Deploying and Upgrading Strimzi

Table of Contents

Chapter 1. Deployment overview

Strimzi simplifies the process of running Apache Kafka in a Kubernetes cluster.

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

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

Upgrade instructions are provided for Strimzi and Kafka upgrades.

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

1.1. Configuring a deployment

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

You might want to review the configuration options available for Kafka components before you deploy Strimzi. For more information on the configuration options, see *Configuring Strimzi*.

1.1.1. Securing Kafka

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

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

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

1.1.2. Monitoring a deployment

Strimzi supports additional deployment options to monitor your deployment.

- Extract metrics and monitor Kafka components by deploying Prometheus and Grafana with your Kafka cluster.
- Extract additional metrics, particularly related to monitoring consumer lag, by deploying Kafka

Exporter with your Kafka cluster.

- Track messages end-to-end by setting up distributed tracing.

1.1.3. CPU and memory resource limits and requests

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

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

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

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

1.2. Strimzi custom resources

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

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

CRDs also allow Strimzi resources to benefit from native Kubernetes features like CLI accessibility and configuration validation.

1.2.1. Strimzi custom resource example

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

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

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

NOTE Access to manage custom resources is limited to Strimzi administrators. For more information, see Designating Strimzi administrators.

A CRD defines a new `kind` of resource, such as `kind:Kafka`, within a Kubernetes cluster.

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

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

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

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

Kafka topic CRD

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

① The metadata for the topic CRD, its name and a label to identify the CRD.

② The specification for this CRD, including the group (domain) name, the plural name and the supported schema version, which are used in the URL to access the API of the topic. The other names are used to identify instance resources in the CLI. For example, `kubectl get kafkatopic my-topic` or `kubectl get kafkatopics`.

③ The shortname can be used in CLI commands. For example, `kubectl get kt` can be used as an abbreviation instead of `kubectl get kafkatopic`.

④ The information presented when using a `get` command on the custom resource.

⑤ The current status of the CRD as described in the schema reference for the resource.

⑥ openAPIV3Schema validation provides validation for the creation of topic custom resources. For example, a topic requires at least one partition and one replica.

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

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

Kafka topic custom resource

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

① The `kind` and `apiVersion` identify the CRD of which the custom resource is an instance.

② A label, applicable only to `KafkaTopic` and `KafkaUser` resources, that defines the name of the Kafka cluster (which is same as the name of the `Kafka` resource) to which a topic or user belongs.

③ The spec shows the number of partitions and replicas for the topic as well as the configuration parameters for the topic itself. In this example, the retention period for a message to remain in the topic and the segment file size for the log are specified.

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

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

After a `KafkaTopic` custom resource is created, the Topic Operator is notified and corresponding Kafka topics are created in Strimzi.

Additional resources

- Extend the Kubernetes API with CustomResourceDefinitions
- Example configuration files provided with Strimzi

1.3. Using the Kafka Bridge to connect with a Kafka cluster

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

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

Additional resources

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

1.4. Document Conventions

User-replaced values

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

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

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

1.5. Additional resources

- Strimzi Overview
- Configuring Strimzi
- Using the Strimzi Kafka Bridge

Chapter 2. Strimzi installation methods

You can install Strimzi on Kubernetes 1.19 and later in three ways.

Installation method	Description
Installation artifacts (YAML files)	Download the release artifacts from the GitHub releases page. Download the `strimzi-<version>.zip` or `strimzi-<version>.tar.gz` archive file. The archive file contains installation artifacts and example configuration files. Deploy the YAML installation artifacts to your Kubernetes cluster using `kubectl`. You start by deploying the Cluster Operator from `install/cluster-operator` to a single namespace, multiple namespaces, or all namespaces. You can also use the `install/` artifacts to deploy the following: • Strimi administrator roles (`strimzi-admin`) • A standalone Topic Operator (`topic-operator`) • A standalone User Operator (`user-operator`) • Strimzi Drain Cleaner (`drain-cleaner`)
OperatorHub.io	Use the **Strimzi Kafka** operator in the OperatorHub.io to deploy the Cluster Operator. You then deploy Strimzi components using custom resources.
Helm chart	Use a Helm chart to deploy the Cluster Operator. You then deploy Strimzi components using custom resources.

For the greatest flexibility, choose the installation artifacts method. The OperatorHub.io method provides a standard configuration and allows you to take advantage of automatic updates. Helm charts provide a convenient way to manage the installation of applications.

Chapter 3. What is deployed with Strimzi

Apache Kafka components are provided for deployment to Kubernetes with the Strimzi distribution. The Kafka components are generally run as clusters for availability.

A typical deployment incorporating Kafka components might include:

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

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

3.1. Order of deployment

The required order of deployment to a Kubernetes cluster is as follows:

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

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

Chapter 4. Preparing for your Strimzi deployment

This section shows how you prepare for a Strimzi deployment, describing:

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

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

4.1. Deployment prerequisites

To deploy Strimzi, you will need the following:

- A Kubernetes 1.19 and later cluster.

  If you do not have access to a Kubernetes cluster, you can install Strimzi with *Minikube*.

- The `kubectl` command-line tool is installed and configured to connect to the running cluster.

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

`oc` and `kubectl` commands

The `oc` command functions as an alternative to `kubectl`. In almost all cases the example `kubectl` commands used in this guide can be done using `oc` simply by replacing the command name (options and arguments remain the same).

In other words, instead of using:

```
kubectl apply -f your-file
```

when using OpenShift you can use:

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

NOTE As an exception to this general rule, `oc` uses `oc adm` subcommands for *cluster*

> *management* functionality, whereas `kubectl` does not make this distinction. For example, the `oc` equivalent of `kubectl taint` is `oc adm taint`.

4.2. Downloading Strimzi release artifacts

To use deployment files to install Strimzi, download and extract the files from the GitHub releases page.

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

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

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

NOTE Additionally, Strimzi container images are available through the Container Registry. However, we recommend that you use the YAML files provided to deploy Strimzi.

4.3. Example configuration and deployment files

Use the example configuration and deployment files provided with Strimzi to deploy Kafka components with different configurations and monitor your deployment. Example configuration files for custom resources contain important properties and values, which you can extend with additional supported configuration properties for your own deployment.

4.3.1. Example files location

The example files are provided with the downloadable release artifacts from the GitHub releases page.

You can also access the example files directly from the `examples` directory.

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

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

4.3.2. Example files provided with Strimzi

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

Examples directory

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

① `KafkaUser` custom resource configuration, which is managed by the User Operator.

② `KafkaTopic` custom resource configuration, which is managed by Topic Operator.

③ Authentication and authorization configuration for Kafka components. Includes example configuration for TLS and SCRAM-SHA-512 authentication. The Keycloak example includes `Kafka` custom resource configuration and a Keycloak realm specification. You can use the example to try Keycloak authorization services. There is also an example with enabled `oauth` authentication and `keycloak` authorization metrics.

④ `Kafka` custom resource configuration for a deployment of Mirror Maker. Includes example configuration for replication policy and synchronization frequency.

⑤ Metrics configuration, including Prometheus installation and Grafana dashboard files.

⑥ `Kafka` custom resource configuration for a deployment of Kafka. Includes example configuration for an ephemeral or persistent single or multi-node deployment.

⑦ `Kafka` custom resource with a deployment configuration for Cruise Control. Includes `KafkaRebalance` custom resources to generate optimizations proposals from Cruise Control, with example configurations to use the default or user optimization goals.

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

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

Additional resources

- Configuring a Strimzi deployment

4.4. Pushing container images to your own registry

Container images for Strimzi are available in the Container Registry. The installation YAML files provided by Strimzi will pull the images directly from the Container Registry.

If you do not have access to the Container Registry or want to use your own container repository:

1. Pull **all** container images listed here

2. Push them into your own registry
3. Update the image names in the YAML files used in deployment

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

Container image	Namespace/Repository	Description
Kafka	• quay.io/strimzi/kafka:0.33.0-kafka-3.2.0 • quay.io/strimzi/kafka:0.33.0-kafka-3.2.1 • quay.io/strimzi/kafka:0.33.0-kafka-3.2.3 • quay.io/strimzi/kafka:0.33.0-kafka-3.3.1 • quay.io/strimzi/kafka:0.33.0-kafka-3.3.2	Strimzi image for running Kafka, including: • Kafka Broker • Kafka Connect • Kafka MirrorMaker • ZooKeeper • TLS Sidecars
Operator	• quay.io/strimzi/operator:0.33.0	Strimzi image for running the operators: • Cluster Operator • Topic Operator • User Operator • Kafka Initializer
Kafka Bridge	• quay.io/strimzi/kafka-bridge:0.24.0	Strimzi image for running the Strimzi kafka Bridge
Strimzi Drain Cleaner	• quay.io/strimzi/drain-cleaner:0.3.1	Strimzi image for running the Strimzi Drain Cleaner
JmxTrans	• quay.io/strimzi/jmxtrans:0.33.0	Strimzi image for running the Strimzi JmxTrans

4.5. Designating Strimzi administrators

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

- `strimzi-view` allows users to view and list Strimzi resources.
- `strimzi-admin` allows users to also create, edit or delete Strimzi resources.

When you install these roles, they will automatically aggregate (add) these rights to the default Kubernetes cluster roles. `strimzi-view` aggregates to the `view` role, and `strimzi-admin` aggregates to the `edit` and `admin` roles. Because of the aggregation, you might not need to assign these roles to

users who already have similar rights.

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

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

Prerequisites

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

Procedure

1. Create the `strimzi-view` and `strimzi-admin` cluster roles in Kubernetes.

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

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

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

4.6. Installing a local Kubernetes cluster with Minikube

Minikube offers an easy way to get started with Kubernetes. If a Kubernetes cluster is unavailable, you can use Minikube to create a local cluster.

You can download and install Minikube from the Kubernetes website, which also provides documentation. Depending on the number of brokers you want to deploy inside the cluster, and whether you want to run Kafka Connect as well, try running Minikube with at least with 4 GB of RAM instead of the default 2 GB.

Once installed, start Minikube using:

```
minikube start --memory 4096
```

To interact with the cluster, install the `kubectl` utility.

Chapter 5. Deploying Strimzi using installation artifacts

Having prepared your environment for a deployment of Strimzi, you can deploy Strimzi to a Kubernetes cluster. Use the installation files provided with the release artifacts.

You can deploy Strimzi 0.33.0 on Kubernetes 1.19 and later.

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

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

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

5.1. Basic deployment path

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

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

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

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

- Cluster Operator (`Deployment`, `ConfigMap`)
- Strimzi CRDs (`CustomResourceDefinition`)
- RBAC resources (`ClusterRole`, `ClusterRoleBinding`, `RoleBinding`)
- Service account (`ServiceAccount`)

The basic deployment path is as follows:

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

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

5.2. Deploying the Cluster Operator

The Cluster Operator is responsible for deploying and managing Kafka clusters within a Kubernetes cluster.

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

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

5.2.1. Specifying the namespaces the Cluster Operator watches

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

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

NOTE The Cluster Operator can watch one, multiple, or all namespaces in a Kubernetes cluster. The Topic Operator and User Operator watch for `KafkaTopic` and `KafkaUser` resources in a single namespace. For more information, see Watching namespaces with Strimzi operators.

The Cluster Operator watches for changes to the following resources:

- `Kafka` for the Kafka cluster.
- `KafkaConnect` for the Kafka Connect cluster.
- `KafkaConnector` for creating and managing connectors in a Kafka Connect cluster.
- `KafkaMirrorMaker` for the Kafka MirrorMaker instance.
- `KafkaMirrorMaker2` for the Kafka MirrorMaker 2.0 instance.

- `KafkaBridge` for the Kafka Bridge instance.
- `KafkaRebalance` for the Cruise Control optimization requests.

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

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

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

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

5.2.2. Deploying the Cluster Operator to watch a single namespace

This procedure shows how to deploy the Cluster Operator to watch Strimzi resources in a single namespace in your Kubernetes cluster.

Prerequisites

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

Procedure

1. Edit the Strimzi installation files to use the namespace the Cluster Operator is going to be installed into.

   For example, in this procedure the Cluster Operator is installed into the namespace `my-cluster-operator-namespace`.

   On Linux, use:

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

   On MacOS, use:

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

2. Deploy the Cluster Operator:

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

3. Check the status of the deployment:

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

Output shows the deployment name and readiness

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

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

5.2.3. Deploying the Cluster Operator to watch multiple namespaces

This procedure shows how to deploy the Cluster Operator to watch Strimzi resources across multiple namespaces in your Kubernetes cluster.

Prerequisites

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

Procedure

1. Edit the Strimzi installation files to use the namespace the Cluster Operator is going to be installed into.

   For example, in this procedure the Cluster Operator is installed into the namespace `my-cluster-operator-namespace`.

   On Linux, use:

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

   On MacOS, use:

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

2. Edit the `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml` file to add a list of all the namespaces the Cluster Operator will watch to the `STRIMZI_NAMESPACE` environment variable.

   For example, in this procedure the Cluster Operator will watch the namespaces `watched-namespace-1`, `watched-namespace-2`, `watched-namespace-3`.

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
      - name: strimzi-cluster-operator
        image: quay.io/strimzi/operator:0.33.0
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: watched-namespace-1,watched-namespace-2,watched-namespace-3
```

3. For each namespace listed, install the `RoleBindings`.

   In this example, we replace `watched-namespace` in these commands with the namespaces listed in the previous step, repeating them for `watched-namespace-1`, `watched-namespace-2`, `watched-namespace-3`:

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

4. Deploy the Cluster Operator:

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

5. Check the status of the deployment:

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

   Output shows the deployment name and readiness

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

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

5.2.4. Deploying the Cluster Operator to watch all namespaces

This procedure shows how to deploy the Cluster Operator to watch Strimzi resources across all namespaces in your Kubernetes cluster.

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

Prerequisites

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

Procedure

1. Edit the Strimzi installation files to use the namespace the Cluster Operator is going to be installed into.

   For example, in this procedure the Cluster Operator is installed into the namespace `my-cluster-operator-namespace`.

   On Linux, use:

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

   On MacOS, use:

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

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

   ```
   apiVersion: apps/v1
   kind: Deployment
   spec:
     # ...
     template:
       spec:
         # ...
         serviceAccountName: strimzi-cluster-operator
         containers:
         - name: strimzi-cluster-operator
           image: quay.io/strimzi/operator:0.33.0
           imagePullPolicy: IfNotPresent
           env:
           - name: STRIMZI_NAMESPACE
             value: "*"
   ```

```
        # ...
```

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

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

4. Deploy the Cluster Operator to your Kubernetes cluster.

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

5. Check the status of the deployment:

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

Output shows the deployment name and readiness

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

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

5.3. Deploying Kafka

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

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

- Kafka cluster
- Topic Operator
- User Operator

When installing Kafka, Strimzi also installs a ZooKeeper cluster and adds the necessary

configuration to connect Kafka with ZooKeeper.

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

5.3.1. Deploying the Kafka cluster

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

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

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

`kafka-persistent.yaml`
: Deploys a persistent cluster with three ZooKeeper and three Kafka nodes.

`kafka-jbod.yaml`
: Deploys a persistent cluster with three ZooKeeper and three Kafka nodes (each using multiple persistent volumes).

`kafka-persistent-single.yaml`
: Deploys a persistent cluster with a single ZooKeeper node and a single Kafka node.

`kafka-ephemeral.yaml`
: Deploys an ephemeral cluster with three ZooKeeper and three Kafka nodes.

`kafka-ephemeral-single.yaml`
: Deploys an ephemeral cluster with three ZooKeeper nodes and a single Kafka node.

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

Ephemeral cluster
: In general, an ephemeral (or temporary) Kafka cluster is suitable for development and testing purposes, not for production. This deployment uses `emptyDir` volumes for storing broker information (for ZooKeeper) and topics or partitions (for Kafka). Using an `emptyDir` volume means that its content is strictly related to the pod life cycle and is deleted when the pod goes down.

Persistent cluster
: A persistent Kafka cluster uses persistent volumes to store ZooKeeper and Kafka data. A `PersistentVolume` is acquired using a `PersistentVolumeClaim` to make it independent of the actual type of the `PersistentVolume`. The `PersistentVolumeClaim` can use a `StorageClass` to trigger automatic volume provisioning. When no `StorageClass` is specified, Kubernetes will try to use the default `StorageClass`.

The following examples show some common types of persistent volumes:

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

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

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

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

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

Default cluster name and specified Kafka versions

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    version: 3.3.2
    #...
    config:
      #...
      log.message.format.version: "3.3"
      inter.broker.protocol.version: "3.3"
  # ...
```

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create and deploy an ephemeral or persistent cluster.

- To create and deploy an ephemeral cluster:

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

- To create and deploy a persistent cluster:

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

2. Check the status of the deployment:

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

Output shows the pod names and readiness

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

`my-cluster` is the name of the Kafka cluster.

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

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

Additional resources

Kafka cluster configuration

5.3.2. Deploying the Topic Operator using the Cluster Operator

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

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

If you use Strimzi to deploy multiple Kafka clusters into the same namespace, enable the Topic Operator for only one Kafka cluster or use the `watchedNamespace` property to configure the Topic

Operators to watch other namespaces.

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

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

Prerequisites

- The Cluster Operator must be deployed.

Procedure

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

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

2. Configure the Topic Operator `spec` using the properties described in `EntityTopicOperatorSpec` schema reference.

   Use an empty object (`{}`) if you want all properties to use their default values.

3. Create or update the resource:

```
kubectl apply -f <kafka_configuration_file>
```

4. Check the status of the deployment:

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

   Output shows the pod name and readiness

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

   `my-cluster` is the name of the Kafka cluster.

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

5.3.3. Deploying the User Operator using the Cluster Operator

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

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

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

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

Prerequisites

- The Cluster Operator must be deployed.

Procedure

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

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

2. Configure the User Operator `spec` using the properties described in `EntityUserOperatorSpec` schema reference.

   Use an empty object (`{}`) if you want all properties to use their default values.

3. Create or update the resource:

```
kubectl apply -f <kafka_configuration_file>
```

4. Check the status of the deployment:

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

Output shows the pod name and readiness

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

`my-cluster` is the name of the Kafka cluster.

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

5.4. Deploying Kafka Connect

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

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

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

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

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

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

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

5.4.1. Deploying Kafka Connect to your Kubernetes cluster

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

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

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

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

- `examples/connect/kafka-connect.yaml`

Prerequisites

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

Procedure

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

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

2. Check the status of the deployment:

```
kubectl get deployments -n <my_cluster_operator_namespace>
```

Output shows the deployment name and readiness

```
NAME                         READY  UP-TO-DATE  AVAILABLE
my-connect-cluster-connect   1/1    1           1
```

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

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

Additional resources

Kafka Connect cluster configuration

5.4.2. Configuring Kafka Connect for multiple instances

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

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

① The Kafka Connect cluster ID within Kafka.

② Kafka topic that stores connector offsets.

③ Kafka topic that stores connector and task status configurations.

④ Kafka topic that stores connector and task status updates.

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

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

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

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

5.4.3. Adding connectors

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

Source connector

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

Sink connector

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

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

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

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

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

- Using Strimzi's `KafkaConnector` custom resource

- Using the Kafka Connect API

You can also create new connector instances using these options.

Building a new container image with connector plugins automatically

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

Prerequisites

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

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

Procedure

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ①
  #...
  build:
    output: ②
      type: docker
      image: my-registry.io/my-org/my-connect-cluster:latest
      pushSecret: my-registry-credentials
    plugins: ③
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/2.1.1.Final/debezium-connector-postgres-2.1.1.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/0.9.0/camel-telegram-kafka-connector-0.9.0-package.tar.gz
            sha512sum:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e8020bf
4187870699819f54ef5859c7846ee4081507f48873479
  #...
```

① The specification for the Kafka Connect cluster.

② (Required) Configuration of the container registry where new images are pushed.

③ (Required) List of connector plugins and their artifacts to add to the new container image. Each plugin must be configured with at least one `artifact`.

2. Create or update the resource:

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

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

Additional resources

- Kafka Connect `Build` schema reference

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

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

You can use the Kafka container image on Container Registry as a base image for creating your own custom image with additional connector plugins.

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

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create a new `Dockerfile` using `quay.io/strimzi/kafka:0.33.0-kafka-3.3.2` as the base image:

```
FROM quay.io/strimzi/kafka:0.33.0-kafka-3.3.2
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Example plugins file

```
$ tree ./my-plugins/
```

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

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

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

2. Build the container image.
3. Push your custom image to your container registry.
4. Point to the new container image.

   You can point to the image in one of the following ways:

   - Edit the `KafkaConnect.spec.image` property of the `KafkaConnect` custom resource.

     If set, this property overrides the `STRIMZI_KAFKA_CONNECT_IMAGES` environment variable in the

Cluster Operator.

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

① The specification for the Kafka Connect cluster.

② The docker image for the pods.

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

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

Additional resources

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

Deploying KafkaConnector resources

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

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

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

Example connectors

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

Strimzi provides an example `KafkaConnector` configuration file (`examples/connect/source-`

connector.yaml) for the example file connector plugins, which creates the following connector instances as KafkaConnector resources:

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

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

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

Prerequisites

- A Kafka Connect deployment
- The Cluster Operator is running

Procedure

1. Add the FileStreamSourceConnector and FileStreamSinkConnector plugins to Kafka Connect in one of the following ways:
   - Configure Kafka Connect to build a new container image with plugins automatically
   - Create a Docker image from the base Kafka Connect image (manually or using continuous integration)
2. Set the strimzi.io/use-connector-resources annotation to true in the Kafka Connect configuration.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect ①
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
    # ...
```

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

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector ①
  labels:
    strimzi.io/cluster: my-connect-cluster ②
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector ③
```

```
  tasksMax: 2 ④
  autoRestart: ⑤
    enabled: true
  config: ⑥
    file: "/opt/kafka/LICENSE" ⑦
    topic: my-topic ⑧
    # ...
```

① Name of the `KafkaConnector` resource, which is used as the name of the connector. Use any name that is valid for a Kubernetes resource.

② Name of the Kafka Connect cluster to create the connector instance in. Connectors must be deployed to the same namespace as the Kafka Connect cluster they link to.

③ Full name or alias of the connector class. This should be present in the image being used by the Kafka Connect cluster.

④ Maximum number of Kafka Connect tasks that the connector can create.

⑤ Enables automatic restarts of failed connectors and tasks.

⑥ Connector configuration as key-value pairs.

⑦ This example source connector configuration reads data from the `/opt/kafka/LICENSE` file.

⑧ Kafka topic to publish the source data to.

4. Create the source `KafkaConnector` in your Kubernetes cluster:

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

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

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

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

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

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

the Kafka Connect cluster.

② Connector configuration as key-value pairs.

③ Temporary file to publish the source data to.

④ Kafka topic to read the source data from.

7. Create the sink `KafkaConnector` in your Kubernetes cluster:

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

8. Check that the connector resources were created:

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

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

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

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

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

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

Source and sink connector configuration options

The connector configuration is defined in the `spec.config` property of the `KafkaConnector` resource.

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

Table 1. Configuration options for the `FileStreamSource` connector class

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

Table 2. Configuration options for `FileStreamSinkConnector` class

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

Manually restarting connectors

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

Prerequisites

- The Cluster Operator is running.

Procedure

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

```
kubectl get KafkaConnector
```

2. Restart the connector by annotating the `KafkaConnector` resource in Kubernetes.

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

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

3. Wait for the next reconciliation to occur (every two minutes by default).

   The Kafka connector is restarted, as long as the annotation was detected by the reconciliation process. When Kafka Connect accepts the restart request, the annotation is removed from the `KafkaConnector` custom resource.

Manually restarting Kafka connector tasks

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

Prerequisites

- The Cluster Operator is running.

Procedure

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

```
kubectl get KafkaConnector
```

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

```
kubectl describe KafkaConnector <kafka_connector_name>
```

3. Use the ID to restart the connector task by annotating the `KafkaConnector` resource in Kubernetes:

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

   In this example, task `0` is restarted.

4. Wait for the next reconciliation to occur (every two minutes by default).

   The Kafka connector task is restarted, as long as the annotation was detected by the reconciliation process. When Kafka Connect accepts the restart request, the annotation is removed from the `KafkaConnector` custom resource.

Exposing the Kafka Connect API

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

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

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

You can add the connector configuration as a JSON object.

Example curl request to add connector configuration

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
```

```
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
    }
}'
```

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

- `LoadBalancer` or `NodePort` type services
- `Ingress` resources
- OpenShift routes

NOTE The connection is insecure, so allow external access advisedly.

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

Selector configuration for the service

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

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

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

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

Example NetworkPolicy to allow requests to the Kafka Connect API

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-custom-connect-network-policy
spec:
  ingress:
  - from:
    - podSelector: ①
        matchLabels:
```

```
          app: my-connector-manager
    ports:
    - port: 8083
      protocol: TCP
  podSelector:
    matchLabels:
      strimzi.io/cluster: my-connect-cluster
      strimzi.io/kind: KafkaConnect
      strimzi.io/name: my-connect-cluster-connect
  policyTypes:
  - Ingress
```

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

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

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

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

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

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

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

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

5.5. Deploying Kafka MirrorMaker

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

5.5.1. Deploying Kafka MirrorMaker to your Kubernetes cluster

This procedure shows how to deploy a Kafka MirrorMaker cluster to your Kubernetes cluster using

the Cluster Operator.

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

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

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

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

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Deploy Kafka MirrorMaker to your Kubernetes cluster:

   For MirrorMaker:

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

   For MirrorMaker 2.0:

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

2. Check the status of the deployment:

   ```
   kubectl get deployments -n <my_cluster_operator_namespace>
   ```

   Output shows the deployment name and readiness

   ```
   NAME                          READY  UP-TO-DATE  AVAILABLE
   my-mirror-maker-mirror-maker  1/1    1           1
   my-mm2-cluster-mirrormaker2   1/1    1           1
   ```

   `my-mirror-maker` is the name of the Kafka MirrorMaker cluster. `my-mm2-cluster` is the name of the Kafka MirrorMaker 2.0 cluster.

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

Additional resources

- Kafka MirrorMaker cluster configuration

5.6. Deploying Kafka Bridge

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

5.6.1. Deploying Kafka Bridge to your Kubernetes cluster

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

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

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

- `examples/bridge/kafka-bridge.yaml`

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Deploy Kafka Bridge to your Kubernetes cluster:

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

2. Check the status of the deployment:

```
kubectl get deployments -n <my_cluster_operator_namespace>
```

Output shows the deployment name and readiness

```
NAME              READY  UP-TO-DATE  AVAILABLE
my-bridge-bridge  1/1    1           1
```

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

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

Additional resources

- Kafka Bridge cluster configuration

- Using the Strimzi Kafka Bridge

5.6.2. Exposing the Kafka Bridge service to your local machine

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

NOTE	Port forwarding is only suitable for development and testing purposes.

Procedure

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

```
kubectl get pods -o name

pod/kafka-consumer
# ...
pod/my-bridge-bridge-7cbd55496b-nclrt
```

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

```
kubectl port-forward pod/my-bridge-bridge-7cbd55496b-nclrt 8080:8080 &
```

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

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

5.6.3. Accessing the Kafka Bridge outside of Kubernetes

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

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

- `LoadBalancer` or `NodePort` type services
- `Ingress` resources
- OpenShift routes

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

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

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

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

5.7. Alternative standalone deployment options for Strimzi operators

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

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

5.7.1. Deploying the standalone Topic Operator

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

A standalone deployment can operate with any Kafka cluster.

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

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

Prerequisites

- You are running a Kafka cluster for the Topic Operator to connect to.

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

Procedure

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

   Example standalone Topic Operator deployment configuration

   ```
   apiVersion: apps/v1
   ```

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

```
            - name: STRIMZI_SECURITY_PROTOCOL ⑱
              value: "SSL"
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2. If you want to connect to Kafka brokers that are using certificates from a public certificate authority, set `STRIMZI_PUBLIC_CA` to `true`. Set this property to `true`, for example, if you are using Amazon AWS MSK service.
3. If you enabled mTLS with the `STRIMZI_TLS_ENABLED` environment variable, specify the keystore and truststore used to authenticate connection to the Kafka cluster.

   Example mTLS configuration

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

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

   ② The password for accessing the truststore.

   ③ The keystore contains the private key for mTLS authentication.

   ④ The password for accessing the keystore.

4. Deploy the Topic Operator.

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

5. Check the status of the deployment:

   ```
   kubectl get deployments
   ```

   Output shows the deployment name and readiness

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

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

5.7.2. Deploying the standalone User Operator

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

management. You can use a standalone User Operator with a Kafka cluster that is not managed by the Cluster Operator.

A standalone deployment can operate with any Kafka cluster.

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

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

Prerequisites

- You are running a Kafka cluster for the User Operator to connect to.

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

Procedure

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

   Example standalone User Operator deployment configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-user-operator
  labels:
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-user-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE ①
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS ②
              value: my-kafka-bootstrap-address:9092
```

```
            - name: STRIMZI_CA_CERT_NAME ③
              value: my-cluster-clients-ca-cert
            - name: STRIMZI_CA_KEY_NAME ④
              value: my-cluster-clients-ca
            - name: STRIMZI_LABELS ⑤
              value: "strimzi.io/cluster=my-cluster"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS ⑥
              value: "120000"
            - name: STRIMZI_WORK_QUEUE_SIZE ⑦
              value: 10000
            - name: STRIMZI_CONTROLLER_THREAD_POOL_SIZE ⑧
              value: 10
            - name: STRIMZI_USER_OPERATIONS_THREAD_POOL_SIZE ⑨
              value: 4
            - name: STRIMZI_LOG_LEVEL ⑩
              value: INFO
            - name: STRIMZI_GC_LOG_ENABLED ⑪
              value: "true"
            - name: STRIMZI_CA_VALIDITY ⑫
              value: "365"
            - name: STRIMZI_CA_RENEWAL ⑬
              value: "30"
            - name: STRIMZI_JAVA_OPTS ⑭
              value: "-Xmx=512M -Xms=256M"
            - name: STRIMZI_JAVA_SYSTEM_PROPERTIES ⑮
              value: "-Djavax.net.debug=verbose -DpropertyName=value"
            - name: STRIMZI_SECRET_PREFIX ⑯
              value: "kafka-"
            - name: STRIMZI_ACLS_ADMIN_API_SUPPORTED ⑰
              value: "true"
            - name: STRIMZI_MAINTENANCE_TIME_WINDOWS ⑱
              value: '* * 8-10 * * ?;* * 14-15 * * ?'
            - name: STRIMZI_KAFKA_ADMIN_CLIENT_CONFIGURATION ⑲
              value: |
                default.api.timeout.ms=120000
                request.timeout.ms=60000
```

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

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

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

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

⑤ The label to identify the `KafkaUser` resources managed by the User Operator. This does not have to be the name of the Kafka cluster. It can be the label assigned to the `KafkaUser`

resource. If you deploy more than one User Operator, the labels must be unique for each. That is, the operators cannot manage the same resources.

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

⑦ The size of the controller event queue. The size of the queue should be at least as big as the maximal amount of users you expect the User Operator to operate. The default is `1024`.

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

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

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

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

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

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

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

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

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

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

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

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

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

Example mTLS configuration

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

① The Kubernetes `Secret` that contains the public key (`ca.crt`) value of the CA that signs Kafka broker certificates.

② The Kubernetes `Secret` that contains the keystore (`entity-operator.p12`) with the private key and certificate for mTLS authentication against the Kafka cluster. The `Secret` must also contain the password (`entity-operator.password`) for accessing the keystore.

3. Deploy the User Operator.

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

4. Check the status of the deployment:

```
kubectl get deployments
```

Output shows the deployment name and readiness

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

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

Chapter 6. Deploying Strimzi from OperatorHub.io

OperatorHub.io is a catalog of Kubernetes operators sourced from multiple providers. It offers you an alternative way to install a stable version of Strimzi.

The Operator Lifecycle Manager is used for the installation and management of all operators published on OperatorHub.io. Operator Lifecycle Manager is a prerequisite for installing the Strimzi Kafka operator

To install Strimzi, locate *Strimzi* from OperatorHub.io, and follow the instructions provided to deploy the Cluster Operator. After you have deployed the Cluster Operator, you can deploy Strimzi components using custom resources. For example, you can deploy the `Kafka` custom resource, and the installed Cluster Operator will create a Kafka cluster.

Upgrades between versions might include manual steps. Always read the release notes before upgrading.

For information on upgrades, see Cluster Operator upgrade options.

WARNING	Make sure you use the appropriate update channel. Installing Strimzi from the default *stable* channel is generally safe. However, we do not recommend enabling *automatic* OLM updates on the stable channel. An automatic upgrade will skip any necessary steps prior to upgrade. For example, to upgrade from 0.22 or earlier you must first update custom resources to support the `v1beta2` API version. Use automatic upgrades only on version-specific channels.

Chapter 7. Deploying Strimzi using Helm

Helm charts are used to package, configure, and deploy Kubernetes resources. Strimzi provides a Helm chart to deploy the Cluster Operator.

After you have deployed the Cluster Operator this way, you can deploy Strimzi components using custom resources. For example, you can deploy the `Kafka` custom resource, and the installed Cluster Operator will create a Kafka cluster.

For information on upgrades, see Cluster Operator upgrade options.

Prerequisites

- The Helm client must be installed on a local machine.
- Helm must be installed to the Kubernetes cluster.

Procedure

1. Add the Strimzi Helm Chart repository:

```
helm repo add strimzi https://strimzi.io/charts/
```

2. Deploy the Cluster Operator using the Helm command line tool:

```
helm install strimzi/strimzi-kafka-operator
```

3. Verify that the Cluster Operator has been deployed successfully using the Helm command line tool:

```
helm ls
```

4. Deploy Kafka and other Kafka components using custom resources.

Chapter 8. Setting up client access to a Kafka cluster

After you have deployed Strimzi, the procedures in this section explain how to:

- Deploy example producer and consumer clients, which you can use to verify your deployment
- Set up client access to a Kafka cluster using listeners

  The steps to set up access to the Kafka cluster for a client outside Kubernetes are more complex, and require familiarity with the Kafka component configuration procedures.

8.1. Deploying example clients

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

Prerequisites

- The Kafka cluster is available for the clients.

Procedure

1. Deploy a Kafka producer.

```
kubectl run kafka-producer -ti --image=quay.io/strimzi/kafka:0.33.0-kafka-3.3.2
--rm=true --restart=Never -- bin/kafka-console-producer.sh --bootstrap-server
cluster-name-kafka-bootstrap:9092 --topic my-topic
```

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

```
kubectl run kafka-consumer -ti --image=quay.io/strimzi/kafka:0.33.0-kafka-3.3.2
--rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server
cluster-name-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

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

8.2. Setting up client access to a Kafka cluster using listeners

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

Kafka listeners provide access. The following listener types are supported:

- `internal` to connect within the same Kubernetes cluster
- `route` to use OpenShift `Route` and the default HAProxy router
- `loadbalancer` to use loadbalancer services
- `nodeport` to use ports on Kubernetes nodes
- `ingress` to use Kubernetes *Ingress* and the Ingress NGINX Controller for Kubernetes
- `cluster-ip` to expose Kafka using per-broker `ClusterIP` services

The type chosen depends on your requirements, and your environment and infrastructure. For example, loadbalancers might not be suitable for certain infrastructure, such as bare metal, where node ports provide a better option.

In this procedure:

1. An external listener is configured for the Kafka cluster, with TLS encryption and mTLS authentication, and Kafka `simple` authorization enabled.
2. A `KafkaUser` is created for the client, with mTLS authentication, and Access Control Lists (ACLs) defined for `simple` authorization.

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

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

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

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

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

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

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

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

You add the certificates to your client configuration.

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

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

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

Prerequisites

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

Procedure

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

    Example listener configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners: ①
    - name: external ②
      port: 9094 ③
      type: <listener_type> ④
      tls: true ⑤
      authentication:
        type: tls ⑥
      configuration: ⑦
        #...
    authorization: ⑧
      type: simple
      superUsers:
        - super-user-name ⑨
  # ...
```

    ① Configuration options for enabling external listeners are described in the Generic Kafka

listener schema reference.

② Name to identify the listener. Must be unique within the Kafka cluster.

③ Port number used by the listener inside Kafka. The port number has to be unique within a given Kafka cluster. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX. Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.

④ External listener type specified as `route`, `loadbalancer`, `nodeport` or `ingress`. An internal listener is specified as `internal` or `cluster-ip`.

⑤ Required. TLS encryption on the listener. For `route` and `ingress` type listeners it must be set to `true`. For mTLS authentication, also use the `authentication` property.

⑥ Client authentication mechanism on the listener. For server and client authentication using mTLS, you specify `tls: true` and `authentication.type: tls`.

⑦ (Optional) Depending on the requirements of the listener type, you can specify additional listener configuration.

⑧ Authorization specified as `simple`, which uses the `AclAuthorizer` Kafka plugin.

⑨ (Optional) Super users can access all brokers regardless of any access restrictions defined in ACLs.

WARNING

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

2. Create or update the `Kafka` resource.

```
kubectl apply -f <kafka_configuration_file>
```

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

A service is created for each Kafka broker pod.

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

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

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

NOTE

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

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

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

For example:

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

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

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

   Example user configuration

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

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

② Authentication specified as mutual `tls`.

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

5. Create or modify the `KafkaUser` resource.

```
kubectl apply -f USER-CONFIG-FILE
```

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

Example secret

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

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

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

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

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

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

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

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

cluster:

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

10. Configure your client with the truststore credentials to verify the identity of the Kafka cluster.

    Specify the public cluster CA certificate.

    Example truststore configuration

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

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

11. Configure your client with the keystore credentials to verify the user when connecting to the Kafka cluster.

    Specify the public certificate and private key.

    Example keystore configuration

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

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

    Example keystore configuration

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

    Additional resources

    - Listener authentication options
    - Kafka authorization options
    - If you are using an authorization server, you can use token-based OAuth 2.0 authentication

and OAuth 2.0 authorization.

Chapter 9. Introducing metrics

You can use Prometheus and Grafana to monitor your Strimzi deployment.

You can monitor your Strimzi deployment by viewing key metrics on dashboards and setting up alerts that trigger under certain conditions. Metrics are available for each of the components of Strimzi.

You can also collect metrics specific to `oauth` authentication and `opa` or `keycloak` authorization. You do this by setting the `enableMetrics` property to `true` in the listener configuration of the `Kafka` resource. For example, set `enableMetrics` to `true` in `spec.kafka.listeners.authentication` and `spec.kafka.authorization`. Similarly, you can enable metrics for `oauth` authentication in the `KafkaBridge`, `KafkaConnect`, `KafkaMirrorMaker`, and `KafkaMirrorMaker2` custom resources.

To provide metrics information, Strimzi uses Prometheus rules and Grafana dashboards.

When configured with a set of rules for each component of Strimzi, Prometheus consumes key metrics from the pods that are running in your cluster. Grafana then visualizes those metrics on dashboards. Strimzi includes example Grafana dashboards that you can customize to suit your deployment.

Depending on your requirements, you can:

- Set up and deploy Prometheus to expose metrics
- Deploy Kafka Exporter to provide additional metrics
- Use Grafana to present the Prometheus metrics

With Prometheus and Grafana set up, you can use the example Grafana dashboards provided by Strimzi for monitoring.

Additionally, you can configure your deployment to track messages end-to-end by setting up distributed tracing.

NOTE	Strimzi provides example installation files for Prometheus and Grafana. You can use these files as a starting point when trying out monitoring of Strimzi. For further support, try engaging with the Prometheus and Grafana developer communities.

Supporting documentation for metrics and monitoring tools

For more information on the metrics and monitoring tools, refer to the supporting documentation:

- Prometheus
- Prometheus configuration
- Kafka Exporter
- Grafana Labs
- Apache Kafka Monitoring describes JMX metrics exposed by Apache Kafka
- ZooKeeper JMX describes JMX metrics exposed by Apache ZooKeeper

9.1. Monitoring consumer lag with Kafka Exporter

Kafka Exporter is an open source project to enhance monitoring of Apache Kafka brokers and clients. You can configure the `Kafka` resource to deploy Kafka Exporter with your Kafka cluster. Kafka Exporter extracts additional metrics data from Kafka brokers related to offsets, consumer groups, consumer lag, and topics. The metrics data is used, for example, to help identify slow consumers. Lag data is exposed as Prometheus metrics, which can then be presented in Grafana for analysis.

Kafka Exporter reads from the `__consumer_offsets` topic, which stores information on committed offsets for consumer groups. For Kafka Exporter to be able to work properly, consumer groups needs to be in use.

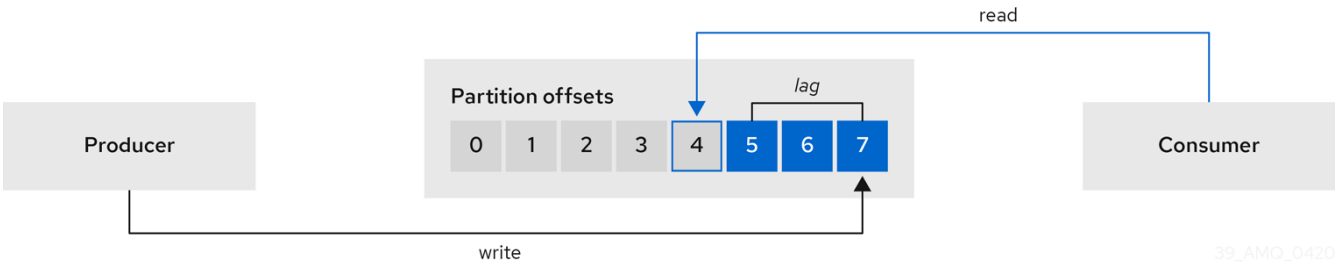
A Grafana dashboard for Kafka Exporter is one of a number of example Grafana dashboards provided by Strimzi.

IMPORTANT	Kafka Exporter provides only additional metrics related to consumer lag and consumer offsets. For regular Kafka metrics, you have to configure the Prometheus metrics in Kafka brokers.

Consumer lag indicates the difference in the rate of production and consumption of messages. Specifically, consumer lag for a given consumer group indicates the delay between the last message in the partition and the message being currently picked up by that consumer.

The lag reflects the position of the consumer offset in relation to the end of the partition log.

Consumer lag between the producer and consumer offset



This difference is sometimes referred to as the *delta* between the producer offset and consumer offset: the read and write positions in the Kafka broker topic partitions.

Suppose a topic streams 100 messages a second. A lag of 1000 messages between the producer offset (the topic partition head) and the last offset the consumer has read means a 10-second delay.

The importance of monitoring consumer lag

For applications that rely on the processing of (near) real-time data, it is critical to monitor consumer lag to check that it does not become too big. The greater the lag becomes, the further the process moves from the real-time processing objective.

Consumer lag, for example, might be a result of consuming too much old data that has not been

purged, or through unplanned shutdowns.

Reducing consumer lag

Use the Grafana charts to analyze lag and to check if actions to reduce lag are having an impact on an affected consumer group. If, for example, Kafka brokers are adjusted to reduce lag, the dashboard will show the *Lag by consumer group* chart going down and the *Messages consumed per minute* chart going up.

Typical actions to reduce lag include:

- Scaling-up consumer groups by adding new consumers
- Increasing the retention time for a message to remain in a topic
- Adding more disk capacity to increase the message buffer

Actions to reduce consumer lag depend on the underlying infrastructure and the use cases Strimzi is supporting. For instance, a lagging consumer is less likely to benefit from the broker being able to service a fetch request from its disk cache. And in certain cases, it might be acceptable to automatically drop messages until a consumer has caught up.

9.2. Monitoring Cruise Control operations

Cruise Control monitors Kafka brokers in order to track the utilization of brokers, topics, and partitions. Cruise Control also provides a set of metrics for monitoring its own performance.

The Cruise Control metrics reporter collects raw metrics data from Kafka brokers. The data is produced to topics that are automatically created by Cruise Control. The metrics are used to generate optimization proposals for Kafka clusters.

Cruise Control metrics are available for real-time monitoring of Cruise Control operations. For example, you can use Cruise Control metrics to monitor the status of rebalancing operations that are running or provide alerts on any anomalies that are detected in an operation's performance.

You expose Cruise Control metrics by enabling the Prometheus JMX Exporter in the Cruise Control configuration.

NOTE	For a full list of available Cruise Control metrics, which are known as *sensors*, see the Cruise Control documentation.

9.2.1. Exposing Cruise Control metrics

If you want to expose metrics on Cruise Control operations, configure the `Kafka` resource to deploy Cruise Control and enable Prometheus metrics in the deployment. You can use your own configuration or use the example `kafka-cruise-control-metrics.yaml` file provided by Strimzi.

You add the configuration to the `metricsConfig` of the `CruiseControl` property in the `Kafka` resource. The configuration enables the Prometheus JMX Exporter to expose Cruise Control metrics through an HTTP endpoint. The HTTP endpoint is scraped by the Prometheus server.

Example metrics configuration for Cruise Control

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
Spec:
  # ...
  cruiseControl:
    # ...
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: cruise-control-metrics
          key: metrics-config.yml
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: cruise-control-metrics
  labels:
    app: strimzi
data:
  metrics-config.yml: |
  # metrics configuration...
```

9.2.2. Viewing Cruise Control metrics

After you expose the Cruise Control metrics, you can use Prometheus or another suitable monitoring system to view information on the metrics data. Strimzi provides an example Grafana dashboard to display visualizations of Cruise Control metrics. The dashboard is a JSON file called `strimzi-cruise-control.json`. The exposed metrics provide the monitoring data when you enable the Grafana dashboard.

Monitoring balancedness scores

Cruise Control metrics include a balancedness score. Balancedness is the measure of how evenly a workload is distributed in a Kafka cluster.

The Cruise Control metric for balancedness score (`balancedness-score`) might differ from the balancedness score in the `KafkaRebalance` resource. Cruise Control calculates each score using `anomaly.detection.goals` which might not be the same as the `default.goals` used in the `KafkaRebalance` resource. The `anomaly.detection.goals` are specified in the `spec.cruiseControl.config` of the `Kafka` custom resource.

NOTE Refreshing the `KafkaRebalance` resource fetches an optimization proposal. The latest cached optimization proposal is fetched if one of the following conditions applies:

- KafkaRebalance `goals` match the goals configured in the `default.goals` section of the `Kafka` resource
- KafkaRebalance `goals` are not specified

Otherwise, Cruise Control generates a new optimization proposal based on KafkaRebalance `goals`. If new proposals are generated with each refresh, this can impact performance monitoring.

Alerts on anomaly detection

Cruise control's *anomaly detector* provides metrics data for conditions that block the generation of optimization goals, such as broker failures. If you want more visibility, you can use the metrics provided by the anomaly detector to set up alerts and send out notifications. You can set up Cruise Control's *anomaly notifier* to route alerts based on these metrics through a specified notification channel. Alternatively, you can set up Prometheus to scrape the metrics data provided by the anomaly detector and generate alerts. Prometheus Alertmanager can then route the alerts generated by Prometheus.

The Cruise Control documentation provides information on `AnomalyDetector` metrics and the anomaly notifier.

9.3. Example metrics files

You can find example Grafana dashboards and other metrics configuration files in the example configuration files provided by Strimzi.

Example metrics files provided with Strimzi

```
metrics
├─── grafana-dashboards ①
│    ├─── strimzi-cruise-control.json
│    ├─── strimzi-kafka-bridge.json
│    ├─── strimzi-kafka-connect.json
│    ├─── strimzi-kafka-exporter.json
│    ├─── strimzi-kafka-mirror-maker-2.json
│    ├─── strimzi-kafka.json
│    ├─── strimzi-operators.json
│    └─── strimzi-zookeeper.json
├─── grafana-install
│    └─── grafana.yaml ②
├─── prometheus-additional-properties
│    └─── prometheus-additional.yaml ③
├─── prometheus-alertmanager-config
│    └─── alert-manager-config.yaml ④
├─── prometheus-install
│     ├─── alert-manager.yaml ⑤
│     ├─── prometheus-rules.yaml ⑥
│     ├─── prometheus.yaml ⑦
│     ├─── strimzi-pod-monitor.yaml ⑧
```

```
├─── kafka-bridge-metrics.yaml ⑨
├─── kafka-connect-metrics.yaml ⑩
├─── kafka-cruise-control-metrics.yaml ⑪
├─── kafka-metrics.yaml ⑫
└─── kafka-mirror-maker-2-metrics.yaml ⑬
```

① Example Grafana dashboards for the different Strimzi components.

② Installation file for the Grafana image.

③ Additional configuration to scrape metrics for CPU, memory and disk volume usage, which comes directly from the Kubernetes cAdvisor agent and kubelet on the nodes.

④ Hook definitions for sending notifications through Alertmanager.

⑤ Resources for deploying and configuring Alertmanager.

⑥ Alerting rules examples for use with Prometheus Alertmanager (deployed with Prometheus).

⑦ Installation resource file for the Prometheus image.

⑧ PodMonitor definitions translated by the Prometheus Operator into jobs for the Prometheus server to be able to scrape metrics data directly from pods.

⑨ Kafka Bridge resource with metrics enabled.

⑩ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Connect.

⑪ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Cruise Control.

⑫ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka and ZooKeeper.

⑬ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Mirror Maker 2.0.

9.3.1. Example Prometheus metrics configuration

Strimzi uses the Prometheus JMX Exporter to expose metrics through an HTTP endpoint, which can be scraped by the Prometheus server.

Grafana dashboards are dependent on Prometheus JMX Exporter relabeling rules, which are defined for Strimzi components in the custom resource configuration.

A label is a name-value pair. Relabeling is the process of writing a label dynamically. For example, the value of a label may be derived from the name of a Kafka server and client ID.

Strimzi provides example custom resource configuration YAML files with relabeling rules. When deploying Prometheus metrics configuration, you can can deploy the example custom resource or copy the metrics configuration to your own custom resource definition.

Table 3. Example custom resources with metrics configuration

Component	Custom resource	Example YAML file
Kafka and ZooKeeper	`Kafka`	`kafka-metrics.yaml`
Kafka Connect	`KafkaConnect`	`kafka-connect-metrics.yaml`

Component	Custom resource	Example YAML file
Kafka MirrorMaker 2.0	`KafkaMirrorMaker2`	`kafka-mirror-maker-2-metrics.yaml`
Kafka Bridge	`KafkaBridge`	`kafka-bridge-metrics.yaml`
Cruise Control	`Kafka`	`kafka-cruise-control-metrics.yaml`

9.3.2. Example Prometheus rules for alert notifications

Example Prometheus rules for alert notifications are provided with the example metrics configuration files provided by Strimzi. The rules are specified in the example `prometheus-rules.yaml` file for use in a Prometheus deployment.

Alerting rules provide notifications about specific conditions observed in metrics. Rules are declared on the Prometheus server, but Prometheus Alertmanager is responsible for alert notifications.

Prometheus alerting rules describe conditions using PromQL expressions that are continuously evaluated.

When an alert expression becomes true, the condition is met and the Prometheus server sends alert data to the Alertmanager. Alertmanager then sends out a notification using the communication method configured for its deployment.

General points about the alerting rule definitions:

- A `for` property is used with the rules to determine the period of time a condition must persist before an alert is triggered.
- A tick is a basic ZooKeeper time unit, which is measured in milliseconds and configured using the `tickTime` parameter of `Kafka.spec.zookeeper.config`. For example, if ZooKeeper `tickTime=3000`, 3 ticks (3 x 3000) equals 9000 milliseconds.
- The availability of the `ZookeeperRunningOutOfSpace` metric and alert is dependent on the Kubernetes configuration and storage implementation used. Storage implementations for certain platforms may not be able to supply the information on available space required for the metric to provide an alert.

Alertmanager can be configured to use email, chat messages or other notification methods. Adapt the default configuration of the example rules according to your specific needs.

Example altering rules

The `prometheus-rules.yaml` file contains example rules for the following components:

- Kafka
- ZooKeeper
- Entity Operator
- Kafka Connect

- Kafka Bridge
- MirrorMaker
- Kafka Exporter

A description of each of the example rules is provided in the file.

9.3.3. Example Grafana dashboards

If you deploy Prometheus to provide metrics, you can use the example Grafana dashboards provided with Strimzi to monitor Strimzi components.

Example dashboards are provided in the `examples/metrics/grafana-dashboards` directory as JSON files.

All dashboards provide JVM metrics, as well as metrics specific to the component. For example, the Grafana dashboard for Strimzi operators provides information on the number of reconciliations or custom resources they are processing.

The example dashboards don't show all the metrics supported by Kafka. The dashboards are populated with a representative set of metrics for monitoring.

Table 4. Example Grafana dashboard files

Component	Example JSON file
Strimzi operators	`strimzi-operators.json`
Kafka	`strimzi-kafka.json`
ZooKeeper	`strimzi-zookeeper.json`
Kafka Connect	`strimzi-kafka-connect.json`
Kafka MirrorMaker 2.0	`strimzi-kafka-mirror-maker-2.json`
Kafka Bridge	`strimzi-kafka-bridge.json`
Cruise Control	`strimzi-cruise-control.json`
Kafka Exporter	`strimzi-kafka-exporter.json`

NOTE When metrics are not available to the Kafka Exporter, because there is no traffic in the cluster yet, the Kafka Exporter Grafana dashboard will show `N/A` for numeric fields and `No data to show` for graphs.

9.4. Using Prometheus with Strimzi

You can use Prometheus to provide monitoring data for the example Grafana dashboards provided with Strimzi.

To expose metrics in Prometheus format, you add configuration to a custom resource. You will also need to make sure that the metrics are scraped by your monitoring stack. Prometheus and Prometheus Alertmanager are used in the examples provided by Strimzi, but you can use also other

compatible tools.

1. Deploying Prometheus metrics configuration
2. Setting up Prometheus
3. Deploying Prometheus Alertmanager

9.4.1. Deploying Prometheus metrics configuration

Deploy Prometheus metrics configuration to use Prometheus with Strimzi. Use the `metricsConfig` property to enable and configure Prometheus metrics.

You can use your own configuration or the example custom resource configuration files provided with Strimzi.

- `kafka-metrics.yaml`
- `kafka-connect-metrics.yaml`
- `kafka-mirror-maker-2-metrics.yaml`
- `kafka-bridge-metrics.yaml`
- `kafka-cruise-control-metrics.yaml`

The example configuration files have relabeling rules and the configuration required to enable Prometheus metrics. Prometheus scrapes metrics from target HTTP endpoints. The example files are a good way to try Prometheus with Strimzi.

To apply the relabeling rules and metrics configuration, do one of the following:

- Copy the example configuration to your own custom resources
- Deploy the custom resource with the metrics configuration

If you want to include Kafka Exporter metrics, add `kafkaExporter` configuration to your `Kafka` resource.

IMPORTANT	Kafka Exporter provides only additional metrics related to consumer lag and consumer offsets. For regular Kafka metrics, you have to configure the Prometheus metrics in Kafka brokers.

This procedure shows how to deploy Prometheus metrics configuration in the `Kafka` resource. The process is the same when using the example files for other resources.

Procedure

1. Deploy the example custom resource with the Prometheus configuration.

   For example, for each `Kafka` resource you apply the `kafka-metrics.yaml` file.

   Deploying the example configuration

   ```
   kubectl apply -f kafka-metrics.yaml
   ```

Alternatively, you can copy the example configuration in `kafka-metrics.yaml` to your own `Kafka` resource.

Copying the example configuration

```
kubectl edit kafka <kafka-configuration-file>
```

Copy the `metricsConfig` property and the `ConfigMap` it references to your `Kafka` resource.

Example metrics configuration for Kafka

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metricsConfig: ①
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: kafka-metrics
          key: kafka-metrics-config.yml
---
kind: ConfigMap ②
apiVersion: v1
metadata:
  name: kafka-metrics
  labels:
    app: strimzi
data:
  kafka-metrics-config.yml: |
  # metrics configuration...
```

① Copy the `metricsConfig` property that references the ConfigMap that contains metrics configuration.

② Copy the whole `ConfigMap` that specifies the metrics configuration.

NOTE

For Kafka Bridge, you specify the `enableMetrics` property and set it to `true`.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
```

```
    # ...
  enableMetrics: true
  # ...
```

2. To deploy Kafka Exporter, add `kafkaExporter` configuration.

   `kafkaExporter` configuration is only specified in the `Kafka` resource.

   Example configuration for deploying Kafka Exporter

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    image: my-registry.io/my-org/my-exporter-cluster:latest ①
    groupRegex: ".*" ②
    topicRegex: ".*" ③
    resources: ④
      requests:
        cpu: 200m
        memory: 64Mi
      limits:
        cpu: 500m
        memory: 128Mi
    logging: debug ⑤
    enableSaramaLogging: true ⑥
    template: ⑦
      pod:
        metadata:
          labels:
            label1: value1
        imagePullSecrets:
          - name: my-docker-credentials
        securityContext:
          runAsUser: 1000001
          fsGroup: 0
        terminationGracePeriodSeconds: 120
    readinessProbe: ⑧
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe: ⑨
      initialDelaySeconds: 15
      timeoutSeconds: 5
# ...
```

① ADVANCED OPTION: Container image configuration, which is recommended only in special situations.

② A regular expression to specify the consumer groups to include in the metrics.

③ A regular expression to specify the topics to include in the metrics.

④ CPU and memory resources to reserve.

⑤ Logging configuration, to log messages with a given severity (debug, info, warn, error, fatal) or above.

⑥ Boolean to enable Sarama logging, a Go client library used by Kafka Exporter.

⑦ Customization of deployment templates and pods.

⑧ Healthcheck readiness probes.

⑨ Healthcheck liveness probes.

NOTE	For Kafka Exporter to be able to work properly, consumer groups need to be in use.

Additional resources

- `KafkaExporterTemplate` schema reference
- `metricsConfig` schema reference

9.4.2. Setting up Prometheus

Prometheus provides an open source set of components for systems monitoring and alert notification.

We describe here how you can use the CoreOS Prometheus Operator to run and manage a Prometheus server that is suitable for use in production environments, but with the correct configuration you can run any Prometheus server.

NOTE	The Prometheus server configuration uses service discovery to discover the pods in the cluster from which it gets metrics. For this feature to work correctly, the service account used for running the Prometheus service pod must have access to the API server so it can retrieve the pod list. For more information, see Discovering services.

Prometheus configuration

Strimzi provides example configuration files for the Prometheus server.

A Prometheus YAML file is provided for deployment:

- `prometheus.yaml`

Additional Prometheus-related configuration is also provided in the following files:

- `prometheus-additional.yaml`
- `prometheus-rules.yaml`
- `strimzi-pod-monitor.yaml`

For Prometheus to obtain monitoring data:

- Deploy the Prometheus Operator

Then use the configuration files to:

- Deploy Prometheus

Alerting rules

The `prometheus-rules.yaml` file provides example alerting rule examples for use with Alertmanager.

Prometheus resources

When you apply the Prometheus configuration, the following resources are created in your Kubernetes cluster and managed by the Prometheus Operator:

- A `ClusterRole` that grants permissions to Prometheus to read the health endpoints exposed by the Kafka and ZooKeeper pods, cAdvisor and the kubelet for container metrics.
- A `ServiceAccount` for the Prometheus pods to run under.
- A `ClusterRoleBinding` which binds the `ClusterRole` to the `ServiceAccount`.
- A `Deployment` to manage the Prometheus Operator pod.
- A `PodMonitor` to manage the configuration of the Prometheus pod.
- A `Prometheus` to manage the configuration of the Prometheus pod.
- A `PrometheusRule` to manage alerting rules for the Prometheus pod.
- A `Secret` to manage additional Prometheus settings.
- A `Service` to allow applications running in the cluster to connect to Prometheus (for example, Grafana using Prometheus as datasource).

Deploying the CoreOS Prometheus Operator

To deploy the Prometheus Operator to your Kafka cluster, apply the YAML bundle resources file from the Prometheus CoreOS repository.

Procedure

1. Download the `bundle.yaml` resources file from the repository.

   On Linux, use:

   ```
   curl -s https://raw.githubusercontent.com/coreos/prometheus-
   operator/master/bundle.yaml | sed -e '/[[:space:]]*namespace: [a-zA-Z0-9-
   ]*$/s/namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' > prometheus-
   operator-deployment.yaml
   ```

   On MacOS, use:

```
curl -s https://raw.githubusercontent.com/coreos/prometheus-
operator/master/bundle.yaml | sed -e '' '/[[:space:]]*namespace: [a-zA-Z0-9-
]*$/s/namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' > prometheus-
operator-deployment.yaml
```

- Replace the example `namespace` with your own.
- Use the latest `master` release as shown, or choose a release that is compatible with your version of Kubernetes (see the Kubernetes compatibility matrix). The `master` release of the Prometheus Operator works with Kubernetes 1.18+.

NOTE	If using OpenShift, specify a release of the OpenShift fork of the Prometheus Operator repository.

2. (Optional) If it is not required, you can manually remove the `spec.template.spec.securityContext` property from the `prometheus-operator-deployment.yaml` file.
3. Deploy the Prometheus Operator:

```
kubectl create -f prometheus-operator-deployment.yaml
```

Deploying Prometheus

Use Prometheus to obtain monitoring data in your Kafka cluster.

You can use your own Prometheus deployment or deploy Prometheus using the example metrics configuration files provided by Strimzi. The example files include a configuration file for a Prometheus deployment and files for Prometheus-related resources:

- `examples/metrics/prometheus-install/prometheus.yaml`
- `examples/metrics/prometheus-install/prometheus-rules.yaml`
- `examples/metrics/prometheus-install/strimzi-pod-monitor.yaml`
- `examples/metrics/prometheus-additional-properties/prometheus-additional.yaml`

The deployment process creates a `ClusterRoleBinding` and discovers an Alertmanager instance in the namespace specified for the deployment.

NOTE	By default, the Prometheus Operator only supports jobs that include an `endpoints` role for service discovery. Targets are discovered and scraped for each endpoint port address. For endpoint discovery, the port address may be derived from service (`role: service`) or pod (`role: pod`) discovery.

Prerequisites

- Check the example alerting rules provided

Procedure

1. Modify the Prometheus installation file (`prometheus.yaml`) according to the namespace

Prometheus is going to be installed into:

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
```

2. Edit the `PodMonitor` resource in `strimzi-pod-monitor.yaml` to define Prometheus jobs that will scrape the metrics data from pods.

   Update the `namespaceSelector.matchNames` property with the namespace where the pods to scrape the metrics from are running.

   `PodMonitor` is used to scrape data directly from pods for Apache Kafka, ZooKeeper, Operators, the Kafka Bridge and Cruise Control.

3. Edit the `prometheus.yaml` installation file to include additional configuration for scraping metrics directly from nodes.

   The Grafana dashboards provided show metrics for CPU, memory and disk volume usage, which come directly from the Kubernetes cAdvisor agent and kubelet on the nodes.

   The Prometheus Operator does not have a monitoring resource like `PodMonitor` for scraping the nodes, so the `prometheus-additional.yaml` file contains the additional configuration needed.

   a. Create a `Secret` resource from the configuration file (`prometheus-additional.yaml` in the `examples/metrics/prometheus-additional-properties` directory):

      ```
      kubectl apply -f prometheus-additional.yaml
      ```

   b. Edit the `additionalScrapeConfigs` property in the `prometheus.yaml` file to include the name of the `Secret` and the `prometheus-additional.yaml` file.

4. Deploy the Prometheus resources:

   ```
   kubectl apply -f strimzi-pod-monitor.yaml
   kubectl apply -f prometheus-rules.yaml
   kubectl apply -f prometheus.yaml
   ```

9.4.3. Deploying Alertmanager

Use Alertmanager to route alerts to a notification service. Prometheus Alertmanager is a component for handling alerts and routing them to a notification service. Alertmanager supports

an essential aspect of monitoring, which is to be notified of conditions that indicate potential issues based on alerting rules.

You can use the example metrics configuration files provided by Strimzi to deploy Alertmanager to send notifications to a Slack channel. A configuration file defines the resources for deploying Alertmanager:

- `examples/metrics/prometheus-install/alert-manager.yaml`

An additional configuration file provides the hook definitions for sending notifications from your Kafka cluster:

- `examples/metrics/prometheus-alertmanager-config/alert-manager-config.yaml`

The following resources are defined on deployment:

- An `Alertmanager` to manage the Alertmanager pod.
- A `Secret` to manage the configuration of the Alertmanager.
- A `Service` to provide an easy to reference hostname for other services to connect to Alertmanager (such as Prometheus).

Prerequisites

- Metrics are configured for the Kafka cluster resource
- Prometheus is deployed

Procedure

1. Create a `Secret` resource from the Alertmanager configuration file (`alert-manager-config.yaml` in the `examples/metrics/prometheus-alertmanager-config` directory):

```
kubectl apply -f alert-manager-config.yaml
```

2. Update the `alert-manager-config.yaml` file to replace the:
   - `slack_api_url` property with the actual value of the Slack API URL related to the application for the Slack workspace
   - `channel` property with the actual Slack channel on which to send notifications
3. Deploy Alertmanager:

```
kubectl apply -f alert-manager.yaml
```

9.4.4. Using metrics with Minikube

When adding Prometheus and Grafana servers to an Apache Kafka deployment using Minikube, the memory available to the virtual machine should be increased (to 4 GB of RAM, for example, instead of the default 2 GB).

For information on how to increase the default amount of memory, see Installing a local Kubernetes cluster with Minikube

Additional resources

- Prometheus - Monitoring Docker Container Metrics using cAdvisor describes how to use cAdvisor (short for container Advisor) metrics with Prometheus to analyze and expose resource usage (CPU, Memory, and Disk) and performance data from running containers within pods on Kubernetes.

9.5. Enabling the example Grafana dashboards

Use Grafana to provide visualizations of Prometheus metrics on customizable dashboards.

You can use your own Grafana deployment or deploy Grafana using the example metrics configuration files provided by Strimzi. The example files include a configuration file for a Grafana deployment

- `examples/metrics/grafana-install/grafana.yaml`

Strimzi also provides example dashboard configuration files for Grafana in JSON format.

- `examples/metrics/grafana-dashboards`

This procedure uses the example Grafana configuration file and example dashboards.

The example dashboards are a good starting point for monitoring key metrics, but they don't show all the metrics supported by Kafka. You can modify the example dashboards or add other metrics, depending on your infrastructure.

NOTE	No alert notification rules are defined.

When accessing a dashboard, you can use the `port-forward` command to forward traffic from the Grafana pod to the host. The name of the Grafana pod is different for each user.

Prerequisites

- Metrics are configured for the Kafka cluster resource
- Prometheus and Prometheus Alertmanager are deployed

Procedure

1. Deploy Grafana.

```
kubectl apply -f grafana.yaml
```

2. Get the details of the Grafana service.

```
kubectl get service grafana
```

For example:

NAME	TYPE	CLUSTER-IP	PORT(S)
grafana	ClusterIP	172.30.123.40	3000/TCP

Note the port number for port forwarding.

3. Use `port-forward` to redirect the Grafana user interface to `localhost:3000`:

```
kubectl port-forward svc/grafana 3000:3000
```

4. In a web browser, access the Grafana login screen using the URL `http://localhost:3000`.

   The Grafana Log In page appears.

5. Enter your user name and password, and then click **Log In**.

   The default Grafana user name and password are both `admin`. After logging in for the first time, you can change the password.

6. In **Configuration** > **Data Sources**, add Prometheus as a *data source*.
   - Specify a name
   - Add *Prometheus* as the type
   - Specify a Prometheus server URL (http://prometheus-operated:9090)

     Save and test the connection when you have added the details.

7. Click the + icon and then click **Import**.
8. In `examples/metrics/grafana-dashboards`, copy the JSON of the dashboard to import.
9. Paste the JSON into the text box, and then click **Load**.
10. Repeat steps 7-9 for the other example Grafana dashboards.

The imported Grafana dashboards are available to view from the **Dashboards** home page.

Chapter 10. Introducing distributed tracing

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

In Strimzi, tracing facilitates the end-to-end tracking of messages: from source systems to Kafka, and then from Kafka to target systems and applications. Distributed tracing complements the monitoring of metrics in Grafana dashboards, as well as the component loggers.

Support for tracing is built in to the following Kafka components:

- MirrorMaker to trace messages from a source cluster to a target cluster
- Kafka Connect to trace messages consumed and produced by Kafka Connect
- Kafka Bridge to trace messages between Kafka and HTTP client applications

Tracing is not supported for Kafka brokers.

You enable and configure tracing for these components through their custom resources. You add tracing configuration using `spec.template` properties.

You enable tracing by specifying a tracing type using the `spec.tracing.type` property:

`opentelemetry`
: Specify `type: opentelemetry` to use OpenTelemetry. By Default, OpenTelemetry uses the OTLP (OpenTelemetry Protocol) exporter and endpoint to get trace data. You can specify other tracing systems supported by OpenTelemetry, including Jaeger tracing. To do this, you change the OpenTelemetry exporter and endpoint in the tracing configuration.

`jaeger`
: Specify `type:jaeger` to use OpenTracing and the Jaeger client to get trace data.

NOTE Support for `type: jaeger` tracing is deprecated. The Jaeger clients are now retired and the OpenTracing project archived. As such, we cannot guarantee their support for future Kafka versions. If possible, we will maintain the support for `type: jaeger` tracing until June 2023 and remove it afterwards. Please migrate to OpenTelemetry as soon as possible.

10.1. Tracing options

Use OpenTelemetry or OpenTracing (deprecated) with the Jaeger tracing system.

OpenTelemetry and OpenTracing provide API specifications that are independent from the tracing or monitoring system.

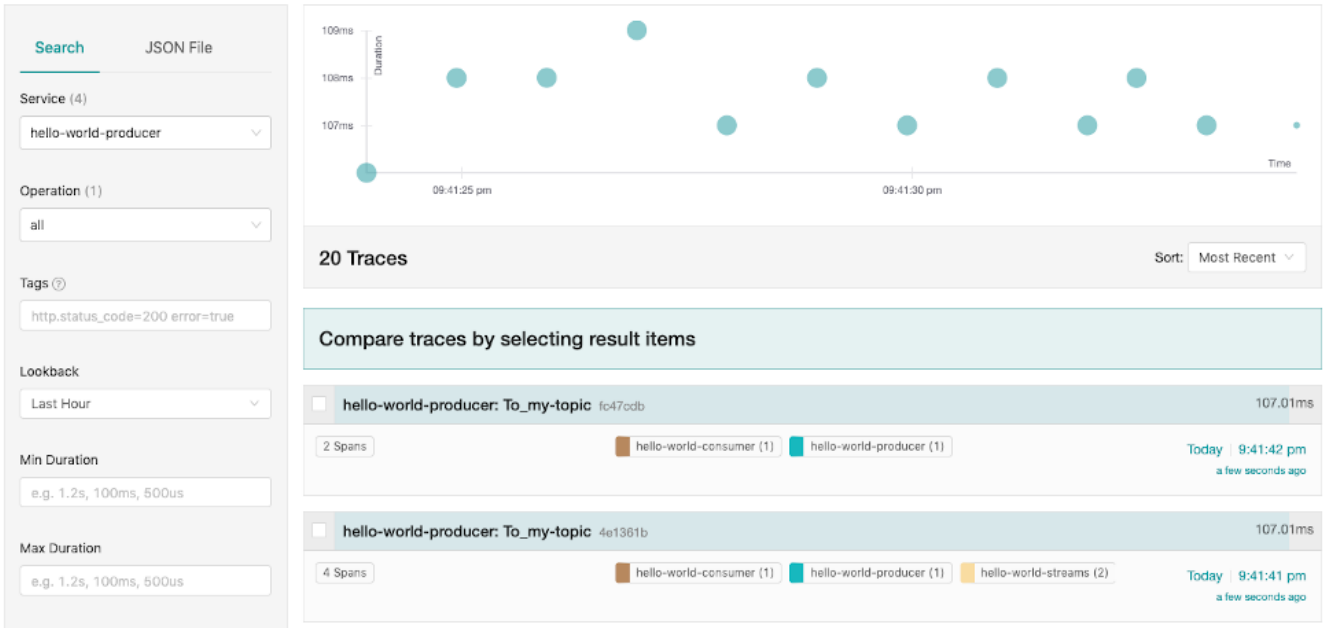
You use the APIs to instrument application code for tracing.

- Instrumented applications generate *traces* for individual requests across the distributed system.
- Traces are composed of *spans* that define specific units of work over time.

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

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

The Jaeger user interface showing a simple query



Additional resources

- Jaeger documentation
- OpenTelemetry documentation
- OpenTracing documentation

10.2. Environment variables for tracing

Use environment variables when you are enabling tracing for Kafka components or initializing a tracer for Kafka clients.

Tracing environment variables are subject to change. For the latest information, see the OpenTelemetry documentation and OpenTracing documentation.

The following tables describe the key environment variables for setting up a tracer.

Table 5. OpenTelemetry environment variables

Property	Required	Description
`OTEL_SERVICE_NAME`	Yes	The name of the Jaeger tracing service for OpenTelemetry.
`OTEL_EXPORTER_JAEGER_ENDPOINT`	Yes	The exporter used for tracing.

Property	Required	Description
`OTEL_TRACES_EXPORTER`	Yes	The exporter used for tracing. Set to `otlp` by default. If using Jaeger tracing, you need to set this environment variable as `jaeger`. If you are using another tracing implementation, specify the exporter used.

Table 6. OpenTracing environment variables

Property	Required	Description
`JAEGER_SERVICE_NAME`	Yes	The name of the Jaeger tracer service.
`JAEGER_AGENT_HOST`	No	The hostname for communicating with the `jaeger-agent` through the User Datagram Protocol (UDP).
`JAEGER_AGENT_PORT`	No	The port used for communicating with the `jaeger-agent` through UDP.

10.3. Setting up distributed tracing

Enable distributed tracing in Kafka components by specifying a tracing type in the custom resource. Instrument tracers in Kafka clients for end-to-end tracking of messages.

To set up distributed tracing, follow these procedures in order:

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

10.3.1. Prerequisites

Before setting up distributed tracing, make sure Jaeger backend components are deployed to your Kubernetes cluster. We recommend using the Jaeger operator for deploying Jaeger on your Kubernetes cluster.

For deployment instructions, see the Jaeger documentation.

NOTE	Setting up tracing for applications and systems beyond Strimzi is outside the scope of this content.

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

Distributed tracing is supported for MirrorMaker, MirrorMaker 2.0, Kafka Connect, and the Strimzi Kafka Bridge. Configure the custom resource of the component to specify and enable a tracer service.

Enabling tracing in a resource triggers the following events:

- Interceptor classes are updated in the integrated consumers and producers of the component.
- For MirrorMaker, MirrorMaker 2.0, and Kafka Connect, the tracing agent initializes a tracer based on the tracing configuration defined in the resource.
- For the Kafka Bridge, a tracer based on the tracing configuration defined in the resource is initialized by the Kafka Bridge itself.

You can enable tracing that uses OpenTelemetry or OpenTracing.

Tracing in MirrorMaker and MirrorMaker 2.0

For MirrorMaker and MirrorMaker 2.0, messages are traced from the source cluster to the target cluster. The trace data records messages entering and leaving the MirrorMaker or MirrorMaker 2.0 component.

Tracing in Kafka Connect

For Kafka Connect, only messages produced and consumed by Kafka Connect are traced. To trace messages sent between Kafka Connect and external systems, you must configure tracing in the connectors for those systems.

Tracing in the Kafka Bridge

For the Kafka Bridge, messages produced and consumed by the Kafka Bridge are traced. Incoming HTTP requests from client applications to send and receive messages through the Kafka Bridge are also traced. To have end-to-end tracing, you must configure tracing in your HTTP clients.

Procedure

Perform these steps for each `KafkaMirrorMaker`, `KafkaMirrorMaker2`, `KafkaConnect`, and `KafkaBridge` resource.

1. In the `spec.template` property, configure the tracer service.
    - Use the tracing environment variables as template configuration properties.
    - For OpenTelemetry, set the `spec.tracing.type` property to `opentelemetry`.
    - For OpenTracing, set the `spec.tracing.type` property to `jaeger`.

Example tracing configuration for Kafka Connect using OpenTelemetry

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

```
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
  #...
```

Example tracing configuration for MirrorMaker using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  #...
  template:
    mirrorMakerContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
#...
```

Example tracing configuration for MirrorMaker 2.0 using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
```

```
#...
```

Example tracing configuration for the Kafka Bridge using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
#...
```

Example tracing configuration for Kafka Connect using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
  #...
```

Example tracing configuration for MirrorMaker using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
```

```
  #...
  template:
    mirrorMakerContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

Example tracing configuration for MirrorMaker 2.0 using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

Example tracing configuration for the Kafka Bridge using OpenTracing

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
```

```
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

2. Create or update the resource:

```
kubectl apply -f <resource_configuration_file>
```

10.3.3. Initializing tracing for Kafka clients

Initialize a tracer, then instrument your client applications for distributed tracing. You can instrument Kafka producer and consumer clients, and Kafka Streams API applications. You can initialize a tracer for OpenTracing or OpenTelemetry.

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

Procedure

In each client application add the dependencies for the tracer:

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

Dependencies for OpenTelemetry

```
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
    <version>1.18.0-alpha</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry.instrumentation</groupId>
  <artifactId>opentelemetry-kafka-clients-{OpenTelemetryKafkaClient}</artifactId>
  <version>1.18.0-alpha</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-otlp</artifactId>
  <version>1.18.0</version>
</dependency>
```

Dependencies for OpenTracing

```
<dependency>
    <groupId>io.jaegertracing</groupId>
    <artifactId>jaeger-client</artifactId>
    <version>1.3.2</version>
</dependency>
```

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-kafka-client</artifactId>
  <version>0.1.15</version>
</dependency>
```

2. Define the configuration of the tracer using the tracing environment variables.
3. Create a tracer, which is initialized with the environment variables:

   Creating a tracer for OpenTelemetry

   ```
   OpenTelemetry ot = GlobalOpenTelemetry.get();
   ```

   Creating a tracer for OpenTracing

   ```
   Tracer tracer = Configuration.fromEnv().getTracer();
   ```

4. Register the tracer as a global tracer:

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

5. Instrument your client:
   - Instrumenting producers and consumers for tracing
   - Instrumenting Kafka Streams applications for tracing

10.3.4. Instrumenting producers and consumers for tracing

Instrument application code to enable tracing in Kafka producers and consumers. Use a decorator pattern or interceptors to instrument your Java producer and consumer application code for tracing. You can then record traces when messages are produced or retrieved from a topic.

OpenTelemetry and OpenTracing instrumentation projects provide classes that support instrumentation of producers and consumers.

Decorator instrumentation

For decorator instrumentation, create a modified producer or consumer instance for tracing. Decorator instrumentation is different for OpenTelemetry and OpenTracing.

Interceptor instrumentation

For interceptor instrumentation, add the tracing capability to the consumer or producer configuration. Interceptor instrumentation is the same for OpenTelemetry and OpenTracing.

Prerequisites

- You have initialized tracing for the client.

You enable instrumentation in producer and consumer applications by adding the tracing JARs

as dependencies to your project.

Procedure

Perform these steps in the application code of each producer and consumer application. Instrument your client application code using either a decorator pattern or interceptors.

- To use a decorator pattern, create a modified producer or consumer instance to send or receive messages.

  You pass the original `KafkaProducer` or `KafkaConsumer` class.

  Example decorator instrumentation for OpenTelemetry

```
// Producer instance
Producer < String, String > op = new KafkaProducer < > (
    configs,
    new StringSerializer(),
    new StringSerializer()
    );
    Producer < String, String > producer = tracing.wrap(op);
KafkaTracing tracing = KafkaTracing.create(GlobalOpenTelemetry.get());
producer.send(...);

//consumer instance
Consumer<String, String> oc = new KafkaConsumer<>(
    configs,
    new StringDeserializer(),
    new StringDeserializer()
    );
    Consumer<String, String> consumer = tracing.wrap(oc);
consumer.subscribe(Collections.singleton("mytopic"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

  Example decorator instrumentation for OpenTracing

```
//producer instance
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
TracingKafkaProducer<Integer, String> tracingProducer = new
TracingKafkaProducer<>(producer, tracer);
TracingKafkaProducer.send(...)

//consumer instance
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);
TracingKafkaConsumer<Integer, String> tracingConsumer = new
TracingKafkaConsumer<>(consumer, tracer);
tracingConsumer.subscribe(Collections.singletonList("mytopic"));
ConsumerRecords<Integer, String> records = tracingConsumer.poll(1000);
```

```
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

- To use interceptors, set the interceptor class in the producer or consumer configuration.

  You use the `KafkaProducer` and `KafkaConsumer` classes in the usual way. The `TracingProducerInterceptor` and `TracingConsumerInterceptor` interceptor classes take care of the tracing capability.

  Example producer configuration using interceptors

```
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingProducerInterceptor.class.getName());

KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
producer.send(...);
```

  Example consumer configuration using interceptors

```
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingConsumerInterceptor.class.getName());

KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("messages"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
tracer);
```

10.3.5. Instrumenting Kafka Streams applications for tracing

Instrument application code to enable tracing in Kafka Streams API applications. Use a decorator pattern or interceptors to instrument your Kafka Streams API applications for tracing. You can then record traces when messages are produced or retrieved from a topic.

Decorator instrumentation

For decorator instrumentation, create a modified Kafka Streams instance for tracing. The OpenTracing instrumentation project provides a `TracingKafkaClientSupplier` class that supports instrumentation of Kafka Streams. You create a wrapped instance of the `TracingKafkaClientSupplier` supplier interface, which provides tracing instrumentation for Kafka Streams. For OpenTelemetry, the process is the same but you need to create a custom `TracingKafkaClientSupplier` class to provide the support.

Interceptor instrumentation

For interceptor instrumentation, add the tracing capability to the Kafka Streams producer and consumer configuration.

Prerequisites

- You have initialized tracing for the client.

  You enable instrumentation in Kafka Streams applications by adding the tracing JARs as dependencies to your project.

- To instrument Kafka Streams with OpenTelemetry, you'll need to write a custom `TracingKafkaClientSupplier`.
- The custom `TracingKafkaClientSupplier` can extend Kafka's `DefaultKafkaClientSupplier`, overriding the producer and consumer creation methods to wrap the instances with the telemetry-related code.

  Example custom `TracingKafkaClientSupplier`

```
private class TracingKafkaClientSupplier extends DefaultKafkaClientSupplier {
    @Override
    public Producer<byte[], byte[]> getProducer(Map<String, Object> config) {
        KafkaTelemetry telemetry =
KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getProducer(config));
    }

    @Override
    public Consumer<byte[], byte[]> getConsumer(Map<String, Object> config) {
        KafkaTelemetry telemetry =
KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getConsumer(config));
    }

    @Override
    public Consumer<byte[], byte[]> getRestoreConsumer(Map<String, Object> config)
{
        return this.getConsumer(config);
    }

    @Override
    public Consumer<byte[], byte[]> getGlobalConsumer(Map<String, Object> config) {
        return this.getConsumer(config);
    }
}
```

Procedure

Perform these steps for each Kafka Streams API application.

- To use a decorator pattern, create an instance of the `TracingKafkaClientSupplier` supplier interface, then provide the supplier interface to `KafkaStreams`.

  Example decorator instrumentation

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
```

```
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
supplier);
streams.start();
```

- To use interceptors, set the interceptor class in the Kafka Streams producer and consumer configuration.

  The `TracingProducerInterceptor` and `TracingConsumerInterceptor` interceptor classes take care of the tracing capability.

  Example producer and consumer configuration using interceptors

```
props.put(StreamsConfig.PRODUCER_PREFIX +
ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingProducerInterceptor.class.getName());
props.put(StreamsConfig.CONSUMER_PREFIX +
ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingConsumerInterceptor.class.getName());
```

10.3.6. Introducing a different OpenTelemetry tracing system

Instead of the default OTLP system, you can specify other tracing systems that are supported by OpenTelemetry. You do this by adding the required artifacts to the Kafka image provided with Strimzi. Any required implementation specific environment variables must also be set. You then enable the new tracing implementation using the `OTEL_TRACES_EXPORTER` environment variable.

This procedure shows how to implement Zipkin tracing.

Procedure

1. Add the tracing artifacts to the `/opt/kafka/libs/` directory of the Strimzi Kafka image.

   You can use the Kafka container image on the Container Registry as a base image for creating a new custom image.

   OpenTelemetry artifact for Zipkin

```
io.opentelemetry:opentelemetry-exporter-zipkin
```

2. Set the tracing exporter and endpoint for the new tracing implementation.

   Example Zikpin tracer configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
```

```
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-zipkin-service
        - name: OTEL_EXPORTER_ZIPKIN_ENDPOINT
          value: http://zipkin-exporter-host-name:9411/api/v2/spans ①
        - name: OTEL_TRACES_EXPORTER
          value: zipkin ②
  tracing:
    type: opentelemetry
#...
```

① Specifies the Zipkin endpoint to connect to.

② The Zipkin exporter.

10.3.7. Custom span names

A tracing *span* is a logical unit of work in Jaeger, with an operation name, start time, and duration. Spans have built-in names, but you can specify custom span names in your Kafka client instrumentation where used.

Specifying custom span names is optional and only applies when using a decorator pattern in producer and consumer client instrumentation or Kafka Streams instrumentation.

Specifying span names for OpenTelemetry

Custom span names cannot be specified directly with OpenTelemetry. Instead, you retrieve span names by adding code to your client application to extract additional tags and attributes.

Example code to extract attributes

```
//Defines attribute extraction for a producer
private static class ProducerAttribExtractor implements AttributesExtractor <
ProducerRecord < ? , ? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ProducerRecord < ? , ? >
producerRecord) {
        set(attributes, AttributeKey.stringKey("prod_start"), "prod1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ProducerRecord < ? , ? >
producerRecord, @Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("prod_end"), "prod2");
    }
}
//Defines attribute extraction for a consumer
private static class ConsumerAttribExtractor implements AttributesExtractor <
ConsumerRecord < ? , ? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ConsumerRecord < ? , ? >
```

```
producerRecord) {
        set(attributes, AttributeKey.stringKey("con_start"), "con1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ConsumerRecord < ? , ? >
producerRecord, @Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("con_end"), "con2");
    }
}
//Extracts the attributes
public static void main(String[] args) throws Exception {
        Map < String, Object > configs = new HashMap < >
(Collections.singletonMap(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"));
        System.setProperty("otel.traces.exporter", "jaeger");
        System.setProperty("otel.service.name", "myapp1");
        KafkaTracing tracing = KafkaTracing.newBuilder(GlobalOpenTelemetry.get())
            .addProducerAttributesExtractors(new ProducerAttribExtractor())
            .addConsumerAttributesExtractors(new ConsumerAttribExtractor())
            .build();
```

Specifying span names for OpenTracing

To specify custom span names for OpenTracing, pass a `BiFunction` object as an additional argument when instrumenting producers and consumers.

For more information on built-in names and specifying custom span names to instrument client application code in a decorator pattern, see the OpenTracing Apache Kafka client instrumentation.

Chapter 11. Upgrading Strimzi

Strimzi can be upgraded to version 0.33.0 to take advantage of new features and enhancements, performance improvements, and security options.

As part of the upgrade, you upgrade Kafka to the latest supported version. Each Kafka release introduces new features, improvements, and bug fixes to your Strimzi deployment.

Strimzi can be downgraded to the previous version if you encounter issues with the newer version.

Released versions of Strimzi are available from the GitHub releases page.

Upgrade downtime and availability

If topics are configured for high availability, upgrading Strimzi should not cause any downtime for consumers and producers that publish and read data from those topics. Highly available topics have a replication factor of at least 3 and partitions distributed evenly among the brokers.

Upgrading Strimzi triggers rolling updates, where all brokers are restarted in turn, at different stages of the process. During rolling updates, not all brokers are online, so overall *cluster availability* is temporarily reduced. A reduction in cluster availability increases the chance that a broker failure will result in lost messages.

11.1. Strimzi upgrade paths

Two Strimzi upgrade paths are possible.

Incremental upgrade

Upgrading Strimzi from the previous minor version to version 0.33.0.

Multi-version upgrade

Upgrading Strimzi from an old version to version 0.33.0 within a single upgrade (skipping one or more intermediate versions).

For example, upgrading from Strimzi 0.24.0 directly to Strimzi 0.33.0.

NOTE	A multi-version Strimzi upgrade might still support the current version of a Kafka deployment.

11.1.1. Supported Kafka versions

Decide which Kafka version to upgrade to before starting the Strimzi upgrade process. You can review supported Kafka versions in the Supported versions table.

- The **Operators** column lists all released Strimzi versions (the Strimzi version is often called the "Operator version").
- The **Kafka versions** column lists the supported Kafka versions for each Strimzi version.

You can only use a Kafka version supported by the version of Strimzi you are using. You can

upgrade to a higher Kafka version as long as it is supported by your version of Strimzi. In some cases, you can also downgrade to a previous supported Kafka version.

11.1.2. Upgrading from a Strimzi version earlier than 0.22

If you are upgrading to the latest version of Strimzi from a version prior to version 0.22, do the following:

1. Upgrade Strimzi to version 0.22 following the standard sequence.
2. Convert Strimzi custom resources to `v1beta2` using the *API conversion tool* provided with Strimzi.
3. Do one of the following:
    - Upgrade Strimzi to a version between 0.23 and 0.26 (where the `ControlPlaneListener` feature gate is disabled by default).
    - Upgrade Strimzi to a version between 0.27 and 0.31 (where the `ControlPlaneListener` feature gate is enabled by default) with the `ControlPlaneListener` feature gate disabled.
4. Enable the `ControlPlaneListener` feature gate.
5. Upgrade to Strimzi 0.33.0 following the standard sequence.

Strimzi custom resources started using the `v1beta2` API version in release 0.22. CRDs and custom resources must be converted **before** upgrading to Strimzi 0.23 or newer. For information on using the API conversion tool, see the Strimzi 0.24.0 upgrade documentation.

NOTE	As an alternative to first upgrading to version 0.22, you can install the custom resources from version 0.22 and then convert the resources.

The `ControlPlaneListener` feature is now permanently enabled in Strimzi. You must upgrade to a version of Strimzi where it is disabled, then enable it using the `STRIMZI_FEATURE_GATES` environment variable in the Cluster Operator configuration.

Disabling the `ControlPlaneListener` feature gate

```
env:
  - name: STRIMZI_FEATURE_GATES
    value: -ControlPlaneListener
```

Enabling the `ControlPlaneListener` feature gate

```
env:
  - name: STRIMZI_FEATURE_GATES
    value: +ControlPlaneListener
```

11.2. Required upgrade sequence

To upgrade brokers and clients without downtime, you *must* complete the Strimzi upgrade

procedures in the following order:

1. Make sure your Kubernetes cluster version is supported.

   Strimzi 0.33.0 requires Kubernetes 1.19 and later.

   You can upgrade Kubernetes with minimal downtime.

2. Upgrade the Cluster Operator.
3. Upgrade all Kafka brokers and client applications to the latest supported Kafka version.
4. Optional: Upgrade consumers and Kafka Streams applications to use the *incremental cooperative rebalance* protocol for partition rebalances.

11.3. Upgrading Kubernetes with minimal downtime

If you are upgrading Kubernetes, refer to the Kubernetes upgrade documentation to check the upgrade path and the steps to upgrade your nodes correctly. Before upgrading Kubernetes, check the supported versions for your version of Strimzi.

When performing your upgrade, you'll want to keep your Kafka clusters available.

You can employ one of the following strategies:

1. Configuring pod disruption budgets
2. Rolling pods by one of these methods:
   a. Using the Strimzi Drain Cleaner
   b. Manually by applying an annotation to your pod

You have to configure the pod disruption budget before using one of the methods to roll your pods.

For Kafka to stay operational, topics must also be replicated for high availability. This requires topic configuration that specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
```

```
    # ...
```

In a highly available environment, the Cluster Operator maintains a minimum number of in-sync replicas for topics during the upgrade process so that there is no downtime.

11.3.1. Rolling pods using the Strimzi Drain Cleaner

You can use the Strimzi Drain Cleaner tool to evict nodes during an upgrade. The Strimzi Drain Cleaner annotates pods with a rolling update pod annotation. This informs the Cluster Operator to perform a rolling update of an evicted pod.

A pod disruption budget allows only a specified number of pods to be unavailable at a given time. During planned maintenance of Kafka broker pods, a pod disruption budget ensures Kafka continues to run in a highly available environment.

You specify a pod disruption budget using a `template` customization for a Kafka component. By default, pod disruption budgets allow only a single pod to be unavailable at a given time.

To do this, you set `maxUnavailable` to `0` (zero). Reducing the maximum pod disruption budget to zero prevents voluntary disruptions, so pods must be evicted manually.

Specifying a pod disruption budget

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    template:
      podDisruptionBudget:
        maxUnavailable: 0
# ...
```

11.3.2. Rolling pods manually while keeping topics available

During an upgrade, you can trigger a manual rolling update of pods through the Cluster Operator. Using `Pod` resources, rolling updates restart the pods of resources with new pods. As with using the Strimzi Drain Cleaner, you'll need to set the `maxUnavailable` value to zero for the pod disruption budget.

You need to watch the pods that need to be drained. You then add a pod annotation to make the update.

Here, the annotation updates a Kafka broker.

Performing a manual rolling update on a Kafka broker pod

```
kubectl annotate pod <cluster_name>-kafka-<index> strimzi.io/manual-rolling-
update=true
```

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

Additional resources

- Kubernetes documentation
- Draining pods using the Strimzi Drain Cleaner
- Replicating topics for high availability
- PodDisruptionBudgetTemplate schema reference
- Performing a rolling update using a pod annotation

11.4. Upgrading the Cluster Operator

Use the same method to upgrade the Cluster Operator as the initial method of deployment.

Using installation files

If you deployed the Cluster Operator using the installation YAML files, perform your upgrade by modifying the Operator installation files, as described in Upgrading the Cluster Operator.

Using the OperatorHub.io

If you deployed Strimzi from OperatorHub.io, use the Operator Lifecycle Manager (OLM) to change the update channel for the Strimzi operators to a new Strimzi version.

Updating the channel starts one of the following types of upgrade, depending on your chosen upgrade strategy:

- An automatic upgrade is initiated
- A manual upgrade that requires approval before installation begins

NOTE	If you subscribe to the *stable* channel, you can get automatic updates without changing channels. However, enabling automatic updates is not recommended because of the potential for missing any pre-installation upgrade steps. Use automatic upgrades only on version-specific channels.

For more information on using OperatorHub.io to upgrade Operators, see the Operator Lifecycle Manager documentation.

Using a Helm chart

If you deployed the Cluster Operator using a Helm chart, use `helm upgrade`.

The `helm upgrade` command does not upgrade the Custom Resource Definitions for Helm. Install

the new CRDs manually after upgrading the Cluster Operator. You can access the CRDs from the GitHub releases page or find them in the `crd` subdirectory inside the Helm Chart.

11.4.1. Upgrading the Cluster Operator returns Kafka version error

If you upgrade the Cluster Operator and get an *unsupported Kafka version* error, your Kafka cluster deployment has an older Kafka version that is not supported by the new operator version. This error applies to all installation methods.

If this error occurs, upgrade Kafka to a supported Kafka version. Change the `spec.kafka.version` in the `Kafka` resource to the supported version.

You can use `kubectl` to check for error messages like this in the `status` of the `Kafka` resource.

Checking the Kafka status for errors

```
kubectl get kafka <kafka_cluster_name> -n <namespace> -o
jsonpath='{.status.conditions}'
```

Replace `<kafka_cluster_name>` with the name of your Kafka cluster and `<namespace>` with the Kubernetes namespace where the pod is running.

11.4.2. Upgrading the Cluster Operator using installation files

This procedure describes how to upgrade a Cluster Operator deployment to use Strimzi 0.33.0.

Follow this procedure if you deployed the Cluster Operator using the installation YAML files.

The availability of Kafka clusters managed by the Cluster Operator is not affected by the upgrade operation.

NOTE	Refer to the documentation supporting a specific version of Strimzi for information on how to upgrade to that version.

Prerequisites

- An existing Cluster Operator deployment is available.
- You have downloaded the release artifacts for Strimzi 0.33.0.

Procedure

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the `/install/cluster-operator` directory). Any changes will be **overwritten** by the new version of the Cluster Operator.
2. Update your custom resources to reflect the supported configuration options available for Strimzi version 0.33.0.
3. Update the Cluster Operator.
    a. Modify the installation files for the new Cluster Operator version according to the namespace the Cluster Operator is running in.

On Linux, use:

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

On MacOS, use:

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

b. If you modified one or more environment variables in your existing Cluster Operator `Deployment`, edit the `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml` file to use those environment variables.

4. When you have an updated configuration, deploy it along with the rest of the installation resources:

```
kubectl replace -f install/cluster-operator
```

Wait for the rolling updates to complete.

5. If the new Operator version no longer supports the Kafka version you are upgrading from, the Cluster Operator returns an error message to say the version is not supported. Otherwise, no error message is returned.
   - If the error message is returned, upgrade to a Kafka version that is supported by the new Cluster Operator version:
     a. Edit the `Kafka` custom resource.
     b. Change the `spec.kafka.version` property to a supported Kafka version.
   - If the error message is *not* returned, go to the next step. You will upgrade the Kafka version later.
6. Get the image for the Kafka pod to ensure the upgrade was successful:

```
kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The image tag shows the new Operator version. For example:

```
quay.io/strimzi/kafka:0.33.0-kafka-3.3.2
```

Your Cluster Operator was upgraded to version 0.33.0 but the version of Kafka running in the cluster it manages is unchanged.

Following the Cluster Operator upgrade, you must perform a Kafka upgrade.

11.5. Upgrading Kafka

After you have upgraded your Cluster Operator to 0.33.0, the next step is to upgrade all Kafka brokers to the latest supported version of Kafka.

Kafka upgrades are performed by the Cluster Operator through rolling updates of the Kafka brokers.

The Cluster Operator initiates rolling updates based on the Kafka cluster configuration.

If `Kafka.spec.kafka.config` contains…	**The Cluster Operator initiates…**
Both the `inter.broker.protocol.version` and the `log.message.format.version`.	A single rolling update. After the update, the `inter.broker.protocol.version` must be updated manually, followed by `log.message.format.version`. Changing each will trigger a further rolling update.
Either the `inter.broker.protocol.version` or the `log.message.format.version`.	Two rolling updates.
No configuration for the `inter.broker.protocol.version` or the `log.message.format.version`.	Two rolling updates.

IMPORTANT

From Kafka 3.0.0, when the `inter.broker.protocol.version` is set to `3.0` or higher, the `log.message.format.version` option is ignored and doesn't need to be set. The `log.message.format.version` property for brokers and the `message.format.version` property for topics are deprecated and will be removed in a future release of Kafka.

As part of the Kafka upgrade, the Cluster Operator initiates rolling updates for ZooKeeper.

- A single rolling update occurs even if the ZooKeeper version is unchanged.
- Additional rolling updates occur if the new version of Kafka requires a new ZooKeeper version.

11.5.1. Kafka versions

Kafka's log message format version and inter-broker protocol version specify, respectively, the log format version appended to messages and the version of the Kafka protocol used in a cluster. To ensure the correct versions are used, the upgrade process involves making configuration changes to existing Kafka brokers and code changes to client applications (consumers and producers).

The following table shows the differences between Kafka versions:

Table 7. Kafka version differences

Kafka version	**Inter-broker protocol version**	**Log message format version**	**ZooKeeper version**
3.2.0	3.2	3.2	3.6.3

Kafka version	Inter-broker protocol version	Log message format version	ZooKeeper version
3.2.1	3.2	3.2	3.6.3
3.2.3	3.2	3.2	3.6.3
3.3.1	3.3	3.3	3.6.3
3.3.2	3.3	3.3	3.6.3

Inter-broker protocol version

In Kafka, the network protocol used for inter-broker communication is called the *inter-broker protocol*. Each version of Kafka has a compatible version of the inter-broker protocol. The minor version of the protocol typically increases to match the minor version of Kafka, as shown in the preceding table.

The inter-broker protocol version is set cluster wide in the `Kafka` resource. To change it, you edit the `inter.broker.protocol.version` property in `Kafka.spec.kafka.config`.

Log message format version

When a producer sends a message to a Kafka broker, the message is encoded using a specific format. The format can change between Kafka releases, so messages specify which version of the message format they were encoded with.

The properties used to set a specific message format version are as follows:

- `message.format.version` property for topics
- `log.message.format.version` property for Kafka brokers

From Kafka 3.0.0, the message format version values are assumed to match the `inter.broker.protocol.version` and don't need to be set. The values reflect the Kafka version used.

When upgrading to Kafka 3.0.0 or higher, you can remove these settings when you update the `inter.broker.protocol.version`. Otherwise, set the message format version based on the Kafka version you are upgrading to.

The default value of `message.format.version` for a topic is defined by the `log.message.format.version` that is set on the Kafka broker. You can manually set the `message.format.version` of a topic by modifying its topic configuration.

11.5.2. Strategies for upgrading clients

The right approach to upgrading your client applications (including Kafka Connect connectors) depends on your particular circumstances.

Consuming applications need to receive messages in a message format that they understand. You can ensure that this is the case in one of two ways:

- By upgrading all the consumers for a topic *before* upgrading any of the producers.
- By having the brokers down-convert messages to an older format.

Using broker down-conversion puts extra load on the brokers, so it is not ideal to rely on down-conversion for all topics for a prolonged period of time. For brokers to perform optimally they should not be down converting messages at all.

Broker down-conversion is configured in two ways:

- The topic-level `message.format.version` configures it for a single topic.
- The broker-level `log.message.format.version` is the default for topics that do not have the topic-level `message.format.version` configured.

Messages published to a topic in a new-version format will be visible to consumers, because brokers perform down-conversion when they receive messages from producers, not when they are sent to consumers.

Common strategies you can use to upgrade your clients are described as follows. Other strategies for upgrading client applications are also possible.

IMPORTANT	The steps outlined in each strategy change slightly when upgrading to Kafka 3.0.0 or later. From Kafka 3.0.0, the message format version values are assumed to match the `inter.broker.protocol.version` and don't need to be set.

Broker-level consumers first strategy

1. Upgrade all the consuming applications.
2. Change the broker-level `log.message.format.version` to the new version.
3. Upgrade all the producing applications.

This strategy is straightforward, and avoids any broker down-conversion. However, it assumes that all consumers in your organization can be upgraded in a coordinated way, and it does not work for applications that are both consumers and producers. There is also a risk that, if there is a problem with the upgraded clients, new-format messages might get added to the message log so that you cannot revert to the previous consumer version.

Topic-level consumers first strategy

For each topic:

1. Upgrade all the consuming applications.
2. Change the topic-level `message.format.version` to the new version.
3. Upgrade all the producing applications.

This strategy avoids any broker down-conversion, and means you can proceed on a topic-by-topic basis. It does not work for applications that are both consumers and producers of the same topic. Again, it has the risk that, if there is a problem with the upgraded clients, new-format messages might get added to the message log.

Topic-level consumers first strategy with down conversion

For each topic:

1. Change the topic-level `message.format.version` to the old version (or rely on the topic defaulting to the broker-level `log.message.format.version`).
2. Upgrade all the consuming and producing applications.
3. Verify that the upgraded applications function correctly.
4. Change the topic-level `message.format.version` to the new version.

This strategy requires broker down-conversion, but the load on the brokers is minimized because it is only required for a single topic (or small group of topics) at a time. It also works for applications that are both consumers and producers of the same topic. This approach ensures that the upgraded producers and consumers are working correctly before you commit to using the new message format version.

The main drawback of this approach is that it can be complicated to manage in a cluster with many topics and applications.

NOTE	It is also possible to apply multiple strategies. For example, for the first few applications and topics the "per-topic consumers first, with down conversion" strategy can be used. When this has proved successful another, more efficient strategy can be considered acceptable to use instead.

11.5.3. Kafka version and image mappings

When upgrading Kafka, consider your settings for the `STRIMZI_KAFKA_IMAGES` environment variable and the `Kafka.spec.kafka.version` property.

- Each `Kafka` resource can be configured with a `Kafka.spec.kafka.version`.
- The Cluster Operator's `STRIMZI_KAFKA_IMAGES` environment variable provides a mapping between the Kafka version and the image to be used when that version is requested in a given `Kafka` resource.
  - If `Kafka.spec.kafka.image` is not configured, the default image for the given version is used.
  - If `Kafka.spec.kafka.image` is configured, the default image is overridden.

WARNING	The Cluster Operator cannot validate that an image actually contains a Kafka broker of the expected version. Take care to ensure that the given image corresponds to the given Kafka version.

11.5.4. Upgrading Kafka brokers and client applications

Upgrade a Strimzi Kafka cluster to the latest supported Kafka version and *inter-broker protocol version*.

You should also choose a strategy for upgrading clients. Kafka clients are upgraded in step 6 of this procedure.

Prerequisites

- The Cluster Operator is up and running.

- Before you upgrade the Strimzi Kafka cluster, check that the `Kafka.spec.kafka.config` properties of the `Kafka` resource do *not* contain configuration options that are not supported in the new Kafka version.

Procedure

1. Update the Kafka cluster configuration:

```
kubectl edit kafka <my_cluster>
```

2. If configured, check that the `inter.broker.protocol.version` and `log.message.format.version` properties are set to the *current* version.

   For example, the current version is 3.2 if upgrading from Kafka version 3.2.3 to 3.3.2:

```
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.2.3
    config:
      log.message.format.version: "3.2"
      inter.broker.protocol.version: "3.2"
      # ...
```

   If `log.message.format.version` and `inter.broker.protocol.version` are not configured, Strimzi automatically updates these versions to the current defaults after the update to the Kafka version in the next step.

   NOTE The value of `log.message.format.version` and `inter.broker.protocol.version` must be strings to prevent them from being interpreted as floating point numbers.

3. Change the `Kafka.spec.kafka.version` to specify the new Kafka version; leave the `log.message.format.version` and `inter.broker.protocol.version` at the defaults for the *current* Kafka version.

   NOTE Changing the `kafka.version` ensures that all brokers in the cluster will be upgraded to start using the new broker binaries. During this process, some brokers are using the old binaries while others have already upgraded to the new ones. Leaving the `inter.broker.protocol.version` unchanged at the current setting ensures that the brokers can continue to communicate with each other throughout the upgrade.

   For example, if upgrading from Kafka 3.2.3 to 3.3.2:

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

```
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.3.2 ①
    config:
      log.message.format.version: "3.2" ②
      inter.broker.protocol.version: "3.2" ③
      # ...
```

① Kafka version is changed to the new version.

② Message format version is unchanged.

③ Inter-broker protocol version is unchanged.

WARNING	You cannot downgrade Kafka if the `inter.broker.protocol.version` for the new Kafka version changes. The inter-broker protocol version determines the schemas used for persistent metadata stored by the broker, including messages written to `__consumer_offsets`. The downgraded cluster will not understand the messages.

4. If the image for the Kafka cluster is defined in the Kafka custom resource, in `Kafka.spec.kafka.image`, update the `image` to point to a container image with the new Kafka version.

   See Kafka version and image mappings

5. Save and exit the editor, then wait for rolling updates to complete.

   Check the progress of the rolling updates by watching the pod state transitions:

   ```
   kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
   ```

   The rolling updates ensure that each pod is using the broker binaries for the new version of Kafka.

6. Depending on your chosen strategy for upgrading clients, upgrade all client applications to use the new version of the client binaries.

   If required, set the `version` property for Kafka Connect and MirrorMaker as the new version of Kafka:

   a. For Kafka Connect, update `KafkaConnect.spec.version`.

   b. For MirrorMaker, update `KafkaMirrorMaker.spec.version`.

   c. For MirrorMaker 2.0, update `KafkaMirrorMaker2.spec.version`.

7. If configured, update the Kafka resource to use the new `inter.broker.protocol.version` version. Otherwise, go to step 9.

For example, if upgrading to Kafka 3.3.2:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.3.2
    config:
      log.message.format.version: "3.2"
      inter.broker.protocol.version: "3.3"
      # ...
```

8. Wait for the Cluster Operator to update the cluster.
9. If configured, update the Kafka resource to use the new `log.message.format.version` version. Otherwise, go to step 10.

   For example, if upgrading to Kafka 3.3.2:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.3.2
    config:
      log.message.format.version: "3.3"
      inter.broker.protocol.version: "3.3"
      # ...
```

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

10. Wait for the Cluster Operator to update the cluster.
    - The Kafka cluster and clients are now using the new Kafka version.
    - The brokers are configured to send messages using the inter-broker protocol version and message format version of the new version of Kafka.

Following the Kafka upgrade, if required, you can upgrade consumers to use the incremental cooperative rebalance protocol.

11.6. Switching to FIPS mode when upgrading Strimzi

Upgrade Strimzi to run in FIPS mode on FIPS-enabled Kubernetes clusters. Until Strimzi 0.33, running on FIPS-enabled Kubernetes clusters was possible only by disabling FIPS mode using the

`FIPS_MODE` environment variable. From release 0.33, Strimzi supports FIPS mode. If you run Strimzi on a FIPS-enabled Kubernetes cluster with the `FIPS_MODE` set to `disabled`, you can enable it by following this procedure.

Prerequisites

- FIPS-enabled Kubernetes cluster
- An existing Cluster Operator deployment with the `FIPS_MODE` environment variable set to `disabled`

Procedure

1. Upgrade the Cluster Operator to version 0.33 or newer but keep the `FIPS_MODE` environment variable set to `disabled`.
2. If you initially deployed a Strimzi version older than 0.30, it might use old encryption and digest algorithms in its PKCS #12 stores, which are not supported with FIPS enabled. To recreate the certificates with updated algorithms, renew the cluster and clients CA certificates.
    a. To renew the CAs generated by the Cluster Operator, add the `force-renew` annotation to the CA secrets to trigger a renewal.
    b. To renew your own CAs, add the new certificate to the CA secret and update the `ca-cert-generation` annotation with a higher incremental value to capture the update.
3. If you use SCRAM-SHA-512 authentication, check the password length of your users. If they are less than 32 characters long, generate a new password in one of the following ways:
    a. Delete the user secret so that the User Operator generates a new one with a new password of sufficient length.
    b. If you provided your password using the `.spec.authentication.password` properties of the `KafkaUser` custom resource, update the password in the Kubernetes secret referenced in the same password configuration. Don't forget to update your clients to use the new passwords.
4. Ensure that the CA certificates are using the correct algorithms and the SCRAM-SHA-512 passwords are of sufficient length. You can then enable the FIPS mode.
5. Remove the `FIPS_MODE` environment variable from the Cluster Operator deployment. This restarts the Cluster Operator and rolls all the operands to enable the FIPS mode. After the restart is complete, all Kafka clusters now run with FIPS mode enabled.

11.7. Upgrading consumers to cooperative rebalancing

You can upgrade Kafka consumers and Kafka Streams applications to use the *incremental cooperative rebalance* protocol for partition rebalances instead of the default *eager rebalance* protocol. The new protocol was added in Kafka 2.4.0.

Consumers keep their partition assignments in a cooperative rebalance and only revoke them at the end of the process, if needed to achieve a balanced cluster. This reduces the unavailability of the consumer group or Kafka Streams application.

NOTE	Upgrading to the incremental cooperative rebalance protocol is optional. The eager rebalance protocol is still supported.

Prerequisites

- You have upgraded Kafka brokers and client applications to Kafka 3.3.2.

Procedure

To upgrade a Kafka consumer to use the incremental cooperative rebalance protocol:

1. Replace the Kafka clients `.jar` file with the new version.
2. In the consumer configuration, append `cooperative-sticky` to the `partition.assignment.strategy`. For example, if the `range` strategy is set, change the configuration to `range, cooperative-sticky`.
3. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.
4. Reconfigure each consumer in the group by removing the earlier `partition.assignment.strategy` from the consumer configuration, leaving only the `cooperative-sticky` strategy.
5. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.

To upgrade a Kafka Streams application to use the incremental cooperative rebalance protocol:

1. Replace the Kafka Streams `.jar` file with the new version.
2. In the Kafka Streams configuration, set the `upgrade.from` configuration parameter to the Kafka version you are upgrading from (for example, 2.3).
3. Restart each of the stream processors (nodes) in turn.
4. Remove the `upgrade.from` configuration parameter from the Kafka Streams configuration.
5. Restart each consumer in the group in turn.

Chapter 12. Downgrading Strimzi

If you are encountering issues with the version of Strimzi you upgraded to, you can revert your installation to the previous version.

If you used the YAML installation files to install Strimzi, you can use the YAML installation files from the previous release to perform the following downgrade procedures:

1. Downgrading the Cluster Operator to a previous version
2. Downgrading Kafka

If the previous version of Strimzi does not support the version of Kafka you are using, you can also downgrade Kafka as long as the log message format versions appended to messages match.

WARNING	If you deployed Strimzi using another installation method, use a supported approach to downgrade Strimzi. Do not use the downgrade instructions provided here. For example, if you installed Strimzi using the Operator Lifecycle Manager (OLM), you can downgrade by changing the deployment channel to an earlier version of Strimzi.

12.1. Downgrading the Cluster Operator to a previous version

If you are encountering issues with Strimzi, you can revert your installation.

This procedure describes how to downgrade a Cluster Operator deployment to a previous version.

Prerequisites

- An existing Cluster Operator deployment is available.
- You have downloaded the installation files for the previous version.

Procedure

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the `/install/cluster-operator` directory). Any changes will be **overwritten** by the previous version of the Cluster Operator.
2. Revert your custom resources to reflect the supported configuration options available for the version of Strimzi you are downgrading to.
3. Update the Cluster Operator.
   a. Modify the installation files for the previous version according to the namespace the Cluster Operator is running in.

      On Linux, use:

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

```
install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

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

b. If you modified one or more environment variables in your existing Cluster Operator `Deployment`, edit the `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml` file to use those environment variables.

4. When you have an updated configuration, deploy it along with the rest of the installation resources:

```
kubectl replace -f install/cluster-operator
```

Wait for the rolling updates to complete.

5. Get the image for the Kafka pod to ensure the downgrade was successful:

```
kubectl get pod my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The image tag shows the new Strimzi version followed by the Kafka version. For example, `NEW-STRIMZI-VERSION-kafka-CURRENT-KAFKA-VERSION`.

Your Cluster Operator was downgraded to the previous version.

12.2. Downgrading Kafka

Kafka version downgrades are performed by the Cluster Operator.

12.2.1. Kafka version compatibility for downgrades

Kafka downgrades are dependent on compatible current and target Kafka versions, and the state at which messages have been logged.

You cannot revert to the previous Kafka version if that version does not support any of the `inter.broker.protocol.version` settings which have *ever been used* in that cluster, or messages have been added to message logs that use a newer `log.message.format.version`.

The `inter.broker.protocol.version` determines the schemas used for persistent metadata stored by the broker, such as the schema for messages written to `__consumer_offsets`. If you downgrade to a version of Kafka that does not understand an `inter.broker.protocol.version` that has ever been previously used in the cluster the broker will encounter data it cannot understand.

If the target downgrade version of Kafka has:

- The *same* `log.message.format.version` as the current version, the Cluster Operator downgrades by performing a single rolling restart of the brokers.
- A *different* `log.message.format.version`, downgrading is only possible if the running cluster has *always* had `log.message.format.version` set to the version used by the downgraded version. This is typically only the case if the upgrade procedure was aborted before the `log.message.format.version` was changed. In this case, the downgrade requires:
  - Two rolling restarts of the brokers if the interbroker protocol of the two versions is different
  - A single rolling restart if they are the same

Downgrading is *not possible* if the new version has ever used a `log.message.format.version` that is not supported by the previous version, including when the default value for `log.message.format.version` is used. For example, this resource can be downgraded to Kafka version 3.2.3 because the `log.message.format.version` has not been changed:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.3.2
    config:
      log.message.format.version: "3.2"
      # ...
```

The downgrade would not be possible if the `log.message.format.version` was set at `"3.3"` or a value was absent, so that the parameter took the default value for a 3.3.2 broker of 3.3.

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

12.2.2. Downgrading Kafka brokers and client applications

Downgrade a Strimzi Kafka cluster to a lower (previous) version of Kafka, such as downgrading from 3.3.2 to 3.2.3.

Prerequisites

- The Cluster Operator is up and running.
- Before you downgrade the Strimzi Kafka cluster, check the following for the `Kafka` resource:
  - IMPORTANT: Compatibility of Kafka versions.
  - `Kafka.spec.kafka.config` does not contain options that are not supported by the Kafka version being downgraded to.
  - `Kafka.spec.kafka.config` has a `log.message.format.version` and `inter.broker.protocol.version` that is supported by the Kafka version being downgraded to.

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

Procedure

1. Update the Kafka cluster configuration.

```
kubectl edit kafka KAFKA-CONFIGURATION-FILE
```

2. Change the `Kafka.spec.kafka.version` to specify the previous version.

   For example, if downgrading from Kafka 3.3.2 to 3.2.3:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.2.3 ①
    config:
      log.message.format.version: "3.2" ②
      inter.broker.protocol.version: "3.2" ③
      # ...
```

   ① Kafka version is changed to the previous version.

   ② Message format version is unchanged.

   ③ Inter-broker protocol version is unchanged.

   NOTE The value of `log.message.format.version` and `inter.broker.protocol.version` must be strings to prevent them from being interpreted as floating point numbers.

3. If the image for the Kafka version is different from the image defined in `STRIMZI_KAFKA_IMAGES` for the Cluster Operator, update `Kafka.spec.kafka.image`.

   See Kafka version and image mappings

4. Save and exit the editor, then wait for rolling updates to complete.

   Check the update in the logs or by watching the pod state transitions:

```
kubectl logs -f CLUSTER-OPERATOR-POD-NAME | grep -E "Kafka version downgrade from
[0-9.]+ to [0-9.]+, phase ([0-9]+) of \1 completed"
```

```
kubectl get pod -w
```

Check the Cluster Operator logs for an `INFO` level message:

```
Reconciliation #NUM(watch) Kafka(NAMESPACE/NAME): Kafka version downgrade from
FROM-VERSION to TO-VERSION, phase 1 of 1 completed
```

5. Downgrade all client applications (consumers) to use the previous version of the client binaries.

   The Kafka cluster and clients are now using the previous Kafka version.

6. If you are reverting back to a version of Strimzi earlier than 0.22, which uses ZooKeeper for the storage of topic metadata, delete the internal topic store topics from the Kafka cluster.

```
kubectl run kafka-admin -ti --image=quay.io/strimzi/kafka:0.33.0-kafka-3.3.2
--rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server
localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog
--delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic
__strimzi_store_topic --delete
```

Additional resources

- Topic Operator topic store

Chapter 13. Finding information on Kafka restarts

After the Cluster Operator restarts a Kafka pod in a Kubernetes cluster, it emits a Kubernetes event into the pod's namespace explaining why the pod restarted. For help in understanding cluster behavior, you can check restart events from the command line.

TIP You can export and monitor restart events using metrics collection tools like Prometheus. Use the metrics tool with an *event exporter* that can export the output in a suitable format.

13.1. Reasons for a restart event

The Cluster Operator initiates a restart event for a specific reason. You can check the reason by fetching information on the restart event.

The reason given depends on whether you are using `StrimziPodSet` or `StatefulSet` resources for the creation and management of pods.

Table 8. Restart reasons

StrimziPodSet	StatefulSet	Description
CaCertHasOldGeneration	CaCertHasOldGeneration	The pod is still using a server certificate signed with an old CA, so needs to be restarted as part of the certificate update.
CaCertRemoved	CaCertRemoved	Expired CA certificates have been removed, and the pod is restarted to run with the current certificates.
CaCertRenewed	CaCertRenewed	CA certificates have been renewed, and the pod is restarted to run with the updated certificates.
ClientCaCertKeyReplaced	ClientCaCertKeyReplaced	The key used to sign clients CA certificates has been replaced, and the pod is being restarted as part of the CA renewal process.
ClusterCaCertKeyReplaced	ClusterCaCertKeyReplaced	The key used to sign the cluster's CA certificates has been replaced, and the pod is being restarted as part of the CA renewal process.
ConfigChangeRequiresRestart	ConfigChangeRequiresRestart	Some Kafka configuration properties are changed dynamically, but others require that the broker be restarted.
CustomListenerCaCertChanged	CustomListenerCaCertChanged	The CA certificate used to secure the Kafka network listeners has changed, and the pod is restarted to use it.

StrimziPodSet	StatefulSet	Description
FileSystemResizeNeeded	FileSystemResizeNeeded	The file system size has been increased, and a restart is needed to apply it.
KafkaCertificatesChanged	KafkaCertificatesChanged	One or more TLS certificates used by the Kafka broker have been updated, and a restart is needed to use them.
ManualRollingUpdate	ManualRollingUpdate	A user annotated the pod, or the `StatefulSet`` or `StrimziPodSet` set it belongs to, to trigger a restart.
PodForceRestartOnError	PodForceRestartOnError	An error occurred that requires a pod restart to rectify.
PodHasOldRevision	JbodVolumesChanged	A disk was added or removed from the Kafka volumes, and a restart is needed to apply the change. When using `StrimziPodSet` resources, the same reason is given if the pod needs to be recreated.
PodHasOldRevision	PodHasOldGeneration	The `StatefulSet` or `StrimziPodSet` that the pod is a member of has been updated, so the pod needs to be recreated. When using `StrimziPodSet` resources, the same reason is given if a disk was added or removed from the Kafka volumes.
PodStuck	PodStuck	The pod is still pending, and is not scheduled or cannot be scheduled, so the operator has restarted the pod in a final attempt to get it running.
PodUnresponsive	PodUnresponsive	Strimzi was unable to connect to the pod, which can indicate a broker not starting correctly, so the operator restarted it in an attempt to resolve the issue.

13.2. Restart event filters

When checking restart events from the command line, you can specify a `field-selector` to filter on Kubernetes event fields.

The following fields are available when filtering events with `field-selector`.

`regardingObject.kind`

The object that was restarted, and for restart events, the kind is always `Pod`.

`regarding.namespace`

The namespace that the pod belongs to.

`regardingObject.name`

The pod's name, for example, `strimzi-cluster-kafka-0`.

`regardingObject.uid`

The unique ID of the pod.

`reason`

The reason the pod was restarted, for example, `JbodVolumesChanged`.

`reportingController`

The reporting component is always `strimzi.io/cluster-operator` for Strimzi restart events.

`source`

`source` is an older version of `reportingController`. The reporting component is always `strimzi.io/cluster-operator` for Strimzi restart events.

`type`

The event type, which is either `Warning` or `Normal`. For Strimzi restart events, the type is `Normal`.

NOTE In older versions of Kubernetes, the fields using the `regarding` prefix might use an `involvedObject` prefix instead. `reportingController` was previously called `reportingComponent`.

13.3. Checking Kafka restarts

Use a `kubectl` command to list restart events initiated by the Cluster Operator. Filter restart events emitted by the Cluster Operator by setting the Cluster Operator as the reporting component using the `reportingController` or `source` event fields.

Prerequisites

- The Cluster Operator is running in the Kubernetes cluster.

Procedure

1. Get all restart events emitted by the Cluster Operator:

```
kubectl -n kafka get events --field-selector
reportingController=strimzi.io/cluster-operator
```

Example showing events returned

```
LAST SEEN   TYPE     REASON                   OBJECT                         MESSAGE
2m          Normal   CaCertRenewed            pod/strimzi-cluster-kafka-0    CA
certificate renewed
58m         Normal   PodForceRestartOnError   pod/strimzi-cluster-kafka-1    Pod
needs to be forcibly restarted due to an error
5m47s       Normal   ManualRollingUpdate      pod/strimzi-cluster-kafka-2    Pod was
```

```
manually annotated to be rolled
```

You can also specify a `reason` or other `field-selector` options to constrain the events returned.

Here, a specific reason is added:

```
kubectl -n kafka get events --field-selector
reportingController=strimzi.io/cluster-operator,reason=PodForceRestartOnError
```

2. Use an output format, such as YAML, to return more detailed information about one or more events.

```
kubectl -n kafka get events --field-selector
reportingController=strimzi.io/cluster-operator,reason=PodForceRestartOnError -o
yaml
```

Example showing detailed events output

```
apiVersion: v1
items:
- action: StrimziInitiatedPodRestart
  apiVersion: v1
  eventTime: "2022-05-13T00:22:34.168086Z"
  firstTimestamp: null
  involvedObject:
      kind: Pod
      name: strimzi-cluster-kafka-1
      namespace: kafka
  kind: Event
  lastTimestamp: null
  message: Pod needs to be forcibly restarted due to an error
  metadata:
      creationTimestamp: "2022-05-13T00:22:34Z"
      generateName: strimzi-event
      name: strimzi-eventwppk6
      namespace: kafka
      resourceVersion: "432961"
      uid: 29fcdb9e-f2cf-4c95-a165-a5efcd48edfc
  reason: PodForceRestartOnError
  reportingController: strimzi.io/cluster-operator
  reportingInstance: strimzi-cluster-operator-6458cfb4c6-6bpdp
  source: {}
  type: Normal
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

The following fields are deprecated, so they are not populated for these events:

- `firstTimestamp`
- `lastTimestamp`
- `source`

Chapter 14. Uninstalling Strimzi

You can uninstall Strimzi using the CLI or by unsubscribing from OperatorHub.io.

Use the same approach you used to install Strimzi.

When you uninstall Strimzi, you will need to identify resources created specifically for a deployment and referenced from the Strimzi resource.

Such resources include:

- Secrets (Custom CAs and certificates, Kafka Connect secrets, and other Kafka secrets)
- Logging `ConfigMaps` (of type `external`)

These are resources referenced by `Kafka`, `KafkaConnect`, `KafkaMirrorMaker`, or `KafkaBridge` configuration.

WARNING	Deleting `CustomResourceDefinitions` results in the garbage collection of the corresponding custom resources (`Kafka`, `KafkaConnect`, `KafkaMirrorMaker`, or `KafkaBridge`) and the resources dependent on them (Deployments, StatefulSets, and other dependent resources).

14.1. Uninstalling Strimzi using the CLI

This procedure describes how to use the `kubectl` command-line tool to uninstall Strimzi and remove resources related to the deployment.

Prerequisites

- Access to a Kubernetes cluster using an account with `cluster-admin` or `strimzi-admin` permissions.
- You have identified the resources to be deleted.

  You can use the following `kubectl` CLI command to find resources and also verify that they have been removed when you have uninstalled Strimzi.

  Command to find resources related to a Strimzi deployment

  ```
  kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
  ```

  Replace *<resource_type>* with the type of the resource you are checking, such as `secret` or `configmap`.

Procedure

1. Delete the Cluster Operator `Deployment`, related `CustomResourceDefinitions`, and `RBAC` resources.

   Specify the installation files used to deploy the Cluster Operator.

```
kubectl delete -f install/cluster-operator
```

2. Delete the resources you identified in the prerequisites.

```
kubectl delete <resource_type> <resource_name> -n <namespace>
```

Replace *<resource_type>* with the type of resource you are deleting and *<resource_name>* with the name of the resource.

Example to delete a secret

```
kubectl delete secret my-cluster-clients-ca -n my-project
```

14.2. Uninstalling Strimzi from OperatorHub.io

This procedure describes how to uninstall Strimzi from OperatorHub.io and remove resources related to the deployment.

You perform the steps using the `kubectl` command-line tool.

Prerequisites

- Access to a Kubernetes cluster using an account with `cluster-admin` or `strimzi-admin` permissions.
- You have identified the resources to be deleted.

  You can use the following `kubectl` CLI command to find resources and also verify that they have been removed when you have uninstalled Strimzi.

  Command to find resources related to a Strimzi deployment

```
kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

  Replace *<resource_type>* with the type of the resource you are checking, such as `secret` or `configmap`.

Procedure

1. Delete the Strimzi subscription.

```
kubectl delete subscription strimzi-cluster-operator -n <namespace>
```

2. Delete the cluster service version (CSV).

```
kubectl delete csv strimzi-cluster-operator.<version>  -n <namespace>
```

3. Remove related CRDs.

```
kubectl get crd -l app=strimzi -o name | xargs kubectl delete
```