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. Additional deployment options described include the steps to introduce 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. How Strimzi supports Kafka

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

Operators simplify the process of:

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

1.2. Strimzi Operators

Strimzi supports Kafka using *Operators* to deploy and manage the components and dependencies of Kafka to Kubernetes.

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

Operators

Strimzi provides Operators for managing a Kafka cluster running within a Kubernetes cluster.

Cluster Operator

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

Entity Operator

Comprises the Topic Operator and User Operator

Topic Operator

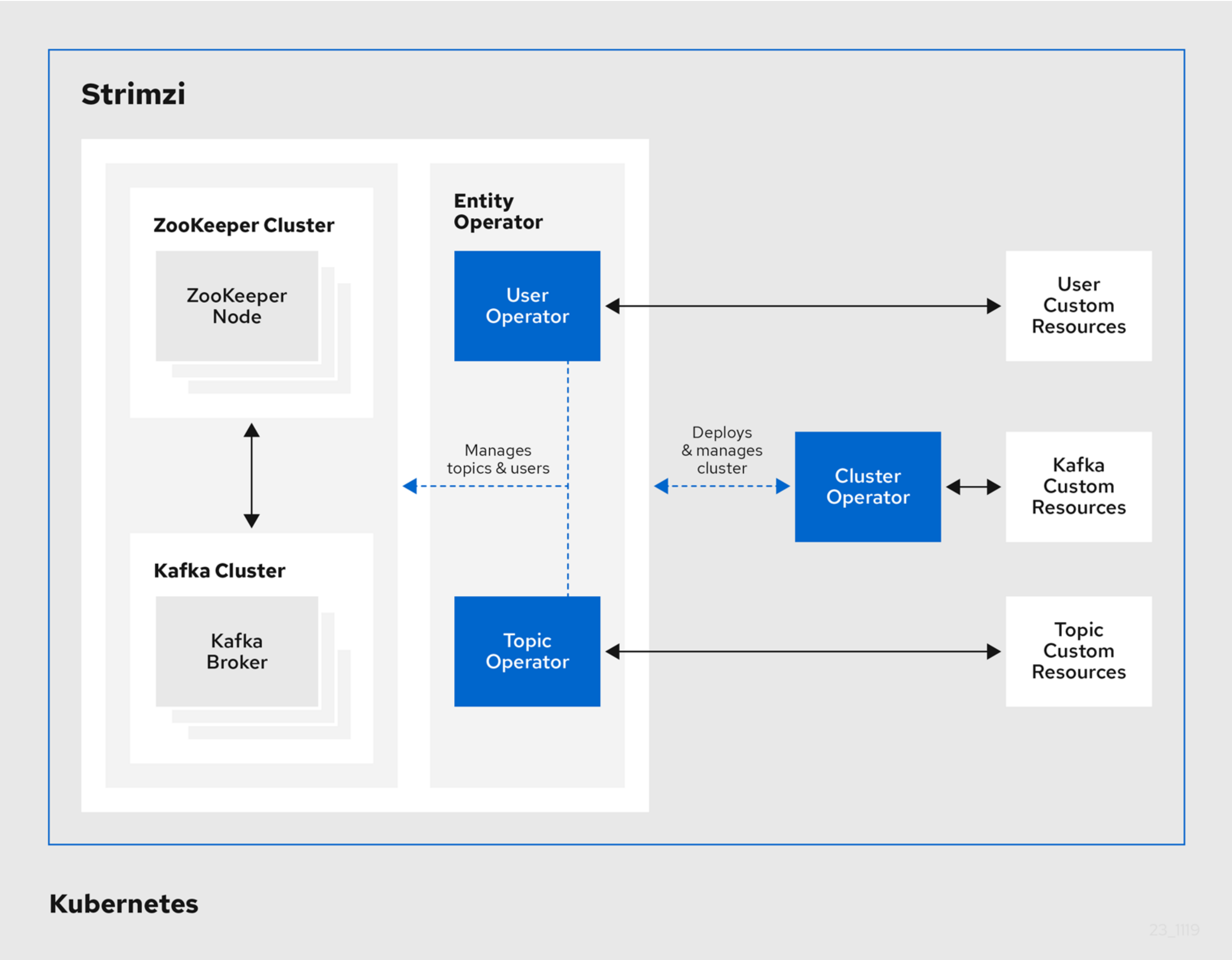
Manages Kafka topics

User Operator

Manages Kafka users

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

Operators within the Strimzi architecture



1.2.1. Cluster Operator

Strimzi uses the Cluster Operator to deploy and manage clusters for:

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

Custom resources are used to deploy the clusters.

For example, to deploy a Kafka cluster:

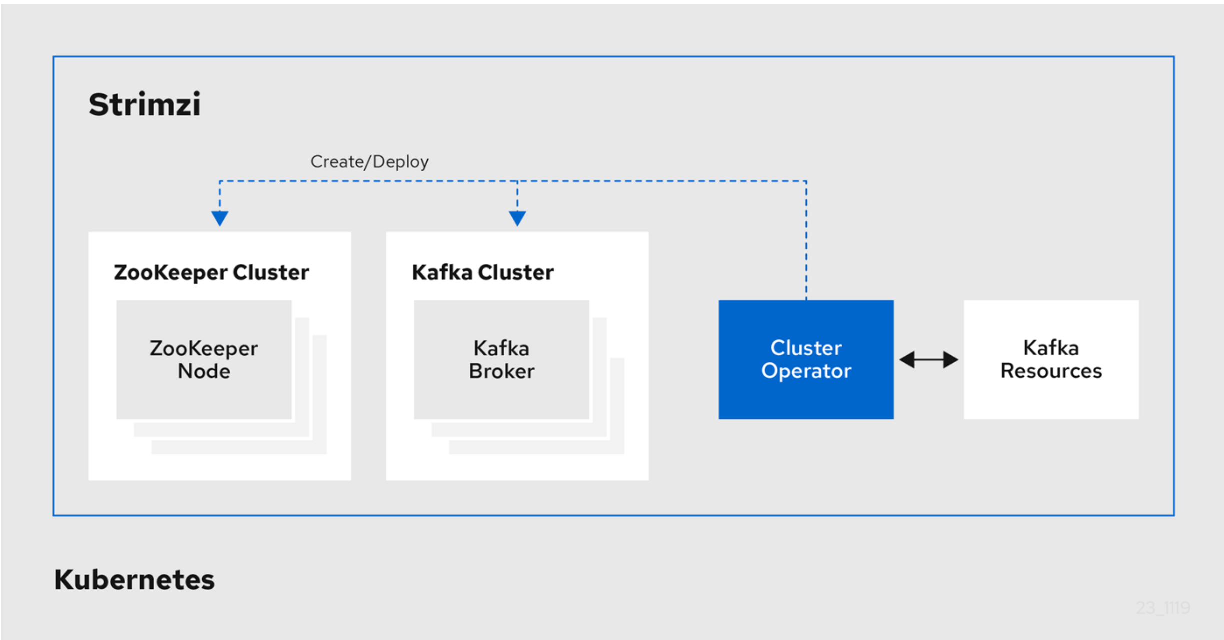
- A `Kafka` resource with the cluster configuration is created within the Kubernetes cluster.
- The Cluster Operator deploys a corresponding Kafka cluster, based on what is declared in the `Kafka` resource.

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

- A Topic Operator to provide operator-style topic management through `KafkaTopic` custom resources
- A User Operator to provide operator-style user management through `KafkaUser` custom resources

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

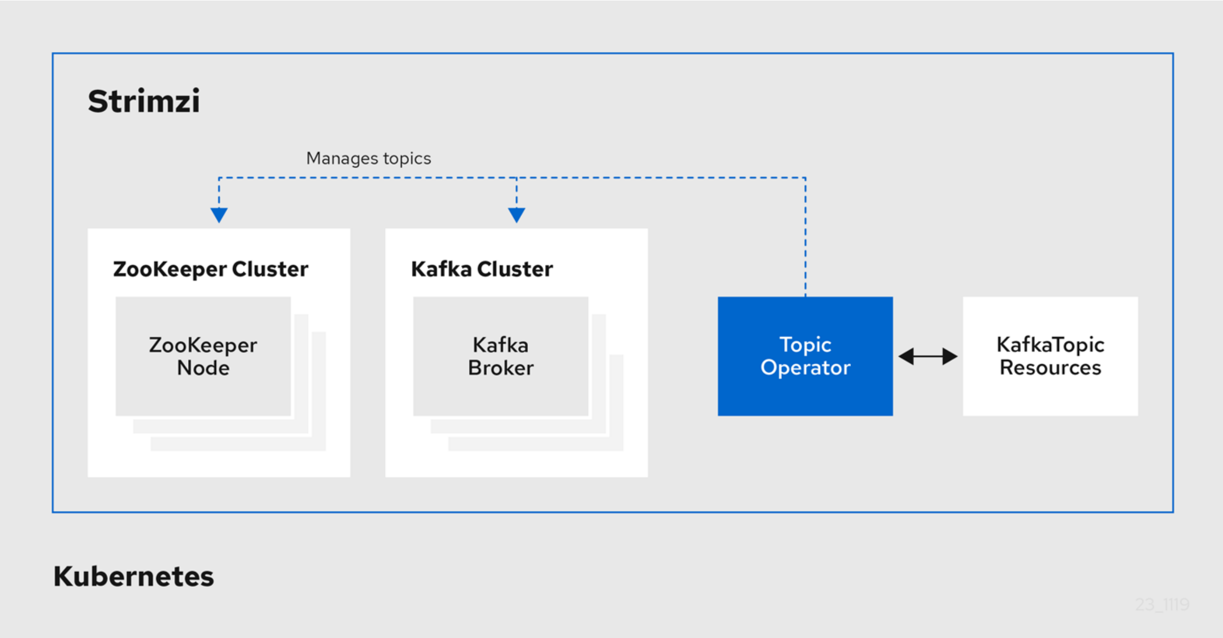
Example architecture for the Cluster Operator



1.2.2. Topic Operator

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

Example architecture for the Topic Operator



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

Specifically, if a `KafkaTopic` is:

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

Working in the other direction, if a topic is:

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

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

The Topic Operator maintains information about each topic in a *topic store*, which is continually synchronized with updates from Kafka topics or Kubernetes `KafkaTopic` custom resources. Updates from operations applied to a local in-memory topic store are persisted to a backup topic store on disk. If a topic is reconfigured or reassigned to other brokers, the `KafkaTopic` will always be up to date.

1.2.3. User Operator

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

For example, if a `KafkaUser` is:

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

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

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

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

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

1.2.4. Feature gates in Strimzi Operators

You can enable and disable some features of operators using *feature gates*.

Feature gates are set in the operator configuration and have three stages of maturity: alpha, beta, or General Availability (GA).

For more information, see Feature gates.

1.3. 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.

Additional resources

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

1.3.1. 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 in the *Deploying and Upgrading Strimzi* guide.

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.

Chapter 2. 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.

2.1. 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

2.2. Additional deployment configuration options

The deployment procedures in this guide describe a deployment using the example installation YAML files provided with Strimzi. The procedures highlight any important configuration considerations, but they do not describe all the configuration options available.

You can use custom resources to refine your deployment.

You may wish to review the configuration options available for Kafka components before you deploy Strimzi. For more information on the configuration through custom resources, see Deployment configuration in the *Using Strimzi* guide.

2.2.1. 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*, which are described in the *Using Strimzi* guide:

- 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.

2.2.2. Monitoring your 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, as described in the *Using Strimzi* guide.

Chapter 3. 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.

3.1. Deployment prerequisites

To deploy Strimzi, make sure that:

- A Kubernetes 1.16 and later cluster is available.

  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`.

3.2. Downloading Strimzi release artifacts

To install Strimzi, download the release artifacts from GitHub.

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.

3.3. 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.

3.3.1. Example files location

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

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.

3.3.2. 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.

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

3.4. 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:0.26.0-kafka-2.8.0 • quay.io/strimzi/kafka:0.26.0-kafka-2.8.1 • quay.io/strimzi/kafka:0.26.0-kafka-3.0.0	Strimzi image for running Kafka, including: • Kafka Broker • Kafka Connect • Kafka MirrorMaker • ZooKeeper • TLS Sidecars
Operator	• quay.io/strimzi/operator:0.26.0	Strimzi image for running the operators: • Cluster Operator • Topic Operator • User Operator • Kafka Initializer
Kafka Bridge	• quay.io/strimzi/kafka-bridge:0.20.3	Strimzi image for running the Strimzi kafka Bridge
JmxTrans	• quay.io/strimzi/jmxtrans:0.26.0	Strimzi image for running the Strimzi JmxTrans

3.5. 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
```

3.6. 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 4. Deploying Strimzi

Having prepared your environment for a deployment of Strimzi, this section shows:

- How to create the Kafka cluster
- Optional procedures to deploy other Kafka components according to your requirements:
  - Kafka Connect
  - Kafka MirrorMaker
  - Kafka Bridge

The procedures assume a Kubernetes cluster is available and running.

This section describes the procedures to deploy Strimzi on Kubernetes 1.16 and later.

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

4.1. Create the Kafka cluster

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.

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. But you can deploy and use the Topic Operator and User Operator as standalone components.

NOTE	The Cluster Operator can watch one, multiple, or all namespaces in a Kubernetes cluster. The Topic Operator and User Operator watch for `KafkaTopics` and `KafkaUsers` in the single namespace of the Kafka cluster deployment.

Deploying a Kafka cluster with the Topic Operator and User Operator

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

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

Deploying a standalone Topic Operator and User Operator

Perform these deployment steps if you want to use the Topic Operator and User Operator with a

Kafka cluster that is **not managed** by Strimzi.

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

4.1.1. Deploying the Cluster Operator

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

The procedures in this section show:

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

Watch options for a Cluster Operator deployment

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

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

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

NOTE Strimzi provides example YAML files to make the deployment process easier.

The Cluster Operator watches for changes to the following resources:

- `Kafka` for the Kafka cluster.
- `KafkaConnect` for the Kafka Connect cluster.
- `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.

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

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

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. Verify that the Cluster Operator was successfully deployed:

```
kubectl get deployments
```

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

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

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:0.26.0
        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/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. Verify that the Cluster Operator was successfully deployed:

   ```
   kubectl get deployments
   ```

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

- This procedure requires use of a Kubernetes user account which is able to create `CustomResourceDefinitions`, `ClusterRoles` and `ClusterRoleBindings`. Use of Role Base Access

Control (RBAC) in the Kubernetes cluster usually means that permission to create, edit, and delete these resources is limited to Kubernetes cluster administrators, such as `system:admin`.

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:0.26.0
           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-entity-operator-
delegation --clusterrole=strimzi-entity-operator --serviceaccount my-cluster-
operator-namespace:strimzi-cluster-operator
```

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

4. Deploy the Cluster Operator to your Kubernetes cluster.

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

5. Verify that the Cluster Operator was successfully deployed:

```
kubectl get deployments
```

Deploying the Cluster Operator using a Helm Chart

As an alternative to using the YAML deployment files, this procedure shows how to deploy the Cluster Operator using a Helm chart provided with Strimzi.

Prerequisites

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

For more information about Helm, see the Helm website.

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
```

Upgrading the Cluster Operator using Helm Chart

To upgrade the Strimzi operator, you can use the `helm upgrade` command. 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 GitHub or find them in the `crd` subdirectory inside the Helm Chart.

Deploying the Cluster Operator from OperatorHub.io

OperatorHub.io is a catalog of Kubernetes Operators sourced from multiple providers. It offers you an alternative way to install stable versions of Strimzi using the Strimzi Kafka Operator.

The Operator Lifecycle Manager is used for the installation and management of all Operators published on OperatorHub.io.

To install Strimzi from OperatorHub.io, locate the *Strimzi Kafka Operator* and follow the instructions provided.

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

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.

4.1.2. Deploying Kafka

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

The procedures in this section show:

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

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

Deploying the Kafka cluster

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

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

Strimzi provides example configuration files. For a Kafka deployment, the following examples are provided:

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

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

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

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

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

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

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

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

The example YAML files specify the latest supported Kafka version, and configuration for its supported log message format version and inter-broker protocol version. Updates to these properties are 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: 3.0.0
    #...
    config:
      #...
      log.message.format.version: 3.0
      inter.broker.protocol.version: 3.0
  # ...
```

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create and deploy an *ephemeral* or *persistent* cluster.

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

   - To create and deploy an *ephemeral* cluster:

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

   - To create and deploy a *persistent* cluster:

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

2. Verify that the Kafka cluster was successfully deployed:

   ```
   kubectl get deployments
   ```

Additional resources

Kafka cluster configuration

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 `KafkaTopics` in the namespace of the Kafka cluster deployment.

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 in the *Using Strimzi* guide.

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:

   Use `kubectl apply`:

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

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 `KafkaUsers` in the namespace of the Kafka cluster deployment.

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 in the *Using Strimzi* guide.

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 in the *Using Strimzi* guide.

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

3. Create or update the resource:

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

4.1.3. 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.

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. Configure the `05-Deployment-strimzi-topic-operator.yaml` deployment file to add the environment variables that enable the Topic Operator to connect to a Kafka cluster.

Prerequisites

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

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 selector to identify the `KafkaTopic` resources managed by the Topic Operator.

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 mutual TLS authentication. Setting this to `false` disables client authentication with TLS 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 TLS with the `STRIMZI_TLS_ENABLED` environment variable, specify the keystore and truststore used to authenticate connection to the Kafka cluster.

Example TLS 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 TLS client authentication.

④ The password for accessing the keystore.

4. Deploy the Topic Operator.

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

5. Verify that the Topic Operator has been deployed successfully.

```
kubectl describe deployment strimzi-topic-operator
```

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

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

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. Edit the `05-Deployment-strimzi-user-operator.yaml` deployment file to add the environment variables that enable the User Operator to connect to a Kafka cluster.

Prerequisites

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

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_LOG_LEVEL ⑦
              value: INFO
            - name: STRIMZI_GC_LOG_ENABLED ⑧
              value: "true"
            - name: STRIMZI_CA_VALIDITY ⑨
              value: "365"
            - name: STRIMZI_CA_RENEWAL ⑩
              value: "30"
            - name: STRIMZI_JAVA_OPTS ⑪
              value: "-Xmx=512M -Xms=256M"
            - name: STRIMZI_JAVA_SYSTEM_PROPERTIES ⑫
              value: "-Djavax.net.debug=verbose -DpropertyName=value"
            - name: STRIMZI_SECRET_PREFIX ⑬
              value: "kafka-"
            - name: STRIMZI_ACLS_ADMIN_API_SUPPORTED ⑭
              value: "true"
```

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 Certificate Authority that signs new user certificates for TLS client authentication.

④ The Kubernetes `Secret` that contains the private key (`ca.key`) value of the Certificate Authority that signs new user certificates for TLS client authentication.

⑤ The label selector used to identify the `KafkaUser` resources managed by the User Operator.

⑥ The interval between periodic reconciliations, in milliseconds. The default is `120000` (2 minutes). The default is `18000` (18 seconds).

⑦ The level for printing logging messages. You can set the level to `ERROR`, `WARNING`, `INFO`, `DEBUG`, or `TRACE`.

⑧ Enables garbage collection (GC) logging. The default is `true`.

⑨ The validity period for the Certificate Authority. The default is `365` days.

⑩ The renewal period for the Certificate Authority. 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.

⑪ (Optional) The Java options used by the JVM running the User Operator

⑫ (Optional) The debugging (`-D`) options set for the User Operator

⑬ (Optional) Prefix for the names of Kubernetes secrets created by the User Operator.

⑭ (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`.

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

   Example TLS configuration

```
# ....
env:
  - name: STRIMZI_CLUSTER_CA_CERT_SECRET_NAME ①
    value: my-cluster-cluster-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 Certificate Authority that signs Kafka broker certificates for TLS client authentication.

② The Kubernetes `Secret` that contains the keystore (`entity-operator.p12`) with the private key and certificate for TLS 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. Verify that the User Operator has been deployed successfully.

```
kubectl describe deployment strimzi-user-operator
```

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

> **NOTE** You might experience a delay with the deployment if you have a slow connection to the Kubernetes cluster and the User Operator images have not been downloaded before.

4.2. Deploy Kafka Connect

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

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

The procedures in this section show how to:

- Deploy a Kafka Connect cluster using a `KafkaConnect` resource
- Run multiple Kafka Connect instances
- Create a Kafka Connect image containing the connectors you need to make your connection
- Create and manage connectors using a KafkaConnector resource or the Kafka Connect REST API
- Deploy a KafkaConnector resource to Kafka Connect
- Restart a Kafka connector by annotating a KafkaConnector resource
- Restart a Kafka connector task by annotating a KafkaConnector resource

> **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.

4.2.1. 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. Verify that Kafka Connect was successfully deployed:

```
kubectl get deployments
```

Additional resources

Kafka Connect cluster configuration

4.2.2. Kafka Connect configuration for multiple instances

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

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

4.2.3. Extending Kafka Connect with connector plug-ins

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

Table 1. File connectors

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

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

- Creating a new container image automatically using Strimzi
- Creating a container image from the Kafka Connect base image (manually or using continuous integration)

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

Creating a new container image automatically using Strimzi

This procedure shows how to 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/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb548045e115e
33a9e69b1b2a352bee24df035a0447cb820077af00c03
      - name: camel-telegram
        artifacts:
          - type: tgz
            url:
https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-
telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
            sha512sum:
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-CONFIG-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 the KafkaConnector custom resources to use the connector plugins you added.

Additional resources

See the *Using Strimzi* guide for more information on:

- Kafka Connect `Build` schema reference

Creating a Docker image from the Kafka Connect base image

This procedure shows how to create a custom image and add it to the `/opt/kafka/plugins` directory.

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

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

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create a new `Dockerfile` using `quay.io/strimzi/kafka:0.26.0-kafka-3.0.0` as the base image:

```
FROM quay.io/strimzi/kafka:0.26.0-kafka-3.0.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Example plug-in file

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

NOTE This example uses the Debezium connectors for MongoDB, MySQL, and PostgreSQL. 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 either:

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

If set, this property overrides the STRIMZI_KAFKA_CONNECT_IMAGES variable in the Cluster Operator.

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

or

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

Additional resources

See the *Using Strimzi* guide for more information on:

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

4.2.4. Creating and managing connectors

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

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

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

Source connector

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

Sink connector

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

Strimzi provides two APIs for creating and managing connectors:

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

Using the APIs, you can:

- Check the status of a connector instance
- Reconfigure a running connector
- Increase or decrease the number of connector tasks for a connector instance
- Restart connectors
- Restart connector tasks, including failed tasks
- Pause a connector instance
- Resume a previously paused connector instance
- Delete a connector instance

KafkaConnector resources

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

You manage a running connector instance by updating its corresponding KafkaConnector resource, and then applying the updates. Annotations are used to manually restart connector instances and connector tasks. You remove a connector by deleting its corresponding KafkaConnector.

To ensure compatibility with earlier versions of Strimzi, KafkaConnectors are disabled by default. To enable them for a Kafka Connect cluster, you must use annotations on the `KafkaConnect` resource. For instructions, see Configuring Kafka Connect in the *Using Strimzi* guide.

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

Strimzi provides example configuration files, including the following example `KafkaConnector` file:

- `examples/connect/source-connector.yaml`.

You can use this example to create and manage a `FileStreamSourceConnector` and a `FileStreamSinkConnector` as described in Deploying the example KafkaConnector resources.

Availability of the Kafka Connect REST API

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

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

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

4.2.5. Deploying the example KafkaConnector resources

Use KafkaConnectors with Kafka Connect to stream data to and from a Kafka cluster.

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

- `examples/connect/source-connector.yaml`.

The file is used to create the following connector instances:

- 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).

> **NOTE**
>
> In a production environment, you prepare container images containing your desired Kafka Connect connectors, as described in Extending Kafka Connect with connector plug-ins.
>
> The `FileStreamSourceConnector` and `FileStreamSinkConnector` are provided as examples. Running these connectors in containers as described here is unlikely to be suitable for production use cases.

Prerequisites

- A Kafka Connect deployment
- KafkaConnectors are enabled in the Kafka Connect deployment
- The Cluster Operator is running

Procedure

1. 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
  config: ⑤
    file: "/opt/kafka/LICENSE" ⑥
    topic: my-topic ⑦
    # ...
```

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 Connector configuration as key-value pairs.

⑥ This example source connector configuration reads data from the `/opt/kafka/LICENSE` file.

⑦ Kafka topic to publish the source data to.

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

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

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

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

4. 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.

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

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

6. 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 your Kafka Connect cluster.

7. 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-CLUSTER-kafka-0 -i -t -- bin/kafka-console-consumer.sh --bootstrap
-server MY-CLUSTER-kafka-bootstrap.NAMESPACE.svc:9092 --topic my-topic --from
-beginning
```

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 2. 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 3. 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.

Additional resources

- Creating and managing connectors

4.2.6. Performing a restart of a Kafka connector

This procedure describes how to manually trigger a restart of a Kafka connector by using a Kubernetes annotation.

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. To restart the connector, annotate the `KafkaConnector` resource in Kubernetes. For example, using `kubectl annotate`:

```
kubectl annotate KafkaConnector KAFKACONNECTOR-NAME strimzi.io/restart=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.

Additional resources

- Creating and managing connectors in the *Deploying and Upgrading* guide.

4.2.7. Performing a restart of a Kafka connector task

This procedure describes how to manually trigger a restart of a Kafka connector task by using a Kubernetes annotation.

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 KAFKACONNECTOR-NAME
```

3. To restart the connector task, annotate the `KafkaConnector` resource in Kubernetes. For example, using `kubectl annotate` to restart task 0:

```
kubectl annotate KafkaConnector KAFKACONNECTOR-NAME strimzi.io/restart-task=0
```

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.

Additional resources

- Creating and managing connectors in the *Deploying and Upgrading* guide.

4.3. Deploy Kafka MirrorMaker

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

4.3.1. 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. Verify that MirrorMaker was successfully deployed:

   ```
   kubectl get deployments
   ```

Additional resources

- Kafka MirrorMaker cluster configuration

4.4. Deploy Kafka Bridge

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

4.4.1. 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. Verify that Kafka Bridge was successfully deployed:

```
kubectl get deployments
```

Additional resources

Kafka Bridge cluster configuration

Chapter 5. Setting up client access to the 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 external client access to the Kafka cluster

  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 described in the *Using Strimzi* guide.

5.1. 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:0.26.0-kafka-3.0.0
--rm=true --restart=Never -- bin/kafka-console-producer.sh --broker-list cluster-
name-kafka-bootstrap:9092 --topic my-topic
```

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

```
kubectl run kafka-consumer -ti --image=quay.io/strimzi/kafka:0.26.0-kafka-3.0.0
--rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server
cluster-name-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

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

5.2. Setting up access for clients outside of Kubernetes

This procedure shows how to configure client access to a Kafka cluster from outside Kubernetes.

Using the address of the Kafka cluster, you can provide external access to a client on a different Kubernetes namespace or outside Kubernetes entirely.

You configure an external Kafka listener to provide the access.

The following external listener types are supported:

- `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 NGINX Ingress Controller for Kubernetes

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

You can configure your listener to use TLS, SCRAM-SHA-512 or OAuth 2.0 authentication. TLS always uses encryption, but it is recommended to also use encryption with SCRAM-SHA-512 and OAuth 2.0 authentication.

You can configure simple, OAuth 2.0, 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 TLS if it is not also enabled in the Kafka configuration.

Strimzi operators automate the configuration process:

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

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

Certificates are available in PKCS #12 (.p12) and PEM (.crt) formats. This procedure shows PKCS #12 certificates.

Prerequisites

- The Kafka cluster is available for the client
- The Cluster Operator and User Operator are running in the cluster
- A client outside the Kubernetes cluster to connect to the Kafka cluster

Procedure

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

   For example:

```
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:
        preferredNodePortAddressType: InternalDNS ⑦
        bootstrap and broker service overrides ⑧
        #...
    authorization: ⑨
      type: simple
      superUsers:
        - super-user-name ⑩
  # ...
```

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`.

⑤ Enables TLS encryption on the listener. Default is `false`. TLS encryption is not required for `route` listeners.

⑥ Authentication specified as `tls`.

⑦ (Optional, for `nodeport` listeners only) Configuration to specify a preference for the first address type used by Strimzi as the node address.

⑧ (Optional) Strimzi automatically determines the addresses to advertise to clients. The addresses are automatically assigned by Kubernetes. You can override bootstrap and broker service addresses if the infrastructure on which you are running Strimzi does not provide the right address. Validation is not performed on the overrides. The override configuration differs according to the listener type. For example, you can override hosts for `route`, DNS names or IP addresses for `loadbalancer`, and node ports for `nodeport`.

⑨ Authorization specified as `simple`, which uses the `AclAuthorizer` Kafka plugin.

⑩ (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-CONFIG-FILE
```

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

A service is created for each Kafka broker pod.

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

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 with the same name as the `Kafka` resource.

3. Find the bootstrap address and port from the status of the `Kafka` resource.

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

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.

     For example:

     ```
     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
             operation: Read
           - resource:
               type: topic
               name: my-topic
               patternType: literal
             operation: Describe
           - resource:
               type: group
               name: my-group
               patternType: literal
             operation: Read
     ```

     ① The label must match the label of the Kafka cluster for the user to be created.

     ② Authentication specified as `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 private and public key for TLS client authentication.

For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: PUBLIC-KEY-OF-THE-CLIENT-CA
  user.crt: USER-CERTIFICATE-CONTAINING-PUBLIC-KEY-OF-USER
  user.key: PRIVATE-KEY-OF-USER
  user.p12: P12-ARCHIVE-FILE-STORING-CERTIFICATES-AND-KEYS
  user.password: PASSWORD-PROTECTING-P12-ARCHIVE
```

6. Extract the public cluster CA certificate to the desired certificate format:

```
kubectl get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.p12}'
| base64 -d > ca.p12
```

7. Extract the password from the password file:

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

8. Configure your client with the authentication details for the public cluster certificates:

Sample client code

```
properties.put("security.protocol","SSL"); ①
properties.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,"/path/to/ca.p12"); ②
properties.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,CA-PASSWORD); ③
properties.put(SslConfigs.SSL_TRUSTSTORE_TYPE_CONFIG,"PKCS12"); ④
```

① Enables TLS encryption (with or without TLS client authentication).

② Specifies the truststore location where the certificates were imported.

③ Specifies the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.

④ Identifies the truststore type.

NOTE Use `security.protocol: SASL_SSL` when using SCRAM-SHA authentication over TLS.

9. Extract the user CA certificate from the user Secret to the desired certificate format:

```
kubectl get secret USER-NAME -o jsonpath='{.data.user\.p12}' | base64 -d > user.p12
```

10. Extract the password from the password file:

```
kubectl get secret USER-NAME -o jsonpath='{.data.user\.password}' | base64 -d >
user.password
```

11. Configure your client with the authentication details for the user CA certificate:

Sample client code

```
properties.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG,"/path/to/user.p12"); ①
properties.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG,"<user.password>"); ②
properties.put(SslConfigs.SSL_KEYSTORE_TYPE_CONFIG,"PKCS12"); ③
```

① Specifies the keystore location where the certificates were imported.

② Specifies the password for accessing the keystore. This property can be omitted if it is not needed by the keystore. The public user certificate is signed by the client CA when it is created.

③ Identifies the keystore type.

12. Add the bootstrap address and port for connecting to the Kafka cluster:

```
bootstrap.servers: BOOTSTRAP-ADDRESS:PORT
```

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 6. Introducing metrics to Kafka

You can use Prometheus and Grafana to monitor your Strimzi 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.

Additional resources

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

6.1. Monitoring consumer lag with Kafka Exporter

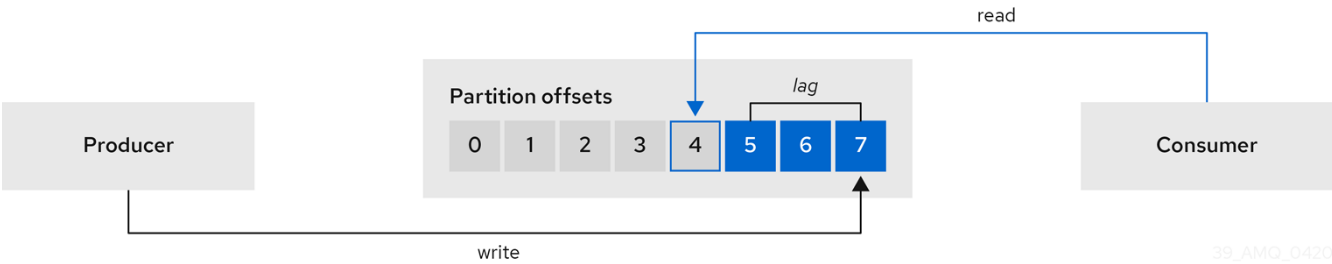
Kafka Exporter is an open source project to enhance monitoring of Apache Kafka brokers and clients. You can configure the `Kafka` resource to deploy Kafka Exporter with your Kafka cluster. Kafka Exporter extracts additional metrics data from Kafka brokers related to offsets, consumer groups, consumer 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.

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.

6.2. 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.

6.2.1. 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 4. Example custom resources with metrics configuration

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

6.2.2. 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.

Kafka alerting rules

`UnderReplicatedPartitions`

Gives the number of partitions for which the current broker is the lead replica but which have fewer replicas than the `min.insync.replicas` configured for their topic. This metric provides insights about brokers that host the follower replicas. Those followers are not keeping up with the leader. Reasons for this could include being (or having been) offline, and over-throttled interbroker replication. An alert is raised when this value is greater than zero, providing information on the under-replicated partitions for each broker.

`AbnormalControllerState`

Indicates whether the current broker is the controller for the cluster. The metric can be 0 or 1. During the life of a cluster, only one broker should be the controller and the cluster always needs to have an active controller. Having two or more brokers saying that they are controllers indicates a problem. If the condition persists, an alert is raised when the sum of all the values for this metric on all brokers is not equal to 1, meaning that there is no active controller (the sum is 0) or more than one controller (the sum is greater than 1).

`UnderMinIsrPartitionCount`

Indicates that the minimum number of in-sync replicas (ISRs) for a lead Kafka broker, specified using `min.insync.replicas`, that must acknowledge a write operation has not been reached. The metric defines the number of partitions that the broker leads for which the in-sync replicas count is less than the minimum in-sync. An alert is raised when this value is greater than zero, providing information on the partition count for each broker that did not achieve the minimum number of acknowledgments.

`OfflineLogDirectoryCount`

Indicates the number of log directories which are offline (for example, due to a hardware failure) so that the broker cannot store incoming messages anymore. An alert is raised when this value is greater than zero, providing information on the number of offline log directories for each broker.

`KafkaRunningOutOfSpace`

Indicates the remaining amount of disk space that can be used for writing data. An alert is raised when this value is lower than 5GiB, providing information on the disk that is running out of space for each persistent volume claim. The threshold value may be changed in `prometheus-rules.yaml`.

ZooKeeper alerting rules

`AvgRequestLatency`

Indicates the amount of time it takes for the server to respond to a client request. An alert is raised when this value is greater than 10 (ticks), providing the actual value of the average request latency for each server.

`OutstandingRequests`

Indicates the number of queued requests in the server. This value goes up when the server receives more requests than it can process. An alert is raised when this value is greater than 10, providing the actual number of outstanding requests for each server.

`ZookeeperRunningOutOfSpace`

Indicates the remaining amount of disk space that can be used for writing data to ZooKeeper. An alert is raised when this value is lower than 5GiB., providing information on the disk that is running out of space for each persistent volume claim.

Kafka Exporter alerting rules

If you perform the steps to introduce metrics to your deployment, you will already have your Kafka cluster configured to use the alert notification rules that support Kafka Exporter.

`UnderReplicatedPartition`

An alert to warn that a topic is under-replicated and the broker is not replicating to enough partitions. The default configuration is for an alert if there are one or more under-replicated partitions for a topic. The alert might signify that a Kafka instance is down or the Kafka cluster is overloaded. A planned restart of the Kafka broker may be required to restart the replication process.

`TooLargeConsumerGroupLag`

An alert to warn that the lag on a consumer group is too large for a specific topic partition. The default configuration is 1000 records. A large lag might indicate that consumers are too slow and are falling behind the producers.

`NoMessageForTooLong`

An alert to warn that a topic has not received messages for a period of time. The default configuration for the time period is 10 minutes. The delay might be a result of a configuration issue preventing a producer from publishing messages to the topic.

6.2.3. Example Grafana dashboards

If you deploy Prometheus to provide metrics, you can use Strimzi's example Grafana dashboards 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 5. 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`

6.3. 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

6.3.1. Deploying Prometheus metrics configuration

Deploy Prometheus metrics configuration to use Prometheus with Strimzi.

You can use your own configuration or Strimzi's example custom resource configuration files.

- `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: my-config-map
          key: my-key
---
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`.

Enabling metrics for Kafka Bridge

```
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.

Additional resources

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

6.3.2. 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 apply -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
   ```

6.3.3. 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
```

6.3.4. 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.

6.4. 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. Point a web browser to `http://localhost:3000`.

   The Grafana Log In page appears.

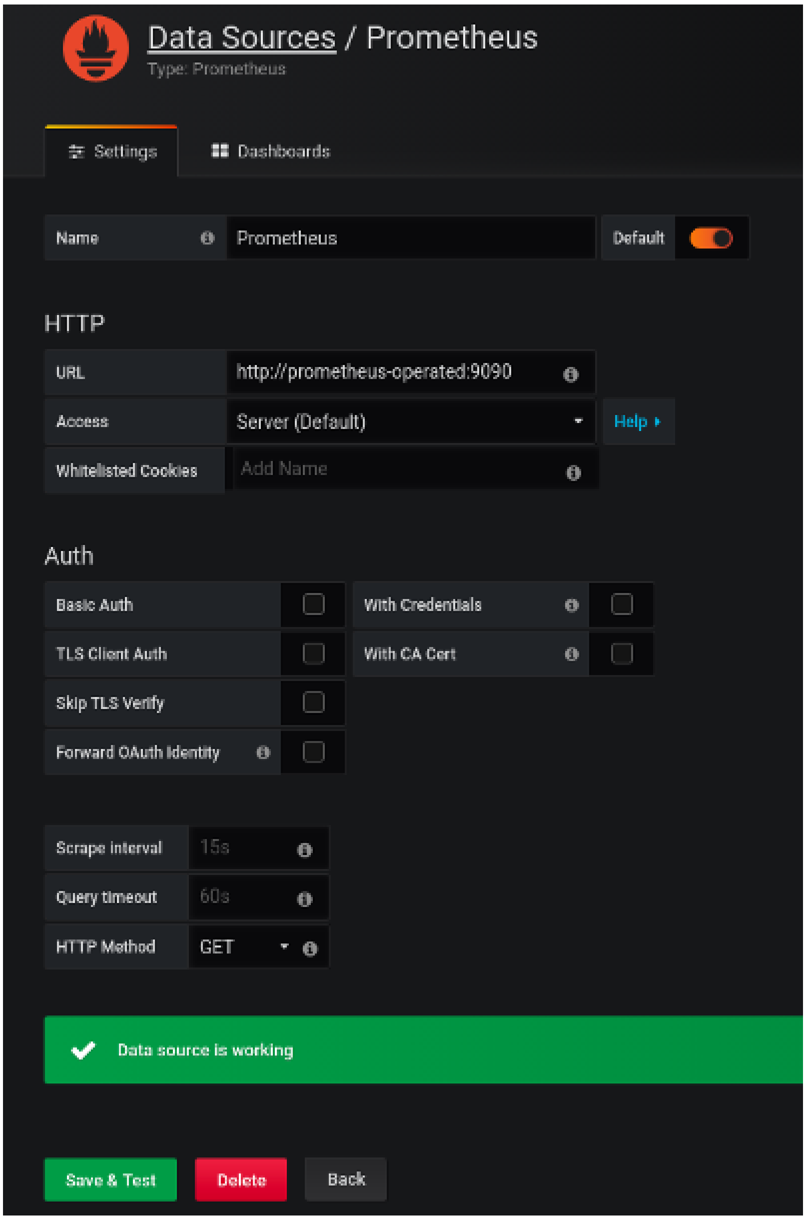
5. Enter your user name and password, and then click **Log In**.

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

6. 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. From **Dashboards › Import**, upload the example dashboards or paste the JSON directly.
8. On the top header, click the dashboard drop-down menu, and then select the dashboard you want to view.

   When the Prometheus server has been collecting metrics for a Strimzi cluster for some time, the dashboards are populated.

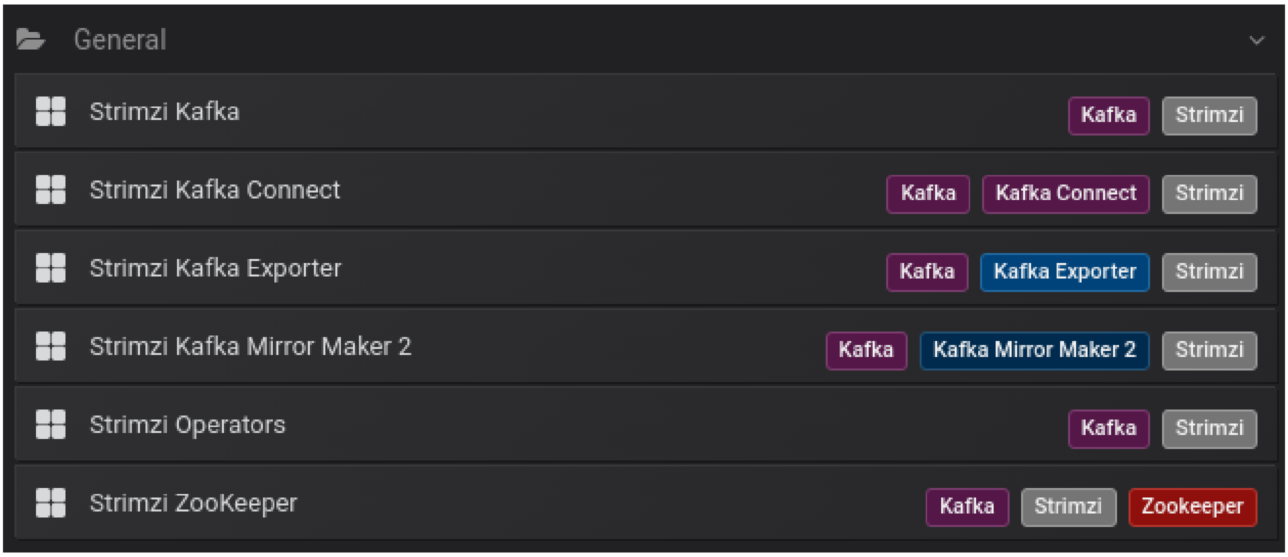
   

   Figure 1. Dashboard selection options

Chapter 7. Upgrading Strimzi

Strimzi can be upgraded to version 0.26.0 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.

Upgrade paths

Two upgrade paths are possible:

Incremental

Upgrading Strimzi from the previous minor version to version 0.26.0.

Multi-version

Upgrading Strimzi from an old version to version 0.26.0 within a single upgrade (skipping one or more intermediate versions).

For example, upgrading from Strimzi 0.20.0 directly to Strimzi 0.22.

Upgrading from a version earlier than 0.22

The `v1beta2` API version for all custom resources was introduced with Strimzi 0.22. For Strimzi 0.23 and newer, the `v1alpha1` and `v1beta1` API versions were removed from all Strimzi custom resources apart from `KafkaTopic` and `KafkaUser`.

If you are upgrading from a Strimzi version prior to version 0.22:

1. Upgrade Strimzi to 0.22
2. Convert the custom resources to `v1beta2`
3. Upgrade Strimzi to 0.23 or newer

NOTE As an alternative, you can install the custom resources from version 0.22, convert the resources, and then upgrade to 0.23 or newer.

Kafka version support

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.

Decide which Kafka version to upgrade to before beginning the Strimzi upgrade process.

NOTE	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.

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.

7.1. 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 0.26.0 requires Kubernetes 1.16 and later.

   - Upgrading Kubernetes with minimal downtime

2. When upgrading from 0.22 or earlier, update existing custom resources to support the `v1beta2` API version.

   - Strimzi custom resource upgrades

3. Update your Cluster Operator to a new Strimzi version.

   The approach you take depends on how you deployed the Cluster Operator.

   - 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.
   - If you deployed the Cluster Operator from OperatorHub.io, use the Operator Lifecycle Manager (OLM) to change the update channel for the Strimzi Operators to a new Strimzi version.

     Depending on your chosen upgrade strategy, after updating the channel, either:

     - An automatic upgrade is initiated
     - A manual upgrade will require approval before the installation begins

       For more information on using OperatorHub.io to upgrade Operators, see the Operator Lifecycle Manager documentation.

◦ 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 GitHub or find them in the `crd` subdirectory inside the Helm Chart.

4. Upgrade all Kafka brokers and client applications to the latest supported Kafka version.
   ◦ Upgrading Kafka
   ◦ Strategies for upgrading clients

Optional: incremental cooperative rebalance upgrade

Consider upgrading consumers and Kafka Streams applications to use the *incremental cooperative rebalance* protocol for partition rebalances.

- Upgrading consumers to cooperative rebalancing

7.2. Upgrading Kubernetes with minimal downtime

If you are upgrading Kubernetes, refer the Kubernertes 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: 1
  config:
    # ...
    default.replication.factor: 3
    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.

7.2.1. 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
# ...
```

7.2.2. 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
- Replicating topics for high availability
- PodDisruptionBudgetTemplate schema reference
- Performing a rolling update using a pod annotation

7.3. Strimzi custom resource upgrades

When upgrading Strimzi to 0.26.0 from 0.22 or earlier, you must ensure that your custom resources are using API version `v1beta2`. You must upgrade the Custom Resource Definitions and the custom resources **before** upgrading to Strimzi 0.23 or newer. To perform the upgrade, you can use the *API* *conversion tool* provided with Strimzi 0.24. For more information, see the Strimzi 0.24.0 upgrade documentation.

7.4. Upgrading the Cluster Operator

This procedure describes how to upgrade a Cluster Operator deployment to use Strimzi 0.26.0.

Follow this procedure if you deployed the Cluster Operator using the installation YAML files rather than OperatorHub.io.

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 0.26.0.

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 0.26.0.
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 a "Version not found" error message. Otherwise, no error message is returned.

   For example:

   ```
   "Version 2.4.0 is not supported. Supported versions are: 2.6.0, 2.6.1, 2.7.0."
   ```

   - 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:0.26.0-kafka-3.0.0
   ```

Your Cluster Operator was upgraded to version 0.26.0 but the version of Kafka running in the cluster it manages is unchanged.

Following the Cluster Operator upgrade, you must perform a Kafka upgrade.

7.5. Upgrading Kafka

After you have upgraded your Cluster Operator to 0.26.0, 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.

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.

Additional resources

- Upgrading the Cluster Operator
- Kafka versions

7.5.1. 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:

Kafka version	Interbroker protocol version	Log message format version	ZooKeeper version
2.8.0	2.8	2.8	3.5.9
2.8.1	2.8	2.8	3.5.9
3.0.0	3.0	3.0	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 format they were encoded with. You can configure a Kafka broker to convert messages from newer format versions to a given older format version before the broker appends the message to the log.

In Kafka, there are two different methods for setting the message format version:

- The `message.format.version` property is set on topics.
- The `log.message.format.version` property is set on Kafka brokers.

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.

The upgrade tasks in this section assume that the message format version is defined by the `log.message.format.version`.

7.5.2. 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.

There are a number of strategies you can use to upgrade your clients:

Consumers first

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.

Per-topic consumers first

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.

Per-topic consumers first, 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.

Other strategies for upgrading client applications are also possible.

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.

7.5.3. 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.

7.5.4. Upgrading Kafka brokers and client applications

This procedure describes how to upgrade a Strimzi Kafka cluster to the latest supported Kafka version.

Compared to your current Kafka version, the new version might support a higher *log message format version* or *inter-broker protocol version*, or both. Follow the steps to upgrade these versions, if required. For more information, see Kafka versions.

You should also choose a strategy for upgrading clients. Kafka clients are upgraded in step 6 of this procedure.

Prerequisites

For the `Kafka` resource to be upgraded, check that:

- The Cluster Operator, which supports both versions of Kafka, is up and running.
- The `Kafka.spec.kafka.config` does *not* contain 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, ensure that `Kafka.spec.kafka.config` has the `log.message.format.version` and `inter.broker.protocol.version` set to the defaults for the *current* Kafka version.

   For example, if upgrading from Kafka version 2.8.0 to 3.0.0:

```
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.8.0
    config:
      log.message.format.version: "2.8"
      inter.broker.protocol.version: "2.8"
      # ...
```

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 ensures that the brokers can continue to communicate with each other throughout the upgrade.

For example, if upgrading from Kafka 2.8.0 to 3.0.0:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.0.0 ①
    config:
      log.message.format.version: "2.8" ②
      inter.broker.protocol.version: "2.8" 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 3.0.0:

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   spec:
     # ...
     kafka:
       version: 3.0.0
       config:
         log.message.format.version: "2.8"
         inter.broker.protocol.version: "3.0"
         # ...
   ```

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 3.0.0:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.0.0
    config:
      log.message.format.version: "3.0"
      inter.broker.protocol.version: "3.0"
      # ...
```

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

7.6. 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 3.0.0.

Procedure

To upgrade a Kafka consumer to use the incremental cooperative rebalance protocol:

1. Replace the Kafka clients `.jar` file with the new version.
2. In the consumer configuration, append `cooperative-sticky` to the `partition.assignment.strategy`. For example, if the `range` strategy is set, change the

configuration to `range, cooperative-sticky`.

3. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.
4. Reconfigure each consumer in the group by removing the earlier `partition.assignment.strategy` from the consumer configuration, leaving only the `cooperative-sticky` strategy.
5. Restart each consumer in the group in turn, waiting for the consumer to rejoin the group after each restart.

To upgrade a Kafka Streams application to use the incremental cooperative rebalance protocol:

1. Replace the Kafka Streams `.jar` file with the new version.
2. In the Kafka Streams configuration, set the `upgrade.from` configuration parameter to the Kafka version you are upgrading from (for example, 2.3).
3. Restart each of the stream processors (nodes) in turn.
4. Remove the `upgrade.from` configuration parameter from the Kafka Streams configuration.
5. Restart each consumer in the group in turn.

Additional resources

- Notable changes in 2.4.0 in the Apache Kafka documentation.

Chapter 8. Downgrading Strimzi

If you are encountering issues with the version of Strimzi you upgraded to, you can revert your installation to the previous version.

You can perform a downgrade to:

1. Revert your Cluster Operator to the previous Strimzi version.
   - Downgrading the Cluster Operator to a previous version
2. Downgrade all Kafka brokers and client applications to the previous Kafka version.
   - 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.

8.1. 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.

8.2. Downgrading Kafka

Kafka version downgrades are performed by the Cluster Operator.

8.2.1. 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 2.8.0 because the `log.message.format.version` has not been changed:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.0.0
    config:
      log.message.format.version: "2.8"
      # ...
```

The downgrade would not be possible if the `log.message.format.version` was set at `"3.0"` or a value was absent (so that the parameter took the default value for a 3.0.0 broker of 3.0).

8.2.2. Downgrading Kafka brokers and client applications

This procedure describes how you can downgrade a Strimzi Kafka cluster to a lower (previous) version of Kafka, such as downgrading from 3.0.0 to 2.8.0.

Prerequisites

For the `Kafka` resource to be downgraded, check:

- IMPORTANT: Compatibility of Kafka versions.
- The Cluster Operator, which supports both versions of Kafka, is up and running.
- The `Kafka.spec.kafka.config` does not contain options that are not supported by the Kafka version being downgraded to.
- The `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.

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 3.0.0 to 2.8.0:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.8.0 ①
    config:
      log.message.format.version: "2.8" ②
      inter.broker.protocol.version: "2.8" ③
      # ...
```

① Kafka version is changed to the previous version.

② Message format version is unchanged.

③ Inter-broker protocol version is unchanged.

NOTE You must format the value of `log.message.format.version` and `inter.broker.protocol.version` as a string to prevent it from being interpreted as a floating point number.

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:0.26.0-kafka-3.0.0
--rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server
localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog
--delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic
__strimzi_store_topic --delete
```

Additional resources

- Topic Operator topic store