.3. USING THE TOPIC OPERATOR

When you create, modify or delete a topic using the **KafkaTopic** resource, the Topic Operator ensures those changes are reflected in the Kafka cluster.

For more information on the **KafkaTopic** resource, see the **KafkaTopic** schema reference.

## Deploying the Topic Operator

You can deploy the Topic Operator using the Cluster Operator or as a standalone operator. You would use a standalone Topic Operator with a Kafka cluster that is not managed by the Cluster Operator.

For deployment instructions, see the following:

- Deploying the Topic Operator using the Cluster Operator (recommended)

- Deploying the standalone Topic Operator

> **IMPORTANT**
>
> To deploy the standalone Topic Operator, you need to set environment variables to connect to a Kafka cluster. These environment variables do not need to be set if you are deploying the Topic Operator using the Cluster Operator as they will be set by the Cluster Operator.

## 7.3.1. Kafka topic resource

The **KafkaTopic** resource is used to configure topics, including the number of partitions and replicas.

The full schema for **KafkaTopic** is described in **KafkaTopic** schema reference.

### 7.3.1.1. Identifying a Kafka cluster for topic handling

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

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
  labels:
    strimzi.io/cluster: my-cluster
```

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

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

### 7.3.1.2. Kafka topic usage recommendations

When working with topics, be consistent. Always operate on either **KafkaTopic** resources or topics directly in OpenShift. Avoid routinely switching between both methods for a given topic.

Use topic names that reflect the nature of the topic, and remember that names cannot be changed later.

If creating a topic in Kafka, use a name that is a valid OpenShift resource name, otherwise the Topic Operator will need to create the corresponding **KafkaTopic** with a name that conforms to the OpenShift rules.

> **NOTE**
>
> For information on the requirements for identifiers and names in OpenShift, refer to Object Names and IDs.

### 7.3.1.3. Kafka topic naming conventions

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

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

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

- The name must not be longer than 249 characters.

- Valid characters for Kafka topics are ASCII alphanumerics, **.**, **_**, and **-**.

- The name cannot be **.** or **..**, though **.** can be used in a name, such as **exampleTopic.** or **.exampleTopic**.

**spec.topicName** must not be changed.

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: topicName-1    1
  # ...
```

**1**    Upper case is invalid in OpenShift.

cannot be changed to:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: name-2
  # ...
```

NOTE

Some Kafka client applications, such as Kafka Streams, can create topics in Kafka programmatically. If those topics have names that are invalid OpenShift resource names, the Topic Operator gives them a valid **metadata.name** based on the Kafka name. Invalid characters are replaced and a hash is appended to the name. For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: mytopic---c55e57fe2546a33f9e603caf57165db4072e827e
spec:
  topicName: myTopic
  # ...
```

## 7.3.2. Topic Operator topic store

The Topic Operator uses Kafka to store topic metadata describing topic configuration as key-value pairs. The *topic store* is based on the Kafka Streams key-value mechanism, which uses Kafka topics to persist the state.

Topic metadata is cached in-memory and accessed locally within the Topic Operator. Updates from operations applied to the local in-memory cache are persisted to a backup topic store on disk. The topic store is continually synchronized with updates from Kafka topics or OpenShift **KafkaTopic** custom resources. Operations are handled rapidly with the topic store set up this way, but should the in-memory cache crash it is automatically repopulated from the persistent storage.

### 7.3.2.1. Internal topic store topics

Internal topics support the handling of topic metadata in the topic store.

**__strimzi_store_topic**

Input topic for storing the topic metadata

**__strimzi-topic-operator-kstreams-topic-store-changelog**

Retains a log of compacted topic store values

> **WARNING**
>
> Do not delete these topics, as they are essential to the running of the Topic Operator.

### 7.3.2.2. Migrating topic metadata from ZooKeeper

In previous releases of AMQ Streams, topic metadata was stored in ZooKeeper. The new process removes this requirement, bringing the metadata into the Kafka cluster, and under the control of the Topic Operator.

When upgrading to AMQ Streams 2.3, the transition to Topic Operator control of the topic store is seamless. Metadata is found and migrated from ZooKeeper, and the old store is deleted.

### 7.3.2.3. Downgrading to an AMQ Streams version that uses ZooKeeper to store topic metadata

If you are reverting back to a version of AMQ Streams earlier than 1.7, which uses ZooKeeper for the storage of topic metadata, you still downgrade your Cluster Operator to the previous version, then downgrade Kafka brokers and client applications to the previous Kafka version as standard.

However, you must also delete the topics that were created for the topic store using a **kafka-admin** command, specifying the bootstrap address of the Kafka cluster. For example:

```
oc run kafka-admin -ti --image=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

The command must correspond to the type of listener and authentication used to access the Kafka cluster.

The Topic Operator will reconstruct the ZooKeeper topic metadata from the state of the topics in Kafka.

### 7.3.2.4. Topic Operator topic replication and scaling

The recommended configuration for topics managed by the Topic Operator is a topic replication factor of 3, and a minimum of 2 in-sync replicas.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10          1
  replicas: 3             2
  config:
    min.insync.replicas: 2   3
  #...
```

**1** The number of partitions for the topic.

**2** The number of replica topic partitions. Currently, this cannot be changed in the **KafkaTopic** resource, but it can be changed using the **kafka-reassign-partitions.sh** tool.

**3** The minimum number of replica partitions that a message must be successfully written to, or an exception is raised.

> **NOTE**
>
> In-sync replicas are used in conjunction with the **acks** configuration for producer applications. The **acks** configuration determines the number of follower partitions a message must be replicated to before the message is acknowledged as successfully received. The Topic Operator runs with **acks=all**, whereby messages must be acknowledged by all in-sync replicas.

When scaling Kafka clusters by adding or removing brokers, replication factor configuration is not changed and replicas are not reassigned automatically. However, you can use the **kafka-reassign-partitions.sh** tool to change the replication factor, and manually reassign replicas to brokers.

Alternatively, though the integration of Cruise Control for AMQ Streams cannot change the replication factor for topics, the optimization proposals it generates for rebalancing Kafka include commands that transfer partition replicas and change partition leadership.

### 7.3.2.5. Handling changes to topics

A fundamental problem that the Topic Operator needs to solve is that there is no single source of truth: both the **KafkaTopic** resource and the Kafka topic can be modified independently of the Topic Operator. Complicating this, the Topic Operator might not always be able to observe changes at each end in real time. For example, when the Topic Operator is down.

To resolve this, the Topic Operator maintains information about each topic in the topic store. When a change happens in the Kafka cluster or OpenShift, it looks at both the state of the other system and the topic store in order to determine what needs to change to keep everything in sync. The same thing happens whenever the Topic Operator starts, and periodically while it is running.

For example, suppose the Topic Operator is not running, and a **KafkaTopic** called *my-topic* is created. When the Topic Operator starts, the topic store does not contain information on my-topic, so it can infer that the **KafkaTopic** was created after it was last running. The Topic Operator creates the topic corresponding to my-topic, and also stores metadata for my-topic in the topic store.

If you update Kafka topic configuration or apply a change through the **KafkaTopic** custom resource, the topic store is updated after the Kafka cluster is reconciled.

The topic store also allows the Topic Operator to manage scenarios where the topic configuration is changed in Kafka topics *and* updated through OpenShift **KafkaTopic** custom resources, as long as the changes are not incompatible. For example, it is possible to make changes to the same topic config key, but to different values. For incompatible changes, the Kafka configuration takes priority, and the **KafkaTopic** is updated accordingly.

> **NOTE**
>
> You can also use the **KafkaTopic** resource to delete topics using a **oc delete -f** *KAFKA-TOPIC-CONFIG-FILE* command. To be able to do this, **delete.topic.enable** must be set to **true** (default) in the **spec.kafka.config** of the Kafka resource.

**Additional resources**

- [Downgrading AMQ Streams](#)

- [Scaling cluster and partition reassignment](#)

- [Cruise Control for cluster rebalancing](#)

### 7.3.3. Configuring Kafka topics

Use the properties of the **KafkaTopic** resource to configure Kafka topics.

You can use **oc apply** to create or modify topics, and **oc delete** to delete existing topics.

For example:

- **oc apply -f** *<topic_config_file>*

- **oc delete KafkaTopic** *<topic_name>*

This procedure shows how to create a topic with 10 partitions and 2 replicas.

## Before you start

It is important that you consider the following before making your changes:

- Kafka does not support decreasing the number of partitions.

- Increasing **spec.partitions** for topics with keys will change how records are partitioned, which can be particularly problematic when the topic uses s*emantic partitioning*.

- AMQ Streams does not support making the following changes through the **KafkaTopic** resource:

  - Using **spec.replicas** to change the number of replicas that were initially specified

  - Changing topic names using **spec.topicName**

## Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.

- A running Topic Operator (typically deployed with the Entity Operator).

- For deleting a topic, **delete.topic.enable=true** (default) in the  **spec.kafka.config** of the **Kafka** resource.

## Procedure

1. Configure the **KafkaTopic** resource.

   ### Example Kafka topic configuration

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaTopic
   metadata:
     name: orders
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     partitions: 10
     replicas: 2
   ```

   ### TIP

   When modifying a topic, you can get the current version of the resource using **oc get kafkatopic orders -o yaml**.

2. Create the **KafkaTopic** resource in OpenShift.

```
oc apply -f <topic_config_file>
```

3. Wait for the ready status of the topic to change to **True**:

```
oc get kafkatopics -o wide -w -n <namespace>
```

**Kafka topic status**

```
NAME        CLUSTER     PARTITIONS  REPLICATION FACTOR READY
my-topic-1  my-cluster  10          3                  True
my-topic-2  my-cluster  10          3
my-topic-3  my-cluster  10          3                  True
```

Topic creation is successful when the **READY** output shows **True**.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or from the Topic Operator logs.
Messages provide details on the reason for the current status.

```
oc get kafkatopics my-topic-2 -o yaml
```

**Details on a topic with a  NotReady status**

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-13T10:14:43.351550Z"
    message: Number of partitions cannot be decreased
    reason: PartitionDecreaseException
    status: "True"
    type: NotReady
```

In this example, the reason the topic is not ready is because the original number of partitions was reduced in the **KafkaTopic** configuration. Kafka does not support this.

After resetting the topic configuration, the status shows the topic is ready.

```
oc get kafkatopics my-topic-2 -o wide -w -n <namespace>
```

**Status update of the topic**

```
NAME        CLUSTER     PARTITIONS  REPLICATION FACTOR READY
my-topic-2  my-cluster  10          3                  True
```

Fetching the details shows no messages

```
oc get kafkatopics my-topic-2 -o yaml
```

**Details on a topic with a  READY status**

```
# ...
```

```
status:
  conditions:
  - lastTransitionTime: '2022-06-13T10:15:03.761084Z'
    status: 'True'
    type: Ready
```

### 7.3.4. Configuring the Topic Operator with resource requests and limits

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

#### Prerequisites

- The Cluster Operator is running.

#### Procedure

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

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

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

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   spec:
     # Kafka and ZooKeeper sections...
     entityOperator:
       topicOperator:
         resources:
           requests:
             cpu: "1"
             memory: 500Mi
           limits:
             cpu: "1"
             memory: 500Mi
   ```

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

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

## 7.4. USING THE USER OPERATOR

When you create, modify or delete a user using the **KafkaUser** resource, the User Operator ensures those changes are reflected in the Kafka cluster.

For more information on the **KafkaUser** resource, see the **KafkaUser** schema reference.

### Deploying the User Operator

You can deploy the User Operator using the Cluster Operator or as a standalone operator. You would use a standalone User Operator with a Kafka cluster that is not managed by the Cluster Operator.

For deployment instructions, see the following:

- Deploying the User Operator using the Cluster Operator (recommended)

- Deploying the standalone User Operator
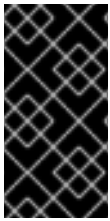
> **IMPORTANT**
>
> To deploy the standalone User Operator, you need to set environment variables to connect to a Kafka cluster. These environment variables do not need to be set if you are deploying the User Operator using the Cluster Operator as they will be set by the Cluster Operator.

## 7.4.1. Configuring Kafka users

Use the properties of the **KafkaUser** resource to configure Kafka users.

You can use **oc apply** to create or modify users, and **oc delete** to delete existing users.

For example:

- **oc apply -f** *<user_config_file>*

- **oc delete KafkaUser** *<user_name>*

Users represent Kafka clients. When you configure Kafka users, you enable the user authentication and authorization mechanisms required by clients to access Kafka. The mechanism used must match the equivalent **Kafka** configuration. For more information on using **Kafka** and **KafkaUser** resources to secure access to Kafka brokers, see Securing access to Kafka brokers .

**Prerequisites**

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.

- A running User Operator (typically deployed with the Entity Operator).

**Procedure**

1. Configure the **KafkaUser** resource.
   This example specifies mTLS authentication and simple authorization using ACLs.

   **Example Kafka user configuration**

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

```
acls:
  # Example consumer Acls for topic my-topic using consumer group my-group
  - resource:
      type: topic
      name: my-topic
      patternType: literal
    operations:
      - Describe
      - Read
    host: "*"
  - resource:
      type: group
      name: my-group
      patternType: literal
    operations:
      - Read
    host: "*"
  # Example Producer Acls for topic my-topic
  - resource:
      type: topic
      name: my-topic
      patternType: literal
    operations:
      - Create
      - Describe
      - Write
    host: "*"
```

2. Create the **KafkaUser** resource in OpenShift.

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

3. Wait for the ready status of the user to change to **True**:

   ```
   oc get kafkausers -o wide -w -n <namespace>
   ```

   **Kafka user status**

   ```
   NAME       CLUSTER     AUTHENTICATION  AUTHORIZATION READY
   my-user-1  my-cluster  tls             simple        True
   my-user-2  my-cluster  tls             simple
   my-user-3  my-cluster  tls             simple        True
   ```

   User creation is successful when the **READY** output shows **True**.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or User Operator logs.
   Messages provide details on the reason for the current status.

   ```
   oc get kafkausers my-user-2 -o yaml
   ```

   **Details on a user with a NotReady status**

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:07:37.238065Z"
    message: Simple authorization ACL rules are configured but not supported in the
      Kafka cluster configuration.
    reason: InvalidResourceException
    status: "True"
    type: NotReady
```

In this example, the reason the user is not ready is because simple authorization is not enabled in the **Kafka** configuration.

### Kafka configuration for simple authorization

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    authorization:
      type: simple
```

After updating the Kafka configuration, the status shows the user is ready.

```
oc get kafkausers my-user-2 -o wide -w -n <namespace>
```

### Status update of the user

```
NAME       CLUSTER     AUTHENTICATION  AUTHORIZATION READY
my-user-2  my-cluster  tls             simple        True
```

Fetching the details shows no messages.

```
oc get kafkausers my-user-2 -o yaml
```

### Details on a user with a **READY** status

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:33:40.166846Z"
    status: "True"
    type: Ready
```

## 7.4.2. Configuring the User Operator with resource requests and limits

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

**Prerequisites**

- The Cluster Operator is running.

**Procedure**

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

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

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

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   spec:
     # Kafka and ZooKeeper sections...
     entityOperator:
       userOperator:
         resources:
           requests:
             cpu: "1"
             memory: 500Mi
           limits:
             cpu: "1"
             memory: 500Mi
   ```

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

# 7.5. CONFIGURING FEATURE GATES

AMQ Streams operators support *feature gates* to enable or disable certain features and functionality. Enabling a feature gate changes the behavior of the relevant operator and introduces the feature to your AMQ Streams deployment.

Feature gates have a default state of either *enabled* or *disabled*.

To modify a feature gate's default state, use the **STRIMZI_FEATURE_GATES** environment variable in the operator's configuration. You can modify multiple feature gates using this single environment variable. Specify a comma-separated list of feature gate names and prefixes. A **+** prefix enables the feature gate and a **-** prefix disables it.

**Example feature gate configuration that enables FeatureGate1 and disables FeatureGate2**

```
env:
  - name: STRIMZI_FEATURE_GATES
    value: +FeatureGate1,-FeatureGate2
```

## 7.5.1. ControlPlaneListener feature gate

The **ControlPlaneListener** feature gate has moved to GA, which means it is now permanently enabled and cannot be disabled. With **ControlPlaneListener** enabled, the connections between the Kafka controller and brokers use an internal *control plane listener* on port 9090. Replication of data between

brokers, as well as internal connections from AMQ Streams operators, Cruise Control, or the Kafka Exporter use the *replication listener* on port 9091.

> **IMPORTANT**
>
> With the **ControlPlaneListener** feature gate permanently enabled, it is no longer possible to upgrade or downgrade directly between AMQ Streams 1.7 and earlier and AMQ Streams 2.3 and newer. You have to upgrade or downgrade through one of the AMQ Streams versions in between.

### 7.5.2. ServiceAccountPatching feature gate

The **ServiceAccountPatching** feature gate has moved to GA, which means it is now permanently enabled and cannot be disabled. With **ServiceAccountPatching** enabled, the Cluster Operator always reconciles service accounts and updates them when needed. For example, when you change service account labels or annotations using the **template** property of a custom resource, the operator automatically updates them on the existing service account resources.

### 7.5.3. UseStrimziPodSets feature gate

The **UseStrimziPodSets** feature gate has a default state of *enabled*.

The **UseStrimziPodSets** feature gate introduces a resource for managing pods called **StrimziPodSet**. When the feature gate is enabled, this resource is used instead of the StatefulSets. AMQ Streams handles the creation and management of pods instead of OpenShift. Using StrimziPodSets instead of StatefulSets provides more control over the functionality.

When this feature gate is disabled, AMQ Streams relies on StatefulSets to create and manage pods for the ZooKeeper and Kafka clusters. AMQ Streams creates the StatefulSet and OpenShift creates the pods according to the StatefulSet definition. When a pod is deleted, OpenShift is responsible for recreating it. The use of StatefulSets has the following limitations:

- Pods are always created or removed based on their index numbers

- All pods in the StatefulSet need to have a similar configuration

- Changing storage configuration for the Pods in the StatefulSet is complicated

**Disabling the UseStrimziPodSets feature gate**

To disable the **UseStrimziPodSets** feature gate, specify **-UseStrimziPodSets** in the **STRIMZI_FEATURE_GATES** environment variable in the Cluster Operator configuration.

> **IMPORTANT**
>
> The **UseStrimziPodSets** feature gate must be disabled when downgrading to AMQ Streams 2.0 and earlier versions.

### 7.5.4. (Preview) UseKRaft feature gate

The **UseKRaft** feature gate has a default state of *disabled*.

The **UseKRaft** feature gate deploys the Kafka cluster in the KRaft (Kafka Raft metadata) mode without ZooKeeper. This feature gate is currently intended only for development and testing.
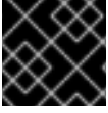
> **IMPORTANT**
>
> **The KRaft mode is not ready for production in Apache Kafka or in AMQ Streams.**

When the **UseKRaft** feature gate is enabled, the Kafka cluster is deployed without ZooKeeper.  **The .spec.zookeeper properties in the Kafka custom resource will be ignored, but still need to be present.** The **UseKRaft** feature gate provides an API that configures Kafka cluster nodes and their roles. The API is still in development and is expected to change before the KRaft mode is production-ready.

Currently, the KRaft mode in AMQ Streams has the following major limitations:

- Moving from Kafka clusters with ZooKeeper to KRaft clusters or the other way around is not supported.

- Upgrades and downgrades of Apache Kafka versions or the AMQ Streams operator are not supported. Users might need to delete the cluster, upgrade the operator and deploy a new Kafka cluster.

- The Topic Operator is not supported. The **spec.entityOperator.topicOperator** property **must be removed** from the **Kafka** custom resource.

- SCRAM-SHA-512 authentication is not supported.

- JBOD storage is not supported. The **type: jbod** storage can be used, but the JBOD array can contain only one disk.

- Liveness and readiness probes are disabled.

- All Kafka nodes have both the **controller** and **broker** KRaft roles. Kafka clusters with separate **controller** and **broker** nodes are not supported.

### Enabling the UseStrimziPodSets feature gate

To enable the **UseKRaft** feature gate, specify **+UseKRaft** in the **STRIMZI_FEATURE_GATES** environment variable in the Cluster Operator configuration.
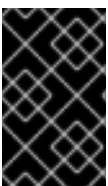
> **IMPORTANT**
>
> The **UseKRaft** feature gate depends on the **UseStrimziPodSets** feature gate. When enabling the **UseKRaft** feature gate, make sure that the **USeStrimziPodSets** feature gate is enabled as well.

## 7.5.5. Feature gate releases

Feature gates have three stages of maturity:

- Alpha — typically disabled by default

- Beta — typically enabled by default

- General Availability (GA) — typically always enabled

Alpha stage features might be experimental or unstable, subject to change, or not sufficiently tested for production use. Beta stage features are well tested and their functionality is not likely to change. GA stage features are stable and should not change in the future. Alpha and beta stage features are

removed if they do not prove to be useful.

- The **ControlPlaneListener** feature gate moved to GA in AMQ Streams 2.3. It is now permanently enabled and cannot be disabled.

- The **ServiceAccountPatching** feature gate moved to GA in AMQ Streams 2.3. It is now permanently enabled and cannot be disabled.

- The **UseStrimziPodSets** feature gate moved to beta stage in AMQ Streams 2.3.

- The **UseKRaft** feature gate is available for development only and does not currently have a planned release for moving to the beta phase.

> **NOTE**
>
> Feature gates might be removed when they reach GA. This means that the feature was incorporated into the AMQ Streams core features and can no longer be disabled.

Table 7.5. Feature gates and the AMQ Streams versions when they moved to alpha, beta, or GA

| Feature gate | Alpha | Beta | GA |
|---|---|---|---|
| **ControlPlaneListener** | 1.8 | 2.0 | 2.3 |
| **ServiceAccountPatching** | 1.8 | 2.0 | 2.3 |
| **UseStrimziPodSets** | 2.1 | 2.3 | - |
| **UseKRaft** | 2.2 | - | - |

If a feature gate is enabled, you may need to disable it before upgrading or downgrading from a specific AMQ Streams version. The following table shows which feature gates you need to disable when upgrading or downgrading AMQ Streams versions.

Table 7.6. Feature gates to disable when upgrading or downgrading AMQ Streams

| Disable Feature gate | Upgrading from AMQ Streams version | Downgrading to AMQ Streams version |
|---|---|---|
| **ControlPlaneListener** | 1.7 and earlier | 1.7 and earlier |
| **UseStrimziPodSets** | - | 2.0 and earlier |

## 7.6. 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 8. CRUISE CONTROL FOR CLUSTER REBALANCING

Cruise Control is an open source system that supports the following Kafka operations:

- Monitoring cluster workload

- Rebalancing a cluster based on predefined constraints

The operations help with running a more balanced Kafka cluster that uses broker pods more efficiently.

A typical cluster can become unevenly loaded over time. Partitions that handle large amounts of message traffic might not be evenly distributed across the available brokers. To rebalance the cluster, administrators must monitor the load on brokers and manually reassign busy partitions to brokers with spare capacity.

Cruise Control automates the cluster rebalancing process. It constructs a *workload model* of resource utilization for the cluster—based on CPU, disk, and network load—and generates optimization proposals (that you can approve or reject) for more balanced partition assignments. A set of configurable optimization goals is used to calculate these proposals.

You can generate optimization proposals in specific modes. The default **full** mode rebalances partitions across all brokers. You can also use the **add-brokers** and **remove-brokers** modes to accommodate changes when scaling a cluster up or down.

When you approve an optimization proposal, Cruise Control applies it to your Kafka cluster. You configure and generate optimization proposals using a **KafkaRebalance** resource. You can configure the resource using an annotation so that optimization proposals are approved automatically or manually.

> **NOTE**
>
> AMQ Streams provides example configuration files for Cruise Control .

## 8.1. CRUISE CONTROL COMPONENTS AND FEATURES

Cruise Control consists of four main components—the Load Monitor, the Analyzer, the Anomaly Detector, and the Executor—and a REST API for client interactions. AMQ Streams utilizes the REST API to support the following Cruise Control features:

- Generating optimization proposals from optimization goals.

- Rebalancing a Kafka cluster based on an optimization proposal.

**Optimization goals**

An optimization goal describes a specific objective to achieve from a rebalance. For example, a goal might be to distribute topic replicas across brokers more evenly. You can change what goals to include through configuration. A goal is defined as a hard goal or soft goal. You can add hard goals through Cruise Control deployment configuration. You also have main, default, and user-provided goals that fit into each of these categories.

- **Hard goals** are preset and must be satisfied for an optimization proposal to be successful.

- **Soft goals** do not need to be satisfied for an optimization proposal to be successful. They can be set aside if it means that all hard goals are met.

- **Main goals** are inherited from Cruise Control. Some are preset as hard goals. Main goals are used in optimization proposals by default.

- **Default goals** are the same as the main goals by default. You can specify your own set of default goals.

- **User-provided goals** are a subset of default goals that are configured for generating a specific optimization proposal.

**Optimization proposals**

Optimization proposals comprise the goals you want to achieve from a rebalance. You generate an optimization proposal to create a summary of proposed changes and the results that are possible with the rebalance. The goals are assessed in a specific order of priority. You can then choose to approve or reject the proposal. You can reject the proposal to run it again with an adjusted set of goals.
You can generate an optimization proposal in one of three modes.

- **full** is the default mode and runs a full rebalance.

- **add-brokers** is the mode you use after adding brokers when scaling up a Kafka cluster.

- **remove-brokers** is the mode you use before removing brokers when scaling down a Kafka cluster.

Other Cruise Control features are not currently supported, including self healing, notifications, write-your-own goals, and changing the topic replication factor.

**Additional resources**

- [Cruise Control documentation](#)

## 8.2. OPTIMIZATION GOALS OVERVIEW

Optimization goals are constraints on workload redistribution and resource utilization across a Kafka cluster. To rebalance a Kafka cluster, Cruise Control uses optimization goals to generate optimization proposals, which you can approve or reject.

### 8.2.1. Goals order of priority

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

1. Rack-awareness

2. Minimum number of leader replicas per broker for a set of topics

3. Replica capacity

4. Capacity goals

   - Disk capacity

   - Network inbound capacity

   - Network outbound capacity

   - CPU capacity

5. Replica distribution

6. Potential network output

7. Resource distribution goals

   - Disk utilization distribution

   - Network inbound utilization distribution

   - Network outbound utilization distribution

   - CPU utilization distribution

8. Leader bytes-in rate distribution

9. Topic replica distribution

10. Leader replica distribution

11. Preferred leader election

12. Intra-broker disk capacity

13. Intra-broker disk usage distribution

For more information on each optimization goal, see Goals in the Cruise Control Wiki.

> **NOTE**
>
> "Write your own" goals and Kafka assigner goals are not yet supported.

## 8.2.2. Goals configuration in AMQ Streams custom resources

You configure optimization goals in **Kafka** and **KafkaRebalance** custom resources. Cruise Control has configurations for hard optimization goals that must be satisfied, as well as main, default, and user-provided optimization goals.

You can specify optimization goals in the following configuration:

- Main goals — **Kafka.spec.cruiseControl.config.goals**

- Hard goals — **Kafka.spec.cruiseControl.config.hard.goals**

- Default goals — **Kafka.spec.cruiseControl.config.default.goals**

- User-provided goals — **KafkaRebalance.spec.goals**
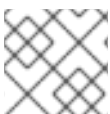
> **NOTE**
>
> Resource distribution goals are subject to capacity limits on broker resources.

## 8.2.3. Hard and soft optimization goals

Hard goals are goals that *must* be satisfied in optimization proposals. Goals that are not configured as hard goals are known as *soft goals*. You can think of soft goals as *best effort* goals: they do *not* need to

be satisfied in optimization proposals, but are included in optimization calculations. An optimization proposal that violates one or more soft goals, but satisfies all hard goals, is valid.

Cruise Control will calculate optimization proposals that satisfy all the hard goals and as many soft goals as possible (in their priority order). An optimization proposal that does *not* satisfy all the hard goals is rejected by Cruise Control and not sent to the user for approval.

> **NOTE**
>
> For example, you might have a soft goal to distribute a topic's replicas evenly across the cluster (the topic replica distribution goal). Cruise Control will ignore this goal if doing so enables all the configured hard goals to be met.

In Cruise Control, the following main optimization goals are preset as hard goals:

> RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal

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

- To inherit the preset hard goals from Cruise Control, do not specify the **hard.goals** property in **Kafka.spec.cruiseControl.config**

- To change the preset hard goals, specify the desired goals in the **hard.goals** property, using their fully-qualified domain names.

Example **Kafka** configuration for hard optimization goals

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      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.

## 8.2.4. Main optimization goals

The *main optimization goals* are available to all users. Goals that are not listed in the main optimization goals are not available for use in Cruise Control operations.

Unless you change the Cruise Control deployment configuration, AMQ Streams will inherit the following main optimization goals from Cruise Control, in descending priority order:

> RackAwareGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal

Some of these goals are preset as hard goals.

To reduce complexity, we recommend that you use the inherited main optimization goals, unless you need to *completely* exclude one or more goals from use in **KafkaRebalance** resources. The priority order of the main optimization goals can be modified, if desired, in the configuration for default optimization goals.

You configure main optimization goals, if necessary, in the Cruise Control deployment configuration: **Kafka.spec.cruiseControl.config.goals**

- To accept the inherited main optimization goals, do not specify the **goals** property in **Kafka.spec.cruiseControl.config**.

- If you need to modify the inherited main optimization goals, specify a list of goals, in descending priority order, in the **goals** configuration option.
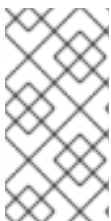
> **NOTE**
>
> If you change the inherited main optimization goals, you must ensure that the hard goals, if configured in the **hard.goals** property in **Kafka.spec.cruiseControl.config**, are a subset of the main optimization goals that you configured. Otherwise, errors will occur when generating optimization proposals.

## 8.2.5. Default optimization goals

Cruise Control uses the *default optimization goals* to generate the *cached optimization proposal*. For more information about the cached optimization proposal, see Section 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 main optimization goals are used as the default optimization goals. In this case, the cached optimization proposal is generated using the main optimization goals.

- To use the main optimization goals as the default goals, do not specify the **default.goals** property in **Kafka.spec.cruiseControl.config**.

- To modify the default optimization goals, edit the **default.goals** property in **Kafka.spec.cruiseControl.config**. You must use a subset of the main optimization goals.

**Example Kafka configuration for default optimization goals**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      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 main optimization goals.

## 8.2.6. User-provided optimization goals

*User-provided optimization goals* narrow down the configured default goals for a particular optimization proposal. You can set them, as required, in **spec.goals** in a **KafkaRebalance** custom resource:

> KafkaRebalance.spec.goals

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

User-provided optimization goals must:

- Include all configured hard goals, or an error occurs

- Be a subset of the main optimization goals

To ignore the configured hard goals when generating an optimization proposal, add the **skipHardGoalCheck: true** property to the **KafkaRebalance** custom resource. See Section 8.6, "Generating optimization proposals".

**Additional resources**

- [Configuring and deploying Cruise Control with Kafka](#)

- [Configurations](#) in the Cruise Control Wiki.

# 8.3. OPTIMIZATION PROPOSALS OVERVIEW

Configure a **KafkaRebalance** resource to generate optimization proposals and apply the suggested changes. An *optimization proposal* is a summary of proposed changes that would produce a more balanced Kafka cluster, with partition workloads distributed more evenly among the brokers.

Each optimization proposal is based on the set of [optimization goals](#) that was used to generate it, subject to any configured [capacity limits on broker resources](#).

All optimization proposals are *estimates* of the impact of a proposed rebalance. You can approve or reject a proposal. You cannot approve a cluster rebalance without first generating the optimization proposal.

You can run optimization proposals in one of the following rebalancing modes:

- **full**

- **add-brokers**

- **remove-brokers**

## 8.3.1. Rebalancing modes

You specify a rebalancing mode using the **spec.mode** property of the **KafkaRebalance** custom resource.

**full**

> The **full** mode runs a full rebalance by moving replicas across all the brokers in the cluster. This is the default mode if the **spec.mode** property is not defined in the **KafkaRebalance** custom resource.

**add-brokers**

> The **add-brokers** mode is used after scaling up a Kafka cluster by adding one or more brokers. Normally, after scaling up a Kafka cluster, new brokers are used to host only the partitions of newly created topics. If no new topics are created, the newly added brokers are not used and the existing brokers remain under the same load. By using the **add-brokers** mode immediately after adding brokers to the cluster, the rebalancing operation moves replicas from existing brokers to the newly added brokers. You specify the new brokers as a list using the **spec.brokers** property of the **KafkaRebalance** custom resource.

**remove-brokers**

> The **remove-brokers** mode is used before scaling down a Kafka cluster by removing one or more brokers. If you scale down a Kafka cluster, brokers are shut down even if they host replicas. This can lead to under-replicated partitions and possibly result in some partitions being under their minimum ISR (in-sync replicas). To avoid this potential problem, the **remove-brokers** mode moves replicas off the brokers that are going to be removed. When these brokers are not hosting replicas anymore, you can safely run the scaling down operation. You specify the brokers you're removing as a list in the **spec.brokers** property in the **KafkaRebalance** custom resource.

In general, use the **full** rebalance mode to rebalance a Kafka cluster by spreading the load across brokers. Use the **add-brokers** and **remove-brokers** modes only if you want to scale your cluster up or down and rebalance the replicas accordingly.

The procedure to run a rebalance is actually the same across the three different modes. The only difference is with specifying a mode through the **spec.mode** property and, if needed, listing brokers that have been added or will be removed through the **spec.brokers** property.

## 8.3.2. The results of an optimization proposal

When an optimization proposal is generated, a summary and broker load is returned.

Summary

> The summary is contained in the **KafkaRebalance** resource. The summary provides an overview of the proposed cluster rebalance and indicates the scale of the changes involved. A summary of a successfully generated optimization proposal is contained in the **Status.OptimizationResult** property of the **KafkaRebalance** resource. The information provided is a summary of the full optimization proposal.

Broker load

> The broker load is stored in a ConfigMap that contains data as a JSON string. The broker load shows before and after values for the proposed rebalance, so you can see the impact on each of the brokers in the cluster.

## 8.3.3. Manually approving or rejecting an optimization proposal

An optimization proposal summary shows the proposed scope of changes.

You can use the name of the **KafkaRebalance** resource to return a summary from the command line.

Returning an optimization proposal summary

```
oc describe kafkarebalance <kafka_rebalance_resource_name> -n <namespace>
```

You can also use the **jq** command line JSON parser tool.

Returning an optimization proposal summary using jq

```
oc get kafkarebalance -o json | jq <jq_query>.
```

Use the summary to decide whether to approve or reject an optimization proposal.

Approving an optimization proposal

> You approve the optimization proposal by setting the **strimzi.io/rebalance** annotation of the **KafkaRebalance** resource to **approve**. Cruise Control applies the proposal to the Kafka cluster and starts a cluster rebalance operation.

Rejecting an optimization proposal

> If you choose not to approve an optimization proposal, you can change the optimization goals or update any of the rebalance performance tuning options , and then generate another proposal. You can use the **strimzi.io/refresh** annotation to generate a new optimization proposal for a **KafkaRebalance** resource.

Use optimization proposals to assess the movements required for a rebalance. For example, a summary describes inter-broker and intra-broker movements. Inter-broker rebalancing moves data between

separate brokers. Intra-broker rebalancing moves data between disks on the same broker when you are using a JBOD storage configuration. Such information can be useful even if you don't go ahead and approve the proposal.

You might reject an optimization proposal, or delay its approval, because of the additional load on a Kafka cluster when rebalancing.

In the following example, the proposal suggests the rebalancing of data between separate brokers. The rebalance involves the movement of 55 partition replicas, totaling 12MB of data, across the brokers. Though the inter-broker movement of partition replicas has a high impact on performance, the total amount of data is not large. If the total data was much larger, you could reject the proposal, or time when to approve the rebalance to limit the impact on the performance of the Kafka cluster.

Rebalance performance tuning options can help reduce the impact of data movement. If you can extend the rebalance period, you can divide the rebalance into smaller batches. Fewer data movements at a single time reduces the load on the cluster.

**Example optimization proposal summary**

```
Name:         my-rebalance
Namespace:    myproject
Labels:       strimzi.io/cluster=my-cluster
Annotations:  API Version:  kafka.strimzi.io/v1alpha1
Kind:         KafkaRebalance
Metadata:
 # ...
Status:
  Conditions:
    Last Transition Time:  2022-04-05T14:36:11.900Z
    Status:                ProposalReady
    Type:                  State
  Observed Generation:     1
  Optimization Result:
    Data To Move MB:  0
    Excluded Brokers For Leadership:
    Excluded Brokers For Replica Move:
    Excluded Topics:
    Intra Broker Data To Move MB:        12
    Monitored Partitions Percentage:     100
    Num Intra Broker Replica Movements:  0
    Num Leader Movements:                24
    Num Replica Movements:               55
    On Demand Balancedness Score After:   82.91290759174306
    On Demand Balancedness Score Before:  78.01176356230222
    Recent Windows:                5
  Session Id:                      a4f833bd-2055-4213-bfdd-ad21f95bf184
```

The proposal will also move 24 partition leaders to different brokers. This requires a change to the ZooKeeper configuration, which has a low impact on performance.

The balancedness scores are measurements of the overall balance of the Kafka cluster before and after the optimization proposal is approved. A balancedness score is based on optimization goals. If all goals are satisfied, the score is 100. The score is reduced for each goal that will not be met. Compare the balancedness scores to see whether the Kafka cluster is less balanced than it could be following a rebalance.

## 8.3.4. Automatically approving an optimization proposal

To save time, you can automate the process of approving optimization proposals. With automation, when you generate an optimization proposal it goes straight into a cluster rebalance.

To enable the optimization proposal auto-approval mechanism, create the **KafkaRebalance** resource with the **strimzi.io/rebalance-auto-approval** annotation set to **true**. If the annotation is not set or set to **false**, the optimization proposal requires manual approval.

### Example rebalance request with auto-approval mechanism enabled

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: true
spec:
  mode: # any mode
  # ...
```

You can still check the status when automatically approving an optimization proposal. The status of the **KafkaRebalance** resource moves to **Ready** when the rebalance is complete.

## 8.3.5. Optimization proposal summary properties

The following table explains the properties contained in the optimization proposal's summary section.

Table 8.1. Properties contained in an optimization proposal summary

| JSON property | Description |
|---|---|
| **numIntraBrokerReplicaMovements** | The total number of partition replicas that will be transferred between the disks of the cluster's brokers.<br><br>**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.<br><br>**Performance impact during rebalance operation**: Relatively high. |

| JSON property | Description |
| --- | --- |
| **onDemandBalancednessScore Before, onDemandBalancednessScore After** | A measurement of the overall *balancedness* of a Kafka Cluster, before and after the optimization proposal was generated.<br><br>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.<br><br>The **Before** score is based on the current configuration of the Kafka cluster. The **After** score is based on the generated optimization proposal. |
| **intraBrokerDataToMoveMB** | The sum of the size of each partition replica that will be moved between disks on the same broker (see also **numIntraBrokerReplicaMovements**).<br><br>**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**).<br><br>**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.<br><br>**Performance impact during rebalance operation:** Relatively low. |
| **excludedBrokersForReplicaMove** | Not yet supported. An empty list is returned. |

## 8.3.6. Broker load properties

The broker load is stored in a ConfigMap (with the same name as the KafkaRebalance custom resource) as a JSON formatted string. This JSON string consists of a JSON object with keys for each broker IDs linking to a number of metrics for each broker. Each metric consist of three values. The first is the metric value before the optimization proposal is applied, the second is the expected value of the metric after the proposal is applied, and the third is the difference between the first two values (after minus before).

> **NOTE**
>
> The ConfigMap appears when the KafkaRebalance resource is in the **ProposalReady** state and remains after the rebalance is complete.

You can use the name of the ConfigMap to view its data from the command line.

**Returning ConfigMap data**

```
oc describe configmaps <my_rebalance_configmap_name> -n <namespace>
```

You can also use the **jq** command line JSON parser tool to extract the JSON string from the ConfigMap.

**Extracting the JSON string from the ConfigMap using jq**

```
oc get configmaps <my_rebalance_configmap_name> -o json | jq '.["data"]
["brokerLoad.json"]|fromjson|.'
```

The following table explains the properties contained in the optimization proposal's broker load ConfigMap:

| JSON property | Description |
| --- | --- |
| **leaders** | The number of replicas on this broker that are partition leaders. |
| **replicas** | The number of replicas on this broker. |
| **cpuPercentage** | The CPU utilization as a percentage of the defined capacity. |
| **diskUsedPercentage** | The disk utilization as a percentage of the defined capacity. |
| **diskUsedMB** | The absolute disk usage in MB. |
| **networkOutRate** | The total network output rate for the broker. |
| **leaderNetworkInRate** | The network input rate for all partition leader replicas on this broker. |
| **followerNetworkInRate** | The network input rate for all follower replicas on this broker. |
| **potentialMaxNetworkOutRate** | The hypothetical maximum network output rate that would be realized if this broker became the leader of all the replicas it currently hosts. |

### 8.3.7. Cached optimization proposal

Cruise Control maintains a *cached optimization proposal* based on the configured default optimization goals. Generated from the workload model, the cached optimization proposal is updated every 15 minutes to reflect the current state of the Kafka cluster. If you generate an optimization proposal using the default optimization goals, Cruise Control returns the most recent cached proposal.

To change the cached optimization proposal refresh interval, edit the **proposal.expiration.ms** setting in the Cruise Control deployment configuration. Consider a shorter interval for fast changing clusters, although this increases the load on the Cruise Control server.

**Additional resources**

- Section 8.2, "Optimization goals overview"

- Section 8.6, "Generating optimization proposals"

- Section 8.7, "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.

### 8.4.1. Partition reassignment commands

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

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

- Partition movement: Involves transferring the partition replica and its data to a new location. Partition movements can take one of two forms:

    - Inter-broker movement: The partition replica is moved to a log directory on a different broker.

    - Intra-broker movement: The partition replica is moved to a different log directory on the same broker.

- Leadership movement: This involves switching the leader of the partition's replicas.

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

### 8.4.2. Replica movement strategies

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

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

- **PrioritizeSmallReplicaMovementStrategy**: Order reassignments in order of ascending size.

- **PrioritizeLargeReplicaMovementStrategy**: Order reassignments in order of descending size.

- **PostponeUrpReplicaMovementStrategy**: Prioritize reassignments for replicas of partitions which have no out-of-sync replicas.

- **PrioritizeMinIsrWithOfflineReplicasStrategy**: Prioritize reassignments with (At/Under)MinISR partitions with offline replicas. This strategy will only work if **cruiseControl.config.concurrency.adjuster.min.isr.check.enabled** is set to **true** in the **Kafka** custom resource's spec.

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

### 8.4.3. Intra-broker disk balancing

Moving a large amount of data between disks on the same broker has less impact than between separate brokers. If you are running a Kafka deployment that uses JBOD storage with multiple disks on the same broker, Cruise Control can balance partitions between the disks.

> **NOTE**
>
> If you are using JBOD storage with a single disk, intra-broker disk balancing will result in a proposal with 0 partition movements since there are no disks to balance between.

To perform an intra-broker disk balance, set **rebalanceDisk** to **true** under the **KafkaRebalance.spec**. When setting **rebalanceDisk** to **true**, do not set a **goals** field in the **KafkaRebalance.spec**, as Cruise Control will automatically set the intra-broker goals and ignore the inter-broker goals. Cruise Control does not perform inter-broker and intra-broker balancing at the same time.

### 8.4.4. Rebalance tuning options

Cruise Control provides several configuration options for tuning the rebalance parameters discussed above. You can set these tuning options when configuring and deploying Cruise Control with Kafka or optimization proposal levels:

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

- The individual rebalance performance configurations can be set under **KafkaRebalance.spec**.

The relevant configurations are summarized in the following table.

**Table 8.2. Rebalance performance tuning configuration**

| Cruise Control properties | KafkaRebalance properties | Default | Description |
|---|---|---|---|
| **num.concurrent.partition.movements.per.broker** | **concurrentPartitionMovementsPerBroker** | 5 | The maximum number of inter-broker partition movements in each partition reassignment batch |
| **num.concurrent.intra.broker.partition.movements** | **concurrentIntraBrokerPartitionMovements** | 2 | The maximum number of intra-broker partition movements in each partition reassignment batch |
| **num.concurrent.leader.movements** | **concurrentLeaderMovements** | 1000 | The maximum number of partition leadership changes in each partition reassignment batch |
| **default.replication.throttle** | **replicationThrottle** | Null (no limit) | The bandwidth (in bytes per second) to assign to partition reassignment |

| Cruise Control properties | KafkaRebalance properties | Default | Description |
|---|---|---|---|
| **default.replica.movement.strategies** | **replicaMovementStrategies** | **BaseReplicaMovementStrategy** | The list of strategies (in priority order) used to determine the order in which partition reassignment commands are executed for generated proposals. For the server setting, use a comma separated string with the fully qualified names of the strategy class (add **com.linkedin.kafka.cruisecontrol.executor.strategy.** to the start of each class name). For the **KafkaRebalance** resource setting use a YAML array of strategy class names. |
| - | **rebalanceDisk** | false | Enables intra-broker disk balancing, which balances disk space utilization between disks on the same broker. Only applies to Kafka deployments that use JBOD storage with multiple disks. |

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

**Additional resources**

- Section 12.2.50, "**CruiseControlSpec** schema reference".

- Section 12.2.133, "**KafkaRebalanceSpec** schema reference".

## 8.5. CONFIGURING AND DEPLOYING CRUISE CONTROL WITH KAFKA

Configure a **Kafka** resource to deploy Cruise Control alongside a Kafka cluster. You can use the **cruiseControl** properties of the **Kafka** resource to configure the deployment. Deploy one instance of Cruise Control per Kafka cluster.

Use **goals** configuration in the Cruise Control **config** to specify optimization goals for generating optimization proposals. You can use **brokerCapacity** to change the default capacity limits for goals related to resource distribution. If brokers are running on nodes with heterogeneous network resources, you can use **overrides** to set network capacity limits for each broker.

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

**Prerequisites**

- An OpenShift cluster

- A running Cluster Operator

**Procedure**

1. Edit the **cruiseControl** property for the **Kafka** resource.
   The properties you can configure are shown in this example configuration:

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
    name: my-cluster
   spec:
    # ...
    cruiseControl:
     brokerCapacity: 1
       inboundNetwork: 10000KB/s
       outboundNetwork: 10000KB/s
       overrides: 2
       - brokers: [0]
         inboundNetwork: 20000KiB/s
         outboundNetwork: 20000KiB/s
       - brokers: [1, 2]
         inboundNetwork: 30000KiB/s
         outboundNetwork: 30000KiB/s
       # ...
     config: 3
       default.goals: > 4
         com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
         com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
         com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
         # ...
   ```

```
        hard.goals: >
          com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
          com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
          # ...
        cpu.balance.threshold: 1.1
        metadata.max.age.ms: 300000
        send.buffer.bytes: 131072
        webserver.http.cors.enabled: true (5)
        webserver.http.cors.origin: "*"
        webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
        # ...
      resources: (6)
        requests:
          cpu: 1
          memory: 512Mi
        limits:
          cpu: 2
          memory: 2Gi
      logging: (7)
          type: inline
          loggers:
            rootLogger.level: "INFO"
      template: (8)
        pod:
          metadata:
            labels:
              label1: value1
          securityContext:
            runAsUser: 1000001
            fsGroup: 0
          terminationGracePeriodSeconds: 120
      readinessProbe: (9)
        initialDelaySeconds: 15
        timeoutSeconds: 5
      livenessProbe:
        initialDelaySeconds: 15
        timeoutSeconds: 5
      metricsConfig: (10)
        type: jmxPrometheusExporter
        valueFrom:
          configMapKeyRef:
            name: cruise-control-metrics
            key: metrics-config.yml
    # ...
```

1. Capacity limits for broker resources .

2. Overrides set network capacity limits for specific brokers when running on nodes with heterogeneous network resources.

3. Cruise Control configuration. Standard Cruise Control configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

4. Optimization goals configuration , which can include configuration for default optimization goals (**default.goals**), main optimization goals ( **goals**), and hard goals ( **hard.goals**).

5    CORS enabled and configured for read-only access to the Cruise Control API.

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

7    Cruise Control loggers and log levels added directly (**inline**) or indirectly (**external**) through a ConfigMap. A custom ConfigMap must be placed under the **log4j.properties** key. Cruise Control has a single logger named **rootLogger.level**. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

8    Template customization. Here a pod is scheduled with additional security attributes.

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

10   Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter (the default metrics exporter).

2. Create or update the resource:

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

3. Check the status of the deployment:

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

   **Output shows the deployment name and readiness**

   ```
   NAME                    READY  UP-TO-DATE  AVAILABLE
   my-cluster-cruise-control 1/1    1          1
   ```

   **my-cluster** is the name of the Kafka cluster.

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

## Auto-created topics

The following table shows the three topics that are automatically created when Cruise Control is deployed. These topics are required for Cruise Control to work properly and must not be deleted or changed. You can change the name of the topic using the specified configuration option.

Table 8.3. Auto-created topics

| Auto-created topic configuration | Default topic name | Created by | Function |
|---|---|---|---|
| **metric.report er.topic** | **strimzi.cruis econtrol.met rics** | AMQ Streams Metrics Reporter | Stores the raw metrics from the Metrics Reporter in each Kafka broker. |

| Auto-created topic configuration | Default topic name | Created by | Function |
|---|---|---|---|
| **partition.metric.sample.store.topic** | **strimzi.cruisecontrol.partitionmetricsamples** | Cruise Control | Stores the derived metrics for each partition. These are created by the Metric Sample Aggregator. |
| **broker.metric.sample.store.topic** | **strimzi.cruisecontrol.modeltrainingsamples** | Cruise Control | Stores the metrics samples used to create the Cluster Workload Model. |

To prevent the removal of records that are needed by Cruise Control, log compaction is disabled in the auto-created topics.

**NOTE**

If the names of the auto-created topics are changed in a Kafka cluster that already has Cruise Control enabled, the old topics will not be deleted and should be manually removed.

**What to do next**

After configuring and deploying Cruise Control, you can generate optimization proposals.

**Additional resources**

- Optimization goals overview

- **CruiseControlSpec** schema reference

## 8.6. GENERATING OPTIMIZATION PROPOSALS

When you create or update a **KafkaRebalance** resource, Cruise Control generates an optimization proposal for the Kafka cluster based on the configured optimization goals. Analyze the information in the optimization proposal and decide whether to approve it. You can use the results of the optimization proposal to rebalance your Kafka cluster.

You can run the optimization proposal in one of the following modes:

- **full** (default)

- **add-brokers**

- **remove-brokers**

The mode you use depends on whether you are rebalancing across all the brokers already running in the Kafka cluster; or you want to rebalance after scaling up or before scaling down your Kafka cluster. For more information, see Rebalancing modes with broker scaling.

**Prerequisites**

- You have deployed Cruise Control to your AMQ Streams cluster.

- You have configured optimization goals and, optionally, capacity limits on broker resources.

**Procedure**

1. Create a **KafkaRebalance** resource and specify the appropriate mode.

   **full** mode (default)

   To use the *default optimization goals* defined in the **Kafka** resource, leave the **spec** property empty. Cruise Control rebalances a Kafka cluster in **full** mode by default.

   **Example configuration with full rebalancing by default**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec: {}
   ```

   You can also run a full rebalance by specifying the **full** mode through the **spec.mode** property.

   **Example configuration specifying full mode**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     mode: full
   ```

   **add-brokers** mode

   If you want to rebalance a Kafka cluster after scaling up, specify the **add-brokers** mode. In this mode, existing replicas are moved to the newly added brokers. You need to specify the brokers as a list.

   **Example configuration specifying add-brokers mode**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
   ```

```
    labels:
      strimzi.io/cluster: my-cluster
  spec:
    mode: add-brokers
    brokers: [3, 4] 1
```

**1**      List of newly added brokers added by the scale up operation. This property is mandatory.

**remove-brokers** mode

If you want to rebalance a Kafka cluster before scaling down, specify the **remove-brokers** mode.
In this mode, replicas are moved off the brokers that are going to be removed. You need to specify the brokers that are being removed as a list.

### Example configuration specifying **remove-brokers** mode

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: remove-brokers
  brokers: [3, 4] 1
```

**1**      List of brokers to be removed by the scale down operation. This property is mandatory.

> **NOTE**
>
> The following steps and the steps to approve or stop a rebalance are the same regardless of the rebalance mode you are using.

2. To configure *user-provided optimization goals* instead of using the default goals, add the **goals** property and enter one or more goals.
   In the following example, rack awareness and replica capacity are configured as user-provided optimization goals:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
```

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

4. (Optional) To approve the optimization proposal automatically, set the **strimzi.io/rebalance-auto-approval** annotation to **true**:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: true
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

5. Create or update the resource:

```
oc apply -f <kafka_rebalance_configuration_file>
```

The Cluster Operator requests the optimization proposal from Cruise Control. This might take a few minutes depending on the size of the Kafka cluster.

6. If you used the automatic approval mechanism, wait for the status of the optimization proposal to change to **Ready**. If you haven't enabled the automatic approval mechanism, wait for the status of the optimization proposal to change to **ProposalReady**:

```
oc get kafkarebalance -o wide -w -n <namespace>
```

**PendingProposal**

A **PendingProposal** status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

**ProposalReady**

A **ProposalReady** status means the optimization proposal is ready for review and approval.

When the status changes to **ProposalReady**, the optimization proposal is ready to approve.

7. Review the optimization proposal.

The optimization proposal is contained in the **Status.Optimization Result** property of the **KafkaRebalance** resource.

```
oc describe kafkarebalance <kafka_rebalance_resource_name>
```

## Example optimization proposal

```
Status:
  Conditions:
    Last Transition Time:  2020-05-19T13:50:12.533Z
    Status:                ProposalReady
    Type:                  State
  Observed Generation:     1
  Optimization Result:
    Data To Move MB:  0
    Excluded Brokers For Leadership:
    Excluded Brokers For Replica Move:
    Excluded Topics:
    Intra Broker Data To Move MB:      0
    Monitored Partitions Percentage:   100
    Num Intra Broker Replica Movements:  0
    Num Leader Movements:              0
    Num Replica Movements:             26
    On Demand Balancedness Score After:   81.8666802863978
    On Demand Balancedness Score Before:  78.01176356230222
    Recent Windows:                    1
  Session Id:                          05539377-ca7b-45ef-b359-e13564f1458c
```

The properties in the **Optimization Result** section describe the pending cluster rebalance operation. For descriptions of each property, see Contents of optimization proposals .

## Insufficient CPU capacity

If a Kafka cluster is overloaded in terms of CPU utilization, you might see an insufficient CPU capacity error in the **KafkaRebalance** status. It's worth noting that this utilization value is unaffected by the **excludedTopics** configuration. Although optimization proposals will not reassign replicas of excluded topics, their load is still considered in the utilization calculation.

## Example CPU utilization error

```
com.linkedin.kafka.cruisecontrol.exception.OptimizationFailureException:
    [CpuCapacityGoal] Insufficient capacity for cpu (Utilization 615.21,
    Allowed Capacity 420.00, Threshold: 0.70). Add at least 3 brokers with
    the same cpu capacity (100.00) as broker-0. Add at least 3 brokers with
    the same cpu capacity (100.00) as broker-0.
```

> **NOTE**
>
> The error shows CPU capacity as a percentage rather than the number of CPU cores. For this reason, it does not directly map to the number of CPUs configured in the Kafka custom resource. It is like having a single *virtual* CPU per broker, which has the cycles of the CPUs configured in **Kafka.spec.kafka.resources.limits.cpu**. This has no effect on the rebalance behavior, since the ratio between CPU utilization and capacity remains the same.

**What to do next**

**Additional resources**

- Section 8.3, "Optimization proposals overview"

## 8.7. APPROVING AN OPTIMIZATION PROPOSAL

You can approve an optimization proposal generated by Cruise Control, if its status is **ProposalReady**. Cruise Control will then apply the optimization proposal to the Kafka cluster, reassigning partitions to brokers and changing partition leadership.

CAUTION

**This is not a dry run.**Before you approve an optimization proposal, you must:

- Refresh the proposal in case it has become out of date.

- Carefully review the contents of the proposal.

**Prerequisites**

- You have generated an optimization proposal from Cruise Control.

- The **KafkaRebalance** custom resource status is **ProposalReady**.

**Procedure**

Perform these steps for the optimization proposal that you want to approve.

1. Unless the optimization proposal is newly generated, check that it is based on current information about the state of the Kafka cluster. To do so, refresh the optimization proposal to make sure it uses the latest cluster metrics:

   a. Annotate the **KafkaRebalance** resource in OpenShift with **strimzi.io/rebalance=refresh**:

      ```
      oc annotate kafkarebalance <kafka_rebalance_resource_name>
      strimzi.io/rebalance=refresh
      ```

2. Wait for the status of the optimization proposal to change to **ProposalReady**:

   ```
   oc get kafkarebalance -o wide -w -n <namespace>
   ```

   **PendingProposal**

   A **PendingProposal** status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

   **ProposalReady**

   A **ProposalReady** status means the optimization proposal is ready for review and approval.

   When the status changes to **ProposalReady**, the optimization proposal is ready to approve.

3. Approve the optimization proposal that you want Cruise Control to apply.

Annotate the **KafkaRebalance** resource in OpenShift with **strimzi.io/rebalance=approve**:

```
oc annotate kafkarebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance=approve
```

4. The Cluster Operator detects the annotated resource and instructs Cruise Control to rebalance the Kafka cluster.

5. Wait for the status of the optimization proposal to change to **Ready**:

```
oc get kafkarebalance -o wide -w -n <namespace>
```

**Rebalancing**

A **Rebalancing** status means the rebalancing is in progress.

**Ready**

A **Ready** status means the rebalance is complete.

**NotReady**

A **NotReady** status means an error occurred—see  Fixing problems with a **KafkaRebalance resource**.

When the status changes to **Ready**, the rebalance is complete.

To use the same **KafkaRebalance** custom resource to generate another optimization proposal, apply the **refresh** annotation to the custom resource. This moves the custom resource to the **PendingProposal** or **ProposalReady** state. You can then review the optimization proposal and approve it, if desired.

**Additional resources**

- Section 8.3, "Optimization proposals overview"

- Section 8.8, "Stopping a cluster rebalance"

## 8.8. STOPPING A CLUSTER REBALANCE

Once started, a cluster rebalance operation might take some time to complete and affect the overall performance of the Kafka cluster.

If you want to stop a cluster rebalance operation that is in progress, apply the **stop** annotation to the **KafkaRebalance** custom resource. This instructs Cruise Control to finish the current batch of partition reassignments and then stop the rebalance. When the rebalance has stopped, completed partition reassignments have already been applied; therefore, the state of the Kafka cluster is different when compared to prior to the start of the rebalance operation. If further rebalancing is required, you should generate a new optimization proposal.

NOTE

The performance of the Kafka cluster in the intermediate (stopped) state might be worse than in the initial state.

**Prerequisites**

- You have [approved the optimization proposal](#) by annotating the **KafkaRebalance** custom resource with **approve**.

- The status of the **KafkaRebalance** custom resource is **Rebalancing**.

**Procedure**

1. Annotate the **KafkaRebalance** resource in OpenShift:

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

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

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

3. Wait until the status changes to **Stopped**.

**Additional resources**

- [Section 8.3, "Optimization proposals overview"](#)

## 8.9. FIXING PROBLEMS WITH A KAFKAREBALANCE RESOURCE

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

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

- A misconfigured parameter in the **KafkaRebalance** resource.

- The **strimzi.io/cluster** label for specifying the Kafka cluster in the **KafkaRebalance** resource is missing.

- The Cruise Control server is not deployed as the **cruiseControl** property in the **Kafka** resource is missing.

- The Cruise Control server is not reachable.

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

**Prerequisites**

- You have [approved an optimization proposal](#).

- The status of the **KafkaRebalance** custom resource for the rebalance operation is **NotReady**.

**Procedure**

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

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

2. Attempt to resolve the issue in the **KafkaRebalance** resource.

3. Annotate the **KafkaRebalance** resource in OpenShift:

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

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

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

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

**Additional resources**

- Section 8.3, "Optimization proposals overview"

# CHAPTER 9. VALIDATING SCHEMAS WITH THE RED HAT BUILD OF APICURIO REGISTRY

You can use the Red Hat build of Apicurio Registry with AMQ Streams.

Apicurio Registry is a datastore for sharing standard event schemas and API designs across API and event-driven architectures. You can use Apicurio Registry to decouple the structure of your data from your client applications, and to share and manage your data types and API descriptions at runtime using a REST interface.

Apicurio Registry stores schemas used to serialize and deserialize messages, which can then be referenced from your client applications to ensure that the messages that they send and receive are compatible with those schemas. Apicurio Registry provides Kafka client serializers/deserializers for Kafka producer and consumer applications. Kafka producer applications use serializers to encode messages that conform to specific event schemas. Kafka consumer applications use deserializers, which validate that the messages have been serialized using the correct schema, based on a specific schema ID.

You can enable your applications to use a schema from the registry. This ensures consistent schema usage and helps to prevent data errors at runtime.

**Additional resources**

- Red Hat build of Apicurio Registry documentation

- Red Hat build of Apicurio Registry is built on the Apicurio Registry open source community project available on GitHub: Apicurio/apicurio-registry

# CHAPTER 10. MANAGING TLS CERTIFICATES

AMQ Streams supports TLS for encrypted communication between Kafka and AMQ Streams components.

Communication is always encrypted between the following components:

- Communication between Kafka and ZooKeeper

- Interbroker communication between Kafka brokers

- Internodal communication between ZooKeeper nodes

- AMQ Streams operator communication with Kafka brokers and ZooKeeper nodes

Communication between Kafka clients and Kafka brokers is encrypted according to how the cluster is configured. For the Kafka and AMQ Streams components, TLS certificates are also used for authentication.

The Cluster Operator automatically sets up and renews TLS certificates to enable encryption and authentication within your cluster. It also sets up other TLS certificates if you want to enable encryption or mTLS authentication between Kafka brokers and clients.

CA (certificate authority) certificates are generated by the Cluster Operator to verify the identities of components and clients. If you don't want to use the CAs generated by the Cluster Operator, you can install your own cluster and clients CA certificates .

You can also provide *Kafka listener certificates* for TLS listeners or external listeners that have TLS encryption enabled. Use Kafka listener certificates to incorporate the security infrastructure you already have in place.

> **NOTE**
>
> Any certificates you provide are not renewed by the Cluster Operator.

Figure 10.1. Example architecture of the communication secured by TLS
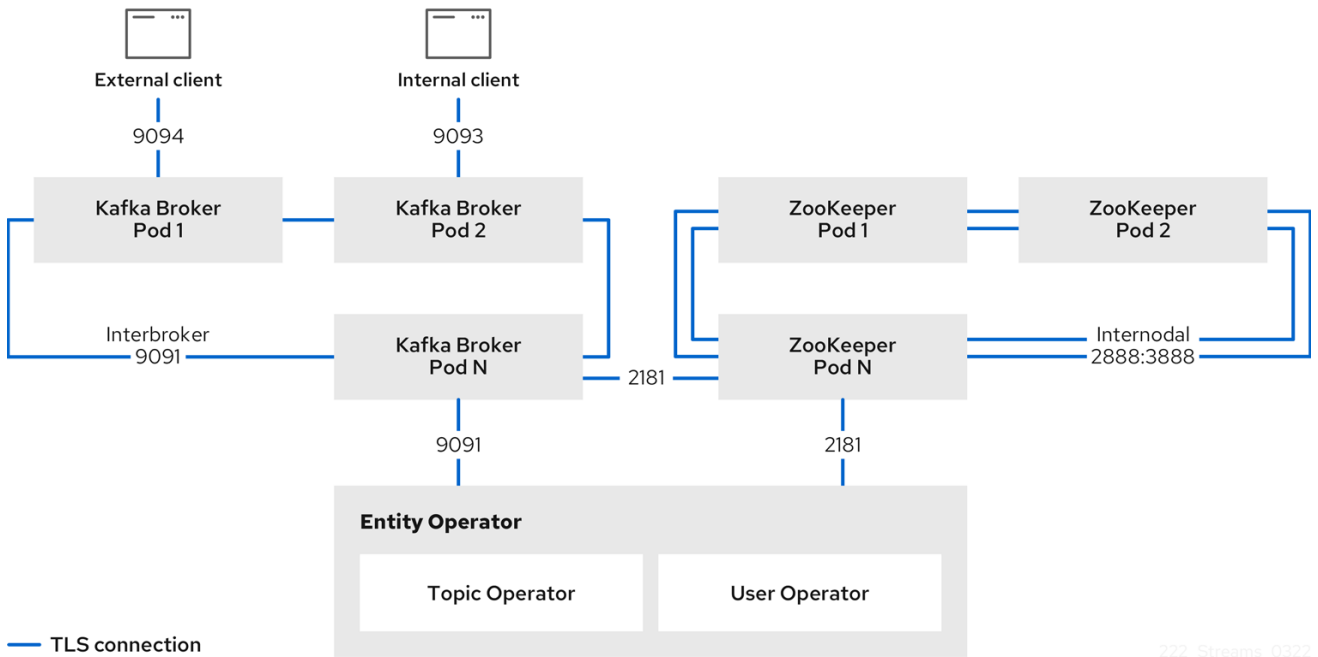


## 10.1. INTERNAL CLUSTER CA AND CLIENTS CA

To support encryption, each AMQ Streams component needs its own private keys and public key certificates. All component certificates are signed by an internal CA (certificate authority) called the *cluster CA*.

Similarly, each Kafka client application connecting to AMQ Streams using mTLS needs to use private keys and certificates. A second internal CA, named the *clients CA*, is used to sign certificates for the Kafka clients.

Both the cluster CA and clients CA have a self-signed public key certificate.

Kafka brokers are configured to trust certificates signed by either the cluster CA or clients CA. Components that clients do not need to connect to, such as ZooKeeper, only trust certificates signed by the cluster CA. Unless TLS encryption for external listeners is disabled, client applications must trust certificates signed by the cluster CA. This is also true for client applications that perform mTLS authentication.

By default, AMQ Streams automatically generates and renews CA certificates issued by the cluster CA or clients CA. You can configure the management of these CA certificates in the **Kafka.spec.clusterCa** and **Kafka.spec.clientsCa** objects.

You can replace the CA certificates for the cluster CA or clients CA with your own. For more information, see Section 10.8.1, "Installing your own CA certificates and private keys" . If you provide your own CA certificates, you must renew them before they expire.

## 10.2. SECRETS GENERATED BY THE OPERATORS

Secrets are created when custom resources are deployed, such as **Kafka** and **KafkaUser**. AMQ Streams uses these secrets to store private and public key certificates for Kafka clusters, clients, and users. The secrets are used for establishing TLS encrypted connections between Kafka brokers, and between brokers and clients. They are also used for mTLS authentication.

Cluster and clients secrets are always pairs: one contains the public key and one contains the private key.

**Cluster secret**

A cluster secret contains the *cluster CA* to sign Kafka broker certificates. Connecting clients use the certificate to establish a TLS encrypted connection with a Kafka cluster. The certificate verifies broker identity.

**Client secret**

A client secret contains the *clients CA* for a user to sign its own client certificate. This allows mutual authentication against the Kafka cluster. The broker validates a client's identity through the certificate.

**User secret**

A user secret contains a private key and certificate. The secret is created and signed by the clients CA when a new user is created. The key and certificate are used to authenticate and authorize the user when accessing the cluster.

## 10.2.1. TLS authentication using keys and certificates in PEM or PKCS #12 format

The secrets created by AMQ Streams provide private keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12 (Public-Key Cryptography Standards) formats. PEM and PKCS #12 are OpenSSL-generated key formats for TLS communications using the SSL protocol.

You can configure mutual TLS (mTLS) authentication that uses the credentials contained in the secrets generated for a Kafka cluster and user.

To set up mTLS, you must first do the following:

- Configure your Kafka cluster with a listener that uses mTLS

- Create a **KafkaUser** that provides client credentials for mTLs

When you deploy a Kafka cluster, a **<cluster_name>-cluster-ca-cert** secret is created with public key to verify the cluster. You use the public key to configure a truststore for the client.

When you create a **KafkaUser**, a **<kafka_user_name>** secret is created with the keys and certificates to verify the user (client). Use these credentials to configure a keystore for the client.

With the Kafka cluster and client set up to use mTLS, you extract credentials from the secrets and add them to your client configuration.

**PEM keys and certificates**

For PEM, you add the following to your client configuration:

**Truststore**

- **ca.crt** from the **<cluster_name>-cluster-ca-cert** secret, which is the CA certificate for the cluster.

**Keystore**

- **user.crt** from the **<kafka_user_name>** secret, which is the public certificate of the user.

- **user.key** from the **<kafka_user_name>** secret, which is the public key of the user.

PKCS #12 keys and certificates

For PKCS #12, you add the following to your client configuration:

**Truststore**

- **ca.p12** from the **<cluster_name>-cluster-ca-cert** secret, which is the CA certificate for the cluster.

- **ca.password** from the **<cluster_name>-cluster-ca-cert** secret, which is the password to access the public cluster CA certificate.

**Keystore**

- **user.p12** from the **<kafka_user_name>** secret, which is the public key certificate of the user.

- **user.password** from the **<kafka_user_name>** secret, which is the password to access the public key certificate of the Kafka user.

PKCS #12 is supported by Java, so you can add the values of the certificates directly to your Java client configuration. You can also reference the certificates from a secure storage location. With PEM files, you must add the certificates directly to the client configuration in single-line format. Choose a format that's suitable for establishing TLS connections between your Kafka cluster and client. Use PKCS #12 if you are unfamiliar with PEM.

> **NOTE**
>
> All keys are 2048 bits in size and, by default, are valid for 365 days from the initial generation. You can change the validity period .

## 10.2.2. Secrets generated by the Cluster Operator

The Cluster Operator generates the following certificates, which are saved as secrets in the OpenShift cluster. AMQ Streams uses these secrets by default.

The cluster CA and clients CA have separate secrets for the private key and public key.

***<cluster_name>*-cluster-ca**

Contains the private key of the cluster CA. AMQ Streams and Kafka components use the private key to sign server certificates.

***<cluster_name>*-cluster-ca-cert**

Contains the public key of the cluster CA. Kafka clients use the public key to verify the identity of the Kafka brokers they are connecting to with TLS server authentication.

***<cluster_name>*-clients-ca**

Contains the private key of the clients CA. Kafka clients use the private key to sign new user certificates for mTLS authentication when connecting to Kafka brokers.

***<cluster_name>*-clients-ca-cert**

Contains the public key of the clients CA. Kafka brokers use the public key to verify the identity of clients accessing the Kafka brokers when mTLS authentication is used.

Secrets for communication between AMQ Streams components contain a private key and a public key certificate signed by the cluster CA.

***<cluster_name>*-kafka-brokers**

Contains the private and public keys for Kafka brokers.

***<cluster_name>*-zookeeper-nodes**

Contains the private and public keys for ZooKeeper nodes.

***<cluster_name>*-cluster-operator-certs**

Contains the private and public keys for encrypting communication between the Cluster Operator and Kafka or ZooKeeper.

***<cluster_name>*-entity-topic-operator-certs**

Contains the private and public keys for encrypting communication between the Topic Operator and Kafka or ZooKeeper.

***<cluster_name>*-entity-user-operator-certs**

Contains the private and public keys for encrypting communication between the User Operator and Kafka or ZooKeeper.

***<cluster_name>*-cruise-control-certs**

Contains the private and public keys for encrypting communication between Cruise Control and Kafka or ZooKeeper.

***<cluster_name>*-kafka-exporter-certs**

Contains the private and public keys for encrypting communication between Kafka Exporter and Kafka or ZooKeeper.

> **NOTE**
>
> You can provide your own server certificates and private keys to connect to to Kafka brokers using *Kafka listener certificates* rather than certificates signed by the cluster CA or clients CA.

## 10.2.3. Cluster CA secrets

Cluster CA secrets are managed by the Cluster Operator in a Kafka cluster.

Only the ***<cluster_name>*-cluster-ca-cert** secret is required by clients. All other cluster secrets are accessed by AMQ Streams components. You can enforce this using OpenShift role-based access controls, if necessary.

> **NOTE**
>
> The CA certificates in ***<cluster_name>*-cluster-ca-cert** must be trusted by Kafka client applications so that they validate the Kafka broker certificates when connecting to Kafka brokers over TLS.

Table 10.1. Fields in the ***<cluster_name>*-cluster-ca** secret

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

Table 10.2. Fields in the ***<cluster_name>*-cluster-ca-cert** secret

| Field | Description |
|---|---|
| **ca.p12** | PKCS #12 store for storing certificates and keys. |
| **ca.password** | Password for protecting the PKCS #12 store. |
| **ca.crt** | The current certificate for the cluster CA. |

Table 10.3. Fields in the *<cluster_name>*-**kafka-brokers** secret

| Field | Description |
|---|---|
| *<cluster_name>*-**kafka-***<num>*.**p12** | PKCS #12 store for storing certificates and keys. |
| *<cluster_name>*-**kafka-***<num>*.**password** | Password for protecting the PKCS #12 store. |
| *<cluster_name>*-**kafka-***<num>*.**crt** | Certificate for a Kafka broker pod *<num>*. Signed by a current or former cluster CA private key in *<cluster_name>*-**cluster-ca**. |
| *<cluster_name>*-**kafka-***<num>*.**key** | Private key for a Kafka broker pod *<num>*. |

Table 10.4. Fields in the *<cluster_name>*-**zookeeper-nodes** secret

| Field | Description |
|---|---|
| *<cluster_name>*-**zookeeper-***<num>*.**p12** | PKCS #12 store for storing certificates and keys. |
| *<cluster_name>*-**zookeeper-***<num>*.**password** | Password for protecting the PKCS #12 store. |
| *<cluster_name>*-**zookeeper-***<num>*.**crt** | Certificate for ZooKeeper node *<num>*. Signed by a current or former cluster CA private key in *<cluster_name>*-**cluster-ca**. |
| *<cluster_name>*-**zookeeper-***<num>*.**key** | Private key for ZooKeeper pod *<num>*. |

Table 10.5. Fields in the *<cluster_name>*-**cluster-operator-certs** secret

| Field | Description |
|---|---|
| **cluster-operator.p12** | PKCS #12 store for storing certificates and keys. |
| **cluster-operator.password** | Password for protecting the PKCS #12 store. |

| Field | Description |
|---|---|
| **cluster-operator.crt** | Certificate for mTLS communication between the Cluster Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**. |
| **cluster-operator.key** | Private key for mTLS communication between the Cluster Operator and Kafka or ZooKeeper. |

Table 10.6. Fields in the ***<cluster_name>*-entity-topic-operator-certs** secret

| Field | Description |
|---|---|
| **entity-operator.p12** | PKCS #12 store for storing certificates and keys. |
| **entity-operator.password** | Password for protecting the PKCS #12 store. |
| **entity-operator.crt** | Certificate for mTLS communication between the Topic Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**. |
| **entity-operator.key** | Private key for mTLS communication between the Topic Operator and Kafka or ZooKeeper. |

Table 10.7. Fields in the ***<cluster_name>*-entity-user-operator-certs** secret

| Field | Description |
|---|---|
| **entity-operator.p12** | PKCS #12 store for storing certificates and keys. |
| **entity-operator.password** | Password for protecting the PKCS #12 store. |
| **entity-operator.crt** | Certificate for mTLS communication between the User Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**. |
| **entity-operator.key** | Private key for mTLS communication between the User Operator and Kafka or ZooKeeper. |

Table 10.8. Fields in the ***<cluster_name>*-cruise-control-certs** secret

| Field | Description |
|---|---|
| **cruise-control.p12** | PKCS #12 store for storing certificates and keys. |
| **cruise-control.password** | Password for protecting the PKCS #12 store. |

| Field | Description |
|---|---|
| **cruise-control.crt** | Certificate for mTLS communication between Cruise Control and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**. |
| **cruise-control.key** | Private key for mTLS communication between the Cruise Control and Kafka or ZooKeeper. |

Table 10.9. Fields in the ***<cluster_name>*-kafka-exporter-certs** secret

| Field | Description |
|---|---|
| **kafka-exporter.p12** | PKCS #12 store for storing certificates and keys. |
| **kafka-exporter.password** | Password for protecting the PKCS #12 store. |
| **kafka-exporter.crt** | Certificate for mTLS communication between Kafka Exporter and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in ***<cluster_name>*-cluster-ca**. |
| **kafka-exporter.key** | Private key for mTLS communication between the Kafka Exporter and Kafka or ZooKeeper. |

## 10.2.4. Clients CA secrets

Clients CA secrets are managed by the Cluster Operator in a Kafka cluster.

The certificates in ***<cluster_name>*-clients-ca-cert** are those which the Kafka brokers trust.

The ***<cluster_name>*-clients-ca** secret is used to sign the certificates of client applications. This secret must be accessible to the AMQ Streams components and for administrative access if you are intending to issue application certificates without using the User Operator. You can enforce this using OpenShift role-based access controls, if necessary.

Table 10.10. Fields in the ***<cluster_name>*-clients-ca** secret

| Field | Description |
|---|---|
| **ca.key** | The current private key for the clients CA. |

Table 10.11. Fields in the ***<cluster_name>*-clients-ca-cert** secret

| Field | Description |
|---|---|
| **ca.p12** | PKCS #12 store for storing certificates and keys. |
| **ca.password** | Password for protecting the PKCS #12 store. |

| Field | Description |
|---|---|
| **ca.crt** | The current certificate for the clients CA. |

## 10.2.5. User secrets generated by the User Operator

User secrets are managed by the User Operator.

When a user is created using the User Operator, a secret is generated using the name of the user.

**Table 10.12. Fields in the *user_name* secret**

| Secret name | Field within secret | Description |
|---|---|---|
| ***<user_name>*** | **user.p12** | PKCS #12 store for storing certificates and keys. |
| | **user.password** | Password for protecting the PKCS #12 store. |
| | **user.crt** | Certificate for the user, signed by the clients CA |
| | **user.key** | Private key for the user |

## 10.2.6. Adding labels and annotations to cluster CA secrets

By configuring the **clusterCaCert** template property in the **Kafka** custom resource, you can add custom labels and annotations to the Cluster CA secrets created by the Cluster Operator. Labels and annotations are useful for identifying objects and adding contextual information. You configure template properties in AMQ Streams custom resources.

**Example template customization to add labels and annotations to secrets**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      clusterCaCert:
        metadata:
          labels:
            label1: value1
            label2: value2
          annotations:
            annotation1: value1
            annotation2: value2
    # ...
```

For more information on configuring template properties, see Section 2.8, "Customizing OpenShift resources".

### 10.2.7. Disabling `ownerReference` in the CA secrets

By default, the cluster and clients CA secrets are created with an **ownerReference** property that is set to the **Kafka** custom resource. This means that, when the **Kafka** custom resource is deleted, the CA secrets are also deleted (garbage collected) by OpenShift.

If you want to reuse the CA for a new cluster, you can disable the **ownerReference** by setting the **generateSecretOwnerReference** property for the cluster and clients CA secrets to **false** in the **Kafka** configuration. When the **ownerReference** is disabled, CA secrets are not deleted by OpenShift when the corresponding **Kafka** custom resource is deleted.

**Example Kafka configuration with disabled `ownerReference` for cluster and clients CAs**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    generateSecretOwnerReference: false
  clientsCa:
    generateSecretOwnerReference: false
# ...
```

**Additional resources**

- **CertificateAuthority** schema reference

## 10.3. CERTIFICATE RENEWAL AND VALIDITY PERIODS

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

For CA certificates automatically created by the Cluster Operator, you can configure the validity period of:

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

- Clients CA certificates in **Kafka.spec.clientsCa.validityDays**

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

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

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

You can configure the renewal period of the certificates created by the Cluster Operator:

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

- Clients CA certificates in **Kafka.spec.clientsCa.renewalDays**

The default renewal period for both certificates is 30 days.

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

### Validity period against renewal period

```
Not Before                    Not After
  |                              |
  |<--------------- validityDays --------------->|
                   <--- renewalDays --->|
```
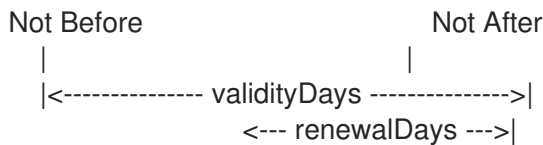
To make a change to the validity and renewal periods after creating the Kafka cluster, you configure and apply the **Kafka** custom resource, and manually renew the CA certificates. If you do not manually renew the certificates, the new periods will be used the next time the certificate is renewed automatically.

### Example Kafka configuration for certificate validity and renewal periods

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
  clientsCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
# ...
```

The behavior of the Cluster Operator during the renewal period depends on the settings for the **generateCertificateAuthority** certificate generation properties for the cluster CA and clients CA.

**true**

> If the properties are set to **true**, a CA certificate is generated automatically by the Cluster Operator, and renewed automatically within the renewal period.

**false**

> If the properties are set to **false**, a CA certificate is not generated by the Cluster Operator. Use this option if you are installing your own certificates.

## 10.3.1. Renewal process with automatically generated CA certificates

The Cluster Operator performs the following processes in this order when renewing CA certificates:

1. Generates a new CA certificate, but retains the existing key.
   The new certificate replaces the old one with the name **ca.crt** within the corresponding **Secret**.

2. Generates new client certificates (for ZooKeeper nodes, Kafka brokers, and the Entity Operator).
   This is not strictly necessary because the signing key has not changed, but it keeps the validity period of the client certificate in sync with the CA certificate.

3. Restarts ZooKeeper nodes so that they will trust the new CA certificate and use the new client certificates.

4. Restarts Kafka brokers so that they will trust the new CA certificate and use the new client certificates.

5. Restarts the Topic and User Operators so that they will trust the new CA certificate and use the new client certificates.
   User certificates are signed by the clients CA. User certificates generated by the User Operator are renewed when the clients CA is renewed.

## 10.3.2. Client certificate renewal

The Cluster Operator is not aware of the client applications using the Kafka cluster.

When connecting to the cluster, and to ensure they operate correctly, client applications must:

- Trust the cluster CA certificate published in the *<cluster>*-cluster-ca-cert Secret.

- Use the credentials published in their *<user-name>* Secret to connect to the cluster.
  The User Secret provides credentials in PEM and PKCS #12 format, or it can provide a password when using SCRAM-SHA authentication. The User Operator creates the user credentials when a user is created.

You must ensure clients continue to work after certificate renewal. The renewal process depends on how the clients are configured.

If you are provisioning client certificates and keys manually, you must generate new client certificates and ensure the new certificates are used by clients within the renewal period. Failure to do this by the end of the renewal period could result in client applications being unable to connect to the cluster.

> **NOTE**
>
> For workloads running inside the same OpenShift cluster and namespace, Secrets can be mounted as a volume so the client Pods construct their keystores and truststores from the current state of the Secrets. For more details on this procedure, see Configuring internal clients to trust the cluster CA.

## 10.3.3. Manually renewing the CA certificates generated by the Cluster Operator

Cluster and clients CA certificates generated by the Cluster Operator auto-renew at the start of their respective certificate renewal periods. However, you can use the **strimzi.io/force-renew** annotation to manually renew one or both of these certificates before the certificate renewal period starts. You might do this for security reasons, or if you have changed the renewal or validity periods for the certificates .

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

> **NOTE**
>
> If you are using your own CA certificates, the **force-renew** annotation cannot be used. Instead, follow the procedure for renewing your own CA certificates .

**Prerequisites**

- The Cluster Operator is running.

- A Kafka cluster in which CA certificates and private keys are installed.

**Procedure**

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

   **Table 10.13. Annotation for the Secret that forces renewal of certificates**

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

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

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

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

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

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

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

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

For example, for a cluster CA certificate:

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

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

**Additional resources**

- Section 10.2, "Secrets generated by the operators"

- Section 2.2.5, "Maintenance time windows for rolling updates"

- Section 12.2.49, "**CertificateAuthority** schema reference"

## 10.3.4. Replacing private keys used by the CA certificates generated by the Cluster Operator

You can replace the private keys used by the cluster CA and clients CA certificates generated by the Cluster Operator. When a private key is replaced, the Cluster Operator generates a new CA certificate for the new private key.

> **NOTE**
>
> If you are using your own CA certificates, the **force-replace** annotation cannot be used. Instead, follow the procedure for renewing your own CA certificates.

**Prerequisites**

- The Cluster Operator is running.

- A Kafka cluster in which CA certificates and private keys are installed.

**Procedure**

- Apply the **strimzi.io/force-replace** annotation to the **Secret** that contains the private key that you want to renew.

  Table 10.14. Commands for replacing private keys

| Private key for | Secret | Annotate command |
| --- | --- | --- |
| Cluster CA | *CLUSTER-NAME*-cluster-ca | **oc annotate secret *CLUSTER-NAME*-cluster-ca strimzi.io/force-replace=true** |

| 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 10.2, "Secrets generated by the operators"

- Section 2.2.5, "Maintenance time windows for rolling updates"

## 10.4. TLS CONNECTIONS

### 10.4.1. ZooKeeper communication

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

Communication between Kafka brokers and ZooKeeper nodes is also encrypted.

### 10.4.2. Kafka inter-broker communication

Communication between Kafka brokers is always encrypted using TLS. The connections between the Kafka controller and brokers use an internal *control plane listener* on port 9090. Replication of data between brokers, as well as internal connections from AMQ Streams operators, Cruise Control, or the Kafka Exporter use the *replication listener* on port 9091. These internal listeners are not available to Kafka clients.

### 10.4.3. Topic and User Operators

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

### 10.4.4. Cruise Control

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

## 10.4.5. Kafka Client connections

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

# 10.5. CONFIGURING INTERNAL CLIENTS TO TRUST THE CLUSTER CA

This procedure describes how to configure a Kafka client that resides inside the OpenShift cluster — connecting to a TLS listener — to trust the cluster CA certificate.

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

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

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

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

**Prerequisites**

- The Cluster Operator must be running.

- There needs to be a **Kafka** resource within the OpenShift cluster.

- You need a Kafka client application inside the OpenShift cluster that will connect using TLS, and needs to trust the cluster CA certificate.

- The client application must be running in the same namespace as the **Kafka** resource.

**Using PKCS #12 format (.p12)**

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

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

```
    volumes:
    - name: secret-volume
      secret:
        secretName: my-cluster-cluster-ca-cert
```

Here we're mounting the following:

- The PKCS #12 file into an exact path, which can be configured

- The password into an environment variable, where it can be used for Java configuration

2. Configure the Kafka client with the following properties:

- A security protocol option:

  - **security.protocol: SSL** when using TLS for encryption (with or without mTLS authentication).

  - **security.protocol: SASL_SSL** when using SCRAM-SHA authentication over TLS.

- **ssl.truststore.location** with the truststore location where the certificates were imported.

- **ssl.truststore.password** with the password for accessing the truststore.

- **ssl.truststore.type=PKCS12** to identify the truststore type.

**Using PEM format (.crt)**

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

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
  - name: client-name
    image: client-name
    volumeMounts:
    - name: secret-volume
      mountPath: /data/crt
  volumes:
  - name: secret-volume
    secret:
      secretName: my-cluster-cluster-ca-cert
```

2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

## 10.6. CONFIGURING EXTERNAL CLIENTS TO TRUST THE CLUSTER CA

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

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

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

The steps describe how to obtain the certificate from the Cluster Secret that verifies the identity of the Kafka cluster.
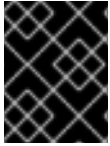
> **IMPORTANT**
>
> The ***<cluster_name>*-cluster-ca-cert** secret contains more than one CA certificate during the CA certificate renewal period. Clients must add *all* of them to their truststores.

**Prerequisites**

- The Cluster Operator must be running.

- There needs to be a **Kafka** resource within the OpenShift cluster.

- You need a Kafka client application outside the OpenShift cluster that will connect using TLS, and needs to trust the cluster CA certificate.

**Using PKCS #12 format (.p12)**

1. Extract the cluster CA certificate and password from the ***<cluster_name>*-cluster-ca-cert** Secret of the Kafka cluster.

   ```
   oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' | base64 -d >
   ca.p12
   ```

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

   Replace *<cluster_name>* with the name of the Kafka cluster.

2. Configure the Kafka client with the following properties:

   - A security protocol option:

     - **security.protocol: SSL** when using TLS.

     - **security.protocol: SASL_SSL** when using SCRAM-SHA authentication over TLS.

   - **ssl.truststore.location** with the truststore location where the certificates were imported.

   - **ssl.truststore.password** with the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.

   - **ssl.truststore.type=PKCS12** to identify the truststore type.

**Using PEM format (.crt)**

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

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

2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

## 10.7. KAFKA LISTENER CERTIFICATES

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

Providing Kafka listener certificates allows you to leverage existing security infrastructure, such as your organization's private CA or a public CA. Kafka clients will need to trust the CA which was used to sign the listener certificate.

You must manually renew Kafka listener certificates when needed.

### 10.7.1. Providing your own Kafka listener certificates for TLS encryption

Listeners provide client access to Kafka brokers. Configure listeners in the **Kafka** resource, including the configuration required for client access using TLS.

By default, the listeners use certificates signed by the internal CA (certificate authority) certificates generated by AMQ Streams. A CA certificate is generated by the Cluster Operator when it creates a Kafka cluster. When you configure a client for TLS, you add the CA certificate to its truststore configuration to verify the Kafka cluster. You can also install and use your own CA certificates. Or you can configure a listener using **brokerCertChainAndKey** properties and use a custom server certificate.

The **brokerCertChainAndKey** properties allow you to access Kafka brokers using your own custom certificates at the listener-level. You create a secret with your own private key and server certificate, then specify the key and certificate in the listener's **brokerCertChainAndKey** configuration. You can use a certificate signed by a public (external) CA or a private CA. If signed by a public CA, you usually won't need to add it to a client's truststore configuration. Custom certificates are not managed by AMQ Streams, so you need to renew them manually.
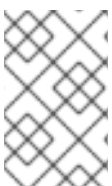
> **NOTE**
>
> Listener certificates are used for TLS encryption and server authentication only. They are not used for TLS client authentication. If you want to use your own certificate for TLS client authentication as well, you must install and use your own clients CA.

**Prerequisites**

- The Cluster Operator is running.

- Each listener requires the following:

  - A compatible server certificate signed by an external CA. (Provide an X.509 certificate in PEM format.)
    You can use one listener certificate for multiple listeners.

  - Subject Alternative Names (SANs) are specified in the certificate for each listener. For more information, see Section 10.7.2, "Alternative subjects in server certificates for Kafka listeners".

If you are not using a self-signed certificate, you can provide a certificate that includes the whole CA chain in the certificate.

You can only use the **brokerCertChainAndKey** properties if TLS encryption ( **tls: true**) is configured for the listener.

**Procedure**

1. Create a **Secret** containing your private key and server certificate:

   ```
   oc create secret generic my-secret --from-file=my-listener-key.key --from-file=my-listener-certificate.crt
   ```

2. Edit the **Kafka** resource for your cluster.
   Configure the listener to use your **Secret**, certificate file, and private key file in the **configuration.brokerCertChainAndKey** property.

   **Example configuration for a loadbalancer external listener with TLS encryption enabled**

   ```
   # ...
   listeners:
     - name: plain
       port: 9092
       type: internal
       tls: false
     - name: external
       port: 9094
       type: loadbalancer
       tls: true
       configuration:
         brokerCertChainAndKey:
           secretName: my-secret
           certificate: my-listener-certificate.crt
           key: my-listener-key.key
   # ...
   ```

   **Example configuration for a TLS listener**

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

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

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

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

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

**Additional resources**

- Chapter 6, *Managing secure access to Kafka*

- Section 10.7.2, "Alternative subjects in server certificates for Kafka listeners"

- Section 12.2.4, "**GenericKafkaListener** schema reference"

## 10.7.2. 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.

### 10.7.2.1. 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
```

### 10.7.2.2. 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 10.15. 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**.<br><br>You can use a matching wildcard name. |
| **loadbalancer** | Addresses of all Kafka broker **loadbalancers** and the bootstrap **loadbalancer** address.<br><br>You can use a matching wildcard name. |
| **NodePort** | Addresses of all OpenShift worker nodes that the Kafka broker pods might be scheduled to.<br><br>You can use a matching wildcard name. |

Additional resources

- Section 10.7.1, "Providing your own Kafka listener certificates for TLS encryption"

## 10.8. USING YOUR OWN CA CERTIFICATES AND PRIVATE KEYS

Install and use your own CA certificates and private keys instead of using the defaults generated by the Cluster Operator. You can replace the cluster and clients CA certificates and private keys.

You can switch to using your own CA certificates and private keys in the following ways:

- Install your own CA certificates and private keys before deploying your Kafka cluster

- Replace the default CA certificates and private keys with your own after deploying a Kafka cluster

The steps to replace the default CA certificates and private keys after deploying a Kafka cluster are the same as those used to renew your own CA certificates and private keys.

If you use your own certificates, they won't be renewed automatically. You need to renew the CA certificates and private keys before they expire.

Renewal options:

- Renew the CA certificates only

- Renew CA certificates and private keys (or replace the defaults)

## 10.8.1. Installing your own CA certificates and private keys

Install your own CA certificates and private keys instead of using the cluster and clients CA certificates and private keys generated by the Cluster Operator.

By default, AMQ Streams uses the following cluster CA and clients CA secrets, which are renewed automatically.

- Cluster CA secrets

  - **<cluster_name>-cluster-ca**

  - **<cluster_name>-cluster-ca-cert**

- Clients CA secrets

  - **<cluster_name>-clients-ca**

  - **<cluster_name>-clients-ca-cert**

To install your own certificates, use the same names.

### Prerequisites

- The Cluster Operator is running.

- A Kafka cluster is not yet deployed.
  If you have already deployed a Kafka cluster, you can replace the default CA certificates with your own.

- Your own X.509 certificates and keys in PEM format for the cluster CA or clients CA.

  - If you want to use a cluster or clients CA which is not a Root CA, you have to include the whole chain in the certificate file. The chain should be in the following order:

    1. The cluster or clients CA

    2. One or more intermediate CAs

    3. The root CA

  - All CAs in the chain should be configured using the X509v3 Basic Constraints extension. Basic Constraints limit the path length of a certificate chain.

- The OpenSSL TLS management tool for converting certificates.

### Before you begin

The Cluster Operator generates keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12 (Public-Key Cryptography Standards) formats. You can add your own certificates in either format.

Some applications cannot use PEM certificates and support only PKCS #12 certificates. If you don't have a cluster certificate in PKCS #12 format, use the OpenSSL TLS management tool to generate one from your **ca.crt** file.

### Example certificate generation command

```
openssl pkcs12 -export -in ca.crt -nokeys -out ca.p12 -password pass:<P12_password> -caname
ca.crt
```

Replace *<P12_password>* with your own password.

### Procedure

1. Create a new secret that contains the CA certificate.

   #### Client secret creation with a certificate in PEM format only

   ```
   oc create secret generic <cluster_name>-clients-ca-cert --from-file=ca.crt=ca.crt
   ```

   #### Cluster secret creation with certificates in PEM and PKCS #12 format

   ```
   oc create secret generic <cluster_name>-cluster-ca-cert \
     --from-file=ca.crt=ca.crt \
     --from-file=ca.p12=ca.p12 \
     --from-literal=ca.password=P12-PASSWORD
   ```

   Replace *<cluster_name>* with the name of your Kafka cluster.

2. Create a new secret that contains the private key.

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

3. Label the secrets.

   ```
   oc label secret CA-CERTIFICATE-SECRET strimzi.io/kind=Kafka
   strimzi.io/cluster=<cluster_name>
   ```

   ```
   oc label secret CA-KEY-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=<cluster_name>
   ```

   - Label **strimzi.io/kind=Kafka** identifies the Kafka custom resource.

   - Label **strimzi.io/cluster=*<cluster_name>*** identifies the Kafka cluster.

4. Annotate the secrets

   ```
   oc annotate secret CA-CERTIFICATE-SECRET strimzi.io/ca-cert-generation=CA-
   CERTIFICATE-GENERATION
   ```

   ```
   oc annotate secret CA-KEY-SECRET strimzi.io/ca-key-generation=CA-KEY-GENERATION
   ```

   - Annotation **strimzi.io/ca-cert-generation=*CA-CERTIFICATE-GENERATION*** defines the generation of a new CA certificate.

   - Annotation **strimzi.io/ca-key-generation=*CA-KEY-GENERATION*** defines the generation of a new CA key.

Start from 0 (zero) as the incremental value (**strimzi.io/ca-cert-generation=0**) for your own CA certificate. Set a higher incremental value when you renew the certificates.

5. Create the **Kafka** resource for your cluster, configuring either the **Kafka.spec.clusterCa** or the **Kafka.spec.clientsCa** object to *not* use generated CAs.

   **Example fragment Kafka resource configuring the cluster CA to use certificates you supply for yourself**

   ```
   kind: Kafka
   version: kafka.strimzi.io/v1beta2
   spec:
     # ...
     clusterCa:
       generateCertificateAuthority: false
   ```

## Additional resources

- Section 10.8.2, "Renewing your own CA certificates".

- Section 10.8.3, "Renewing or replacing CA certificates and private keys with your own".

- Section 10.7.1, "Providing your own Kafka listener certificates for TLS encryption".

## 10.8.2. Renewing your own CA certificates

If you are using your own CA certificates, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire.

Perform the steps in this procedure when you are renewing CA certificates and continuing with the same private key. If you are renewing your own CA certificates *and* private keys, see Section 10.8.3, "Renewing or replacing CA certificates and private keys with your own".

The procedure describes the renewal of CA certificates in PEM format.

### Prerequisites

- The Cluster Operator is running.

- You have new cluster or clients X.509 certificates in PEM format.

### Procedure

1. Update the **Secret** for the CA certificate.
   Edit the existing secret to add the new CA certificate and update the certificate generation annotation value.

   ```
   oc edit secret <ca_certificate_secret_name>
   ```

   *<ca_certificate_secret_name>* is the name of the **Secret**, which is *<kafka_cluster_name>*-**cluster-ca-cert** for the cluster CA certificate and *<kafka_cluster_name>*-**clients-ca-cert** for the clients CA certificate.

The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named **my-cluster**.

**Example secret configuration for a cluster CA certificate**

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F...  1
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0"  2
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

**1** Current base64–encoded CA certificate

**2** Current CA certificate generation annotation value

2. Encode your new CA certificate into base64.

   ```
   cat <path_to_new_certificate> | base64
   ```

3. Update the CA certificate.
   Copy the base64–encoded CA certificate from the previous step as the value for the **ca.crt** property under **data**.

4. Increase the value of the CA certificate generation annotation.
   Update the **strimzi.io/ca-cert-generation** annotation with a higher incremental value. For example, change **strimzi.io/ca-cert-generation=0** to **strimzi.io/ca-cert-generation=1**. If the **Secret** is missing the annotation, the value is treated as **0**, so add the annotation with a value of **1**.

   When AMQ Streams generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The **strimzi.io/ca-cert-generation** has to be incremented on each CA certificate renewal.

5. Save the secret with the new CA certificate and certificate generation annotation value.

   **Example secret configuration updated with a new CA certificate**

   ```
   apiVersion: v1
   kind: Secret
   data:
     ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F...  1
   metadata:
     annotations:
   ```

```
    strimzi.io/ca-cert-generation: "1" 2
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

**1**     New base64-encoded CA certificate

**2**     New CA certificate generation annotation value

On the next reconciliation, the Cluster Operator performs a rolling update of ZooKeeper, Kafka, and other components to trust the new CA certificate.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

## 10.8.3. Renewing or replacing CA certificates and private keys with your own

If you are using your own CA certificates and private keys, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire. You can also use the same procedure to replace the CA certificates and private keys generated by the AMQ Streams operators with your own.

Perform the steps in this procedure when you are renewing or replacing CA certificates and private keys. If you are only renewing your own CA certificates, see Section 10.8.2, "Renewing your own CA certificates".

The procedure describes the renewal of CA certificates and private keys in PEM format.

Before going through the following steps, make sure that the CN (Common Name) of the new CA certificate is different from the current one. For example, when the Cluster Operator renews certificates automatically it adds a *v<version_number>* suffix to identify a version. Do the same with your own CA certificate by adding a different suffix on each renewal. By using a different key to generate a new CA certificate, you retain the current CA certificate stored in the **Secret**.

### Prerequisites

- The Cluster Operator is running.

- You have new cluster or clients X.509 certificates and keys in PEM format.

### Procedure

1. Pause the reconciliation of the **Kafka** custom resource.

   a. Annotate the custom resource in OpenShift, setting the **pause-reconciliation** annotation to **true**:

      ```
      oc annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="true"
      ```

      For example, for a **Kafka** custom resource named **my-cluster**:

```
oc annotate Kafka my-cluster strimzi.io/pause-reconciliation="true"
```

b. Check that the status conditions of the custom resource show a change to **ReconciliationPaused**:

```
oc describe Kafka <name_of_custom_resource>
```

The **type** condition changes to **ReconciliationPaused** at the **lastTransitionTime**.

2. Update the **Secret** for the CA certificate.

   a. Edit the existing secret to add the new CA certificate and update the certificate generation annotation value.

   ```
   oc edit secret <ca_certificate_secret_name>
   ```

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

   The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named **my-cluster**.

   **Example secret configuration for a cluster CA certificate**

   ```
   apiVersion: v1
   kind: Secret
   data:
     ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F...  1
   metadata:
     annotations:
       strimzi.io/ca-cert-generation: "0"  2
     labels:
       strimzi.io/cluster: my-cluster
       strimzi.io/kind: Kafka
     name: my-cluster-cluster-ca-cert
     #...
   type: Opaque
   ```

   **1** Current base64-encoded CA certificate

   **2** Current CA certificate generation annotation value

   b. Rename the current CA certificate to retain it.
   Rename the current **ca.crt** property under **data** as **ca-*<date>*.crt**, where *<date>* is the certificate expiry date in the format *YEAR-MONTH-DAYTHOUR-MINUTE-SECONDZ*. For example **ca-2022-01-26T17-32-00Z.crt:**. Leave the value for the property as it is to retain the current CA certificate.

   c. Encode your new CA certificate into base64.

   ```
   cat <path_to_new_certificate> | base64
   ```

   d. Update the CA certificate.
Create a new **ca.crt** property under **data** and copy the base64-encoded CA certificate from the previous step as the value for **ca.crt** property.

   e. Increase the value of the CA certificate generation annotation.
Update the **strimzi.io/ca-cert-generation** annotation with a higher incremental value. For example, change **strimzi.io/ca-cert-generation=0** to **strimzi.io/ca-cert-generation=1**. If the **Secret** is missing the annotation, the value is treated as **0**, so add the annotation with a value of **1**.

When AMQ Streams generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The **strimzi.io/ca-cert-generation** has to be incremented on each CA certificate renewal.

   f. Save the secret with the new CA certificate and certificate generation annotation value.

**Example secret configuration updated with a new CA certificate**

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F...       1
  ca-2022-01-26T17-32-00Z.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F...   2
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1"   3
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

   **1**    New base64-encoded CA certificate

   **2**    Old base64-encoded CA certificate

   **3**    New CA certificate generation annotation value

3. Update the **Secret** for the CA key used to sign your new CA certificate.

   a. Edit the existing secret to add the new CA key and update the key generation annotation value.

```
oc edit secret <ca_key_name>
```

*<ca_key_name>* is the name of CA key, which is ***<kafka_cluster_name>*-cluster-ca** for the cluster CA key and ***<kafka_cluster_name>*-clients-ca** for the clients CA key.

The following example shows a secret for a cluster CA key that's associated with a Kafka cluster named **my-cluster**.

**Example secret configuration for a cluster CA key**

```
apiVersion: v1
kind: Secret
data:
  ca.key: SA1cKF1GFDzOliPOIUQBHDNFGDFS... 1
metadata:
  annotations:
    strimzi.io/ca-key-generation: "0" 2
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque
```

**1**     Current base64-encoded CA key

**2**     Current CA key generation annotation value

b. Encode the CA key into base64.

```
cat <path_to_new_key> | base64
```

c. Update the CA key.
   Copy the base64-encoded CA key from the previous step as the value for the **ca.key** property under **data**.

d. Increase the value of the CA key generation annotation.
   Update the **strimzi.io/ca-key-generation** annotation with a higher incremental value. For example, change **strimzi.io/ca-key-generation=0** to **strimzi.io/ca-key-generation=1**. If the **Secret** is missing the annotation, it is treated as **0**, so add the annotation with a value of **1**.

   When AMQ Streams generates certificates, the key generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates together with a new CA key, set the annotation with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates and keys. The **strimzi.io/ca-key-generation** has to be incremented on each CA certificate renewal.

4. Save the secret with the new CA key and key generation annotation value.

**Example secret configuration updated with a new CA key**

```
apiVersion: v1
kind: Secret
data:
  ca.key: AB0cKF1GFDzOliPOIUQWERZJQ0F... 1
metadata:
  annotations:
    strimzi.io/ca-key-generation: "1" 2
  labels:
    strimzi.io/cluster: my-cluster
```

```
  strimzi.io/kind: Kafka
 name: my-cluster-cluster-ca
 #...
type: Opaque
```

**1** New base64-encoded CA key

**2** New CA key generation annotation value

5. Resume from the pause.
   To resume the **Kafka** custom resource reconciliation, set the **pause-reconciliation** annotation to **false**.

   ```
   oc annotate --overwrite Kafka <name_of_custom_resource> strimzi.io/pause-
   reconciliation="false"
   ```

   You can also do the same by removing the **pause-reconciliation** annotation.

   ```
   oc annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation-
   ```

On the next reconciliation, the Cluster Operator performs a rolling update of ZooKeeper, Kafka, and other components to trust the new CA certificate. When the rolling update is complete, the Cluster Operator will start a new one to generate new server certificates signed by the new CA key.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

# CHAPTER 11. MANAGING AMQ STREAMS

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

## 11.1. 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.

### 11.1.1. 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 11.1. Long and short names for each AMQ Streams resource

| AMQ Streams resource | Long name | Short name |
|---|---|---|
| Kafka | kafka | k |
| Kafka Topic | kafkatopic | kt |
| Kafka User | kafkauser | ku |
| Kafka Connect | kafkaconnect | kc |
| Kafka Connector | kafkaconnector | kctr |
| Kafka Mirror Maker | kafkamirrormaker | kmm |
| Kafka Mirror Maker 2 | kafkamirrormaker2 | kmm2 |
| Kafka Bridge | kafkabridge | kb |

| AMQ Streams resource | Long name | Short name |
|---|---|---|
| Kafka Rebalance | kafkarebalance | kr |

### 11.1.1.1. 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.

### 11.1.1.2. Querying the status of sub-resources

There are other values you can pass to the **-o** option. For example, by using **-o yaml** you get the output in YAML format. Using **-o json** will return it as JSON.

You can see all the options in **oc get --help**.

One of the most useful options is the JSONPath support, which allows you to pass JSONPath expressions to query the Kubernetes API. A JSONPath expression can extract or navigate specific parts of any resource.

For example, you can use the JSONPath expression **{.status.listeners[?(@.name=="tls")].bootstrapServers}** to get the bootstrap address from the status of the Kafka custom resource and use it in your Kafka clients.

Here, the command finds the **bootstrapServers** value of the listener named **tls**:

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

my-cluster-kafka-bootstrap.myproject.svc:9093
```

By changing the name condition you can also get the address of the other Kafka listeners.

You can use **jsonpath** to extract any other property or group of properties from any custom resource.

## 11.1.2. AMQ Streams custom resource status information

Several resources have a **status** property, as described in the following table.

Table 11.2. Custom resource status properties

| AMQ Streams resource | Schema reference | Publishes status information on... |
|---|---|---|
| **Kafka** | Section 12.2.56, "**KafkaStatus** schema reference" | The Kafka cluster. |
| **KafkaConnect** | Section 12.2.87, "**KafkaConnectStatus** schema reference" | The Kafka Connect cluster, if deployed. |
| **KafkaConnector** | Section 12.2.125, "**KafkaConnectorStatus** schema reference" | KafkaConnector resources, if deployed. |
| **KafkaMirrorMaker** | Section 12.2.113, "**KafkaMirrorMakerStatus** schema reference" | The Kafka MirrorMaker tool, if deployed. |
| **KafkaTopic** | Section 12.2.91, "**KafkaTopicStatus** schema reference" | Kafka topics in your Kafka cluster. |
| **KafkaUser** | Section 12.2.107, "**KafkaUserStatus** schema reference" | Kafka users in your Kafka cluster. |

| AMQ Streams resource | Schema reference | Publishes status information on... |
|---|---|---|
| **KafkaBridge** | Section 12.2.122, "**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:

- **KafkaStatus** provides information on listener addresses, and the id of the Kafka cluster.

- **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/v1beta2
kind: Kafka
metadata:
spec:
  # ...
status:
  conditions: 1
  - lastTransitionTime: 2021-07-23T23:46:57+0000
    status: "True"
    type: Ready 2
```

```
    observedGeneration: 4 ❸
    listeners: ❹
    - 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
    clusterId: CLUSTER-ID ❺
   # ...
```

❶ 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.

❷ The **Ready** condition indicates whether the Cluster Operator currently considers the Kafka cluster able to handle traffic.

❸ The **observedGeneration** indicates the generation of the **Kafka** custom resource that was last reconciled by the Cluster Operator.

❹ The **listeners** describe the current Kafka bootstrap addresses by type.

❺ The Kafka cluster id.

> **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 11.1.3, "Finding the status of a custom resource" .

### 11.1.3. 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 11.1.2, "AMQ Streams custom resource status information"](#)

- For more information about using JSONPath, see [JSONPath support](#).

## 11.2. PAUSING RECONCILIATION OF CUSTOM RESOURCES

Sometimes it is useful to pause the reconciliation of custom resources managed by AMQ Streams Operators, so that you can perform fixes or make updates. If reconciliations are paused, any changes made to custom resources are ignored by the Operators until the pause ends.

If you want to pause reconciliation of a custom resource, set the **strimzi.io/pause-reconciliation** annotation to **true** in its configuration. This instructs the appropriate Operator to pause reconciliation of the custom resource. For example, you can apply the annotation to the **KafkaConnect** resource so that reconciliation by the Cluster Operator is paused.

You can also create a custom resource with the pause annotation enabled. The custom resource is created, but it is ignored.

#### Prerequisites

- The AMQ Streams Operator that manages the custom resource is running.

#### Procedure

1. Annotate the custom resource in OpenShift, setting **pause-reconciliation** to **true**:

   ```
   oc annotate <kind_of_custom_resource> <name_of_custom_resource> strimzi.io/pause-reconciliation="true"
   ```

   For example, for the **KafkaConnect** custom resource:

```
oc annotate KafkaConnect my-connect strimzi.io/pause-reconciliation="true"
```

2. Check that the status conditions of the custom resource show a change to
   **ReconciliationPaused**:

   ```
   oc describe <kind_of_custom_resource> <name_of_custom_resource>
   ```

   The **type** condition changes to **ReconciliationPaused** at the **lastTransitionTime**.

   **Example custom resource with a paused reconciliation condition type**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnect
   metadata:
     annotations:
       strimzi.io/pause-reconciliation: "true"
       strimzi.io/use-connector-resources: "true"
     creationTimestamp: 2021-03-12T10:47:11Z
     #...
   spec:
     # ...
   status:
     conditions:
     - lastTransitionTime: 2021-03-12T10:47:41.689249Z
       status: "True"
       type: ReconciliationPaused
   ```

**Resuming from pause**

- To resume reconciliation, you can set the annotation to **false**, or remove the annotation.

**Additional resources**

- Customizing OpenShift resources

- Finding the status of a custom resource

## 11.3. EVICTING PODS WITH AMQ STREAMS DRAIN CLEANER

Kafka and ZooKeeper pods might be evicted during OpenShift upgrades, maintenance or pod rescheduling. If your Kafka broker and ZooKeeper pods were deployed by AMQ Streams, you can use the AMQ Streams Drain Cleaner tool to handle the pod evictions. Since the AMQ Streams Drain Cleaner will handle the eviction instead of OpenShift, you need to set the **podDisruptionBudget** for your Kafka deployment to **0** (zero). OpenShift will then no longer be allowed to evict the pod automatically.

By deploying the AMQ Streams Drain Cleaner, you can use the Cluster Operator to move Kafka pods instead of OpenShift. The Cluster Operator ensures that topics are never under–replicated. Kafka can remain operational during the eviction process. The Cluster Operator waits for topics to synchronize, as the OpenShift worker nodes drain consecutively.

An admission webhook notifies the AMQ Streams Drain Cleaner of pod eviction requests to the Kubernetes API. The AMQ Streams Drain Cleaner then adds a rolling update annotation to the pods to be drained. This informs the Cluster Operator to perform a rolling update of an evicted pod.

> **NOTE**
>
> If you are not using the AMQ Streams Drain Cleaner, you can add pod annotations to perform rolling updates manually.

## Webhook configuration

The AMQ Streams Drain Cleaner deployment files include a **ValidatingWebhookConfiguration** resource file. The resource provides the configuration for registering the webhook with the Kubernetes API.

The configuration defines the **rules** for the Kubernetes API to follow in the event of a pod eviction request. The rules specify that only **CREATE** operations related to **pods/eviction** sub-resources are intercepted. If these rules are met, the API forwards the notification.

The **clientConfig** points to the AMQ Streams Drain Cleaner service and **/drainer** endpoint that exposes the webhook. The webhook uses a secure TLS connection, which requires authentication. The **caBundle** property specifies the certificate chain to validate HTTPS communication. Certificates are encoded in Base64.

## Webhook configuration for pod eviction notifications

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
# ...
webhooks:
  - name: strimzi-drain-cleaner.strimzi.io
    rules:
      - apiGroups:   [""]
        apiVersions: ["v1"]
        operations:  ["CREATE"]
        resources:   ["pods/eviction"]
        scope:       "Namespaced"
    clientConfig:
      service:
        namespace: "strimzi-drain-cleaner"
        name: "strimzi-drain-cleaner"
        path: /drainer
        port: 443
        caBundle: Cg==
    # ...
```

### 11.3.1. Prerequisites

To deploy and use the AMQ Streams Drain Cleaner, you need to download the deployment files.

The AMQ Streams Drain Cleaner deployment files are available from the AMQ Streams software downloads page.

### 11.3.2. Deploying the AMQ Streams Drain Cleaner

Deploy the AMQ Streams Drain Cleaner to the OpenShift cluster where the Cluster Operator and Kafka cluster are running.

**Prerequisites**

- You have downloaded the AMQ Streams Drain Cleaner deployment files .

- You have a highly available Kafka cluster deployment running with OpenShift worker nodes that you would like to update.

- Topics are replicated for high availability.
  Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

### Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

## Excluding ZooKeeper

If you don't want to include ZooKeeper, you can remove the **--zookeeper** command option from the AMQ Streams Drain Cleaner **Deployment** configuration file.

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-drain-cleaner
      containers:
        - name: strimzi-drain-cleaner
          # ...
          command:
            - "/application"
            - "-Dquarkus.http.host=0.0.0.0"
            - "--kafka"
            - "--zookeeper"    1
          # ...
```

1 Remove this option to exclude ZooKeeper from AMQ Streams Drain Cleaner operations.

### Procedure

1. Configure a pod disruption budget of **0** (zero) for your Kafka deployment using **template** settings in the **Kafka** resource.

### Specifying a pod disruption budget

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    template:
      podDisruptionBudget:
        maxUnavailable: 0

    # ...
  zookeeper:
    template:
      podDisruptionBudget:
        maxUnavailable: 0
    # ...
```

Reducing the maximum pod disruption budget to zero prevents OpenShift from automatically evicting the pods in case of voluntary disruptions, leaving the AMQ Streams Drain Cleaner and AMQ Streams Cluster Operator to roll the pod which will be scheduled by OpenShift on a different worker node.

Add the same configuration for ZooKeeper if you want to use AMQ Streams Drain Cleaner to drain ZooKeeper nodes.

2. Update the **Kafka** resource:

   ```
   oc apply -f <kafka-configuration-file>
   ```

3. Deploy the AMQ Streams Drain Cleaner.

   - To run the Drain Cleaner on OpenShift, apply the resources in the **/install/drain-cleaner/openshift** directory.

     ```
     oc apply -f ./install/drain-cleaner/openshift
     ```

## 11.3.3. Using the AMQ Streams Drain Cleaner

Use the AMQ Streams Drain Cleaner in combination with the Cluster Operator to move Kafka broker or ZooKeeper pods from nodes that are being drained. When you run the AMQ Streams Drain Cleaner, it annotates pods with a rolling update pod annotation. The Cluster Operator performs rolling updates based on the annotation.

### Prerequisites

- You have deployed the AMQ Streams Drain Cleaner .

### Procedure

1. Drain a specified OpenShift node hosting the Kafka broker or ZooKeeper pods.

```
oc get nodes
oc drain <name-of-node> --delete-emptydir-data --ignore-daemonsets --timeout=6000s --
force
```

2. Check the eviction events in the AMQ Streams Drain Cleaner log to verify that the pods have been annotated for restart.

**AMQ Streams Drain Cleaner log show annotations of pods**

```
INFO ... Received eviction webhook for Pod my-cluster-zookeeper-2 in namespace my-
project
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project will be annotated for restart
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project found and annotated for
restart

INFO ... Received eviction webhook for Pod my-cluster-kafka-0 in namespace my-project
INFO ... Pod my-cluster-kafka-0 in namespace my-project will be annotated for restart
INFO ... Pod my-cluster-kafka-0 in namespace my-project found and annotated for restart
```

3. Check the reconciliation events in the Cluster Operator log to verify the rolling updates.

**Cluster Operator log shows rolling updates**

```
INFO  PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster): Rolling Pod
my-cluster-zookeeper-2
INFO  PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster): Rolling Pod
my-cluster-kafka-0
INFO  AbstractOperator:500 - Reconciliation #13(timer) Kafka(my-project/my-cluster):
reconciled
```

# 11.4. MANUALLY STARTING ROLLING UPDATES OF KAFKA AND ZOOKEEPER CLUSTERS

AMQ Streams supports the use of annotations on resources to manually trigger a rolling update of Kafka and ZooKeeper clusters through the Cluster Operator. Rolling updates restart the pods of the resource with new ones.

Manually performing a rolling update on a specific pod or set of pods is usually only required in exceptional circumstances. However, rather than deleting the pods directly, if you perform the rolling update through the Cluster Operator you ensure the following:

- The manual deletion of the pod does not conflict with simultaneous Cluster Operator operations, such as deleting other pods in parallel.

- The Cluster Operator logic handles the Kafka configuration specifications, such as the number of in–sync replicas.

## 11.4.1. Prerequisites

To perform a manual rolling update, you need a running Cluster Operator and Kafka cluster.

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

- Cluster Operator

- Kafka cluster

## 11.4.2. Performing a rolling update using a pod management annotation

This procedure describes how to trigger a rolling update of a Kafka cluster or ZooKeeper cluster.

To trigger the update, you add an annotation to the resource you are using to manage the pods running on the cluster. You annotate the **StatefulSet** or **StrimziPodSet** resource (if you enabled the UseStrimziPodSets feature gate).

### Procedure

1. Find the name of the resource that controls the Kafka or ZooKeeper pods you want to manually update.
   For example, if your Kafka cluster is named *my-cluster*, the corresponding names are *my-cluster-kafka* and *my-cluster-zookeeper*.

2. Use **oc annotate** to annotate the appropriate resource in OpenShift.

   **Annotating a StatefulSet**

   ```
   oc annotate statefulset <cluster_name>-kafka strimzi.io/manual-rolling-update=true

   oc annotate statefulset <cluster_name>-zookeeper strimzi.io/manual-rolling-update=true
   ```

   **Annotating a StrimziPodSet**

   ```
   oc annotate strimzipodset <cluster_name>-kafka strimzi.io/manual-rolling-update=true

   oc annotate strimzipodset <cluster_name>-zookeeper strimzi.io/manual-rolling-update=true
   ```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of all pods within the annotated resource is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of all the pods is complete, the annotation is removed from the resource.

## 11.4.3. Performing a rolling update using a Pod annotation

This procedure describes how to manually trigger a rolling update of an existing Kafka cluster or ZooKeeper cluster using an OpenShift **Pod** annotation. When multiple pods are annotated, consecutive rolling updates are performed within the same reconciliation run.

### Prerequisites

You can perform a rolling update on a Kafka cluster regardless of the topic replication factor used. But for Kafka to stay operational during the update, you'll need the following:

- A highly available Kafka cluster deployment running with nodes that you wish to update.

- Topics replicated for high availability.
  Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

### Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

#### Procedure

1. Find the name of the Kafka or ZooKeeper **Pod** you want to manually update.
   For example, if your Kafka cluster is named *my-cluster*, the corresponding **Pod** names are *my-cluster-kafka-index* and *my-cluster-zookeeper-index*. The *index* starts at zero and ends at the total number of replicas minus one.

2. Annotate the **Pod** resource in OpenShift.
   Use **oc annotate**:

   ```
   oc annotate pod cluster-name-kafka-index strimzi.io/manual-rolling-update=true
   ```

   ```
   oc annotate pod cluster-name-zookeeper-index strimzi.io/manual-rolling-update=true
   ```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of the annotated **Pod** is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of a pod is complete, the annotation is removed from the **Pod**.

## 11.5. 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
```

### 11.5.1. 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.

## 11.6. 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

## 11.6.1. 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
```
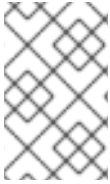
> **NOTE**
>
> If you are using *Retain* as the reclaim policy, but you want to delete an entire cluster, you need to delete the PVs manually. Otherwise they will not be deleted, and may cause unnecessary expenditure on resources.

## 11.6.2. 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.

## 11.6.3. Recovering a deleted cluster from persistent volumes

This procedure describes how to recover a deleted cluster from persistent volumes (PVs).

In this situation, the Topic Operator identifies that topics exist in Kafka, but the **KafkaTopic** resources do not exist.

When you get to the step to recreate your cluster, you have two options:

1. Use *Option 1* when you can recover all **KafkaTopic** resources.
   The **KafkaTopic** resources must therefore be recovered before the cluster is started so that the corresponding topics are not deleted by the Topic Operator.

2. Use *Option 2* when you are unable to recover all **KafkaTopic** resources.
   In this case, you deploy your cluster without the Topic Operator, delete the Topic Operator topic store metadata, and then redeploy the Kafka cluster with the Topic Operator so it can recreate the **KafkaTopic** resources from the corresponding topics.

> **NOTE**
>
> If the Topic Operator is not deployed, you only need to recover the **PersistentVolumeClaim** (PVC) resources.

### Before you begin

In this procedure, it is essential that PVs are mounted into the correct PVC to avoid data corruption. A **volumeName** is specified for the PVC and this must match the name of the PV.

For more information, see:

- Persistent 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 create -f install/cluster-operator -n my-project
```

6. Recreate your cluster.
   Follow the steps depending on whether or not you have all the **KafkaTopic** resources needed to recreate your cluster.

   *Option 1*: If you have **all** the **KafkaTopic** resources that existed before you lost your cluster, including internal topics such as committed offsets from **__consumer_offsets**:

   1. Recreate all **KafkaTopic** resources.
      It is essential that you recreate the resources before deploying the cluster, or the Topic Operator will delete the topics.

   2. Deploy the Kafka cluster.
      For example:

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

   *Option 2*: If you do not have all the **KafkaTopic** resources that existed before you lost your cluster:

   1. Deploy the Kafka cluster, as with the first option, but without the Topic Operator by removing the **topicOperator** property from the Kafka resource before deploying.
      If you include the Topic Operator in the deployment, the Topic Operator will delete all the topics.

   2. Delete the internal topic store topics from the Kafka cluster:

      ```
      oc run kafka-admin -ti --image=registry.redhat.io/amq7/amq-streams-kafka-33-
      rhel8:2.3.0 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server
      localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete
      && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --
      delete
      ```

      The command must correspond to the type of listener and authentication used to access the Kafka cluster.

   3. Enable the Topic Operator by redeploying the Kafka cluster with the **topicOperator** property to recreate the **KafkaTopic** resources.
      For example:

      ```
      apiVersion: kafka.strimzi.io/v1beta2
      kind: Kafka
      metadata:
        name: my-cluster
      spec:
        #...
        entityOperator:
          topicOperator: {}  1
          #...
      ```

   **1** Here we show the default configuration, which has no additional properties. You specify the required configuration using the properties described in Section 12.2.45, "**EntityTopicOperatorSpec** schema reference".

7. Verify the recovery by listing the **KafkaTopic** resources:

```
oc get KafkaTopic
```

## 11.7. SETTING LIMITS ON BROKERS USING THE KAFKA STATIC QUOTA PLUGIN

Use the *Kafka Static Quota* plugin to set throughput and storage limits on brokers in your Kafka cluster. You enable the plugin and set limits by configuring the **Kafka** resource. You can set a byte-rate threshold and storage quotas to put limits on the clients interacting with your brokers.

You can set byte-rate thresholds for producer and consumer bandwidth. The total limit is distributed across all clients accessing the broker. For example, you can set a byte-rate threshold of 40 MBps for producers. If two producers are running, they are each limited to a throughput of 20 MBps.

Storage quotas throttle Kafka disk storage limits between a soft limit and hard limit. The limits apply to all available disk space. Producers are slowed gradually between the soft and hard limit. The limits prevent disks filling up too quickly and exceeding their capacity. Full disks can lead to issues that are hard to rectify. The hard limit is the maximum storage limit.

> **NOTE**
>
> For JBOD storage, the limit applies across all disks. If a broker is using two 1 TB disks and the quota is 1.1 TB, one disk might fill and the other disk will be almost empty.

**Prerequisites**

- The Cluster Operator that manages the Kafka cluster is running.

**Procedure**

1. Add the plugin properties to the **config** of the **Kafka** resource.
   The plugin properties are shown in this example configuration.

   **Example Kafka Static Quota plugin configuration**

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       config:
         client.quota.callback.class: io.strimzi.kafka.quotas.StaticQuotaCallback    1
         client.quota.callback.static.produce: 1000000    2
         client.quota.callback.static.fetch: 1000000    3
         client.quota.callback.static.storage.soft: 400000000000    4
         client.quota.callback.static.storage.hard: 500000000000    5
         client.quota.callback.static.storage.check-interval: 5    6
   ```

   **1**    Loads the Kafka Static Quota plugin.

**2** Sets the producer byte-rate threshold. 1 MBps in this example.

**3** Sets the consumer byte-rate threshold. 1 MBps in this example.

**4** Sets the lower soft limit for storage. 400 GB in this example.

**5** Sets the higher hard limit for storage. 500 GB in this example.

**6** Sets the interval in seconds between checks on storage. 5 seconds in this example. You can set this to 0 to disable the check.

2. Update the resource.

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

**Additional resources**

- Setting user quotas

## 11.8. FREQUENTLY ASKED QUESTIONS

### 11.8.1. Questions related to the Cluster Operator

#### 11.8.1.1. 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?

#### 11.8.1.2. 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**.

### 11.8.1.3. 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.

### 11.8.1.4. 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.

### 11.8.1.5. Why is hostname verification failing when connecting to NodePorts using TLS?

Currently, off-cluster access using NodePorts with TLS encryption enabled does not support TLS hostname verification. As a result, the clients that verify the hostname will fail to connect. For example, the Java client will fail with the following exception:

```
Caused by: java.security.cert.CertificateException: No subject alternative names matching IP address
168.72.15.231 found
 at sun.security.util.HostnameChecker.matchIP(HostnameChecker.java:168)
 at sun.security.util.HostnameChecker.match(HostnameChecker.java:94)
 at sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:455)
 at sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:436)
 at sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:252)
 at sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:136)
 at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1501)
 ... 17 more
```

To connect, you must disable hostname verification. In the Java client, you can do this by setting the configuration option **ssl.endpoint.identification.algorithm** to an empty string.

When configuring the client using a properties file, you can do it this way:

```
ssl.endpoint.identification.algorithm=
```

When configuring the client directly in Java, set the configuration option to an empty string:

```
props.put("ssl.endpoint.identification.algorithm", "");
```

# CHAPTER 12. CUSTOM RESOURCE API REFERENCE

## 12.1. COMMON CONFIGURATION PROPERTIES

Common configuration properties apply to more than one resource.

### 12.1.1. 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.

### 12.1.2. 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, ingress, 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.

### 12.1.3. 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 encryption 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.

### 12.1.4. 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 encryption, 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 mTLS authentication, see the **KafkaClientAuthenticationTls** schema reference.

### 12.1.5. resources

Configure resource *requests* and *limits* to control resources for AMQ Streams containers. You can specify requests and limits for **memory** and **cpu** resources. The requests should be enough to ensure a stable performance of Kafka.

How you configure resources in a production environment depends on a number of factors. For example, applications are likely to be sharing resources in your OpenShift cluster.

For Kafka, the following aspects of a deployment can impact the resources you need:

- Throughput and size of messages

- The number of network threads handling messages

- The number of producers and consumers

- The number of topics and partitions

The values specified for resource requests are reserved and always available to the container. Resource limits specify the maximum resources that can be consumed by a given container. The amount between the request and limit is not reserved and might not be always available. A container can use the resources up to the limit only when they are available. Resource limits are temporary and can be reallocated.

**Resource requests and limits**
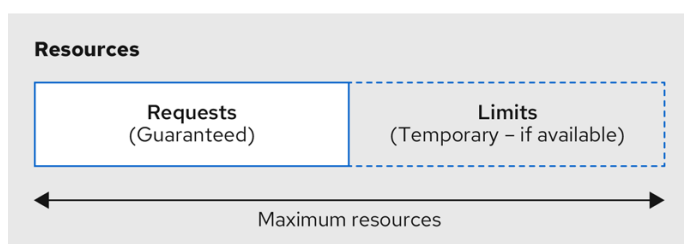


If you set limits without requests or vice versa, OpenShift uses the same value for both. Setting equal requests and limits for resources guarantees quality of service, as OpenShift will not kill containers unless they exceed their limits.

You can configure resource requests and limits for one or more supported resources.

**Example resource configuration**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    #...
    resources:
      requests:
        memory: 64Gi
        cpu: "8"
      limits:
        memory: 64Gi
        cpu: "12"
  entityOperator:
```

```
#...
topicOperator:
  #...
  resources:
    requests:
      memory: 512Mi
      cpu: "1"
    limits:
      memory: 512Mi
      cpu: "1"
```

Resource requests and limits for the Topic Operator and User Operator are set in the **Kafka** resource.

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

> **NOTE**
>
> AMQ Streams uses the OpenShift syntax for specifying **memory** and **cpu** resources. For more information about managing computing resources on OpenShift, see Managing Compute Resources for Containers.

**Memory resources**

When configuring memory resources, consider the total requirements of the components. Kafka runs inside a JVM and uses an operating system page cache to store message data before writing to disk. The memory request for Kafka should fit the JVM heap and page cache. You can configure the **jvmOptions** property to control the minimum and maximum heap size.

Other components don't rely on the page cache. You can configure memory resources without configuring the **jvmOptions** to control the heap size.

Memory requests and limits are specified in megabytes, gigabytes, mebibytes, and gibibytes. Use the following suffixes in the specification:

- **M** for megabytes

- **G** for gigabytes

- **Mi** for mebibytes

- **Gi** for gibibytes

**Example resources using different memory units**

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

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

CPU resources

A CPU request should be enough to give a reliable performance at any time. CPU requests and limits are specified as *cores* or *millicpus/millicores*.

CPU cores are specified as integers (**5** CPU core) or decimals (**2.5** CPU core). 1000 *millicores* is the same as **1** CPU core.

Example CPU units

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

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

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

### 12.1.6. 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**

- **KafkaMirrorMaker.spec**

- **KafkaMirrorMaker2.spec**

- **KafkaBridge.spec**

**Configuring the image property for Kafka, Kafka Connect, and Kafka MirrorMaker**

Kafka, Kafka Connect, 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_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 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-rhel8-operator:2.3.0** container image.

- For User Operator:

    1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.

    2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0** container image.

- For Entity Operator TLS sidecar:

    1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.

    2. **registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0** container image.

- For Kafka Exporter:

    1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE** environment variable from the Cluster Operator configuration.

    2. **registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0** container image.

- For Kafka Bridge:

    1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE** environment variable from the Cluster Operator configuration.

    2. **registry.redhat.io/amq7/amq-streams-bridge-rhel8:2.3.0** container image.

- For Kafka broker initializer:

    1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** environment variable from the Cluster Operator configuration.

    2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0** container image.

## Example container image configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

## 12.1.7. **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 the Probe schema reference.

### 12.1.8. metricsConfig

Use the **metricsConfig** property to enable and configure Prometheus metrics.

The **metricsConfig** property contains a reference to a ConfigMap that has additional configurations 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 further configuration, you can reference a ConfigMap containing an empty file under **metricsConfig.valueFrom.configMapKeyRef.key**. When referencing an empty file, all metrics are exposed as long as they have not been renamed.

### Example ConfigMap with metrics configuration for Kafka

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: my-configmap
data:
  my-key: |
    lowercaseOutputName: true
    rules:
    # 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"
# further configuration
```

## Example metrics configuration for Kafka

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: my-key
    # ...
  zookeeper:
    # ...
```

When metrics are enabled, they are exposed on port 9404.

When the **metricsConfig** (or deprecated **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.

## 12.1.9. **jvmOptions**

The following AMQ Streams components run inside a Java Virtual Machine (JVM):

- Apache Kafka

- Apache ZooKeeper

- Apache Kafka Connect

- Apache Kafka MirrorMaker

- AMQ Streams Kafka Bridge

To optimize their performance on different platforms and architectures, you configure the **jvmOptions** property in the following resources:

- **Kafka.spec.kafka**

- **Kafka.spec.zookeeper**

- **Kafka.spec.entityOperator.userOperator**

- **Kafka.spec.entityOperator.topicOperator**

- **Kafka.spec.cruiseControl**

- **KafkaConnect.spec**

- **KafkaMirrorMaker.spec**

- **KafkaMirrorMaker2.spec**

- **KafkaBridge.spec**

You can specify the following options in your configuration:

**-Xms**

Minimum initial allocation heap size when the JVM starts

**-Xmx**

Maximum heap size

**-XX**

Advanced runtime options for the JVM

**javaSystemProperties**

Additional system properties

**gcLoggingEnabled**

Enables garbage collector logging
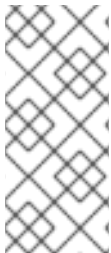
> **NOTE**
>
> The units accepted by JVM settings, such as **-Xmx** and **-Xms**, are the same units accepted by the JDK **java** binary in the corresponding image. Therefore, **1g** or **1G** means 1,073,741,824 bytes, and **Gi** is not a valid unit suffix. This is different from the units used for 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.

**-Xms and -Xmx options**

In addition to setting memory request and limit values for your containers, you can use the **-Xms** and **-Xmx** JVM options to set specific heap sizes for your JVM. Use the **-Xms** option to set an initial heap size and the **-Xmx** option to set a maximum heap size.

Specify heap size to have more control over the memory allocated to your JVM. Heap sizes should make the best use of a container's memory limit (and request) without exceeding it. Heap size and any other memory requirements need to fit within a specified memory limit. If you don't specify heap size in your configuration, but you configure a memory resource limit (and request), the Cluster Operator imposes default heap sizes automatically. The Cluster Operator sets default maximum and minimum heap values based on a percentage of the memory resource configuration.

The following table shows the default heap values.

**Table 12.1. Default heap settings for components**

| Component | Percent of available memory allocated to the heap | Maximum limit |
|---|---|---|
| Kafka | 50% | 5 GB |
| ZooKeeper | 75% | 2 GB |
| Kafka Connect | 75% | None |
| MirrorMaker 2.0 | 75% | None |
| MirrorMaker | 75% | None |
| Cruise Control | 75% | None |
| Kafka Bridge | 50% | 31 Gi |

If a memory limit (and request) is not specified, a JVM's minimum heap size is set to **128M**. The JVM's maximum heap size is not defined to allow the memory to increase as needed. This is ideal for single node environments in test and development.

Setting an appropriate memory request can prevent the following:

- OpenShift killing a container if there is pressure on memory from other pods running on the node.

- OpenShift scheduling a container to a node with insufficient memory. If **-Xms** is set to **-Xmx**, the container will crash immediately; if not, the container will crash at a later time.

In this example, the JVM uses 2 GiB (=2,147,483,648 bytes) for its heap. Total JVM memory usage can be a lot more than the maximum heap size.

### Example -Xmx and -Xms configuration

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

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.

> **IMPORTANT**
>
> Containers performing lots of disk I/O, such as Kafka broker containers, require available memory for use as an operating system page cache. For such containers, the requested memory should be significantly higher than the memory used by the JVM.

### -XX option

**-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

**Example -XX configuration**

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

**JVM options resulting from the -XX configuration**

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

> **NOTE**
>
> When no **-XX** options are specified, the default Apache Kafka configuration of **KAFKA_JVM_PERFORMANCE_OPTS** is used.

**javaSystemProperties**

**javaSystemProperties** are used to configure additional Java system properties, such as debugging utilities.

**Example javaSystemProperties configuration**

```
jvmOptions:
  javaSystemProperties:
    - name: javax.net.debug
      value: ssl
```

For more information about the **jvmOptions**, see the **JvmOptions** schema reference.

## 12.1.10. 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 GC logging configuration**

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

## 12.2. SCHEMA PROPERTIES

### 12.2.1. Kafka schema reference

| Property | Description |
| --- | --- |
| spec<br><br>**KafkaSpec** | The specification of the Kafka and ZooKeeper clusters, and Topic Operator. |
| status<br><br>**KafkaStatus** | The status of the Kafka and ZooKeeper clusters, and Topic Operator. |

## 12.2.2. KafkaSpec schema reference

Used in: **Kafka**

| Property | Description |
| --- | --- |
| kafka<br><br>**KafkaClusterSpec** | Configuration of the Kafka cluster. |
| zookeeper<br><br>**ZookeeperClusterSpec** | Configuration of the ZooKeeper cluster. |
| entityOperator<br><br>**EntityOperatorSpec** | Configuration of the Entity Operator. |
| clusterCa<br><br>**CertificateAuthority** | Configuration of the cluster certificate authority. |
| clientsCa<br><br>**CertificateAuthority** | Configuration of the clients certificate authority. |
| cruiseControl<br><br>**CruiseControlSpec** | Configuration for Cruise Control deployment. Deploys a Cruise Control instance when specified. |
| kafkaExporter<br><br>**KafkaExporterSpec** | Configuration of the Kafka Exporter. Kafka Exporter can provide additional metrics, for example lag of consumer group at topic/partition. |

| Property | Description |
| --- | --- |
| maintenanceTimeWindows | A list of time windows for maintenance tasks (that is, certificates renewal). Each time window is defined by a cron expression. |
| string array | |

### 12.2.3. KafkaClusterSpec schema reference

Used in: **KafkaSpec**

Full list of **KafkaClusterSpec** schema properties

Configures a Kafka cluster.

#### 12.2.3.1. 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/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
    # ...
  zookeeper:
    # ...
```

#### 12.2.3.2. config

Use the **config** properties to configure Kafka broker options as keys.

Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

Configuration options that cannot be configured relate to:

- Security (Encryption, Authentication, and Authorization)

- Listener configuration

- Broker ID configuration

- Configuration of log data directories

- Inter-broker communication

- ZooKeeper connectivity

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, 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/v1beta2
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
    # ...
```

### 12.2.3.3. brokerRackInitImage

When rack awareness is enabled, Kafka broker pods use init container to collect the labels from the OpenShift cluster nodes. The container image used for this container can be configured using the **brokerRackInitImage** property. When the **brokerRackInitImage** field is missing, the following images are used in order of priority:

1. Container image specified in **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** environment variable in the Cluster Operator configuration.

2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0** container image.

Example **brokerRackInitImage** configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    brokerRackInitImage: my-org/my-image:latest
    # ...
```

**NOTE**

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 registry used by AMQ Streams. In this 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.

### 12.2.3.4. logging

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**

Kafka 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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**. Both **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. A ConfigMap using the exact logging configuration specified is created with the custom resource when the Cluster Operator is running, then recreated after each reconciliation. If you do not specify a custom ConfigMap, default logging settings are used. If a specific logger value is not set, upper-level logger settings are inherited for that logger. 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/v1beta2
kind: Kafka
```

```
spec:
  # ...
  kafka:
    # ...
    logging:
      type: inline
      loggers:
        kafka.root.logger.level: "INFO"
    # ...
```

## External logging

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: kafka-log4j.properties
  # ...
```

Any available loggers that are not configured have their level set to **OFF**.

If Kafka was deployed using the Cluster Operator, changes to Kafka 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.

### 12.2.3.5. **KafkaClusterSpec** schema properties

| Property | Description |
| --- | --- |
| version<br><br>string | The kafka broker version. Defaults to 3.3.1. Consult the user documentation to understand the process required to upgrade or downgrade the version. |
| replicas<br><br>integer | The number of pods in the cluster. |
| image<br><br>string | The docker image for the pods. The default value depends on the configured **Kafka.spec.kafka.version**. |
| listeners | Configures listeners of Kafka brokers. |

| Property | Description |
|---|---|
| **GenericKafkaListener** array | |
| config | 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., log.dir, zookeeper.connect, zookeeper.set.acl, zookeeper.ssl, zookeeper.clientCnxnSocket, authorizer., super.user, cruise.control.metrics.topic, cruise.control.metrics.reporter.bootstrap.servers,node.id, process.roles, controller. (with the exception of: zookeeper.connection.timeout.ms, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols, sasl.server.max.receive.size,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,cruise.control.metrics.topic.min.insync.replicas,controller.quorum.election.backoff.max.ms, controller.quorum.election.timeout.ms, controller.quorum.fetch.timeout.ms). |
| map | |
| 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, jbod]. |
| **EphemeralStorage**, **PersistentClaimStorage**, **JbodStorage** | |
| authorization | 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, custom]. |
| **KafkaAuthorizationSimple**, **KafkaAuthorizationOpa**, **KafkaAuthorizationKeycloak**, **KafkaAuthorizationCustom** | |
| rack | Configuration of the **broker.rack** broker config. |
| **Rack** | |
| brokerRackInitImage | The image of the init container used for initializing the **broker.rack**. |
| string | |
| livenessProbe | Pod liveness checking. |
| **Probe** | |

| Property | Description |
| --- | --- |
| readinessProbe<br><br>**Probe** | Pod readiness checking. |
| jvmOptions<br><br>**JvmOptions** | JVM Options for pods. |
| jmxOptions<br><br>**KafkaJmxOptions** | JMX Options for Kafka brokers. |
| resources<br><br>ResourceRequirements | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |
| metricsConfig<br><br>**JmxPrometheusExporterMetrics** | Metrics configuration. The type depends on the value of the **metricsConfig.type** property within the given object, which must be one of [jmxPrometheusExporter]. |
| logging<br><br>**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]. |
| template<br><br>**KafkaClusterTemplate** | Template for Kafka cluster resources. The template allows users to specify how the **StatefulSet**, **Pods**, and **Services** are generated. |

### 12.2.4. GenericKafkaListener schema reference

Used in: **KafkaClusterSpec**

Full list of **GenericKafkaListener** schema properties

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/v1beta2
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: 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
#...
```

### 12.2.4.1. 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.

### 12.2.4.2. type

The type is set as **internal**, or for external listeners, as **route**, **loadbalancer**, **nodeport**, **ingress** or **cluster-ip**. You can also configure a **cluster-ip** listener, a type of internal listener you can use to build custom access mechanisms.

**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 Ingress NGINX 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: 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
    #...
```

**NOTE**

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

**loadbalancer**

Configures an external listener to expose Kafka using a **Loadbalancer** type **Service**. 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 a **NodePort** type **Service**.
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.

## cluster-ip

Configures an internal listener to expose Kafka using a per-broker **ClusterIP** type **Service**.
The listener does not use a headless service and its DNS names to route traffic to Kafka brokers. You can use this type of listener to expose a Kafka cluster when using the headless service is unsuitable. You might use it with a custom access mechanism, such as one that uses a specific Ingress controller or the OpenShift Gateway API.

A new **ClusterIP** service is created for each Kafka broker pod. The service is assigned a **ClusterIP** address to serve as a Kafka *bootstrap* address with a per-broker port number. For example, you can configure the listener to expose a Kafka cluster over an Nginx Ingress Controller with TCP port configuration.

**Example cluster-ip listener configuration**

```
#...
spec:
  kafka:
    #...
    listeners:
      - name: external-cluster-ip
        type: cluster-ip
        tls: false
        port: 9096
    #...
```

### 12.2.4.3. 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** and **cluster-ip** 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[?
(@.name=="<listener_name>")].bootstrapServers}{"\n"}'
```

> **NOTE**
>
> Listeners cannot be configured to use the ports set aside for interbroker communication (9090 and 9091) and metrics (9404).

### 12.2.4.4. tls

The TLS property is required.

By default, TLS encryption is not enabled. To enable it, set the **tls** property to **true**.

For **route** and **ingress** type listeners, TLS encryption must be enabled.

### 12.2.4.5. authentication

Authentication for the listener can be specified as:

- mTLS (**tls**)

- SCRAM-SHA-512 (**scram-sha-512**)

- Token-based OAuth 2.0 (**oauth**)

- Custom (**custom**)

### 12.2.4.6. 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.

In the following 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** property is the same as the **from** property in **NetworkPolicy** resources.

Exanmple network policy configuration

```
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
  # ...
```

### 12.2.4.7. GenericKafkaListener schema properties

| Property | Description |
|---|---|
| name<br><br>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<br><br>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<br><br>string (one of [ingress, internal, route, loadbalancer, cluster-ip, nodeport]) | Type of the listener. Currently the supported types are **internal**, **route**, **loadbalancer**, **nodeport** and **ingress**.<br><br>- **internal** type exposes Kafka internally only within the OpenShift cluster.<br>- **route** type uses OpenShift Routes to expose Kafka.<br>- **loadbalancer** type uses LoadBalancer type services to expose Kafka.<br>- **nodeport** type uses NodePort type services to expose Kafka.<br>- **ingress** type uses OpenShift Nginx Ingress to expose Kafka with TLS passthrough.<br>- **cluster-ip** type uses a per-broker **ClusterIP** service. |
| tls<br><br>boolean | Enables TLS encryption on the listener. This is a required property. |
| authentication<br><br>**KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**, **KafkaListenerAuthenticationCustom** | 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, custom]. |
| configuration | Additional listener configuration. |

| Property | Description |
|---|---|
| **GenericKafkaListenerConfiguration** | |
| networkPolicyPeers<br><br>**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. For more information, see the external documentation for networking.k8s.io/v1 networkpolicypeer. |

### 12.2.5. KafkaListenerAuthenticationTls schema reference

Used in: **GenericKafkaListener**

The **type** property is a discriminator that distinguishes use of the **KafkaListenerAuthenticationTls** type from **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**, **KafkaListenerAuthenticationCustom**. It must have the value **tls** for the type **KafkaListenerAuthenticationTls**.

| Property | Description |
|---|---|
| type<br><br>string | Must be **tls**. |

### 12.2.6. KafkaListenerAuthenticationScramSha512 schema reference

Used in: **GenericKafkaListener**

The **type** property is a discriminator that distinguishes use of the **KafkaListenerAuthenticationScramSha512** type from **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationOAuth**, **KafkaListenerAuthenticationCustom**. It must have the value **scram-sha-512** for the type **KafkaListenerAuthenticationScramSha512**.

| Property | Description |
|---|---|
| type<br><br>string | Must be **scram-sha-512**. |

### 12.2.7. KafkaListenerAuthenticationOAuth schema reference

Used in: **GenericKafkaListener**

The **type** property is a discriminator that distinguishes use of the **KafkaListenerAuthenticationOAuth** type from **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationCustom**. It must have the value **oauth** for the type **KafkaListenerAuthenticationOAuth**.

| Property | Description |
| --- | --- |
| accessTokenIsJwt<br><br>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<br><br>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**. |
| checkAudience<br><br>boolean | Enable or disable audience checking. Audience checks identify the recipients of tokens. If audience checking is enabled, the OAuth Client ID also has to be configured using the **clientId** property. The Kafka broker will reject tokens that do not have its **clientId** in their **aud** (audience) claim.Default value is **false**. |
| checkIssuer<br><br>boolean | Enable or disable issuer checking. By default issuer is checked using the value configured by **validIssuerUri**. Default value is **true**. |
| clientAudience<br><br>string | The audience to use when making requests to the authorization server's token endpoint. Used for inter-broker authentication and for configuring OAuth 2.0 over PLAIN using the **clientId** and **secret** method. |
| clientId<br><br>string | OAuth Client ID which the Kafka broker can use to authenticate against the authorization server and use the introspect endpoint URI. |
| clientScope<br><br>string | The scope to use when making requests to the authorization server's token endpoint. Used for inter-broker authentication and for configuring OAuth 2.0 over PLAIN using the **clientId** and **secret** method. |
| clientSecret<br><br>**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. |
| connectTimeoutSeconds<br><br>integer | The connect timeout in seconds when connecting to authorization server. If not set, the effective connect timeout is 60 seconds. |

| Property | Description |
|---|---|
| customClaimCheck<br><br>string | JsonPath filter query to be applied to the JWT token or to the response of the introspection endpoint for additional token validation. Not set by default. |
| disableTlsHostnameVerification<br><br>boolean | Enable or disable TLS hostname verification. Default value is **false**. |
| enableECDSA<br><br>boolean | The **enableECDSA property has been deprecated.** Enable or disable ECDSA support by installing BouncyCastle crypto provider. ECDSA support is always enabled. The BouncyCastle libraries are no longer packaged with AMQ Streams. Value is ignored. |
| enableMetrics<br><br>boolean | Enable or disable OAuth metrics. Default value is **false**. |
| enableOauthBearer<br><br>boolean | Enable or disable OAuth authentication over SASL_OAUTHBEARER. Default value is **true**. |
| enablePlain<br><br>boolean | Enable or disable OAuth authentication over SASL_PLAIN. There is no re-authentication support when this mechanism is used. Default value is **false**. |
| failFast<br><br>boolean | Enable or disable termination of Kafka broker processes due to potentially recoverable runtime errors during startup. Default value is **true**. |
| fallbackUserNameClaim<br><br>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<br><br>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. |

| Property | Description |
| --- | --- |
| groupsClaim<br><br>string | JsonPath query used to extract groups for the user during authentication. Extracted groups can be used by a custom authorizer. By default no groups are extracted. |
| groupsClaimDelimiter<br><br>string | A delimiter used to parse groups when they are extracted as a single String value rather than a JSON array. Default value is ',' (comma). |
| introspectionEndpointUri<br><br>string | URI of the token introspection endpoint which can be used to validate opaque non-JWT tokens. |
| jwksEndpointUri<br><br>string | URI of the JWKS certificate endpoint, which can be used for local JWT validation. |
| jwksExpirySeconds<br><br>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. |
| jwksIgnoreKeyUse<br><br>boolean | Flag to ignore the 'use' attribute of **key** declarations in a JWKS endpoint response. Default value is **false**. |
| jwksMinRefreshPauseSeconds<br><br>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<br><br>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<br><br>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. This option only applies to SASL_OAUTHBEARER authentication mechanism (when **enableOauthBearer** is **true**). |
| readTimeoutSeconds<br><br>integer | The read timeout in seconds when connecting to authorization server. If not set, the effective read timeout is 60 seconds. |
| tlsTrustedCertificates<br><br>**CertSecretSource** array | Trusted certificates for TLS connection to the OAuth server. |
| tokenEndpointUri<br><br>string | URI of the Token Endpoint to use with SASL_PLAIN mechanism when the client authenticates with **clientId** and a **secret**. If set, the client can authenticate over SASL_PLAIN by either setting **username** to **clientId**, and setting **password** to client **secret**, or by setting **username** to account username, and **password** to access token prefixed with **$accessToken:**. If this option is not set, the **password** is always interpreted as an access token (without a prefix), and **username** as the account username (a so called 'no-client-credentials' mode). |
| type<br><br>string | Must be **oauth**. |
| userInfoEndpointUri<br><br>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<br><br>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 | URI of the token issuer used for authentication. |

| Property | Description |
|---|---|
| string | |
| validTokenType | Valid value for the **token_type** attribute returned by the Introspection Endpoint. No default value, and not checked by default. |
| string | |

## 12.2.8. **GenericSecretSource** schema reference

Used in: **KafkaClientAuthenticationOAuth**, **KafkaListenerAuthenticationCustom**, **KafkaListenerAuthenticationOAuth**

| Property | Description |
|---|---|
| key | The key under which the secret value is stored in the OpenShift Secret. |
| string | |
| secretName | The name of the OpenShift Secret containing the secret value. |
| string | |

## 12.2.9. **CertSecretSource** schema reference

Used in: **ClientTls**, **KafkaAuthorizationKeycloak**, **KafkaClientAuthenticationOAuth**, **KafkaListenerAuthenticationOAuth**

| Property | Description |
|---|---|
| certificate | The name of the file certificate in the Secret. |
| string | |
| secretName | The name of the Secret containing the certificate. |
| string | |

## 12.2.10. **KafkaListenerAuthenticationCustom** schema reference

Used in: **GenericKafkaListener**

Full list of **KafkaListenerAuthenticationCustom** schema properties

To configure custom authentication, set the **type** property to **custom**.

Custom authentication allows for any type of kafka-supported authentication to be used.

### Example custom OAuth authentication configuration

```
spec:
  kafka:
    config:
      principal.builder.class: SimplePrincipal.class
    listeners:
      - name: oauth-bespoke
        port: 9093
        type: internal
        tls: true
        authentication:
          type: custom
          sasl: true
          listenerConfig:
            oauthbearer.sasl.client.callback.handler.class: client.class
            oauthbearer.sasl.server.callback.handler.class: server.class
            oauthbearer.sasl.login.callback.handler.class: login.class
            oauthbearer.connections.max.reauth.ms: 999999999
            sasl.enabled.mechanisms: oauthbearer
            oauthbearer.sasl.jaas.config: |
              org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required ;
          secrets:
            - name: example
```

A protocol map is generated that uses the **sasl** and **tls** values to determine which protocol to map to the listener.

- SASL = True, TLS = True → SASL_SSL

- SASL = False, TLS = True → SSL

- SASL = True, TLS = False → SASL_PLAINTEXT

- SASL = False, TLS = False → PLAINTEXT

#### 12.2.10.1. listenerConfig

Listener configuration specified using **listenerConfig** is prefixed with **listener.name.*<listener_name>-<port>***. For example, **sasl.enabled.mechanisms** becomes **listener.name.*<listener_name>-<port>*.sasl.enabled.mechanisms**.

#### 12.2.10.2. secrets

Secrets are mounted to **/opt/kafka/custom-authn-secrets/custom-listener-*<listener_name>-<port>*/*<secret_name>*** in the Kafka broker nodes' containers.

For example, the mounted secret (**example**) in the example configuration would be located at **/opt/kafka/custom-authn-secrets/custom-listener-oauth-bespoke-9093/example**.

#### 12.2.10.3. Principal builder

You can set a custom principal builder in the Kafka cluster configuration. However, the principal builder is subject to the following requirements:

- The specified principal builder class must exist on the image. *Before* building your own, check if one already exists. You'll need to rebuild the AMQ Streams images with the required classes.

- No other listener is using **oauth** type authentication. This is because an OAuth listener appends its own principle builder to the Kafka configuration.

- The specified principal builder is compatible with AMQ Streams.

Custom principal builders must support peer certificates for authentication, as AMQ Streams uses these to manage the Kafka cluster.

> **NOTE**
>
> Kafka's default principal builder class supports the building of principals based on the names of peer certificates. The custom principal builder should provide a principal of type **user** using the name of the SSL peer certificate.

The following example shows a custom principal builder that satisfies the OAuth requirements of AMQ Streams.

**Example principal builder for custom OAuth configuration**

```java
public final class CustomKafkaPrincipalBuilder implements KafkaPrincipalBuilder {

    public KafkaPrincipalBuilder() {}

    @Override
    public KafkaPrincipal build(AuthenticationContext context) {
        if (context instanceof SslAuthenticationContext) {
            SSLSession sslSession = ((SslAuthenticationContext) context).session();
            try {
                return new KafkaPrincipal(
                    KafkaPrincipal.USER_TYPE, sslSession.getPeerPrincipal().getName());
            } catch (SSLPeerUnverifiedException e) {
                throw new IllegalArgumentException("Cannot use an unverified peer for authentication", e);
            }
        }

        // Create your own KafkaPrincipal here
        ...
    }
}
```

### 12.2.10.4. KafkaListenerAuthenticationCustom schema properties

The **type** property is a discriminator that distinguishes use of the **KafkaListenerAuthenticationCustom** type from **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth**. It must have the value **custom** for the type **KafkaListenerAuthenticationCustom**.

| Property | Description |
| --- | --- |

| Property | Description |
| --- | --- |
| listenerConfig<br><br>map | Configuration to be used for a specific listener. All values are prefixed with listener.name.*<listener_name>.* |
| sasl<br><br>boolean | Enable or disable SASL on this listener. |
| secrets<br><br>[GenericSecretSource](#) array | Secrets to be mounted to /opt/kafka/custom-authn-secrets/custom-listener-*<listener_name>*-*<port>*/*<secret_name>*. |
| type<br><br>string | Must be **custom**. |

## 12.2.11. GenericKafkaListenerConfiguration schema reference

Used in: **GenericKafkaListener**

Full list of **GenericKafkaListenerConfiguration** schema properties

Configuration for Kafka listeners.

### 12.2.11.1. brokerCertChainAndKey

The **brokerCertChainAndKey** property is only used with listeners that have TLS encryption enabled. You can use the property to provide 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
  # ...
```

### 12.2.11.2. 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**.

### 12.2.11.3. 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
      # ...
  # ...
```

### 12.2.11.4. class

The **class** property is only used with **ingress** listeners. You can configure 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: true
    configuration:
      class: nginx-internal
    # ...
  # ...
```

### 12.2.11.5. 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
      # ...
  # ...
```

### 12.2.11.6. 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 7.2.3, "Configuring the Cluster Operator with environment variables" for more details.

### 12.2.11.7. **GenericKafkaListenerConfiguration** schema properties

| Property | Description |
|---|---|
| brokerCertChainAndKey<br><br>**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<br><br>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<br><br>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. This field can be used only with **loadbalancer** type listener. |
| bootstrap<br><br>**GenericKafkaListenerConfigurationBootstrap** | Bootstrap configuration. |
| brokers<br><br>**GenericKafkaListenerConfigurationBroker**<br>array | Per-broker configurations. |
| ipFamilyPolicy<br><br>string (one of [RequireDualStack, SingleStack, PreferDualStack]) | Specifies the IP Family Policy used by the service. Available options are **SingleStack**, **PreferDualStack** and **RequireDualStack**. **SingleStack** is for a single IP family. **PreferDualStack** is for two IP families on dual-stack configured clusters or a single IP family on single-stack clusters. **RequireDualStack** fails unless there are two IP families on dual-stack configured clusters. If unspecified, OpenShift will choose the default value based on the service type. Available on OpenShift 1.20 and newer. |

| Property | Description |
|---|---|
| ipFamilies<br><br>string (one or more of [IPv6, IPv4]) array | Specifies the IP Families used by the service. Available options are **IPv4** and **IPv6. If unspecified, OpenShift will choose the default value based on the `ipFamilyPolicy** setting. Available on OpenShift 1.20 and newer. |
| createBootstrapService<br><br>boolean | Whether to create the bootstrap service or not. The bootstrap service is created by default (if not specified differently). This field can be used with the **loadBalancer** type listener. |
| class<br><br>string | Configures the **Ingress** class that defines which **Ingress** controller will be used. This field can be used only with **ingress** type listener. If not specified, the default Ingress controller will be used. |
| finalizers<br><br>string array | A list of finalizers which will be configured for the **LoadBalancer** type Services created for this listener. If supported by the platform, the finalizer **service.kubernetes.io/load-balancer-cleanup** to make sure that the external load balancer is deleted together with the service.For more information, see [https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/#garbage-collecting-load-balancers](https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/#garbage-collecting-load-balancers). This field can be used only with **loadbalancer** type listeners. |
| maxConnectionCreationRate<br><br>integer | The maximum connection creation rate we allow in this listener at any time. New connections will be throttled if the limit is reached. |
| maxConnections<br><br>integer | The maximum number of connections we allow for this listener in the broker at any time. New connections are blocked if the limit is reached. |

| Property | Description |
|---|---|
| preferredNodePortAddressType<br><br>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):<br><br>    ● **ExternalDNS**<br><br>    ● **ExternalIP**<br><br>    ● **InternalDNS**<br><br>    ● **InternalIP**<br><br>    ● **Hostname**<br><br>This field is used to select the preferred address type, which is checked first. If no address is found for this address type, the other types are checked in the default order. This field can only be used with **nodeport** type listener. |
| useServiceDnsDomain<br><br>boolean | Configures whether the OpenShift service DNS domain should be used or not. If set to **true**, the generated addresses will 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. |

## 12.2.12. **CertAndKeySecretSource** schema reference

Used in: **GenericKafkaListenerConfiguration**, **KafkaClientAuthenticationTls**

| Property | Description |
|---|---|
| certificate<br><br>string | The name of the file certificate in the Secret. |
| key<br><br>string | The name of the private key in the Secret. |
| secretName<br><br>string | The name of the Secret containing the certificate. |

### 12.2.13. GenericKafkaListenerConfigurationBootstrap schema reference

Used in: **GenericKafkaListenerConfiguration**

Full list of **GenericKafkaListenerConfigurationBootstrap** schema properties

Broker service equivalents of **nodePort**, **host**, **loadBalancerIP** and **annotations** properties are configured in the **GenericKafkaListenerConfigurationBroker** schema.

#### 12.2.13.1. 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 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
  # ...
```

#### 12.2.13.2. 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
# ...
```

### 12.2.13.3. 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
  # ...
```

### 12.2.13.4. 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
  # ...
```

### 12.2.13.5. annotations

Use the **annotations** property to add annotations to OpenShift resources related to the listeners. You can use these annotations, for example, 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"
  # ...
```

### 12.2.13.6. **GenericKafkaListenerConfigurationBootstrap** schema properties

| Property | Description |
| --- | --- |
| alternativeNames<br><br>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<br><br>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<br><br>integer | Node port for the bootstrap service. This field can be used only with **nodeport** type listener. |

| Property | Description |
|---|---|
| loadBalancerIP<br><br>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<br><br>map | Annotations that will be added to the **Ingress**, **Route**, 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**, **route**, or **ingress** type listeners. |
| labels<br><br>map | Labels that will be added to the **Ingress**, **Route**, or **Service** resource. This field can be used only with **loadbalancer**, **nodeport**, **route**, or **ingress** type listeners. |

## 12.2.14. GenericKafkaListenerConfigurationBroker schema reference

Used in: **GenericKafkaListenerConfiguration**

Full list of **GenericKafkaListenerConfigurationBroker** schema properties

You can see example configuration for the **nodePort**, **host**, **loadBalancerIP** and **annotations** properties in the **GenericKafkaListenerConfigurationBootstrap** schema, which configures bootstrap service overrides.

### 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 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 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
# ...
```

### 12.2.14.1. **GenericKafkaListenerConfigurationBroker** schema properties

| Property | Description |
|---|---|
| broker<br><br>integer | ID of the kafka broker (broker identifier). Broker IDs start from 0 and correspond to the number of broker replicas. |
| advertisedHost<br><br>string | The host name which will be used in the brokers' **advertised.brokers**. |
| advertisedPort<br><br>integer | The port number which will be used in the brokers' **advertised.brokers**. |
| host<br><br>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<br><br>integer | Node port for the per-broker service. This field can be used only with **nodeport** type listener. |
| loadBalancerIP<br><br>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. |

| Property | Description |
|----------|-------------|
| annotations<br><br>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. |
| labels<br><br>map | Labels that will be added to the **Ingress**, **Route**, or **Service** resource. This field can be used only with **loadbalancer**, **nodeport**, **route**, or **ingress** type listeners. |

### 12.2.15. EphemeralStorage schema reference

Used in: **JbodStorage**, **KafkaClusterSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes use of the **EphemeralStorage** type from **PersistentClaimStorage**. It must have the value **ephemeral** for the type **EphemeralStorage**.

| Property | Description |
|----------|-------------|
| id<br><br>integer | Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'. |
| sizeLimit<br><br>string | When type=ephemeral, defines the total amount of local storage required for this EmptyDir volume (for example 1Gi). |
| type<br><br>string | Must be **ephemeral**. |

### 12.2.16. PersistentClaimStorage schema reference

Used in: **JbodStorage**, **KafkaClusterSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes use of the **PersistentClaimStorage** type from **EphemeralStorage**. It must have the value **persistent-claim** for the type **PersistentClaimStorage**.

| Property | Description |
|----------|-------------|
| type | Must be **persistent-claim**. |

| Property | Description |
| --- | --- |
| string | |
| size | When type=persistent-claim, defines the size of the persistent volume claim (i.e 1Gi). Mandatory when type=persistent-claim. |
| string | |
| selector | Specifies a specific persistent volume to use. It contains key:value pairs representing labels for selecting such a volume. |
| map | |
| deleteClaim | Specifies if the persistent volume claim has to be deleted when the cluster is un-deployed. |
| boolean | |
| class | The storage class to use for dynamic volume allocation. |
| string | |
| id | Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'. |
| integer | |
| overrides | Overrides for individual brokers. The **overrides** field allows to specify a different configuration for different brokers. |
| **PersistentClaimStorageOverride** array | |

### 12.2.17. **PersistentClaimStorageOverride** schema reference

Used in: **PersistentClaimStorage**

| Property | Description |
| --- | --- |
| class | The storage class to use for dynamic volume allocation for this broker. |
| string | |
| broker | Id of the kafka broker (broker identifier). |
| integer | |

### 12.2.18. **JbodStorage** schema reference

Used in: **KafkaClusterSpec**

The **type** property is a discriminator that distinguishes use of the **JbodStorage** type 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 | |

## 12.2.19. KafkaAuthorizationSimple schema reference

Used in: **KafkaClusterSpec**

Full list of **KafkaAuthorizationSimple** schema properties

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.

### 12.2.19.1. 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**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: simple
      superUsers:
        - CN=client_1
        - user_2
        - CN=client_3
    # ...
```
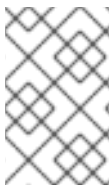
> **NOTE**
>
> The **super.user** configuration option in the **config** property in **Kafka.spec.kafka** is ignored. Designate super users in the **authorization** property instead. For more information, see Kafka broker configuration .

### 12.2.19.2. KafkaAuthorizationSimple schema properties

The **type** property is a discriminator that distinguishes use of the **KafkaAuthorizationSimple** type from **KafkaAuthorizationOpa**, **KafkaAuthorizationKeycloak**, **KafkaAuthorizationCustom**. 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 | |

## 12.2.20. KafkaAuthorizationOpa schema reference

Used in: **KafkaClusterSpec**

Full list of **KafkaAuthorizationOpa** schema properties

To use Open Policy Agent authorization, set the **type** property in the **authorization** section to the value **opa**, and configure OPA properties as required. AMQ Streams uses Open Policy Agent plugin for Kafka authorization as the authorizer. For more information about the format of the input data and policy examples, see Open Policy Agent plugin for Kafka authorization .

### 12.2.20.1. 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.**

### 12.2.20.2. 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.

### 12.2.20.3. initialCacheCapacity

Initial capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request. Defaults to **5000**.

### 12.2.20.4. maximumCacheSize

Maximum capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request. Defaults to **50000**.

### 12.2.20.5. 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).

### 12.2.20.6. 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**

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

### 12.2.20.7. KafkaAuthorizationOpa schema properties

The **type** property is a discriminator that distinguishes use of the **KafkaAuthorizationOpa** type from **KafkaAuthorizationSimple**, **KafkaAuthorizationKeycloak**, **KafkaAuthorizationCustom**. 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 | |

| Property | Description |
| --- | --- |
| allowOnError<br><br>boolean | Defines whether a Kafka client should be allowed or denied by default when the authorizer fails to query the Open Policy Agent, for example, when it is temporarily unavailable). Defaults to **false** – all actions will be denied. |
| initialCacheCapacity<br><br>integer | Initial capacity of the local cache used by the authorizer to avoid querying the Open Policy Agent for every request Defaults to **5000**. |
| maximumCacheSize<br><br>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<br><br>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<br><br>string array | List of super users, which is specifically a list of user principals that have unlimited access rights. |
| enableMetrics<br><br>boolean | Defines whether the Open Policy Agent authorizer plugin should provide metrics. Defaults to **false**. |

## 12.2.21. **KafkaAuthorizationKeycloak** schema reference

Used in: **KafkaClusterSpec**

The **type** property is a discriminator that distinguishes use of the **KafkaAuthorizationKeycloak** type from **KafkaAuthorizationSimple**, **KafkaAuthorizationOpa**, **KafkaAuthorizationCustom**. It must have the value **keycloak** for the type **KafkaAuthorizationKeycloak**.

| Property | Description |
| --- | --- |
| type<br><br>string | Must be **keycloak**. |
| clientId<br><br>string | OAuth Client ID which the Kafka client can use to authenticate against the OAuth server and use the token endpoint URI. |

| Property | Description |
| --- | --- |
| tokenEndpointUri<br><br>string | Authorization server token endpoint URI. |
| tlsTrustedCertificates<br><br>**CertSecretSource** array | Trusted certificates for TLS connection to the OAuth server. |
| disableTlsHostnameVerification<br><br>boolean | Enable or disable TLS hostname verification. Default value is **false**. |
| delegateToKafkaAcls<br><br>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<br><br>integer | The time between two consecutive grants refresh runs in seconds. The default value is 60. |
| grantsRefreshPoolSize<br><br>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<br><br>string array | List of super users. Should contain list of user principals which should get unlimited access rights. |
| connectTimeoutSeconds<br><br>integer | The connect timeout in seconds when connecting to authorization server. If not set, the effective connect timeout is 60 seconds. |
| readTimeoutSeconds<br><br>integer | The read timeout in seconds when connecting to authorization server. If not set, the effective read timeout is 60 seconds. |
| enableMetrics<br><br>boolean | Enable or disable OAuth metrics. Default value is **false**. |

## 12.2.22. KafkaAuthorizationCustom schema reference

Used in: **KafkaClusterSpec**

Full list of **KafkaAuthorizationCustom** schema properties

To use custom authorization in AMQ Streams, you can configure your own **Authorizer** plugin to define Access Control Lists (ACLs).

ACLs allow you to define which users have access to which resources at a granular level.

Configure the **Kafka** custom resource to use custom authorization. Set the **type** property in the **authorization** section to the value **custom**, and the set following properties.

> **IMPORTANT**
>
> The custom authorizer must implement the **org.apache.kafka.server.authorizer.Authorizer** interface, and support configuration of **super.users** using the super.users configuration property.

### 12.2.22.1. authorizerClass

(Required) Java class that implements the **org.apache.kafka.server.authorizer.Authorizer** interface to support custom ACLs.

### 12.2.22.2. 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.

You can add configuration for initializing the custom authorizer using **Kafka.spec.kafka.config**.

**An example of custom authorization configuration under Kafka.spec**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: custom
      authorizerClass: io.mycompany.CustomAuthorizer
      superUsers:
        - CN=client_1
        - user_2
        - CN=client_3
    # ...
    config:
      authorization.custom.property1=value1
      authorization.custom.property2=value2
    # ...
```

In addition to the **Kafka** custom resource configuration, the JAR file containing the custom authorizer class along with its dependencies must be available on the classpath of the Kafka broker.

The AMQ Streams Maven build process provides a mechanism to add custom third-party libraries to the generated Kafka broker container image by adding them as dependencies in the **pom.xml** file under the **docker-images/kafka/kafka-thirdparty-libs** directory. The directory contains different folders for different Kafka versions. Choose the appropriate folder. Before modifying the **pom.xml** file, the third-party library must be available in a Maven repository, and that Maven repository must be accessible to the AMQ Streams build process.

> **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.

Custom authorization can make use of group membership information extracted from the JWT token during authentication when using **oauth** authentication and configuring **groupsClaim** configuration attribute. Groups are available on the **OAuthKafkaPrincipal** object during authorize() call as follows:

```
public List<AuthorizationResult> authorize(AuthorizableRequestContext requestContext,
List<Action> actions) {

    KafkaPrincipal principal = requestContext.principal();
    if (principal instanceof OAuthKafkaPrincipal) {
        OAuthKafkaPrincipal p = (OAuthKafkaPrincipal) principal;

        for (String group: p.getGroups()) {
            System.out.println("Group: " + group);
        }
    }
}
```

### 12.2.22.3. KafkaAuthorizationCustom schema properties

The **type** property is a discriminator that distinguishes use of the **KafkaAuthorizationCustom** type from **KafkaAuthorizationSimple**, **KafkaAuthorizationOpa**, **KafkaAuthorizationKeycloak**. It must have the value **custom** for the type **KafkaAuthorizationCustom**.

| Property | Description |
|---|---|
| type | Must be **custom**. |
| string | |
| authorizerClass | Authorization implementation class, which must be available in classpath. |
| string | |
| superUsers | List of super users, which are user principals with unlimited access rights. |
| string array | |

| Property | Description |
| --- | --- |
| supportsAdminApi | Indicates whether the custom authorizer supports the APIs for managing ACLs using the Kafka Admin API. Defaults to **false**. |
| boolean | |

## 12.2.23. Rack schema reference

Used in: **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**

Full list of **Rack** schema properties

The **rack** option configures rack awareness. A *rack* can represent an availability zone, data center, or an actual rack in your data center. The *rack* is configured through a **topologyKey**. **topologyKey** identifies a label on OpenShift nodes that contains the name of the topology in its value. An example of such a label is **topology.kubernetes.io/zone** (or **failure-domain.beta.kubernetes.io/zone** on older OpenShift versions), which contains the name of the availability zone in which the OpenShift node runs. You can configure your Kafka cluster to be aware of the *rack* in which it runs, and enable additional features such as spreading partition replicas across different racks or consuming messages from the closest replicas.

For more information about OpenShift node labels, see Well-Known Labels, Annotations and Taints . Consult your OpenShift administrator regarding the node label that represents the zone or rack into which the node is deployed.

### 12.2.23.1. Spreading partition replicas across racks

When rack awareness is configured, AMQ Streams will set **broker.rack** configuration for each Kafka broker. The **broker.rack** configuration assigns a rack ID to each broker. When **broker.rack** is configured, Kafka brokers will spread partition replicas across as many different racks as possible. When replicas are spread across multiple racks, the probability that multiple replicas will fail at the same time is lower than if they would be in the same rack. Spreading replicas improves resiliency, and is important for availability and reliability. To enable rack awareness in Kafka, add the **rack** option to the **.spec.kafka** section of the **Kafka** custom resource as shown in the example below.

**Example rack configuration for Kafka**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    # ...
```

> **NOTE**
>
> The *rack* in which brokers are running can change in some cases when the pods are deleted or restarted. As a result, the replicas running in different racks might then share the same rack. Use Cruise Control and the **KafkaRebalance** resource with the **RackAwareGoal** to make sure that replicas remain distributed across different racks.

When rack awareness is enabled in the **Kafka** custom resource, AMQ Streams will automatically add the OpenShift **preferredDuringSchedulingIgnoredDuringExecution** affinity rule to distribute the Kafka brokers across the different racks. However, the *preferred* rule does not guarantee that the brokers will be spread. Depending on your exact OpenShift and Kafka configurations, you should add additional **affinity** rules or configure **topologySpreadConstraints** for both ZooKeeper and Kafka to make sure the nodes are properly distributed accross as many racks as possible. For more information see Section 2.9, "Configuring pod scheduling".

### 12.2.23.2. Consuming messages from the closest replicas

Rack awareness can also be used in consumers to fetch data from the closest replica. This is useful for reducing the load on your network when a Kafka cluster spans multiple datacenters and can also reduce costs when running Kafka in public clouds. However, it can lead to increased latency.

In order to be able to consume from the closest replica, rack awareness has to be configured in the Kafka cluster, and the **RackAwareReplicaSelector** has to be enabled. The replica selector plugin provides the logic that enables clients to consume from the nearest replica. The default implementation uses **LeaderSelector** to always select the leader replica for the client. Specify **RackAwareReplicaSelector** for the **replica.selector.class** to switch from the default implementation.

**Example rack configuration with enabled replica-aware selector**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    config:
      # ...
      replica.selector.class: org.apache.kafka.common.replica.RackAwareReplicaSelector
    # ...
```

In addition to the Kafka broker configuration, you also need to specify the **client.rack** option in your consumers. The **client.rack** option should specify the *rack ID* in which the consumer is running. **RackAwareReplicaSelector** associates matching **broker.rack** and **client.rack** IDs, to find the nearest replica and consume from it. If there are multiple replicas in the same rack, **RackAwareReplicaSelector** always selects the most up-to-date replica. If the rack ID is not specified, or if it cannot find a replica with the same rack ID, it will fall back to the leader replica.

Figure 12.1. Example showing client consuming from replicas in the same availability zone
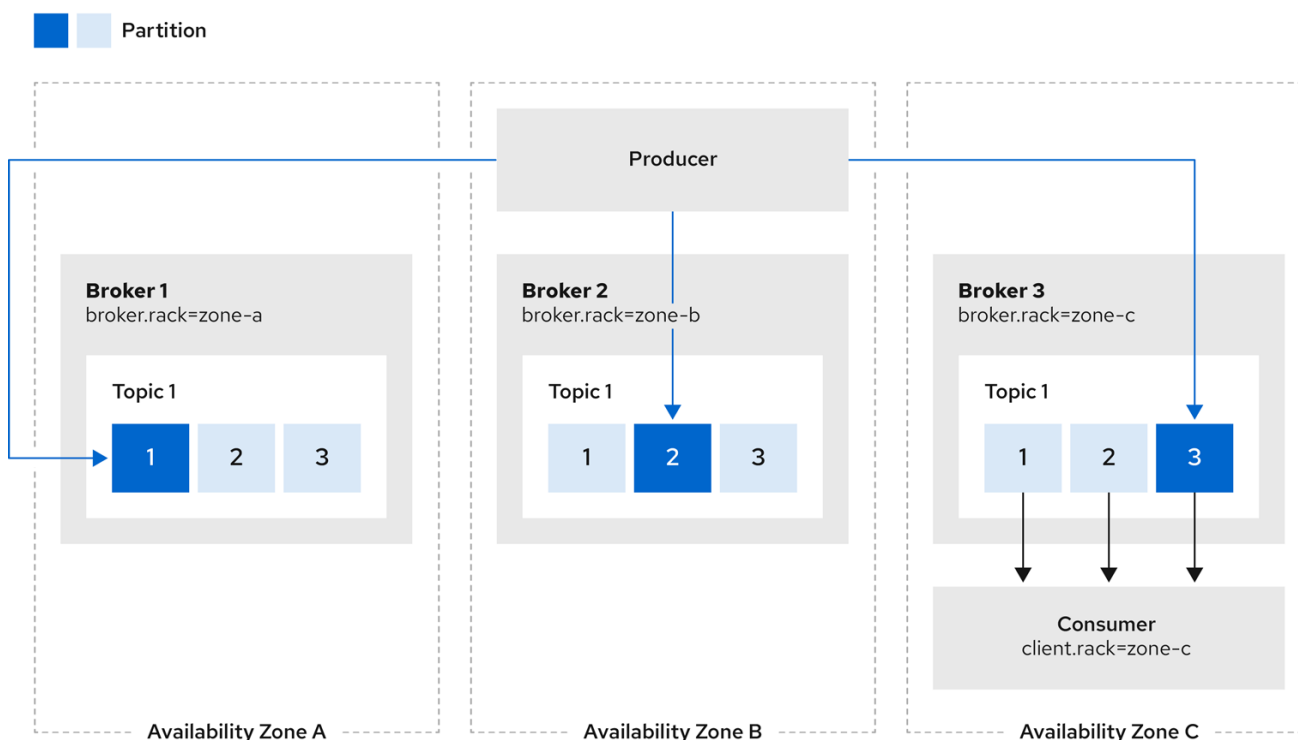


You can also configure Kafka Connect, MirrorMaker 2.0 and Kafka Bridge so that connectors consume messages from the closest replicas. You enable rack awareness in the **KafkaConnect**, **KafkaMirrorMaker2**, and **KafkaBridge** custom resources. The configuration does does not set affinity rules, but you can also configure **affinity** or **topologySpreadConstraints**. For more information see Section 2.9, "Configuring pod scheduling".

When deploying Kafka Connect using AMQ Streams, you can use the **rack** section in the **KafkaConnect** custom resource to automatically configure the **client.rack** option.

## Example **rack** configuration for Kafka Connect

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
# ...
spec:
  # ...
  rack:
    topologyKey: topology.kubernetes.io/zone
  # ...
```

When deploying MirrorMaker 2 using AMQ Streams, you can use the **rack** section in the **KafkaMirrorMaker2** custom resource to automatically configure the **client.rack** option.

## Example **rack** configuration for MirrorMaker 2.0

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
# ...
spec:
```

```
# ...
rack:
  topologyKey: topology.kubernetes.io/zone
# ...
```

When deploying Kafka Bridge using AMQ Streams, you can use the **rack** section in the **KafkaBridge** custom resource to automatically configure the **client.rack** option.

### Example rack configuration for Kafka Bridge

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
# ...
spec:
  # ...
  rack:
    topologyKey: topology.kubernetes.io/zone
  # ...
```

### 12.2.23.3. Rack schema properties

| Property | Description |
|----------|-------------|
| topologyKey<br><br>string | A key that matches labels assigned to the OpenShift cluster nodes. The value of the label is used to set a broker's **broker.rack** config, and the **client.rack** config for Kafka Connect or MirrorMaker 2.0. |

### 12.2.24. Probe schema reference

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectSpec**, **KafkaExporterSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **TlsSidecar**, **ZookeeperClusterSpec**

| Property | Description |
|----------|-------------|
| failureThreshold<br><br>integer | Minimum consecutive failures for the probe to be considered failed after having succeeded. Defaults to 3. Minimum value is 1. |
| initialDelaySeconds<br><br>integer | The initial delay before first the health is first checked. Default to 15 seconds. Minimum value is 0. |
| periodSeconds<br><br>integer | How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1. |

| Property | Description |
|---|---|
| successThreshold<br><br>integer | 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. |
| timeoutSeconds<br><br>integer | The timeout for each attempted health check. Default to 5 seconds. Minimum value is 1. |

## 12.2.25. **JvmOptions** schema reference

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **ZookeeperClusterSpec**

| Property | Description |
|---|---|
| -XX<br><br>map | A map of -XX options to the JVM. |
| -Xms<br><br>string | -Xms option to to the JVM. |
| -Xmx<br><br>string | -Xmx option to to the JVM. |
| gcLoggingEnabled<br><br>boolean | Specifies whether the Garbage Collection logging is enabled. The default is false. |
| javaSystemProperties<br><br>**SystemProperty** array | A map of additional system properties which will be passed using the **-D** option to the JVM. |

## 12.2.26. **SystemProperty** schema reference

Used in: **JvmOptions**

| Property | Description |
| --- | --- |
| name | The system property name. |
| string | |
| value | The system property value. |
| string | |

### 12.2.27. **KafkaJmxOptions** schema reference

Used in: **KafkaClusterSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **ZookeeperClusterSpec**

Full list of **KafkaJmxOptions** schema properties

Configures JMX connection options.

Get JMX metrics from Kafka brokers, ZooKeeper nodes, Kafka Connect, and MirrorMaker 2.0. by connecting to port 9999. Use the **jmxOptions** property to configure a password-protected or an unprotected JMX port. Using password protection prevents unauthorized pods from accessing the port.

You can then obtain metrics about the component.

For example, for each Kafka broker you can obtain bytes-per-second usage data from clients, or the request rate of the network of the broker.

To enable security for the JMX port, set the **type** parameter in the **authentication** field to **password**.

#### Example password-protected JMX configuration for Kafka brokers and ZooKeeper nodes

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jmxOptions:
      authentication:
        type: "password"
    # ...
  zookeeper:
    # ...
    jmxOptions:
      authentication:
        type: "password"
    #...
```

You can then deploy a pod into a cluster and obtain JMX metrics using the headless service by specifying which broker you want to address.

For example, to get JMX metrics from broker *0* you specify:

> "*CLUSTER-NAME*-kafka-0.*CLUSTER-NAME*-kafka-brokers"

*CLUSTER-NAME*-kafka-0 is name of the broker pod, and *CLUSTER-NAME*-kafka-brokers is the name of the headless service to return the IPs of the broker pods.

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.

For an unprotected JMX port, use an empty object **{}** to open the JMX port on the headless service. You deploy a pod and obtain metrics in the same way as for the protected port, but in this case any pod can read from the JMX port.

**Example open port JMX configuration for Kafka brokers and ZooKeeper nodes**

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

**Additional resources**

- For more information on the Kafka component metrics exposed using JMX, see the Apache Kafka documentation.

### 12.2.27.1. **KafkaJmxOptions** schema properties

| Property | Description |
| --- | --- |
| authentication<br><br>**KafkaJmxAuthenticationPassword** | Authentication configuration for connecting to the JMX port. The type depends on the value of the **authentication.type** property within the given object, which must be one of [password]. |

### 12.2.28. **KafkaJmxAuthenticationPassword** schema reference

Used in: **KafkaJmxOptions**

The **type** property is a discriminator that distinguishes use of the **KafkaJmxAuthenticationPassword** type 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 | |

### 12.2.29. **JmxPrometheusExporterMetrics** schema reference

Used in: **CruiseControlSpec**, **KafkaClusterSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes use of the **JmxPrometheusExporterMetrics** type from other subtypes which may be added in the future. It must have the value **jmxPrometheusExporter** for the type **JmxPrometheusExporterMetrics**.

| Property | Description |
| --- | --- |
| type | Must be **jmxPrometheusExporter**. |
| string | |
| valueFrom **ExternalConfigurationReference** | ConfigMap entry where the Prometheus JMX Exporter configuration is stored. For details of the structure of this configuration, see the Prometheus JMX Exporter. |

### 12.2.30. **ExternalConfigurationReference** schema reference

Used in: **ExternalLogging**, **JmxPrometheusExporterMetrics**

| Property | Description |
| --- | --- |
| configMapKeyRef **ConfigMapKeySelector** | Reference to the key in the ConfigMap containing the configuration. For more information, see the external documentation for core/v1 configmapkeyselector. |

### 12.2.31. **InlineLogging** schema reference

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes use of the **InlineLogging** type 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 | |

### 12.2.32. **ExternalLogging** schema reference

Used in: **CruiseControlSpec**, **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes use of the **ExternalLogging** type from **InlineLogging**. It must have the value **external** for the type **ExternalLogging**.

| Property | Description |
| --- | --- |
| type | Must be **external**. |
| string | |
| valueFrom | **ConfigMap** entry where the logging configuration is stored. |
| **ExternalConfigurationReference** | |

### 12.2.33. **KafkaClusterTemplate** schema reference

Used in: **KafkaClusterSpec**

| Property | Description |
| --- | --- |
| statefulset | Template for Kafka **StatefulSet**. |
| **StatefulSetTemplate** | |
| pod | Template for Kafka **Pods**. |
| **PodTemplate** | |
| bootstrapService | Template for Kafka bootstrap **Service**. |
| **InternalServiceTemplate** | |

| Property | Description |
| --- | --- |
| brokersService<br><br>**InternalServiceTemplate** | Template for Kafka broker **Service**. |
| externalBootstrapService<br><br>**ResourceTemplate** | Template for Kafka external bootstrap **Service**. |
| perPodService<br><br>**ResourceTemplate** | Template for Kafka per-pod **Services** used for access from outside of OpenShift. |
| externalBootstrapRoute<br><br>**ResourceTemplate** | Template for Kafka external bootstrap **Route**. |
| perPodRoute<br><br>**ResourceTemplate** | Template for Kafka per-pod **Routes** used for access from outside of OpenShift. |
| externalBootstrapIngress<br><br>**ResourceTemplate** | Template for Kafka external bootstrap **Ingress**. |
| perPodIngress<br><br>**ResourceTemplate** | Template for Kafka per-pod **Ingress** used for access from outside of OpenShift. |
| persistentVolumeClaim<br><br>**ResourceTemplate** | Template for all Kafka **PersistentVolumeClaims**. |
| podDisruptionBudget<br><br>**PodDisruptionBudgetTemplate** | Template for Kafka **PodDisruptionBudget**. |
| kafkaContainer<br><br>**ContainerTemplate** | Template for the Kafka broker container. |
| initContainer<br><br>**ContainerTemplate** | Template for the Kafka init container. |

| Property | Description |
| --- | --- |
| clusterCaCert<br><br>**ResourceTemplate** | Template for Secret with Kafka Cluster certificate public key. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the Kafka service account. |
| jmxSecret<br><br>**ResourceTemplate** | Template for Secret of the Kafka Cluster JMX authentication. |
| clusterRoleBinding<br><br>**ResourceTemplate** | Template for the Kafka ClusterRoleBinding. |
| podSet<br><br>**ResourceTemplate** | Template for Kafka **StrimziPodSet** resource. |

## 12.2.34. **StatefulSetTemplate** schema reference

Used in: **KafkaClusterTemplate**, **ZookeeperClusterTemplate**

| Property | Description |
| --- | --- |
| metadata<br><br>**MetadataTemplate** | Metadata applied to the resource. |
| podManagementPolicy<br><br>string (one of [OrderedReady, Parallel]) | PodManagementPolicy which will be used for this StatefulSet. Valid values are **Parallel** and **OrderedReady**. Defaults to **Parallel**. |

## 12.2.35. **MetadataTemplate** schema reference

Used in: **BuildConfigTemplate**, **DeploymentTemplate**, **InternalServiceTemplate**, **PodDisruptionBudgetTemplate**, **PodTemplate**, **ResourceTemplate**, **StatefulSetTemplate**

Full list of **MetadataTemplate** schema properties

**Labels** and **Annotations** are used to identify and organize resources, and are configured in the **metadata** property.

For example:

```
# ...
template:
  pod:
    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.

### 12.2.35.1. **MetadataTemplate** schema properties

| 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 | |

### 12.2.36. **PodTemplate** schema reference

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Full list of **PodTemplate** schema properties

Configures the template for Kafka pods.

### Example **PodTemplate** configuration

```
# ...
template:
  pod:
    metadata:
      labels:
        label1: value1
      annotations:
        anno1: value1
    imagePullSecrets:
      - name: my-docker-credentials
    securityContext:
      runAsUser: 1000001
```

```
      fsGroup: 0
    terminationGracePeriodSeconds: 120
  # ...
```

### 12.2.36.1. 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/v1beta2
kind: KafkaConnect
#...
spec:
  # ...
  template:
    pod:
      hostAliases:
      - ip: "192.168.1.86"
        hostnames:
        - "my-host-1"
        - "my-host-2"
      #...
```

### 12.2.36.2. PodTemplate schema properties

| Property | Description |
| --- | --- |
| metadata<br><br>**MetadataTemplate** | Metadata applied to the resource. |
| imagePullSecrets<br><br>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. For more information, see the external documentation for core/v1 localobjectreference. |
| securityContext<br><br>PodSecurityContext | Configures pod-level security attributes and common container settings. For more information, see the external documentation for core/v1 podsecuritycontext. |

| Property | Description |
| --- | --- |
| terminationGracePeriodSeconds<br><br>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<br><br>Affinity | The pod's affinity rules. For more information, see the external documentation for core/v1 affinity. |
| tolerations<br><br>Toleration array | The pod's tolerations. For more information, see the external documentation for core/v1 toleration. |
| priorityClassName<br><br>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<br><br>string | The name of the scheduler used to dispatch this **Pod**. If not specified, the default scheduler will be used. |
| hostAliases<br><br>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. For more information, see the external documentation for core/v1 hostalias. |
| tmpDirSizeLimit<br><br>string | Defines the total amount (for example **1Gi**) of local storage required for temporary EmptyDir volume (/**tmp**). Default value is **5Mi**. |
| enableServiceLinks<br><br>boolean | Indicates whether information about services should be injected into Pod's environment variables. |
| topologySpreadConstraints<br><br>TopologySpreadConstraint array | The pod's topology spread constraints. For more information, see the external documentation for core/v1 topologyspreadconstraint. |

## 12.2.37. **InternalServiceTemplate** schema reference

Used in: **CruiseControlTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **ZookeeperClusterTemplate**

| Property | Description |
| --- | --- |
| metadata<br><br>**MetadataTemplate** | Metadata applied to the resource. |
| ipFamilyPolicy<br><br>string (one of [RequireDualStack, SingleStack, PreferDualStack]) | Specifies the IP Family Policy used by the service. Available options are **SingleStack**, **PreferDualStack** and **RequireDualStack**. **SingleStack** is for a single IP family. **PreferDualStack** is for two IP families on dual-stack configured clusters or a single IP family on single-stack clusters. **RequireDualStack** fails unless there are two IP families on dual-stack configured clusters. If unspecified, OpenShift will choose the default value based on the service type. Available on OpenShift 1.20 and newer. |
| ipFamilies<br><br>string (one or more of [IPv6, IPv4]) array | Specifies the IP Families used by the service. Available options are **IPv4** and **IPv6. If unspecified, OpenShift will choose the default value based on the `ipFamilyPolicy** setting. Available on OpenShift 1.20 and newer. |

## 12.2.38. ResourceTemplate schema reference

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **KafkaUserTemplate**, **ZookeeperClusterTemplate**

| Property | Description |
| --- | --- |
| metadata<br><br>**MetadataTemplate** | Metadata applied to the resource. |

## 12.2.39. PodDisruptionBudgetTemplate schema reference

Used in: **CruiseControlTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Full list of **PodDisruptionBudgetTemplate** schema properties

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.

An example of **PodDisruptionBudget** template

```
# ...
template:
  podDisruptionBudget:
    metadata:
      labels:
        key1: label1
        key2: label2
      annotations:
        key1: label1
        key2: label2
    maxUnavailable: 1
  # ...
```

### 12.2.39.1. **PodDisruptionBudgetTemplate** schema properties

| Property | Description |
|---|---|
| metadata<br><br>**MetadataTemplate** | Metadata to apply to the **PodDisruptionBudgetTemplate** resource. |
| maxUnavailable<br><br>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. |

### 12.2.40. **ContainerTemplate** schema reference

Used in: **CruiseControlTemplate**, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Full list of **ContainerTemplate** schema properties

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.

### 12.2.40.1. **ContainerTemplate** schema properties

| Property | Description |
|---|---|
| env<br><br>**ContainerEnvVar** array | Environment variables which should be applied to the container. |
| securityContext<br><br>**SecurityContext** | Security context for the container. For more information, see the external documentation for core/v1 securitycontext. |

### 12.2.41. **ContainerEnvVar** schema reference

Used in: **ContainerTemplate**

| Property | Description |
|---|---|
| name<br><br>string | The environment variable key. |
| value<br><br>string | The environment variable value. |

### 12.2.42. **ZookeeperClusterSpec** schema reference

Used in: **KafkaSpec**

Full list of **ZookeeperClusterSpec** schema properties

Configures a ZooKeeper cluster.

### 12.2.42.1. **config**

Use the **config** properties to configure ZooKeeper options as keys.

Standard Apache ZooKeeper configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

Configuration options that cannot be configured relate to:

- Security (Encryption, Authentication, and Authorization)

- Listener configuration

- Configuration of data directories

- ZooKeeper cluster composition

The values can be one of the following JSON types:

- String

- Number

- Boolean

You can specify and configure the options listed in the ZooKeeper documentation with the exception of those 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 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 ZooKeeper.

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 ZooKeeper configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
```

```
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
  # ...
```

### 12.2.42.2. logging

ZooKeeper has a configurable logger:

- **zookeeper.root.logger**

ZooKeeper 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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**. Both **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. A ConfigMap using the exact logging configuration specified is created with the custom resource when the Cluster Operator is running, then recreated after each reconciliation. If you do not specify a custom ConfigMap, default logging settings are used. If a specific logger value is not set, upper-level logger settings are inherited for that logger. 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/v1beta2
kind: Kafka
spec:
  # ...
  zookeeper:
    # ...
    logging:
      type: inline
      loggers:
        zookeeper.root.logger: "INFO"
    # ...
```

### External logging

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  zookeeper:
    # ...
    logging:
      type: external
      valueFrom:
        configMapKeyRef:
```

```
        name: customConfigMap
        key: zookeeper-log4j.properties
  # ...
```

## Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

### 12.2.42.3. **ZookeeperClusterSpec** schema properties

| Property | Description |
| --- | --- |
| replicas<br><br>integer | The number of pods in the cluster. |
| image<br><br>string | The docker image for the pods. |
| storage<br><br>**EphemeralStorage**, **PersistentClaimStorage** | 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]. |
| config<br><br>map | The ZooKeeper broker config. Properties with the following prefixes cannot be set: server., dataDir, dataLogDir, clientPort, authProvider, quorum.auth, requireClientAuthScheme, snapshot.trust.empty, standaloneEnabled, reconfigEnabled, 4lw.commands.whitelist, secureClientPort, ssl., serverCnxnFactory, sslQuorum (with the exception of: ssl.protocol, ssl.quorum.protocol, ssl.enabledProtocols, ssl.quorum.enabledProtocols, ssl.ciphersuites, ssl.quorum.ciphersuites, ssl.hostnameVerification, ssl.quorum.hostnameVerification). |
| livenessProbe<br><br>**Probe** | Pod liveness checking. |
| readinessProbe<br><br>**Probe** | Pod readiness checking. |
| jvmOptions<br><br>**JvmOptions** | JVM Options for pods. |

| Property | Description |
|---|---|
| jmxOptions<br><br>**KafkaJmxOptions** | JMX Options for Zookeeper nodes. |
| resources<br><br>ResourceRequirements | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |
| metricsConfig<br><br>**JmxPrometheusExporterMetrics** | Metrics configuration. The type depends on the value of the **metricsConfig.type** property within the given object, which must be one of [jmxPrometheusExporter]. |
| logging<br><br>**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<br><br>**ZookeeperClusterTemplate** | Template for ZooKeeper cluster resources. The template allows users to specify how the **StatefulSet**, **Pods**, and **Services** are generated. |

## 12.2.43. **ZookeeperClusterTemplate** schema reference

Used in: **ZookeeperClusterSpec**

| Property | Description |
|---|---|
| statefulset<br><br>**StatefulSetTemplate** | Template for ZooKeeper **StatefulSet**. |
| pod<br><br>**PodTemplate** | Template for ZooKeeper **Pods**. |
| clientService<br><br>**InternalServiceTemplate** | Template for ZooKeeper client **Service**. |
| nodesService<br><br>**InternalServiceTemplate** | Template for ZooKeeper nodes **Service**. |

| Property | Description |
|---|---|
| persistentVolumeClaim<br><br>**ResourceTemplate** | Template for all ZooKeeper **PersistentVolumeClaims**. |
| podDisruptionBudget<br><br>**PodDisruptionBudgetTemplate** | Template for ZooKeeper **PodDisruptionBudget**. |
| zookeeperContainer<br><br>**ContainerTemplate** | Template for the ZooKeeper container. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the ZooKeeper service account. |
| jmxSecret<br><br>**ResourceTemplate** | Template for Secret of the Zookeeper Cluster JMX authentication. |
| podSet<br><br>**ResourceTemplate** | Template for ZooKeeper **StrimziPodSet** resource. |

## 12.2.44. **EntityOperatorSpec** schema reference

Used in: **KafkaSpec**

| Property | Description |
|---|---|
| topicOperator<br><br>**EntityTopicOperatorSpec** | Configuration of the Topic Operator. |
| userOperator<br><br>**EntityUserOperatorSpec** | Configuration of the User Operator. |
| tlsSidecar<br><br>**TlsSidecar** | TLS sidecar configuration. |

| Property | Description |
|---|---|
| template <br><br> **EntityOperatorTemplate** | Template for Entity Operator resources. The template allows users to specify how a **Deployment** and **Pod** is generated. |

## 12.2.45. **EntityTopicOperatorSpec** schema reference

Used in: **EntityOperatorSpec**

Full list of **EntityTopicOperatorSpec** schema properties

Configures the Topic Operator.

### 12.2.45.1. **logging**

The Topic Operator has a configurable logger:

- **rootLogger.level**

The Topic Operator uses the Apache **log4j2** logger implementation.

Use the **logging** property in the **entityOperator.topicOperator** field of the Kafka resource **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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j2.properties**. Both **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. A ConfigMap using the exact logging configuration specified is created with the custom resource when the Cluster Operator is running, then recreated after each reconciliation. If you do not specify a custom ConfigMap, default logging settings are used. If a specific logger value is not set, upper-level logger settings are inherited for that logger. 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/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
```

```
    reconciliationIntervalSeconds: 60
    logging:
      type: inline
      loggers:
        rootLogger.level: INFO
  # ...
```

## External logging

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: external
        valueFrom:
          configMapKeyRef:
            name: customConfigMap
            key: topic-operator-log4j2.properties
  # ...
```

## Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

### 12.2.45.2. **EntityTopicOperatorSpec** schema properties

| 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 | |

| Property | Description |
| --- | --- |
| zookeeperSessionTimeoutSeconds<br><br>integer | Timeout for the ZooKeeper session. |
| startupProbe<br><br>**Probe** | Pod startup checking. |
| livenessProbe<br><br>**Probe** | Pod liveness checking. |
| readinessProbe<br><br>**Probe** | Pod readiness checking. |
| resources<br><br>ResourceRequirements | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |
| topicMetadataMaxAttempts<br><br>integer | The number of attempts at getting topic metadata. |
| logging<br><br>**InlineLogging**, **ExternalLogging** | Logging configuration. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external]. |
| jvmOptions<br><br>**JvmOptions** | JVM Options for pods. |

## 12.2.46. **EntityUserOperatorSpec** schema reference

Used in: **EntityOperatorSpec**

Full list of **EntityUserOperatorSpec** schema properties

Configures the User Operator.

### 12.2.46.1. **logging**

The User Operator has a configurable logger:

- **rootLogger.level**

The User Operator uses the Apache **log4j2** logger implementation.

Use the **logging** property in the **entityOperator.userOperator** field of 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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j2.properties**. Both **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. A ConfigMap using the exact logging configuration specified is created with the custom resource when the Cluster Operator is running, then recreated after each reconciliation. If you do not specify a custom ConfigMap, default logging settings are used. If a specific logger value is not set, upper-level logger settings are inherited for that logger. 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/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
  # ...
```

### External logging

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-topic-namespace
```

```
    reconciliationIntervalSeconds: 60
    logging:
      type: external
      valueFrom:
        configMapKeyRef:
          name: customConfigMap
          key: user-operator-log4j2.properties
  # ...
```

## Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

### 12.2.46.2. **EntityUserOperatorSpec** schema properties

| Property | Description |
|---|---|
| watchedNamespace<br><br>string | The namespace the User Operator should watch. |
| image<br><br>string | The image to use for the User Operator. |
| reconciliationIntervalSeconds<br><br>integer | Interval between periodic reconciliations. |
| zookeeperSessionTimeoutSeconds<br><br>integer | **The zookeeperSessionTimeoutSeconds property has been deprecated.** This property has been deprecated because ZooKeeper is not used anymore by the User Operator. Timeout for the ZooKeeper session. |
| secretPrefix<br><br>string | The prefix that will be added to the KafkaUser name to be used as the Secret name. |
| livenessProbe<br><br>**Probe** | Pod liveness checking. |
| readinessProbe<br><br>**Probe** | Pod readiness checking. |
| resources | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |

| Property | Description |
| --- | --- |
| ResourceRequirements | |
| logging <br><br> **InlineLogging**, **ExternalLogging** | Logging configuration. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external]. |
| jvmOptions <br><br> **JvmOptions** | JVM Options for pods. |

### 12.2.47. TlsSidecar schema reference

Used in: **CruiseControlSpec**, **EntityOperatorSpec**

Full list of **TlsSidecar** schema properties

Configures a TLS sidecar, which 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 communication between components and ZooKeeper.

The TLS sidecar is used in the Entity Operator.

The TLS sidecar is configured using the **tlsSidecar** property in **Kafka.spec.entityOperator**.

The TLS sidecar supports the following additional options:

- **image**

- **resources**

- **logLevel**

- **readinessProbe**

- **livenessProbe**

The **resources** property specifies the memory and CPU resources allocated for the TLS sidecar.

The **image** property configures the container image which will be used.

The **readinessProbe** and **livenessProbe** properties configure healthcheck probes for the TLS sidecar.

The **logLevel** property specifies the logging level. The following logging levels are supported:

- emerg

- alert

- crit

- err

- warning

- notice

- info

- debug

The default value is *notice.*

## Example TLS sidecar configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  entityOperator:
    # ...
    tlsSidecar:
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
    # ...
```

### 12.2.47.1. TlsSidecar schema properties

| Property | Description |
| --- | --- |
| image | The docker image for the container. |
| string | |
| livenessProbe **Probe** | Pod liveness checking. |
| logLevel | The log level for the TLS sidecar. Default value is **notice**. |
| string (one of [emerg, debug, crit, err, alert, warning, notice, info]) | |
| readinessProbe **Probe** | Pod readiness checking. |

| Property | Description |
|---|---|
| resources<br><br>**ResourceRequirements** | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |

### 12.2.48. EntityOperatorTemplate schema reference

Used in: **EntityOperatorSpec**

| Property | Description |
|---|---|
| deployment<br><br>**ResourceTemplate** | Template for Entity Operator **Deployment**. |
| pod<br><br>**PodTemplate** | Template for Entity Operator **Pods**. |
| topicOperatorContainer<br><br>**ContainerTemplate** | Template for the Entity Topic Operator container. |
| userOperatorContainer<br><br>**ContainerTemplate** | Template for the Entity User Operator container. |
| tlsSidecarContainer<br><br>**ContainerTemplate** | Template for the Entity Operator TLS sidecar container. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the Entity Operator service account. |

### 12.2.49. 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<br><br>boolean | If true then Certificate Authority certificates will be generated automatically. Otherwise the user will need to provide a Secret with the CA certificate. Default is true. |
| generateSecretOwnerReference<br><br>boolean | If **true**, the Cluster and Client CA Secrets are configured with the **ownerReference** set to the **Kafka** resource. If the **Kafka** resource is deleted when **true**, the CA Secrets are also deleted. If **false**, the **ownerReference** is disabled. If the **Kafka** resource is deleted when **false**, the CA Secrets are retained and available for reuse. Default is **true**. |
| validityDays<br><br>integer | The number of days generated certificates should be valid for. The default is 365. |
| renewalDays<br><br>integer | The number of days in the certificate renewal period. This is the number of days before the a certificate expires during which renewal actions may be performed. When **generateCertificateAuthority** is true, this will cause the generation of a new certificate. When **generateCertificateAuthority** is true, this will cause extra logging at WARN level about the pending certificate expiry. Default is 30. |
| certificateExpirationPolicy<br><br>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. |

## 12.2.50. **CruiseControlSpec** schema reference

Used in: **KafkaSpec**

Full list of **CruiseControlSpec** schema properties

Configures a Cruise Control cluster.

Configuration options relate to:

- Goals configuration
- Capacity limits for resource distribution goals

### 12.2.50.1. **config**

Use the **config** properties to configure Cruise Control options as keys.

Standard Cruise Control configuration may be provided, restricted to those properties not managed directly by AMQ Streams.

Configuration options that cannot be configured relate to the following:

- Security (Encryption, Authentication, and Authorization)

- Connection to the Kafka cluster

- Client ID configuration

- ZooKeeper connectivity

- Web server configuration

- Self healing

The values can be one of the following JSON types:

- String

- Number

- Boolean

You can specify and configure the options listed in the Cruise Control documentation with the exception of those options that are managed directly by AMQ Streams. See the description of the **config** property for a list of forbidden prefixes.

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 Cruise Control.

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 **webserver** properties to enable Cross-Origin Resource Sharing (CORS).

Example Cruise Control configuration

```
apiVersion: kafka.strimzi.io/v1beta2
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
      webserver.http.cors.enabled: true
      webserver.http.cors.origin: "*"
      webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
    # ...
```

### 12.2.50.2. Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is a HTTP mechanism for controlling access to REST APIs. Restrictions can be on access methods or originating URLs of client applications. You can enable CORS with Cruise Control using the **webserver.http.cors.enabled** property in the  **config**. When enabled, CORS permits read access to the Cruise Control REST API from applications that have different originating URLs than AMQ Streams. This allows applications from specified origins to use **GET** requests to fetch information about the Kafka cluster through the Cruise Control API. For example, applications can fetch information on the current cluster load or the most recent optimization proposal. **POST** requests are not permitted.

> **NOTE**
>
> For more information on using CORS with Cruise Control, see REST APIs in the Cruise Control Wiki.

### Enabling CORS for Cruise Control

You enable and configure CORS in **Kafka.spec.cruiseControl.config**.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    config:
      webserver.http.cors.enabled: true  1
      webserver.http.cors.origin: "*"  2
      webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"  3

      # ...
```

**1**  Enables CORS.

**2**  Specifies permitted origins for the **Access-Control-Allow-Origin** HTTP response header. You can use a wildcard or specify a single origin as a URL. If you use a wildcard, a response is returned following requests from any origin.

**3**  Exposes specified header names for the **Access-Control-Expose-Headers** HTTP response header. Applications in permitted origins can read responses with the specified headers.

### 12.2.50.3. Cruise Control REST API security

The Cruise Control REST API is secured with HTTP Basic authentication and SSL to protect the cluster against potentially destructive Cruise Control operations, such as decommissioning Kafka brokers. We recommend that Cruise Control in AMQ Streams is **only used with these settings enabled**.

However, it is possible to disable these settings by specifying the following Cruise Control configuration:

- To disable the built-in HTTP Basic authentication, set **webserver.security.enable** to **false**.

- To disable the built-in SSL, set **webserver.ssl.enable** to **false**.

**Cruise Control configuration to disable API authorization, authentication, and SSL**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    config:
      webserver.security.enable: false
      webserver.ssl.enable: false
  # ...
```

### 12.2.50.4. brokerCapacity

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:

- **cpu** – CPU resource in millicores or CPU cores (Default: 1)

- **inboundNetwork** – Inbound network throughput in byte units per second (Default: 10000KiB/s)

- **outboundNetwork** – Outbound network throughput in byte units per second (Default: 10000KiB/s)

For network throughput, use an integer value with standard OpenShift byte units (K, M, G) or their bibyte (power of two) equivalents (Ki, Mi, Gi) per second.

> **NOTE**
>
> Disk and CPU capacity limits are automatically generated by AMQ Streams, so you do not need to set them. In order to guarantee accurate rebalance proposals when using CPU goals, you can set CPU requests equal to CPU limits in **Kafka.spec.kafka.resources**. That way, all CPU resources are reserved upfront and are always available. This configuration allows Cruise Control to properly evaluate the CPU utilization when preparing the rebalance proposals based on CPU goals. In cases where you cannot set CPU requests equal to CPU limits in **Kafka.spec.kafka.resources**, you can set the CPU capacity manually for the same accuracy.

**Example Cruise Control brokerCapacity configuration using bibyte units**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    brokerCapacity:
      cpu: "2"
      inboundNetwork: 10000KiB/s
      outboundNetwork: 10000KiB/s
    # ...
```

### 12.2.50.5. Capacity overrides

Brokers might be running on nodes with heterogeneous network or CPU resources. If that's the case, specify **overrides** that set the network capacity and CPU limits for each broker. The overrides ensure an accurate rebalance between the brokers. Override capacity limits can be set for the following broker resources:

- **cpu** – CPU resource in millicores or CPU cores (Default: 1)

- **inboundNetwork** – Inbound network throughput in byte units per second (Default: 10000KiB/s)

- **outboundNetwork** – Outbound network throughput in byte units per second (Default: 10000KiB/s)

**An example of Cruise Control capacity overrides configuration using bibyte units**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    brokerCapacity:
      cpu: "1"
      inboundNetwork: 10000KiB/s
      outboundNetwork: 10000KiB/s
      overrides:
      - brokers: [0]
        cpu: "2.755"
        inboundNetwork: 20000KiB/s
        outboundNetwork: 20000KiB/s
      - brokers: [1, 2]
        cpu: 3000m
        inboundNetwork: 30000KiB/s
        outboundNetwork: 30000KiB/s
```

For more information, refer to the BrokerCapacity schema reference .

### 12.2.50.6. Logging configuration

Cruise Control has its own configurable logger:

- **rootLogger.level**

Cruise Control uses the Apache **log4j2** 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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**. Both **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. A ConfigMap using the exact logging configuration specified is created with the custom resource when the Cluster Operator is running, then recreated after each reconciliation. If you do not specify a custom ConfigMap, default logging settings are used. If a specific logger value is not set, upper-level logger settings are inherited for that logger. Here we see examples of **inline** and **external** logging.

### Inline logging

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
  cruiseControl:
    # ...
    logging:
      type: inline
      loggers:
        rootLogger.level: "INFO"
    # ...
```

### External logging

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
  cruiseControl:
    # ...
    logging:
      type: external
      valueFrom:
        configMapKeyRef:
          name: customConfigMap
          key: cruise-control-log4j.properties
    # ...
```

### Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

### 12.2.50.7. **CruiseControlSpec** schema properties

| Property | Description |
| --- | --- |
| image<br><br>string | The docker image for the pods. |
| tlsSidecar<br><br>**TlsSidecar** | **The tlsSidecar property has been deprecated.** TLS sidecar configuration. |
| resources<br><br>ResourceRequirements | CPU and memory resources to reserve for the Cruise Control container. For more information, see the external documentation for core/v1 resourcerequirements. |
| livenessProbe<br><br>**Probe** | Pod liveness checking for the Cruise Control container. |
| readinessProbe<br><br>**Probe** | Pod readiness checking for the Cruise Control container. |
| jvmOptions<br><br>**JvmOptions** | JVM Options for the Cruise Control container. |
| logging<br><br>**InlineLogging**, **ExternalLogging** | Logging configuration (Log4j 2) 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<br><br>**CruiseControlTemplate** | Template to specify how Cruise Control resources, **Deployments** and **Pods**, are generated. |
| brokerCapacity<br><br>**BrokerCapacity** | The Cruise Control **brokerCapacity** configuration. |

| Property | Description |
|----------|-------------|
| config<br><br>map | The Cruise Control configuration. For a full list of configuration options refer to [https://github.com/linkedin/cruise-control/wiki/Configurations](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, capacity.config.file, self.healing., ssl., kafka.broker.failure.detection.enable, topic.config.provider.class (with the exception of: ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols, webserver.http.cors.enabled, webserver.http.cors.origin, webserver.http.cors.exposeheaders, webserver.security.enable, webserver.ssl.enable). |
| metricsConfig<br><br>**JmxPrometheusExporterMetrics** | Metrics configuration. The type depends on the value of the **metricsConfig.type** property within the given object, which must be one of [jmxPrometheusExporter]. |

### 12.2.51. CruiseControlTemplate schema reference

Used in: **CruiseControlSpec**

| Property | Description |
|----------|-------------|
| deployment<br><br>**ResourceTemplate** | Template for Cruise Control **Deployment**. |
| pod<br><br>**PodTemplate** | Template for Cruise Control **Pods**. |
| apiService<br><br>**InternalServiceTemplate** | Template for Cruise Control API **Service**. |
| podDisruptionBudget<br><br>**PodDisruptionBudgetTemplate** | Template for Cruise Control **PodDisruptionBudget**. |

| Property | Description |
|---|---|
| cruiseControlContainer<br><br>**ContainerTemplate** | Template for the Cruise Control container. |
| tlsSidecarContainer<br><br>**ContainerTemplate** | The **tlsSidecarContainer** property has been deprecated. Template for the Cruise Control TLS sidecar container. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the Cruise Control service account. |

## 12.2.52. **BrokerCapacity** schema reference

Used in: **CruiseControlSpec**

| Property | Description |
|---|---|
| disk<br><br>string | The **disk** property has been deprecated. The Cruise Control disk capacity setting has been deprecated, is ignored, and will be removed in the future Broker capacity for disk in bytes. Use a number value with either standard OpenShift byte units (K, M, G, or T), their bibyte (power of two) equivalents (Ki, Mi, Gi, or Ti), or a byte value with or without E notation. For example, 100000M, 100000Mi, 104857600000, or 1e+11. |
| cpuUtilization<br><br>integer | The **cpuUtilization** property has been deprecated. The Cruise Control CPU capacity setting has been deprecated, is ignored, and will be removed in the future Broker capacity for CPU resource utilization as a percentage (0 - 100). |
| cpu<br><br>string | Broker capacity for CPU resource in cores or millicores. For example, 1, 1.500, 1500m. For more information on valid CPU resource units see https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu. |
| inboundNetwork<br><br>string | Broker capacity for inbound network throughput in bytes per second. Use an integer value with standard OpenShift byte units (K, M, G) or their bibyte (power of two) equivalents (Ki, Mi, Gi) per second. For example, 10000KiB/s. |

| Property | Description |
|---|---|
| outboundNetwork<br><br>string | Broker capacity for outbound network throughput in bytes per second. Use an integer value with standard OpenShift byte units (K, M, G) or their bibyte (power of two) equivalents (Ki, Mi, Gi) per second. For example, 10000KiB/s. |
| overrides<br><br>**BrokerCapacityOverride** array | Overrides for individual brokers. The **overrides** property lets you specify a different capacity configuration for different brokers. |

### 12.2.53. **BrokerCapacityOverride** schema reference

Used in: **BrokerCapacity**

| Property | Description |
|---|---|
| brokers<br><br>integer array | List of Kafka brokers (broker identifiers). |
| cpu<br><br>string | Broker capacity for CPU resource in cores or millicores. For example, 1, 1.500, 1500m. For more information on valid CPU resource units see https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu. |
| inboundNetwork<br><br>string | Broker capacity for inbound network throughput in bytes per second. Use an integer value with standard OpenShift byte units (K, M, G) or their bibyte (power of two) equivalents (Ki, Mi, Gi) per second. For example, 10000KiB/s. |
| outboundNetwork<br><br>string | Broker capacity for outbound network throughput in bytes per second. Use an integer value with standard OpenShift byte units (K, M, G) or their bibyte (power of two) equivalents (Ki, Mi, Gi) per second. For example, 10000KiB/s. |

### 12.2.54. **KafkaExporterSpec** schema reference

Used in: **KafkaSpec**

| Property | Description |
| --- | --- |
| image<br><br>string | The docker image for the pods. |
| groupRegex<br><br>string | Regular expression to specify which consumer groups to collect. Default value is **.***. |
| topicRegex<br><br>string | Regular expression to specify which topics to collect. Default value is **.***. |
| resources<br><br>**ResourceRequirements** | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |
| logging<br><br>string | Only log messages with the given severity or above. Valid levels: [**info**, **debug**, **trace**]. Default log level is **info**. |
| enableSaramaLogging<br><br>boolean | Enable Sarama logging, a Go client library used by the Kafka Exporter. |
| template<br><br>**KafkaExporterTemplate** | Customization of deployment templates and pods. |
| livenessProbe<br><br>**Probe** | Pod liveness check. |
| readinessProbe<br><br>**Probe** | Pod readiness check. |

## 12.2.55. **KafkaExporterTemplate** schema reference

Used in: **KafkaExporterSpec**

| Property | Description |
| --- | --- |
| deployment | Template for Kafka Exporter **Deployment**. |

| Property | Description |
|---|---|
| **ResourceTemplate** | |
| pod<br><br>**PodTemplate** | Template for Kafka Exporter **Pods**. |
| service<br><br>**ResourceTemplate** | The **service** property has been deprecated. The Kafka Exporter service has been removed. Template for Kafka Exporter **Service**. |
| container<br><br>**ContainerTemplate** | Template for the Kafka Exporter container. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the Kafka Exporter service account. |

## 12.2.56. **KafkaStatus** schema reference

Used in: **Kafka**

| Property | Description |
|---|---|
| conditions<br><br>**Condition** array | List of status conditions. |
| observedGeneration<br><br>integer | The generation of the CRD that was last reconciled by the operator. |
| listeners<br><br>**ListenerStatus** array | Addresses of the internal and external listeners. |
| clusterId<br><br>string | Kafka cluster Id. |

## 12.2.57. **Condition** schema reference

Used in: **KafkaBridgeStatus**, **KafkaConnectorStatus**, **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. |
| 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 | |

## 12.2.58. ListenerStatus schema reference

Used in: **KafkaStatus**

| Property | Description |
|---|---|
| type | The **type** property has been deprecated, and should now be configured using **name**. The name of the listener. |
| string | |
| name | The name of the listener. |
| 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 | |

| Property | Description |
|---|---|
| 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 | |

### 12.2.59. **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. |
| integer | |

### 12.2.60. **KafkaConnect** schema reference

| Property | Description |
|---|---|
| spec | The specification of the Kafka Connect cluster. |
| **KafkaConnectSpec** | |
| status | The status of the Kafka Connect cluster. |
| **KafkaConnectStatus** | |

### 12.2.61. **KafkaConnectSpec** schema reference

Used in: **KafkaConnect**

Full list of **KafkaConnectSpec** schema properties

Configures a Kafka Connect cluster.

#### 12.2.61.1. **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** 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** 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/v1beta2
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.

#### 12.2.61.2. logging

Kafka Connect 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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**. Both **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. A ConfigMap using the exact logging configuration specified is created with the custom resource when the Cluster Operator is running, then recreated after each reconciliation. If you do not specify a custom ConfigMap, default logging settings are used. If a specific logger value is not set, upper-level logger settings are inherited for that logger. 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/v1beta2
kind: KafkaConnect
spec:
  # ...
  logging:
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  # ...
```

## External logging

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: connect-logging.log4j
  # ...
```

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.

### 12.2.61.3. **KafkaConnectSpec** schema properties

| Property | Description |
|---|---|
| version<br><br>string | The Kafka Connect version. Defaults to 3.3.1. Consult the user documentation to understand the process required to upgrade or downgrade the version. |
| replicas<br><br>integer | The number of pods in the Kafka Connect group. |
| image<br><br>string | The docker image for the pods. |
| bootstrapServers<br><br>string | Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>*:_<port>_ pairs. |
| tls<br><br>**ClientTls** | TLS configuration. |
| authentication<br><br>**KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **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-256, scram-sha-512, plain, oauth]. |
| config<br><br>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<br><br>ResourceRequirements | The maximum limits for CPU and memory resources and the requested initial resources. For more information, see the external documentation for core/v1 resourcerequirements. |
| livenessProbe<br><br>**Probe** | Pod liveness checking. |
| readinessProbe | Pod readiness checking. |

| Property | Description |
|---|---|
| **Probe** | |
| jvmOptions<br><br>**JvmOptions** | JVM Options for pods. |
| jmxOptions<br><br>**KafkaJmxOptions** | JMX Options. |
| logging<br><br>**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]. |
| clientRackInitImage<br><br>string | The image of the init container used for initializing the **client.rack**. |
| rack<br><br>**Rack** | Configuration of the node label which will be used as the **client.rack** consumer configuration. |
| tracing<br><br>**JaegerTracing**, **OpenTelemetryTracing** | 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, opentelemetry]. |
| template<br><br>**KafkaConnectTemplate** | Template for Kafka Connect and Kafka Mirror Maker 2 resources. The template allows users to specify how the **Deployment**, **Pods** and **Service** are generated. |
| externalConfiguration<br><br>**ExternalConfiguration** | Pass data from Secrets or ConfigMaps to the Kafka Connect pods and use them to configure connectors. |
| build<br><br>**Build** | Configures how the Connect container image should be built. Optional. |
| metricsConfig<br><br>**JmxPrometheusExporterMetrics** | Metrics configuration. The type depends on the value of the **metricsConfig.type** property within the given object, which must be one of [jmxPrometheusExporter]. |

## 12.2.62. ClientTls schema reference

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Full list of **ClientTls** schema properties

Configures TLS trusted certificates for connecting KafkaConnect, KafkaBridge, KafkaMirror, KafkaMirrorMaker2 to the cluster.

### 12.2.62.1. trustedCertificates

Provide a list of secrets using the **trustedCertificates** property.

### 12.2.62.2. ClientTls schema properties

| Property | Description |
|---|---|
| trustedCertificates | Trusted certificates for TLS connection. |
| **CertSecretSource** array | |

## 12.2.63. KafkaClientAuthenticationTls schema reference

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Full list of **KafkaClientAuthenticationTls** schema properties

To configure mTLS authentication, set the **type** property to the value **tls**. mTLS uses a TLS certificate to authenticate.

### 12.2.63.1. 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
```
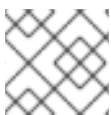
> **NOTE**
>
> mTLS authentication can only be used with TLS connections.

**Example mTLS configuration**

```
authentication:
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: my-public-tls-certificate-file.crt
    key: private.key
```

### 12.2.63.2. KafkaClientAuthenticationTls schema properties

The **type** property is a discriminator that distinguishes use of the **KafkaClientAuthenticationTls** type from **KafkaClientAuthenticationScramSha256**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**. It must have the value **tls** for the type **KafkaClientAuthenticationTls**.

| Property | Description |
|---|---|
| certificateAndKey<br><br>**CertAndKeySecretSource** | Reference to the **Secret** which holds the certificate and private key pair. |
| type<br><br>string | Must be **tls**. |

### 12.2.64. KafkaClientAuthenticationScramSha256 schema reference

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Full list of **KafkaClientAuthenticationScramSha256** schema properties

To configure SASL-based SCRAM-SHA-256 authentication, set the **type** property to **scram-sha-256**. The SCRAM-SHA-256 authentication mechanism requires a username and password.

#### 12.2.64.1. username

Specify the username in the **username** property.

#### 12.2.64.2. 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-256 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**.
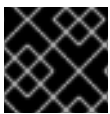
> **IMPORTANT**
>
> Do not specify the actual password in the **password** property.

**Example SASL-based SCRAM-SHA-256 client authentication configuration for Kafka Connect**

```
authentication:
  type: scram-sha-256
  username: my-connect-username
  passwordSecret:
    secretName: my-connect-secret-name
    password: my-connect-password-field
```

### 12.2.64.3. KafkaClientAuthenticationScramSha256 schema properties

| Property | Description |
|---|---|
| passwordSecret<br><br>**PasswordSecretSource** | Reference to the **Secret** which holds the password. |
| type<br><br>string | Must be **scram-sha-256**. |
| username<br><br>string | Username used for the authentication. |

### 12.2.65. PasswordSecretSource schema reference

Used in: **KafkaClientAuthenticationOAuth**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationScramSha256**, **KafkaClientAuthenticationScramSha512**

| Property | Description |
|---|---|
| password<br><br>string | The name of the key in the Secret under which the password is stored. |
| secretName<br><br>string | The name of the Secret containing the password. |

## 12.2.66. KafkaClientAuthenticationScramSha512 schema reference

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Full list of **KafkaClientAuthenticationScramSha512** schema properties

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.

### 12.2.66.1. username

Specify the username in the **username** property.

### 12.2.66.2. 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: LFTIyFRFIMmU2N2Tm
```

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
```

### 12.2.66.3. **KafkaClientAuthenticationScramSha512** schema properties

| 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 | |

### 12.2.67. **KafkaClientAuthenticationPlain** schema reference

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Full list of **KafkaClientAuthenticationPlain** schema properties

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.

### 12.2.67.1. username

Specify the username in the **username** property.

### 12.2.67.2. 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: LFTIyFRFIMmU2N2Tm
```

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
```

### 12.2.67.3. KafkaClientAuthenticationPlain schema properties

The **type** property is a discriminator that distinguishes use of the **KafkaClientAuthenticationPlain** type from **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationOAuth**. It must have the value **plain** for the type **KafkaClientAuthenticationPlain**.

| Property | Description |
|---|---|
| passwordSecret<br><br>**PasswordSecretSource** | Reference to the **Secret** which holds the password. |
| type<br><br>string | Must be **plain**. |
| username<br><br>string | Username used for the authentication. |

### 12.2.68. KafkaClientAuthenticationOAuth schema reference

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2ClusterSpec**, **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Full list of **KafkaClientAuthenticationOAuth** schema properties

To configure OAuth client authentication, set the **type** property to **oauth**.

OAuth authentication can be configured using one of the following options:

- Client ID and secret

- Client ID and refresh token

- Access token

- Username and password

- 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
```

Optionally, **scope** and **audience** can be specified if needed.

## 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
```

## Username and password

OAuth username and password configuration uses the OAuth *Resource Owner Password Grant* mechanism. The mechanism is deprecated, and is only supported to enable integration in environments where client credentials (ID and secret) cannot be used. You might need to use user accounts if your access management system does not support another approach or user accounts are required for authentication.

A typical approach is to create a special user account in your authorization server that represents your client application. You then give the account a long randomly generated password and a very limited set

of permissions. For example, the account can only connect to your Kafka cluster, but is not allowed to use any other services or login to the user interface.

Consider using a refresh token mechanism first.

You can configure the address of your authorization server in the **tokenEndpointUri** property together with the client ID, username and the password used in authentication. The OAuth client will connect to the OAuth server, authenticate using the username, the password, the client ID, and optionally even the client secret to obtain an access token which it will use to authenticate with the Kafka broker.

In the **passwordSecret** property, specify a link to a **Secret** containing the password.

Normally, you also have to configure a **clientId** using a public OAuth client. If you are using a confidential OAuth client, you also have to configure a **clientSecret**.

### An example of OAuth client authentication using username and a password with a public client

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  username: my-username
  passwordSecret:
    secretName: my-password-secret-name
    password: my-password-field-name
  clientId: my-public-client-id
```

### An example of OAuth client authentication using a username and a password with a confidential client

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  username: my-username
  passwordSecret:
    secretName: my-password-secret-name
    password: my-password-field-name
  clientId: my-confidential-client-id
  clientSecret:
    secretName: my-confidential-client-oauth-secret
    key: client-secret
```

Optionally, **scope** and **audience** can be specified if needed.

### 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 resource. 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
```

#### 12.2.68.1. KafkaClientAuthenticationOAuth schema properties

The **type** property is a discriminator that distinguishes use of the **KafkaClientAuthenticationOAuth** type from **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**. It must have the value **oauth** for the type **KafkaClientAuthenticationOAuth**.

| Property | Description |
| --- | --- |
| accessToken<br><br>**GenericSecretSource** | Link to OpenShift Secret containing the access token which was obtained from the authorization server. |
| accessTokenIsJwt<br><br>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**. |
| audience<br><br>string | OAuth audience to use when authenticating against the authorization server. Some authorization servers require the audience to be explicitly set. The possible values depend on how the authorization server is configured. By default, **audience** is not specified when performing the token endpoint request. |
| clientId | |

| Property | Description |
|---|---|
| string | OAuth Client ID which the Kafka client can use to authenticate against the OAuth server and use the token endpoint URI. |
| clientSecret<br><br>**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. |
| connectTimeoutSeconds<br><br>integer | The connect timeout in seconds when connecting to authorization server. If not set, the effective connect timeout is 60 seconds. |
| disableTlsHostnameVerification<br><br>boolean | Enable or disable TLS hostname verification. Default value is **false**. |
| enableMetrics<br><br>boolean | Enable or disable OAuth metrics. Default value is **false**. |
| maxTokenExpirySeconds<br><br>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. |
| passwordSecret<br><br>**PasswordSecretSource** | Reference to the **Secret** which holds the password. |
| readTimeoutSeconds<br><br>integer | The read timeout in seconds when connecting to authorization server. If not set, the effective read timeout is 60 seconds. |
| refreshToken<br><br>**GenericSecretSource** | Link to OpenShift Secret containing the refresh token which can be used to obtain access token from the authorization server. |
| scope<br><br>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. |

| Property | Description |
|---|---|
| tlsTrustedCertificates<br><br>**CertSecretSource** array | Trusted certificates for TLS connection to the OAuth server. |
| tokenEndpointUri<br><br>string | Authorization server token endpoint URI. |
| type<br><br>string | Must be **oauth**. |
| username<br><br>string | Username used for the authentication. |

## 12.2.69. **JaegerTracing** schema reference

**The type JaegerTracing has been deprecated.**

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**

The **type** property is a discriminator that distinguishes use of the **JaegerTracing** type from **OpenTelemetryTracing**. It must have the value **jaeger** for the type **JaegerTracing**.

| Property | Description |
|---|---|
| type<br><br>string | Must be **jaeger**. |

## 12.2.70. **OpenTelemetryTracing** schema reference

Used in: **KafkaBridgeSpec**, **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**, **KafkaMirrorMakerSpec**

The **type** property is a discriminator that distinguishes use of the **OpenTelemetryTracing** type from **JaegerTracing**. It must have the value **opentelemetry** for the type **OpenTelemetryTracing**.

| Property | Description |
|---|---|
| type<br><br>string | Must be **opentelemetry**. |

### 12.2.71. **KafkaConnectTemplate** schema reference

Used in: **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**

| Property | Description |
| --- | --- |
| deployment<br><br>**DeploymentTemplate** | Template for Kafka Connect **Deployment**. |
| pod<br><br>**PodTemplate** | Template for Kafka Connect **Pods**. |
| apiService<br><br>**InternalServiceTemplate** | Template for Kafka Connect API **Service**. |
| connectContainer<br><br>**ContainerTemplate** | Template for the Kafka Connect container. |
| initContainer<br><br>**ContainerTemplate** | Template for the Kafka init container. |
| podDisruptionBudget<br><br>**PodDisruptionBudgetTemplate** | Template for Kafka Connect **PodDisruptionBudget**. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the Kafka Connect service account. |
| clusterRoleBinding<br><br>**ResourceTemplate** | Template for the Kafka Connect ClusterRoleBinding. |
| buildPod<br><br>**PodTemplate** | Template for Kafka Connect Build **Pods**. The build pod is used only on OpenShift. |
| buildContainer<br><br>**ContainerTemplate** | Template for the Kafka Connect Build container. The build container is used only on OpenShift. |

| Property | Description |
|---|---|
| buildConfig<br><br>**BuildConfigTemplate** | Template for the Kafka Connect BuildConfig used to build new container images. The BuildConfig is used only on OpenShift. |
| buildServiceAccount<br><br>**ResourceTemplate** | Template for the Kafka Connect Build service account. |
| jmxSecret<br><br>**ResourceTemplate** | Template for Secret of the Kafka Connect Cluster JMX authentication. |

### 12.2.72. **DeploymentTemplate** schema reference

Used in: **KafkaBridgeTemplate**, **KafkaConnectTemplate**, **KafkaMirrorMakerTemplate**

| Property | Description |
|---|---|
| metadata<br><br>**MetadataTemplate** | Metadata applied to the resource. |
| deploymentStrategy<br><br>string (one of [RollingUpdate, Recreate]) | DeploymentStrategy which will be used for this Deployment. Valid values are **RollingUpdate** and **Recreate**. Defaults to **RollingUpdate**. |

### 12.2.73. **BuildConfigTemplate** schema reference

Used in: **KafkaConnectTemplate**

| Property | Description |
|---|---|
| metadata<br><br>**MetadataTemplate** | Metadata to apply to the **PodDisruptionBudgetTemplate** resource. |
| pullSecret<br><br>string | Container Registry Secret with the credentials for pulling the base image. |

### 12.2.74. **ExternalConfiguration** schema reference

Used in: **KafkaConnectSpec**, **KafkaMirrorMaker2Spec**

Full list of **ExternalConfiguration** schema properties

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**.

When applied, the environment variables and volumes are available for use when developing your connectors.

### 12.2.74.1. **env**

Use the **env** property 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/v1beta2
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 is for a connector to communicate with Amazon AWS. The connector needs to be able to read the **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY**.

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

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

### 12.2.74.2. volumes

Use volumes to mount ConfigMaps or Secrets to a Kafka Connect pod.

Using volumes instead of environment variables is useful in the following scenarios:

- Mounting a properties file that is used to configure Kafka Connect connectors

- Mounting truststores or keystores with TLS certificates

Volumes are mounted inside the Kafka Connect containers on the path **/opt/kafka/external-configuration/<volume-name>**. For example, the files from a volume named **connector-config** will appear in the directory **/opt/kafka/external-configuration/connector-config**.

Configuration *providers* load values from outside the configuration. Use a provider mechanism to avoid passing restricted information over the Kafka Connect REST interface.

- **FileConfigProvider** loads configuration values from properties in a file.

- **DirectoryConfigProvider** loads configuration values from separate files within a directory structure.

Use a comma-separated list if you want to add more than one provider, including custom providers. You can use custom providers to load values from other file locations.

### Using **FileConfigProvider** to load property values

In this example, a Secret named **mysecret** contains connector properties that specify a database name and password:

**Example Secret with database properties**

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |-  1
    dbUsername: my-username  2
    dbPassword: my-password
```

**1**    The connector configuration in properties file format.

**2**    Database username and password properties used in the configuration.

The Secret and the **FileConfigProvider** configuration provider are specified in the Kafka Connect configuration.

- The Secret is mounted to a volume named **connector-config**.

- **FileConfigProvider** is given the alias **file**.

**Example external volumes set to values from a Secret**

```
apiVersion: kafka.strimzi.io/v1beta2
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 is used to define other configuration parameters.

**2**    **FileConfigProvider** 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 a ConfigMap or Secret.

**4**    The name of the Secret.

Placeholders for the property values in the Secret are referenced in the connector configuration. The placeholder structure is **file:*PATH-AND-FILE-NAME*:*PROPERTY***. **FileConfigProvider** reads and

extracts the database *username* and *password* property values from the mounted Secret in connector configurations.

## Example connector configuration showing placeholders for external values

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    database.hostname: 192.168.99.1
    database.port: "3306"
    database.user: "${file:/opt/kafka/external-configuration/connector-config/mysecret:dbUsername}"
    database.password: "${file:/opt/kafka/external-configuration/connector-config/mysecret:dbPassword}"
    database.server.id: "184054"
    #...
```

## Using **DirectoryConfigProvider** to load property values from separate files

In this example, a **Secret** contains TLS truststore and keystore user credentials in separate files.

## Example Secret with user credentials

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

The Secret and the **DirectoryConfigProvider** configuration provider are specified in the Kafka Connect configuration.

- The Secret is mounted to a volume named **connector-config**.

- **DirectoryConfigProvider** is given the alias **directory**.

## Example external volumes set for user credentials files

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

```
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: directory
    config.providers.directory.class: org.apache.kafka.common.config.provider.DirectoryConfigProvider
```

**1**

```
  #...
  externalConfiguration:
    volumes:
      - name: cluster-ca
        secret:
          secretName: my-cluster-cluster-ca-cert
      - name: my-user
        secret:
          secretName: my-user
```

**1** The **DirectoryConfigProvider** provides values from files in a directory. The parameter uses the alias from **config.providers**, taking the form **config.providers.${alias}.class**.

Placeholders for the credentials are referenced in the connector configuration. The placeholder structure is **directory:*PATH*:*FILE-NAME***. **DirectoryConfigProvider** reads and extracts the credentials from the mounted Secret in connector configurations.

**Example connector configuration showing placeholders for external values**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    # ...
    database.history.producer.security.protocol: SSL
    database.history.producer.ssl.truststore.type: PEM
    database.history.producer.ssl.truststore.certificates: "${directory:/opt/kafka/external-configuration/cluster-ca:ca.crt}"
    database.history.producer.ssl.keystore.type: PEM
    database.history.producer.ssl.keystore.certificate.chain: "${directory:/opt/kafka/external-configuration/my-user:user.crt}"
    database.history.producer.ssl.keystore.key: "${directory:/opt/kafka/external-configuration/my-user:user.key}"
    #...
```

## 12.2.74.3. **ExternalConfiguration** schema properties

| Property | Description |
|---|---|
| env<br><br>**ExternalConfigurationEnv** array | Makes data from a Secret or ConfigMap available in the Kafka Connect pods as environment variables. |
| volumes<br><br>**ExternalConfigurationVolumeSource** array | Makes data from a Secret or ConfigMap available in the Kafka Connect pods as volumes. |

## 12.2.75. **ExternalConfigurationEnv** schema reference

Used in: **ExternalConfiguration**

| Property | Description |
|---|---|
| name<br><br>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<br><br>**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. |

## 12.2.76. **ExternalConfigurationEnvVarSource** schema reference

Used in: **ExternalConfigurationEnv**

| Property | Description |
|---|---|
| configMapKeyRef<br><br>ConfigMapKeySelector | Reference to a key in a ConfigMap. For more information, see the external documentation for core/v1 configmapkeyselector. |
| secretKeyRef<br><br>SecretKeySelector | Reference to a key in a Secret. For more information, see the external documentation for core/v1 secretkeyselector. |

## 12.2.77. **ExternalConfigurationVolumeSource** schema reference

Used in: **ExternalConfiguration**

| Property | Description |
|----------|-------------|
| configMap<br><br>ConfigMapVolumeSource | Reference to a key in a ConfigMap. Exactly one Secret or ConfigMap has to be specified. For more information, see the external documentation for core/v1 configmapvolumesource. |
| name<br><br>string | Name of the volume which will be added to the Kafka Connect pods. |
| secret<br><br>SecretVolumeSource | Reference to a key in a Secret. Exactly one Secret or ConfigMap has to be specified. For more information, see the external documentation for core/v1 secretvolumesource. |

## 12.2.78. **Build schema reference**

Used in: **KafkaConnectSpec**

Full list of **Build** schema properties

Configures additional connectors for Kafka Connect deployments.

### 12.2.78.1. output

To build new container images with additional connector plugins, AMQ Streams requires a container registry where the images can be pushed to, stored, and pulled from. AMQ Streams does not run its own container registry, so a registry must be provided. AMQ Streams supports private container registries as well as public registries such as Quay or Docker Hub. The container registry is configured in the **.spec.build.output** section of the **KafkaConnect** custom resource. The **output** configuration, which is required, supports two types: **docker** and **imagestream**.

### Using Docker registry

To use a Docker registry, you have to specify the **type** as **docker**, and the **image** field with the full name of the new container image. The full name must include:

- The address of the registry

- Port number (if listening on a non-standard port)

- The tag of the new container image

Example valid container image names:

- **docker.io/my-org/my-image/my-tag**

- **quay.io/my-org/my-image/my-tag**

- **image-registry.image-registry.svc:5000/myproject/kafka-connect-build:latest**

Each Kafka Connect deployment must use a separate image, which can mean different tags at the most basic level.

If the registry requires authentication, use the **pushSecret** to set a name of the Secret with the registry credentials. For the Secret, use the **kubernetes.io/dockerconfigjson** type and a **.dockerconfigjson** file to contain the Docker credentials. For more information on pulling an image from a private registry, see Create a Secret based on existing Docker credentials .

### Example **output** configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      type: docker 1
      image: my-registry.io/my-org/my-connect-cluster:latest 2
      pushSecret: my-registry-credentials 3
  #...
```

**1** (Required) Type of output used by AMQ Streams.

**2** (Required) Full name of the image used, including the repository and tag.

**3** (Optional) Name of the secret with the container registry credentials.

### Using OpenShift ImageStream

Instead of Docker, you can use OpenShift ImageStream to store a new container image. The ImageStream has to be created manually before deploying Kafka Connect. To use ImageStream, set the **type** to **imagestream**, and use the **image** property to specify the name of the ImageStream and the tag used. For example, **my-connect-image-stream:latest**.

### Example **output** configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      type: imagestream 1
      image: my-connect-build:latest 2
  #...
```

**1** (Required) Type of output used by AMQ Streams.

**2** (Required) Name of the ImageStream and tag.

### 12.2.78.2. **plugins**

Connector plugins are a set of files that define the implementation required to connect to certain types of external system. The connector plugins required for a container image must be configured using the **.spec.build.plugins** property of the **KafkaConnect** custom resource. Each connector plugin must have a name which is unique within the Kafka Connect deployment. Additionally, the plugin artifacts must be listed. These artifacts are downloaded by AMQ Streams, added to the new container image, and used in the Kafka Connect deployment. The connector plugin artifacts can also include additional components, such as (de)serializers. Each connector plugin is downloaded into a separate directory so that the different connectors and their dependencies are properly *sandboxed*. Each plugin must be configured with at least one **artifact**.

**Example plugins configuration with two connector plugins**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins: 1
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb548045e115e
33a9e69b1b2a352bee24df035a0447cb820077af00c03
      - name: camel-telegram
        artifacts:
          - type: tgz
            url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-
telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
            sha512sum:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e8020bf4
187870699819f54ef5859c7846ee4081507f48873479
      #...
```

**1**    (Required) List of connector plugins and their artifacts.

AMQ Streams supports the following types of artifacts:

- JAR files, which are downloaded and used directly

- TGZ archives, which are downloaded and unpacked

- ZIP archives, which are downloaded and unpacked

- Maven artifacts, which uses Maven coordinates

- Other artifacts, which are downloaded and used directly

> **IMPORTANT**
>
> AMQ Streams does not perform any security scanning of the downloaded artifacts. For security reasons, you should first verify the artifacts manually, and configure the checksum verification to make sure the same artifact is used in the automated build and in the Kafka Connect deployment.

## Using JAR artifacts

JAR artifacts represent a JAR file that is downloaded and added to a container image. To use a JAR artifacts, set the **type** property to **jar**, and specify the download location using the **url** property.

Additionally, you can specify a SHA-512 checksum of the artifact. If specified, AMQ Streams will verify the checksum of the artifact while building the new container image.

## Example JAR artifact

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: jar 1
            url: https://my-domain.tld/my-jar.jar 2
            sha512sum: 589...ab4 3
          - type: jar
            url: https://my-domain.tld/my-jar2.jar
  #...
```

**1** (Required) Type of artifact.

**2** (Required) URL from which the artifact is downloaded.

**3** (Optional) SHA-512 checksum to verify the artifact.

## Using TGZ artifacts

TGZ artifacts are used to download TAR archives that have been compressed using Gzip compression. The TGZ artifact can contain the whole Kafka Connect connector, even when comprising multiple different files. The TGZ artifact is automatically downloaded and unpacked by AMQ Streams while building the new container image. To use TGZ artifacts, set the **type** property to **tgz**, and specify the download location using the **url** property.

Additionally, you can specify a SHA-512 checksum of the artifact. If specified, AMQ Streams will verify the checksum before unpacking it and building the new container image.

## Example TGZ artifact

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: tgz ❶
            url: https://my-domain.tld/my-connector-archive.tgz ❷
            sha512sum: 158...jg10 ❸
    #...
```

❶  (Required) Type of artifact.

❷  (Required) URL from which the archive is downloaded.

❸  (Optional) SHA-512 checksum to verify the artifact.

## Using ZIP artifacts

ZIP artifacts are used to download ZIP compressed archives. Use ZIP artifacts in the same way as the TGZ artifacts described in the previous section. The only difference is you specify **type: zip** instead of **type: tgz**.

## Using Maven artifacts

**maven** artifacts are used to specify connector plugin artifacts as Maven coordinates. The Maven coordinates identify plugin artifacts and dependencies so that they can be located and fetched from a Maven repository.

> **NOTE**
>
> The Maven repository must be accessible for the connector build process to add the artifacts to the container image.

## Example Maven artifact

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
```

```
      - type: maven ❶
        repository: https://mvnrepository.com ❷
        group: org.apache.camel.kafkaconnector ❸
        artifact: camel-kafka-connector ❹
        version: 0.11.0 ❺
    #...
```

❶ (Required) Type of artifact.

❷ (Optional) Maven repository to download the artifacts from. If you do not specify a repository, Maven Central repository is used by default.

❸ (Required) Maven group ID.

❹ (Required) Maven artifact type.

❺ (Required) Maven version number.

### Using other artifacts

**other** artifacts represent any kind of file that is downloaded and added to a container image. If you want to use a specific name for the artifact in the resulting container image, use the **fileName** field. If a file name is not specified, the file is named based on the URL hash.

Additionally, you can specify a SHA-512 checksum of the artifact. If specified, AMQ Streams will verify the checksum of the artifact while building the new container image.

### Example other artifact

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: other ❶
            url: https://my-domain.tld/my-other-file.ext ❷
            sha512sum: 589...ab4 ❸
            fileName: name-the-file.ext ❹
    #...
```

❶ (Required) Type of artifact.

❷ (Required) URL from which the artifact is downloaded.

❸ (Optional) SHA-512 checksum to verify the artifact.

❹ (Optional) The name under which the file is stored in the resulting container image.

### 12.2.78.3. **Build** schema properties

| Property | Description |
|---|---|
| output<br><br>**DockerOutput**, **ImageStreamOutput** | Configures where should the newly built image be stored. Required. The type depends on the value of the **output.type** property within the given object, which must be one of [docker, imagestream]. |
| resources<br><br>**ResourceRequirements** | CPU and memory resources to reserve for the build. For more information, see the external documentation for core/v1 resourcerequirements. |
| plugins<br><br>**Plugin** array | List of connector plugins which should be added to the Kafka Connect. Required. |

### 12.2.79. **DockerOutput** schema reference

Used in: **Build**

The **type** property is a discriminator that distinguishes use of the **DockerOutput** type from **ImageStreamOutput**. It must have the value **docker** for the type **DockerOutput**.

| Property | Description |
|---|---|
| image<br><br>string | The full name which should be used for tagging and pushing the newly built image. For example **quay.io/my-organization/my-custom-connect:latest**. Required. |
| pushSecret<br><br>string | Container Registry Secret with the credentials for pushing the newly built image. |
| additionalKanikoOptions<br><br>string array | Configures additional options which will be passed to the Kaniko executor when building the new Connect image. Allowed options are: --customPlatform, --insecure, --insecure-pull, --insecure-registry, --log-format, --log-timestamp, --registry-mirror, --reproducible, --single-snapshot, --skip-tls-verify, --skip-tls-verify-pull, --skip-tls-verify-registry, --verbosity, --snapshotMode, --use-new-run. These options will be used only on OpenShift where the Kaniko executor is used. They will be ignored on OpenShift. The options are described in the Kaniko GitHub repository. Changing this field does not trigger new build of the Kafka Connect image. |
| type | Must be **docker**. |

| Property | Description |
| --- | --- |
| string | |

### 12.2.80. **ImageStreamOutput** schema reference

Used in: **Build**

The **type** property is a discriminator that distinguishes use of the **ImageStreamOutput** type from **DockerOutput**. It must have the value **imagestream** for the type **ImageStreamOutput**.

| Property | Description |
| --- | --- |
| image | The name and tag of the ImageStream where the newly built image will be pushed. For example **my-custom-connect:latest**. Required. |
| string | |
| type | Must be **imagestream**. |
| string | |

### 12.2.81. **Plugin** schema reference

Used in: **Build**

| Property | Description |
| --- | --- |
| name | The unique name of the connector plugin. Will be used to generate the path where the connector artifacts will be stored. The name has to be unique within the KafkaConnect resource. The name has to follow the following pattern: **^[a-z][-_a-z0-9]*[a-z]$**. Required. |
| string | |
| artifacts | List of artifacts which belong to this connector plugin. Required. |
| **JarArtifact**, **TgzArtifact**, **ZipArtifact**, **MavenArtifact**, **OtherArtifact** array | |

### 12.2.82. **JarArtifact** schema reference

Used in: **Plugin**

| Property | Description |
|---|---|
| url<br><br>string | URL of the artifact which will be downloaded. AMQ Streams does not do any security scanning of the downloaded artifacts. For security reasons, you should first verify the artifacts manually and configure the checksum verification to make sure the same artifact is used in the automated build. Required for **jar**, **zip**, **tgz** and **other** artifacts. Not applicable to the **maven** artifact type. |
| sha512sum<br><br>string | SHA512 checksum of the artifact. Optional. If specified, the checksum will be verified while building the new container. If not specified, the downloaded artifact will not be verified. Not applicable to the **maven** artifact type. |
| insecure<br><br>boolean | By default, connections using TLS are verified to check they are secure. The server certificate used must be valid, trusted, and contain the server name. By setting this option to **true**, all TLS verification is disabled and the artifact will be downloaded, even when the server is considered insecure. |
| type<br><br>string | Must be **jar**. |

## 12.2.83. **TgzArtifact** schema reference

Used in: **Plugin**

| Property | Description |
|---|---|
| url<br><br>string | URL of the artifact which will be downloaded. AMQ Streams does not do any security scanning of the downloaded artifacts. For security reasons, you should first verify the artifacts manually and configure the checksum verification to make sure the same artifact is used in the automated build. Required for **jar**, **zip**, **tgz** and **other** artifacts. Not applicable to the **maven** artifact type. |
| sha512sum<br><br>string | SHA512 checksum of the artifact. Optional. If specified, the checksum will be verified while building the new container. If not specified, the downloaded artifact will not be verified. Not applicable to the **maven** artifact type. |

| Property | Description |
| --- | --- |
| insecure<br><br>boolean | By default, connections using TLS are verified to check they are secure. The server certificate used must be valid, trusted, and contain the server name. By setting this option to **true**, all TLS verification is disabled and the artifact will be downloaded, even when the server is considered insecure. |
| type<br><br>string | Must be **tgz**. |

## 12.2.84. **ZipArtifact** schema reference

Used in: **Plugin**

| Property | Description |
| --- | --- |
| url<br><br>string | URL of the artifact which will be downloaded. AMQ Streams does not do any security scanning of the downloaded artifacts. For security reasons, you should first verify the artifacts manually and configure the checksum verification to make sure the same artifact is used in the automated build. Required for **jar**, **zip**, **tgz** and **other** artifacts. Not applicable to the **maven** artifact type. |
| sha512sum<br><br>string | SHA512 checksum of the artifact. Optional. If specified, the checksum will be verified while building the new container. If not specified, the downloaded artifact will not be verified. Not applicable to the **maven** artifact type. |
| insecure<br><br>boolean | By default, connections using TLS are verified to check they are secure. The server certificate used must be valid, trusted, and contain the server name. By setting this option to **true**, all TLS verification is disabled and the artifact will be downloaded, even when the server is considered insecure. |
| type<br><br>string | Must be **zip**. |

## 12.2.85. **MavenArtifact** schema reference

Used in: **Plugin**

The **type** property is a discriminator that distinguishes use of the **MavenArtifact** type from **JarArtifact**, **TgzArtifact**, **ZipArtifact**, **OtherArtifact**. It must have the value **maven** for the type **MavenArtifact**.

| Property | Description |
| --- | --- |
| repository<br><br>string | Maven repository to download the artifact from. Applicable to the **maven** artifact type only. |
| group<br><br>string | Maven group id. Applicable to the **maven** artifact type only. |
| artifact<br><br>string | Maven artifact id. Applicable to the **maven** artifact type only. |
| version<br><br>string | Maven version number. Applicable to the **maven** artifact type only. |
| type<br><br>string | Must be **maven**. |

## 12.2.86. **OtherArtifact** schema reference

Used in: **Plugin**

| Property | Description |
| --- | --- |
| url<br><br>string | URL of the artifact which will be downloaded. AMQ Streams does not do any security scanning of the downloaded artifacts. For security reasons, you should first verify the artifacts manually and configure the checksum verification to make sure the same artifact is used in the automated build. Required for **jar**, **zip**, **tgz** and **other** artifacts. Not applicable to the **maven** artifact type. |
| sha512sum<br><br>string | SHA512 checksum of the artifact. Optional. If specified, the checksum will be verified while building the new container. If not specified, the downloaded artifact will not be verified. Not applicable to the **maven** artifact type. |

| Property | Description |
| --- | --- |
| fileName | Name under which the artifact will be stored. |
| string | |
| insecure | By default, connections using TLS are verified to check they are secure. The server certificate used must be valid, trusted, and contain the server name. By setting this option to **true**, all TLS verification is disabled and the artifact will be downloaded, even when the server is considered insecure. |
| boolean | |
| type | Must be **other**. |
| string | |

## 12.2.87. **KafkaConnectStatus** schema reference

Used in: **KafkaConnect**

| 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 | |
| labelSelector | Label selector for pods providing this resource. |
| string | |
| replicas | The current number of pods being used to provide this resource. |
| integer | |

### 12.2.88. **ConnectorPlugin** schema reference

Used in: **KafkaConnectStatus**, **KafkaMirrorMaker2Status**

| Property | Description |
| --- | --- |
| type | The type of the connector plugin. The available types are **sink** and **source**. |
| string | |
| version | The version of the connector plugin. |
| string | |
| class | The class of the connector plugin. |
| string | |

### 12.2.89. **KafkaTopic** schema reference

| Property | Description |
| --- | --- |
| spec | The specification of the topic. |
| **KafkaTopicSpec** | |
| status | The status of the topic. |
| **KafkaTopicStatus** | |

### 12.2.90. **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. When absent this will default to the broker configuration for **num.partitions**. |
| integer | |
| replicas | The number of replicas the topic should have. When absent this will default to the broker configuration for **default.replication.factor**. |

| Property | Description |
| --- | --- |
| integer | |
| config | The topic configuration. |
| map | |
| topicName | 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. |
| string | |

### 12.2.91. **KafkaTopicStatus** schema reference

Used in: **KafkaTopic**

| Property | Description |
| --- | --- |
| conditions | List of status conditions. |
| **Condition** array | |
| observedGeneration | The generation of the CRD that was last reconciled by the operator. |
| integer | |
| topicName | Topic name. |
| string | |

### 12.2.92. **KafkaUser** schema reference

| Property | Description |
| --- | --- |
| spec | The specification of the user. |
| **KafkaUserSpec** | |
| status | The status of the Kafka User. |
| **KafkaUserStatus** | |

### 12.2.93. **KafkaUserSpec** schema reference

Used in: **KafkaUser**

| Property | Description |
| --- | --- |
| authentication<br><br>**KafkaUserTlsClientAuthentication**,<br>**KafkaUserTlsExternalClientAuthentication**,<br>**KafkaUserScramSha512ClientAuthentication** | Authentication mechanism enabled for this Kafka user. The supported authentication mechanisms are **scram-sha-512**, **tls**, and **tls-external**.<br><br><ul><li>**scram-sha-512** generates a secret with SASL SCRAM-SHA-512 credentials.</li><li>**tls** generates a secret with user certificate for mutual TLS authentication.</li><li>**tls-external** does not generate a user certificate. But prepares the user for using mutual TLS authentication using a user certificate generated outside the User Operator. ACLs and quotas set for this user are configured in the **CN=\<username\>** format.</li></ul>Authentication is optional. If authentication is not configured, no credentials are generated. ACLs and quotas set for the user are configured in the **\<username\>** format suitable for SASL authentication. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, tls-external, scram-sha-512]. |
| authorization<br><br>**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<br><br>**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<br><br>**KafkaUserTemplate** | Template to specify how Kafka User **Secrets** are generated. |

### 12.2.94. KafkaUserTlsClientAuthentication schema reference

Used in: **KafkaUserSpec**

The **type** property is a discriminator that distinguishes use of the **KafkaUserTlsClientAuthentication** type from **KafkaUserTlsExternalClientAuthentication**, **KafkaUserScramSha512ClientAuthentication**. It must have the value **tls** for the type **KafkaUserTlsClientAuthentication**.

| Property | Description |
|----------|-------------|
| type | Must be **tls**. |
| string | |

### 12.2.95. **KafkaUserTlsExternalClientAuthentication** schema reference

Used in: **KafkaUserSpec**

The **type** property is a discriminator that distinguishes use of the
**KafkaUserTlsExternalClientAuthentication** type from **KafkaUserTlsClientAuthentication**,
**KafkaUserScramSha512ClientAuthentication**. It must have the value **tls-external** for the type
**KafkaUserTlsExternalClientAuthentication**.

| Property | Description |
|----------|-------------|
| type | Must be **tls-external**. |
| string | |

### 12.2.96. **KafkaUserScramSha512ClientAuthentication** schema reference

Used in: **KafkaUserSpec**

The **type** property is a discriminator that distinguishes use of the
**KafkaUserScramSha512ClientAuthentication** type from **KafkaUserTlsClientAuthentication**,
**KafkaUserTlsExternalClientAuthentication**. It must have the value **scram-sha-512** for the type
**KafkaUserScramSha512ClientAuthentication**.

| Property | Description |
|----------|-------------|
| password<br><br>**Password** | Specify the password for the user. If not set, a new password is generated by the User Operator. |
| type | Must be **scram-sha-512**. |
| string | |

### 12.2.97. **Password** schema reference

Used in: **KafkaUserScramSha512ClientAuthentication**

| Property | Description |
|---|---|
| valueFrom | Secret from which the password should be read. |
| **PasswordSource** | |

### 12.2.98. `PasswordSource` schema reference

Used in: **Password**

| Property | Description |
|---|---|
| secretKeyRef | Selects a key of a Secret in the resource's namespace. For more information, see the external documentation for core/v1 secretkeyselector. |
| SecretKeySelector | |

### 12.2.99. `KafkaUserAuthorizationSimple` schema reference

Used in: **KafkaUserSpec**

The **type** property is a discriminator that distinguishes use of the **KafkaUserAuthorizationSimple** type 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 | |

### 12.2.100. `AclRule` schema reference

Used in: **KafkaUserAuthorizationSimple**

Full list of **AclRule** schema properties

Configures access control rules for a **KafkaUser** when brokers are using the **AclAuthorizer**.

#### Example **KafkaUser** configuration with authorization

```
apiVersion: kafka.strimzi.io/v1beta2
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
          operations:
            - Read
            - Describe
        - resource:
            type: group
            name: my-group
            patternType: prefix
          operations:
            - Read
```

### 12.2.100.1. 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 **name** value as a prefix and then apply the rule to all resources with names starting with that value.

When **patternType** is set as **literal**, you can set the name to `*` to indicate that the rule applies to all resources.

**Example ACL rule that allows the user to read messages from all topics**

```
  acls:
    - resource:
        type: topic
        name: "*"
```

```
    patternType: literal
operations:
  - Read
```

### 12.2.100.2. type

The **type** of rule, which is to **allow** or **deny** (not currently supported) an operations.

The **type** field is optional. If **type** is unspecified, the ACL rule is treated as an **allow** rule.

### 12.2.100.3. operations

Specify a list of **operations** 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.

### 12.2.100.4. 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.

### 12.2.100.5. **AclRule** schema properties

| Property | Description |
|---|---|
| host | The host from which the action described in the ACL rule is allowed or denied. |
| string | |
| operation | **The operation property has been deprecated, and should now be configured using spec.authorization.acls[*].operations.** Operation which will be allowed or denied. Supported operations are: Read, Write, Create, Delete, Alter, Describe, ClusterAction, AlterConfigs, DescribeConfigs, IdempotentWrite and All. |
| string (one of [Read, Write, Delete, Alter, Describe, All, IdempotentWrite, ClusterAction, Create, AlterConfigs, DescribeConfigs]) | |
| operations | List of operations which will be allowed or denied. Supported operations are: Read, Write, Create, Delete, Alter, Describe, ClusterAction, AlterConfigs, DescribeConfigs, IdempotentWrite and All. |
| string (one or more of [Read, Write, Delete, Alter, Describe, All, IdempotentWrite, ClusterAction, Create, AlterConfigs, DescribeConfigs]) array | |
| resource | Indicates the resource for which given ACL rule applies. The type depends on the value of the **resource.type** property within the given object, which must be one of [topic, group, cluster, transactionalId]. |
| **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource** | |
| type | 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**. |
| string (one of [allow, deny]) | |

### 12.2.101. **AclRuleTopicResource** schema reference

Used in: **AclRule**

The **type** property is a discriminator that distinguishes use of the **AclRuleTopicResource** type from **AclRuleGroupResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource**. It must have the value **topic** for the type **AclRuleTopicResource**.

| Property | Description |
|---|---|
| type | Must be **topic**. |
| string | |
| name | Name of resource for which given ACL rule applies. Can be combined with **patternType** field to use prefix pattern. |
| string | |

| Property | Description |
|---|---|
| patternType  string (one of [prefix, literal]) | Describes the pattern used in the resource field. The supported types are **literal** and **prefix**. With **literal** pattern type, the resource field will be used as a definition of a full topic name. With **prefix** pattern type, the resource name will be used only as a prefix. Default value is **literal**. |

### 12.2.102. **AclRuleGroupResource** schema reference

Used in: **AclRule**

The **type** property is a discriminator that distinguishes use of the **AclRuleGroupResource** type 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**. |

### 12.2.103. **AclRuleClusterResource** schema reference

Used in: **AclRule**

The **type** property is a discriminator that distinguishes use of the **AclRuleClusterResource** type from **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleTransactionalIdResource**. It must have the value **cluster** for the type **AclRuleClusterResource**.

| Property | Description |
|---|---|
| type  string | Must be **cluster**. |

## 12.2.104. **AclRuleTransactionalIdResource** schema reference

Used in: **AclRule**

The **type** property is a discriminator that distinguishes use of the **AclRuleTransactionalIdResource** type 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]) | |

## 12.2.105. **KafkaUserQuotas** schema reference

Used in: **KafkaUserSpec**

Full list of **KafkaUserQuotas** schema properties

Kafka allows a user to set **quotas** to control the use of resources by clients.

### 12.2.105.1. **quotas**

You can configure your clients to use the following types of quotas:

- *Network usage* quotas specify the byte rate threshold for each group of clients sharing a quota.

- *CPU utilization* quotas specify a window for broker requests from clients. The window is the percentage of time for clients to make requests. A client makes requests on the I/O threads and network threads of the broker.

- *Partition mutation* quotas limit the number of partition mutations which clients are allowed to make per second.

A partition mutation quota prevents Kafka clusters from being overwhelmed by concurrent topic operations. Partition mutations occur in response to the following types of user requests:

- Creating partitions for a new topic

- Adding partitions to an existing topic

- Deleting partitions from a topic

You can configure a partition mutation quota to control the rate at which mutations are accepted for user requests.

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.

**Example Kafka user quota configuration**

```
spec:
  quotas:
    producerByteRate: 1048576
    consumerByteRate: 2097152
    requestPercentage: 55
    controllerMutationRate: 10
```

For more information about Kafka user quotas, refer to the Apache Kafka documentation.

### 12.2.105.2. KafkaUserQuotas schema properties

| Property | Description |
| --- | --- |
| consumerByteRate<br><br>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. |
| controllerMutationRate<br><br>number | A quota on the rate at which mutations are accepted for the create topics request, the create partitions request and the delete topics request. The rate is accumulated by the number of partitions created or deleted. |
| producerByteRate<br><br>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<br><br>integer | A quota on the maximum CPU utilization of each client group as a percentage of network and I/O threads. |

### 12.2.106. KafkaUserTemplate schema reference

Used in: **KafkaUserSpec**

Full list of **KafkaUserTemplate** schema properties

Specify additional labels and annotations for the secret created by the User Operator.

### An example showing the KafkaUserTemplate

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

### 12.2.106.1. KafkaUserTemplate schema properties

| Property | Description |
|---|---|
| secret<br><br>**ResourceTemplate** | Template for KafkaUser resources. The template allows users to specify how the **Secret** with password or TLS certificates is generated. |

### 12.2.107. KafkaUserStatus schema reference

Used in: **KafkaUser**

| Property | Description |
|---|---|
| conditions<br><br>**Condition** array | List of status conditions. |
| observedGeneration<br><br>integer | The generation of the CRD that was last reconciled by the operator. |
| username<br><br>string | Username. |

| Property | Description |
| --- | --- |
| secret | The name of **Secret** where the credentials are stored. |
| string | |

### 12.2.108. **KafkaMirrorMaker** schema reference

**The type KafkaMirrorMaker has been deprecated.** Please use **KafkaMirrorMaker2** instead.

| Property | Description |
| --- | --- |
| spec<br><br>**KafkaMirrorMakerSpec** | The specification of Kafka MirrorMaker. |
| status<br><br>**KafkaMirrorMakerStatus** | The status of Kafka MirrorMaker. |

### 12.2.109. **KafkaMirrorMakerSpec** schema reference

Used in: **KafkaMirrorMaker**

Full list of **KafkaMirrorMakerSpec** schema properties

Configures Kafka MirrorMaker.

#### 12.2.109.1. **include**

Use the **include** 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.

#### 12.2.109.2. **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.

#### 12.2.109.3. **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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. Inside the ConfigMap, the logging configuration is described using **log4j.properties**. Both **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. A ConfigMap using the exact logging configuration specified is created with the custom resource when the Cluster Operator is running, then recreated after each reconciliation. If you do not specify a custom ConfigMap, default logging settings are used. If a specific logger value is not set, upper-level logger settings are inherited for that logger. For more information about log levels, see Apache logging services.

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: inline
    loggers:
      mirrormaker.root.logger: "INFO"
  # ...
```

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: mirror-maker-log4j.properties
  # ...
```

### Garbage collector (GC)

Garbage collector logging can also be enabled (or disabled) using the **jvmOptions** property.

### 12.2.109.4. **KafkaMirrorMakerSpec** schema properties

| Property | Description |
| --- | --- |

| Property | Description |
| --- | --- |
| version<br><br>string | The Kafka MirrorMaker version. Defaults to 3.3.1. Consult the documentation to understand the process required to upgrade or downgrade the version. |
| replicas<br><br>integer | The number of pods in the **Deployment**. |
| image<br><br>string | The docker image for the pods. |
| consumer<br><br>**KafkaMirrorMakerConsumerSpec** | Configuration of source cluster. |
| producer<br><br>**KafkaMirrorMakerProducerSpec** | Configuration of target cluster. |
| resources<br><br>ResourceRequirements | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |
| whitelist<br><br>string | The **whitelist** property has been deprecated, and should now be configured using **spec.include**. 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 expression **A\|B**. Or, as a special case, you can mirror all topics using the regular expression **\***. You can also specify multiple regular expressions separated by commas. |
| include<br><br>string | 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 expression **A\|B**. Or, as a special case, you can mirror all topics using the regular expression **\***. You can also specify multiple regular expressions separated by commas. |
| jvmOptions<br><br>**JvmOptions** | JVM Options for pods. |

| Property | Description |
|---|---|
| logging<br><br>**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]. |
| metricsConfig<br><br>**JmxPrometheusExporterMetrics** | Metrics configuration. The type depends on the value of the **metricsConfig.type** property within the given object, which must be one of [jmxPrometheusExporter]. |
| tracing<br><br>**JaegerTracing**, **OpenTelemetryTracing** | 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, opentelemetry]. |
| template<br><br>**KafkaMirrorMakerTemplate** | Template to specify how Kafka MirrorMaker resources, **Deployments** and **Pods**, are generated. |
| livenessProbe<br><br>**Probe** | Pod liveness checking. |
| readinessProbe<br><br>**Probe** | Pod readiness checking. |

## 12.2.110. **KafkaMirrorMakerConsumerSpec** schema reference

Used in: **KafkaMirrorMakerSpec**

Full list of **KafkaMirrorMakerConsumerSpec** schema properties

Configures a MirrorMaker consumer.

### 12.2.110.1. **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.

### 12.2.110.2. **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.

### 12.2.110.3. 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.

### 12.2.110.4. 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.

### 12.2.110.5. **KafkaMirrorMakerConsumerSpec** schema properties

| Property | Description |
|---|---|
| numStreams<br><br>integer | Specifies the number of consumer stream threads to create. |
| offsetCommitInterval<br><br>integer | Specifies the offset auto-commit interval in ms. Default value is 60000. |
| bootstrapServers<br><br>string | A list of host:port pairs for establishing the initial connection to the Kafka cluster. |
| groupId<br><br>string | A unique string that identifies the consumer group this consumer belongs to. |
| authentication<br><br>**KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **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-256, scram-sha-512, plain, oauth]. |

| Property | Description |
| --- | --- |
| config map | The MirrorMaker consumer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, group.id, sasl., security., interceptor.classes (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols). |
| tls<br><br>**ClientTls** | TLS configuration for connecting MirrorMaker to the cluster. |

### 12.2.111. **KafkaMirrorMakerProducerSpec** schema reference

Used in: **KafkaMirrorMakerSpec**

Full list of **KafkaMirrorMakerProducerSpec** schema properties

Configures a MirrorMaker producer.

#### 12.2.111.1. **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.

#### 12.2.111.2. **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.

### 12.2.111.3. **KafkaMirrorMakerProducerSpec** schema properties

| 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-256, scram-sha-512, plain, oauth]. |

| Property | Description |
| --- | --- |
| **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth** | |
| config<br><br>map | 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). |
| tls<br><br>**ClientTls** | TLS configuration for connecting MirrorMaker to the cluster. |

### 12.2.112. KafkaMirrorMakerTemplate schema reference

Used in: **KafkaMirrorMakerSpec**

| Property | Description |
| --- | --- |
| deployment<br><br>**DeploymentTemplate** | Template for Kafka MirrorMaker **Deployment**. |
| pod<br><br>**PodTemplate** | Template for Kafka MirrorMaker **Pods**. |
| podDisruptionBudget<br><br>**PodDisruptionBudgetTemplate** | Template for Kafka MirrorMaker **PodDisruptionBudget**. |
| mirrorMakerContainer<br><br>**ContainerTemplate** | Template for Kafka MirrorMaker container. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the Kafka MirrorMaker service account. |

### 12.2.113. KafkaMirrorMakerStatus schema reference

Used in: **KafkaMirrorMaker**

| Property | Description |
|---|---|
| conditions<br><br>**Condition** array | List of status conditions. |
| observedGeneration<br><br>integer | The generation of the CRD that was last reconciled by the operator. |
| labelSelector<br><br>string | Label selector for pods providing this resource. |
| replicas<br><br>integer | The current number of pods being used to provide this resource. |

## 12.2.114. **KafkaBridge** schema reference

| Property | Description |
|---|---|
| spec<br><br>**KafkaBridgeSpec** | The specification of the Kafka Bridge. |
| status<br><br>**KafkaBridgeStatus** | The status of the Kafka Bridge. |

## 12.2.115. **KafkaBridgeSpec** schema reference

Used in: **KafkaBridge**

Full list of **KafkaBridgeSpec** schema properties

Configures a Kafka Bridge cluster.

Configuration options relate to:

- Kafka cluster bootstrap address

- Security (Encryption, Authentication, and Authorization)

- Consumer configuration

- Producer configuration

- HTTP configuration

### 12.2.115.1. 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.valueFrom.configMapKeyRef.name** property to the name of the ConfigMap containing the external logging configuration. The **logging.valueFrom.configMapKeyRef.name** and **logging.valueFrom.configMapKeyRef.key** properties are mandatory. Default logging is used if the **name** or **key** is not set. 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/v1beta2
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/v1beta2
kind: KafkaBridge
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: bridge-logj42.properties
  # ...
```

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.

### 12.2.115.2. **KafkaBridgeSpec** schema properties

| Property | Description |
|---|---|
| replicas<br><br>integer | The number of pods in the **Deployment**. |
| image<br><br>string | The docker image for the pods. |
| bootstrapServers<br><br>string | A list of host:port pairs for establishing the initial connection to the Kafka cluster. |
| tls<br><br>**ClientTls** | TLS configuration for connecting Kafka Bridge to the cluster. |
| authentication<br><br>**KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **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-256, scram-sha-512, plain, oauth]. |
| http<br><br>**KafkaBridgeHttpConfig** | The HTTP related configuration. |
| adminClient<br><br>**KafkaBridgeAdminClientSpec** | Kafka AdminClient related configuration. |
| consumer<br><br>**KafkaBridgeConsumerSpec** | Kafka consumer related configuration. |
| producer<br><br>**KafkaBridgeProducerSpec** | Kafka producer related configuration. |

| Property | Description |
|---|---|
| resources<br><br>**ResourceRequirements** | CPU and memory resources to reserve. For more information, see the external documentation for core/v1 resourcerequirements. |
| jvmOptions<br><br>**JvmOptions** | **Currently not supported** JVM Options for pods. |
| logging<br><br>**InlineLogging**, **ExternalLogging** | Logging configuration for Kafka Bridge. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external]. |
| clientRackInitImage<br><br>string | The image of the init container used for initializing the **client.rack**. |
| rack<br><br>**Rack** | Configuration of the node label which will be used as the client.rack consumer configuration. |
| enableMetrics<br><br>boolean | Enable the metrics for the Kafka Bridge. Default is false. |
| livenessProbe<br><br>**Probe** | Pod liveness checking. |
| readinessProbe<br><br>**Probe** | Pod readiness checking. |
| template<br><br>**KafkaBridgeTemplate** | Template for Kafka Bridge resources. The template allows users to specify how a **Deployment** and **Pod** is generated. |
| tracing<br><br>**JaegerTracing**, **OpenTelemetryTracing** | 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, opentelemetry]. |

### 12.2.116. `KafkaBridgeHttpConfig` schema reference

Used in: **KafkaBridgeSpec**

Full list of **KafkaBridgeHttpConfig** schema properties

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.

### 12.2.116.1. 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/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  http:
    port: 8080
    cors:
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  # ...
```

### 12.2.116.2. KafkaBridgeHttpConfig schema properties

| Property | Description |
|---|---|
| port | The port which is the server listening on. |
| integer | |
| cors | CORS configuration for the HTTP Bridge. |
| **KafkaBridgeHttpCors** | |

### 12.2.117. KafkaBridgeHttpCors schema reference

Used in: **KafkaBridgeHttpConfig**

| Property | Description |
|---|---|
| allowedOrigins | List of allowed origins. Java regular expressions can be used. |
| string array | |

| Property | Description |
| --- | --- |
| allowedMethods | List of allowed HTTP methods. |
| string array | |

## 12.2.118. **KafkaBridgeAdminClientSpec** schema reference

Used in: **KafkaBridgeSpec**

| Property | Description |
| --- | --- |
| config | The Kafka AdminClient configuration used for AdminClient instances created by the bridge. |
| map | |

## 12.2.119. **KafkaBridgeConsumerSpec** schema reference

Used in: **KafkaBridgeSpec**

Full list of **KafkaBridgeConsumerSpec** schema properties

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/v1beta2
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
    # ...
```

### 12.2.119.1. **KafkaBridgeConsumerSpec** schema properties

| 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 | |

### 12.2.120. **KafkaBridgeProducerSpec** schema reference

Used in: **KafkaBridgeSpec**

Full list of **KafkaBridgeProducerSpec** schema properties

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/v1beta2
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
    # ...
```

### 12.2.120.1. **KafkaBridgeProducerSpec** schema properties

| Property | Description |
| --- | --- |

| Property | Description |
|----------|-------------|
| config<br><br>map | The Kafka producer configuration used for producer instances created by the bridge. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, sasl., security. (with the exception of: ssl.endpoint.identification.algorithm, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols). |

## 12.2.121. **KafkaBridgeTemplate** schema reference

Used in: **KafkaBridgeSpec**

| Property | Description |
|----------|-------------|
| deployment<br><br>**DeploymentTemplate** | Template for Kafka Bridge **Deployment**. |
| pod<br><br>**PodTemplate** | Template for Kafka Bridge **Pods**. |
| apiService<br><br>**InternalServiceTemplate** | Template for Kafka Bridge API **Service**. |
| podDisruptionBudget<br><br>**PodDisruptionBudgetTemplate** | Template for Kafka Bridge **PodDisruptionBudget**. |
| bridgeContainer<br><br>**ContainerTemplate** | Template for the Kafka Bridge container. |
| serviceAccount<br><br>**ResourceTemplate** | Template for the Kafka Bridge service account. |
| initContainer<br><br>**ContainerTemplate** | Template for the Kafka Bridge init container. |

## 12.2.122. **KafkaBridgeStatus** schema reference

Used in: **KafkaBridge**

| Property | Description |
|---|---|
| conditions<br><br>**Condition** array | List of status conditions. |
| observedGeneration<br><br>integer | The generation of the CRD that was last reconciled by the operator. |
| url<br><br>string | The URL at which external client applications can access the Kafka Bridge. |
| labelSelector<br><br>string | Label selector for pods providing this resource. |
| replicas<br><br>integer | The current number of pods being used to provide this resource. |

## 12.2.123. **KafkaConnector** schema reference

| Property | Description |
|---|---|
| spec<br><br>**KafkaConnectorSpec** | The specification of the Kafka Connector. |
| status<br><br>**KafkaConnectorStatus** | The status of the Kafka Connector. |

## 12.2.124. **KafkaConnectorSpec** schema reference

Used in: **KafkaConnector**

| Property | Description |
|---|---|
| class<br><br>string | The Class for the Kafka Connector. |

| Property | Description |
|---|---|
| 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 | |

### 12.2.125. 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 | |
| connectorStatus | The connector status, as reported by the Kafka Connect REST API. |
| map | |
| tasksMax | The maximum number of tasks for the Kafka Connector. |
| integer | |
| topics | The list of topics used by the Kafka Connector. |
| string array | |

### 12.2.126. KafkaMirrorMaker2 schema reference

| Property | Description |
|---|---|

| Property | Description |
|---|---|
| spec<br><br>**KafkaMirrorMaker2Spec** | The specification of the Kafka MirrorMaker 2.0 cluster. |
| status<br><br>**KafkaMirrorMaker2Status** | The status of the Kafka MirrorMaker 2.0 cluster. |

### 12.2.127. **KafkaMirrorMaker2Spec** schema reference

Used in: **KafkaMirrorMaker2**

| Property | Description |
|---|---|
| version<br><br>string | The Kafka Connect version. Defaults to 3.3.1. Consult the user documentation to understand the process required to upgrade or downgrade the version. |
| replicas<br><br>integer | The number of pods in the Kafka Connect group. |
| image<br><br>string | The docker image for the pods. |
| connectCluster<br><br>string | The cluster alias used for Kafka Connect. The alias must match a cluster in the list at **spec.clusters**. |
| clusters<br><br>**KafkaMirrorMaker2ClusterSpec** array | Kafka clusters for mirroring. |
| mirrors<br><br>**KafkaMirrorMaker2MirrorSpec** array | Configuration of the MirrorMaker 2.0 connectors. |
| resources<br><br>ResourceRequirements | The maximum limits for CPU and memory resources and the requested initial resources. For more information, see the external documentation for core/v1 resourcerequirements. |
| livenessProbe | Pod liveness checking. |

| Property | Description |
|---|---|
| **Probe** | |
| readinessProbe | Pod readiness checking. |
| **Probe** | |
| jvmOptions | JVM Options for pods. |
| **JvmOptions** | |
| jmxOptions | JMX Options. |
| **KafkaJmxOptions** | |
| logging | 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]. |
| **InlineLogging**, **ExternalLogging** | |
| clientRackInitImage | The image of the init container used for initializing the **client.rack**. |
| string | |
| rack | Configuration of the node label which will be used as the **client.rack** consumer configuration. |
| **Rack** | |
| tracing | 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, opentelemetry]. |
| **JaegerTracing**, **OpenTelemetryTracing** | |
| template | Template for Kafka Connect and Kafka Mirror Maker 2 resources. The template allows users to specify how the **Deployment**, **Pods** and **Service** are generated. |
| **KafkaConnectTemplate** | |
| externalConfiguration | Pass data from Secrets or ConfigMaps to the Kafka Connect pods and use them to configure connectors. |
| **ExternalConfiguration** | |
| metricsConfig | Metrics configuration. The type depends on the value of the **metricsConfig.type** property within the given object, which must be one of [jmxPrometheusExporter]. |
| **JmxPrometheusExporterMetrics** | |

## 12.2.128. **KafkaMirrorMaker2ClusterSpec** schema reference

Used in: **KafkaMirrorMaker2Spec**

Full list of **KafkaMirrorMaker2ClusterSpec** schema properties

Configures Kafka clusters for mirroring.

### 12.2.128.1. **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.

### 12.2.128.2. **KafkaMirrorMaker2ClusterSpec** schema properties

| Property | Description |
| --- | --- |
| alias<br><br>string | Alias used to reference the Kafka cluster. |
| bootstrapServers<br><br>string | A comma-separated list of **host:port** pairs for establishing the connection to the Kafka cluster. |
| tls<br><br>**ClientTls** | TLS configuration for connecting MirrorMaker 2.0 connectors to a cluster. |
| authentication<br><br>**KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **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-256, scram-sha-512, plain, oauth]. |
| config<br><br>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). |

### 12.2.129. **KafkaMirrorMaker2MirrorSpec** schema reference

Used in: **KafkaMirrorMaker2Spec**

| Property | Description |
| --- | --- |
| sourceCluster<br><br>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<br><br>string | The alias of the target cluster used by the Kafka MirrorMaker 2.0 connectors. The alias must match a cluster in the list at **spec.clusters**. |
| sourceConnector<br><br>**KafkaMirrorMaker2ConnectorSpec** | The specification of the Kafka MirrorMaker 2.0 source connector. |
| heartbeatConnector<br><br>**KafkaMirrorMaker2ConnectorSpec** | The specification of the Kafka MirrorMaker 2.0 heartbeat connector. |
| checkpointConnector<br><br>**KafkaMirrorMaker2ConnectorSpec** | The specification of the Kafka MirrorMaker 2.0 checkpoint connector. |
| topicsPattern<br><br>string | A regular expression matching the topics to be mirrored, for example, "topic1\|topic2\|topic3". Comma-separated lists are also supported. |
| topicsBlacklistPattern<br><br>string | The **topicsBlacklistPattern** property has been deprecated, and should now be configured using **.spec.mirrors.topicsExcludePattern.** A regular expression matching the topics to exclude from mirroring. Comma-separated lists are also supported. |
| topicsExcludePattern<br><br>string | A regular expression matching the topics to exclude from mirroring. Comma-separated lists are also supported. |
| groupsPattern<br><br>string | A regular expression matching the consumer groups to be mirrored. Comma-separated lists are also supported. |

| Property | Description |
| --- | --- |
| groupsBlacklistPattern<br><br>string | The **groupsBlacklistPattern** property has been deprecated, and should now be configured using **.spec.mirrors.groupsExcludePattern**. A regular expression matching the consumer groups to exclude from mirroring. Comma-separated lists are also supported. |
| groupsExcludePattern<br><br>string | A regular expression matching the consumer groups to exclude from mirroring. Comma-separated lists are also supported. |

## 12.2.130. **KafkaMirrorMaker2ConnectorSpec** schema reference

Used in: **KafkaMirrorMaker2MirrorSpec**

| Property | Description |
| --- | --- |
| tasksMax<br><br>integer | The maximum number of tasks for the Kafka Connector. |
| config<br><br>map | The Kafka Connector configuration. The following properties cannot be set: connector.class, tasks.max. |
| pause<br><br>boolean | Whether the connector should be paused. Defaults to false. |

## 12.2.131. **KafkaMirrorMaker2Status** schema reference

Used in: **KafkaMirrorMaker2**

| Property | Description |
| --- | --- |
| conditions<br><br>**Condition** array | List of status conditions. |
| observedGeneration<br><br>integer | The generation of the CRD that was last reconciled by the operator. |

| Property | Description |
|---|---|
| url<br><br>string | The URL of the REST API endpoint for managing and monitoring Kafka Connect connectors. |
| connectorPlugins<br><br>**ConnectorPlugin** array | The list of connector plugins available in this Kafka Connect deployment. |
| connectors<br><br>map array | List of MirrorMaker 2.0 connector statuses, as reported by the Kafka Connect REST API. |
| labelSelector<br><br>string | Label selector for pods providing this resource. |
| replicas<br><br>integer | The current number of pods being used to provide this resource. |

### 12.2.132. **KafkaRebalance** schema reference

| Property | Description |
|---|---|
| spec<br><br>**KafkaRebalanceSpec** | The specification of the Kafka rebalance. |
| status<br><br>**KafkaRebalanceStatus** | The status of the Kafka rebalance. |

### 12.2.133. **KafkaRebalanceSpec** schema reference

Used in: **KafkaRebalance**

| Property | Description |
|---|---|

| Property | Description |
| --- | --- |
| mode<br><br>string (one of [remove-brokers, full, add-brokers]) | Mode to run the rebalancing. The supported modes are **full**, **add-brokers**, **remove-brokers**. If not specified, the **full** mode is used by default.<br><br>• **full** mode runs the rebalancing across all the brokers in the cluster.<br><br>• **add-brokers** mode can be used after scaling up the cluster to move some replicas to the newly added brokers.<br><br>• **remove-brokers** mode can be used before scaling down the cluster to move replicas out of the brokers to be removed. |
| brokers<br><br>integer array | The list of newly added brokers in case of scaling up or the ones to be removed in case of scaling down to use for rebalancing. This list can be used only with rebalancing mode **add-brokers** and **removed-brokers**. It is ignored with**full** mode. |
| goals<br><br>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<br><br>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. |
| rebalanceDisk<br><br>boolean | Enables intra-broker disk balancing, which balances disk space utilization between disks on the same broker. Only applies to Kafka deployments that use JBOD storage with multiple disks. When enabled, inter-broker balancing is disabled. Default is false. |
| excludedTopics<br><br>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 format consult the documentation for that class. |

| Property | Description |
|---|---|
| concurrentPartitionMovementsPerBroker<br><br>integer | The upper bound of ongoing partition replica movements going into/out of each broker. Default is 5. |
| concurrentIntraBrokerPartitionMovements<br><br>integer | The upper bound of ongoing partition replica movements between disks within each broker. Default is 2. |
| concurrentLeaderMovements<br><br>integer | The upper bound of ongoing partition leadership movements. Default is 1000. |
| replicationThrottle<br><br>integer | The upper bound, in bytes per second, on the bandwidth used to move replicas. There is no limit by default. |
| replicaMovementStrategies<br><br>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. |

### 12.2.134. KafkaRebalanceStatus schema reference

Used in: **KafkaRebalance**

| Property | Description |
|---|---|
| conditions<br><br>**Condition** array | List of status conditions. |
| observedGeneration<br><br>integer | The generation of the CRD that was last reconciled by the operator. |
| sessionId<br><br>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<br><br>map | A JSON object describing the optimization result. |

# APPENDIX A. USING YOUR SUBSCRIPTION

AMQ Streams is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

## Accessing Your Account

1. Go to [access.redhat.com](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](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](access.redhat.com/downloads).

2. Locate the **AMQ Streams for Apache Kafka** entries in the **INTEGRATION AND AUTOMATION** category.

3. Select the desired AMQ Streams product. The **Software Downloads** page opens.

4. Click the **Download** link for your component.

## Installing packages with DNF

To install a package and all the package dependencies, use:

```
dnf install <package_name>
```

To install a previously-downloaded package from a local directory, use:

```
dnf install <path_to_download_package>
```

*Revised on 2023-01-17 10:56:27 UTC*