Strimzi Documentation (0.8.0)

Table of Contents

1. Overview of Strimzi

Strimzi makes it easy to run Apache Kafka on OpenShift or Kubernetes. Apache Kafka is a popular platform for streaming data delivery and processing. For more information about Apache Kafka, see the Apache Kafka website.

Strimzi is based on Apache Kafka 2.0.0 and consists of three main components:

Cluster Operator
Responsible for deploying and managing Apache Kafka clusters within OpenShift or Kubernetes cluster.

Topic Operator
Responsible for managing Kafka topics within a Kafka cluster running within OpenShift or Kubernetes cluster.

User Operator
Responsible for managing Kafka users within a Kafka cluster running within OpenShift or Kubernetes cluster.

This guide describes how to install and use Strimzi.

1.1. Kafka Key Features

- Scalability and performance

- Designed for horizontal scalability
- Message ordering guarantee
  - At partition level
- Message rewind/replay
  - "Long term" storage
  - Allows to reconstruct application state by replaying the messages
  - Combined with compacted topics allows to use Kafka as key-value store

1.2. Document Conventions

Replaceables
In this document, replaceable text is styled in monospace and italics.

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

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

2. Getting started with Strimzi

Strimzi works on all types of clusters, from public and private clouds on to local deployments intended for development. This guide expects that an OpenShift or Kubernetes cluster is available and the `kubectl` and `oc` command-line tools are installed and configured to connect to the running cluster.

Table 1. Supported Versions

Product	Version
Kubernetes	1.9 and later
OpenShift Origin	3.9 and later
Apache Kafka	2.0.0

When no existing OpenShift or Kubernetes cluster is available, `Minikube` or `Minishift` can be used to create a local cluster. More details can be found in Installing Kubernetes and OpenShift clusters.

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

2.1. Strimzi downloads

Strimzi releases are available to download from GitHub. The release artefacts contain documentation, installation, and example `.yaml` files for deployment on OpenShift or Kubernetes. The installation, and example files are used throughout this documentation. Additionally, a Helm Chart is provided for deploying the Cluster Operator using Helm. The container images are available through the Docker Hub.

2.2. Cluster Operator

Strimzi uses the Cluster Operator to deploy and manage Kafka (including Zookeeper) and Kafka Connect clusters. The Cluster Operator is deployed inside of the Kubernetes or OpenShift cluster. To deploy a Kafka cluster, a `Kafka` resource with the cluster configuration has to be created within the Kubernetes or OpenShift cluster. Based on what is declared inside of the `Kafka` resource, the Cluster Operator deploys a corresponding Kafka cluster. For more information about the different configuration options supported by the `Kafka` resource, see Kafka cluster configuration

Note	Strimzi contains example YAML files, which make deploying a Cluster Operator easier.

2.2.1. Overview of the Cluster Operator component

The Cluster Operator is in charge of deploying a Kafka cluster alongside a Zookeeper ensemble. As part of the Kafka cluster, it can also deploy the topic operator which provides operator-style topic management via `KafkaTopic` custom resources. The Cluster Operator is also able to deploy a Kafka

Connect cluster which connects to an existing Kafka cluster. On OpenShift such a cluster can be deployed using the Source2Image feature, providing an easy way of including more connectors.

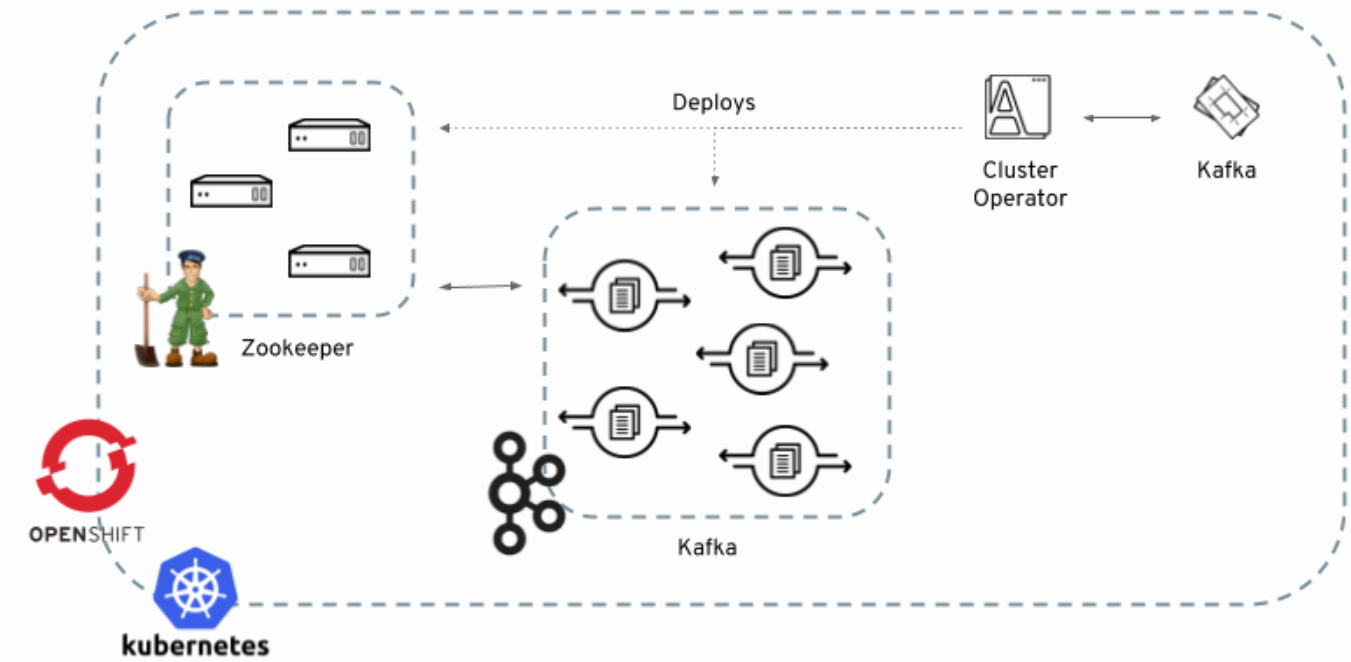


Figure 1. Example Architecture diagram of the Cluster Operator.

When the Cluster Operator is up, it starts to *watch* for certain OpenShift or Kubernetes resources containing the desired Kafka or Kafka Connect cluster configuration. By default, it watches only in the same namespace or project where it is installed. The Cluster Operator can be configured to watch for more OpenShift projects or Kubernetes namespaces. Cluster Operator watches the following resources:

- A `Kafka` resource for the Kafka cluster.
- A `KafkaConnect` resource for the Kafka Connect cluster.
- A `KafkaConnectS2I` resource for the Kafka Connect cluster with Source2Image support.

When a new `Kafka`, `KafkaConnect`, or `KafkaConnectS2I` resource is created in the OpenShift or Kubernetes cluster, the operator gets the cluster description from the desired resource and starts creating a new Kafka or Kafka Connect cluster by creating the necessary other OpenShift or Kubernetes resources, such as StatefulSets, Services, ConfigMaps, and so on.

Every time the desired resource is updated by the user, the operator performs corresponding updates on the OpenShift or Kubernetes resources which make up the Kafka or Kafka Connect cluster. Resources are either patched or deleted and then re-created in order to make the Kafka or Kafka Connect cluster reflect the state of the desired cluster resource. This might cause a rolling update which might lead to service disruption.

Finally, when the desired resource is deleted, the operator starts to undeploy the cluster and delete all the related OpenShift or Kubernetes resources.

2.2.2. Deploying the Cluster Operator to Kubernetes

Prerequisites

- Modify the installation files according to the namespace the Cluster Operator is going to be installed in.

  On Linux, use:

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

  On MacOS, use:

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

Procedure

1. Deploy the Cluster Operator

   ```
   kubectl apply -f install/cluster-operator -n _my-namespace_
   ```

2.2.3. Deploying the Cluster Operator to OpenShift

Prerequisites

- A user with `cluster-admin` role needs to be used, for example, `system:admin`.
- Modify the installation files according to the namespace the Cluster Operator is going to be installed in.

  On Linux, use:

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

On MacOS, use:

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

Procedure

1. Deploy the Cluster Operator

```
oc apply -f install/cluster-operator -n _my-project_
oc apply -f examples/templates/cluster-operator -n _my-project_
```

2.2.4. Deploying the Cluster Operator to watch multiple namespaces

Prerequisites

- Edit the installation files according to the OpenShift project or Kubernetes namespace the Cluster Operator is going to be installed in.

  On Linux, use:

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

  On MacOS, use:

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

Procedure

1. Edit the file `install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml` and in the environment variable `STRIMZI_NAMESPACE` list all the OpenShift projects or Kubernetes namespaces where Cluster Operator should watch for resources. For example:

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
      - name: strimzi-cluster-operator
        image: strimzi/cluster-operator:latest
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: myproject,myproject2,myproject3
```

2. For all namespaces or projects which should be watched by the Cluster Operator, install the `RoleBindings`. Replace the *`my-namespace`* or *`my-project`* with the OpenShift project or Kubernetes namespace used in the previous step.

   On Kubernetes this can be done using `kubectl apply`:

```
kubectl apply -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -n my-namespace
kubectl apply -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-operator-delegation.yaml -n my-namespace
kubectl apply -f install/cluster-operator/032-RoleBinding-strimzi-cluster-operator-topic-operator-delegation.yaml -n my-namespace
```

   On OpenShift this can be done using `oc apply`:

```
oc apply -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -n my-project
oc apply -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-operator-delegation.yaml -n my-project
oc apply -f install/cluster-operator/032-RoleBinding-strimzi-cluster-operator-topic-operator-delegation.yaml -n my-project
```

3. Deploy the Cluster Operator

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

2.2.5. Deploying the Cluster Operator using Helm Chart

Prerequisites

- Helm client has to be installed on the local machine.
- Helm has to be installed in the OpenShift or Kubernetes cluster.

Procedure

1. Add the Strimzi Helm Chart repository:

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

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

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

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

```
helm ls
```

Additional resources

- For more information about Helm, see the Helm website.

2.3. Kafka cluster

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

Strimzi provides two options for Kafka cluster deployment:

Ephemeral
is suitable only for development and testing purposes and not for production. This deployment uses `emptyDir` volumes for storing broker information (Zookeeper) and topics or partitions (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
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 HostPath volumes on Minikube or 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.

2.3.1. Deploying the Kafka cluster to Kubernetes

Prerequisites

- Before deploying a Kafka cluster, the Cluster Operator must be deployed.

Procedure

1. If you are planning to use the Kafka broker for development or testing, create an ephemeral cluster

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

2. If you are planning to use the Kafka cluster in production, create a persistent cluster

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator
- For more information about the different configuration options supported by the `Kafka` resource, see Kafka cluster configuration

2.3.2. Deploying the Kafka cluster to OpenShift

OpenShift provides a template for deploying the Kafka cluster either in the OpenShift console or on the command-line.

Prerequisites

- Before deploying a Kafka cluster, the Cluster Operator must be deployed.

Procedure

1. If you are planning to use the Kafka cluster for development or testing, create an ephemeral cluster

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

2. If you are planning to use the Kafka cluster in production, create a persistent cluster

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator For more information about the different configuration options supported by the `Kafka` resource, see Kafka cluster configuration

2.4. Kafka Connect

The Cluster Operator deploys a Kafka Connect cluster, which can be used with your Kafka broker deployment. It is implemented as a `Deployment` with a configurable number of workers. The default image currently contains only the `FileStreamSinkConnector` and `FileStreamSourceConnector` connectors. The REST interface for managing the Kafka Connect cluster is exposed internally within the OpenShift or Kubernetes cluster as a `kafka-connect` service on port `8083`.

Example `KafkaConnect` resources and the details about the `KafkaConnect` format for deploying Kafka Connect can be found in Kafka Connect cluster configuration and Kafka Connect cluster with Source2Image support.

2.4.1. Deploying Kafka Connect to Kubernetes

Prerequisites

- Before deploying Kafka Connect, the Cluster Operator must be deployed.

Procedure

- Deploy Kafka Connect on Kubernetes by creating the corresponding `KafkaConnect` resource.

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator

2.4.2. Deploying Kafka Connect to OpenShift

On OpenShift, Kafka Connect is provided in the form of a template. It can be deployed from the template using the command-line or through the OpenShift console.

Prerequisites

- Before deploying Kafka Connect, the Cluster Operator must be deployed.

Procedure

- Create a Kafka Connect cluster from the command-line:

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator

2.4.3. Using Kafka Connect with plugins

Strimzi container images for Kafka Connect contain, by default, only the `FileStreamSinkConnector` and `FileStreamSourceConnector` connectors which are part of Apache Kafka.

To facilitate deployment with 3rd party connectors, Kafka Connect is configured to automatically load all plugins or connectors that are present in the `/opt/kafka/plugins` directory during startup.

There are two ways of adding custom plugins into this directory:

- Using a custom Docker image
- Using the OpenShift build system with the Strimzi S2I

Create a new image based on our base image

Strimzi provides its own Docker image for running Kafka Connect, which can be found on Docker Hub as `strimzi/kafka-connect:0.8.0`. This image can be used as a base image for building a new custom image with additional plugins.

The following procedure describes the process for creating such a custom image.

Procedure

1. Create a new `Dockerfile` using `strimzi/kafka-connect:0.8.0` as the base image:

```
FROM strimzi/kafka-connect:0.8.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER kafka:kafka
```

2. Build the container image and upload it to the appropriate container image repository.
3. Set the `KafkaConnect.spec.image` property of the KafkaConnect custom resource or the `STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE` variable to point to the new container image.

Additional resources

- For more information about the `STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE` variable, see Cluster Operator Configuration.
- For more information about the `KafkaConnect.spec.image property`, see Container images.

Using OpenShift builds and S2I to create new images

OpenShift supports builds, which can be used together with the Source-to-Image (S2I) framework to create new container images. An OpenShift build takes a builder image with S2I support together with source code and binaries provided by the user and uses them to build a new container image. The newly created container image is stored in OpenShift's local container image repository and can be used in deployments. Strimzi provides a Kafka Connect builder image, which can be found on Docker Hub as `strimzi/kafka-connect-s2i:0.8.0` with this S2I support. It takes user-provided binaries (with plugins and connectors) and creates a new Kafka Connect image. This enhanced Kafka Connect image can be used with the Kafka Connect deployment.

The S2I deployment provided as an OpenShift template. It can be deployed from the template using the command-line or the OpenShift console.

Procedure

1. Create a Kafka Connect S2I cluster from the command-line

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

2. Once the cluster is deployed, a new build can be triggered from the command-line by creating a directory with Kafka Connect plugins:

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

```
    ├── protobuf-java-2.6.1.jar
    └── README.md
```

3. Start a new image build using the prepared directory:

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

Note	The name of the build will be changed according to the cluster name of the deployed Kafka Connect cluster.

4. Once the build is finished, the new image will be used automatically by the Kafka Connect deployment.

2.5. Kafka Mirror Maker

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

For information about example resources and the format for deploying Kafka Mirror Maker, see Kafka Mirror Maker configuration.

2.5.1. Deploying Kafka Connect to Kubernetes

Prerequisites

- Before deploying Kafka Mirror Maker, the Cluster Operator must be deployed.

Procedure

- Deploy Kafka Mirror Maker on Kubernetes by creating the corresponding `KafkaMirrorMaker` resource.

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator

2.5.2. Deploying Kafka Mirror Maker to OpenShift

On OpenShift, Kafka Mirror Maker is provided in the form of a template. It can be deployed from the template using the command-line or through the OpenShift console.

Prerequisites

- Before deploying Kafka Mirror Maker, the Cluster Operator must be deployed.

Procedure

- Create a Kafka Mirror Maker cluster from the command-line:

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator

2.6. Deploying example clients

Prerequisites

- An existing Kafka cluster for the client to connect to.

Procedure

1. Deploy the producer.

```
oc run kafka-producer -ti --image=strimzi/kafka:0.8.0 --restart=Never \-- bin/kafka-console-producer.sh --broker-list cluster-n
```

2. Type your message into the console where the producer is running.

3. Press Enter to send the message.

4. Deploy the consumer.

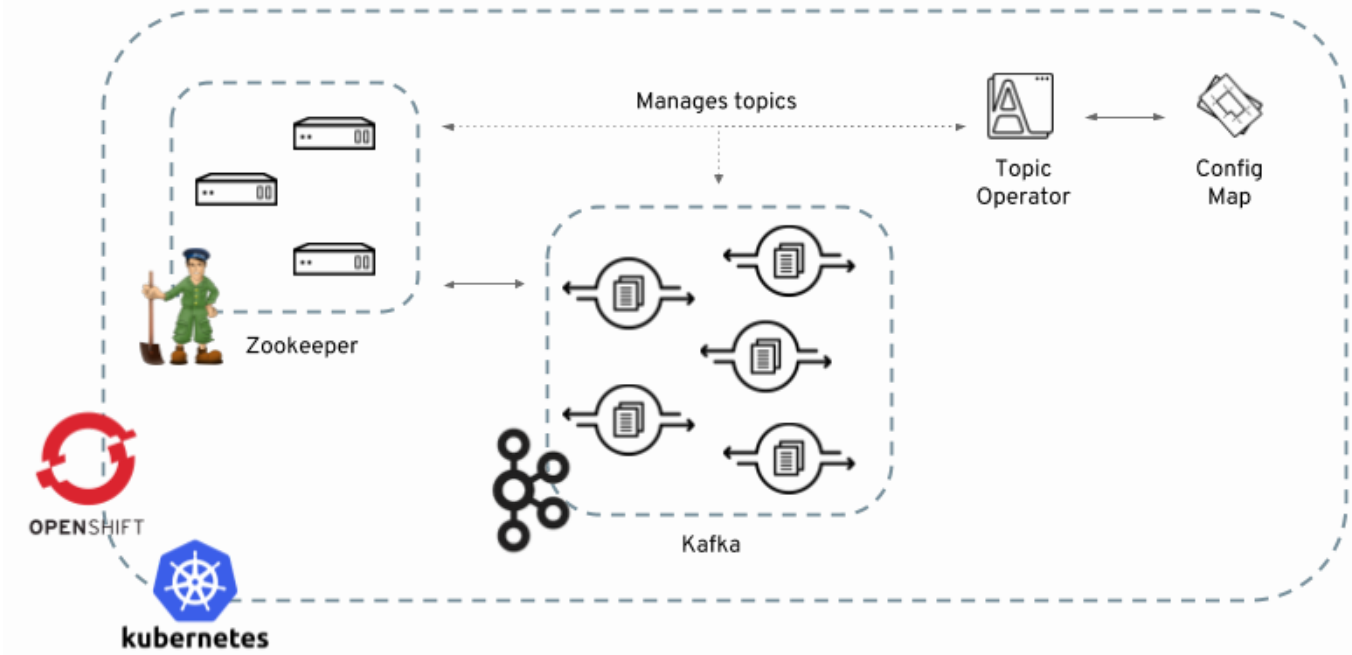
```
oc run kafka-consumer -ti --image=strimzi/kafka:0.8.0 --restart=Never \-- bin/kafka-console-consumer.sh --bootstrap-server clust
```

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

2.7. Topic Operator

2.7.1. Overview of the Topic Operator component

The Topic Operator provides a way of managing topics in a Kafka cluster via OpenShift or Kubernetes resources.



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

Specifically:

- if a `KafkaTopic` is created, the operator will create the topic it describes
- if a `KafkaTopic` is deleted, the operator will delete the topic it describes
- if a `KafkaTopic` is changed, the operator will update the topic it describes

And also, in the other direction:

- if a topic is created within the Kafka cluster, the operator will create a `KafkaTopic` describing it
- if a topic is deleted from the Kafka cluster, the operator will create the `KafkaTopic` describing it
- if a topic in the Kafka cluster is changed, the operator will update the `KafkaTopic` describing it

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

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

For more details about creating, modifying and deleting topics, see Using the Topic Operator.

2.7.2. Deploying the Topic Operator using the Cluster Operator

Prerequisites

- A running Cluster Operator
- A `Kafka` resource to be created or updated

Procedure

1. Topic Operator can be included in the Entity Operator. Edit the `Kafka` resource ensuring it has a `Kafka.spec.entityOperator` object that configures the Entity Operator.
2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about deploying the Entity Operator, see Entity Operator.

- For more information about the `Kafka.spec.entityOperator` object used to configure the Topic Operator when deployed by the Cluster Operator, see `EntityOperatorSpec` schema reference.

2.8. User Operator

The User Operator provides a way of managing Kafka users via OpenShift or Kubernetes resources.

2.8.1. Overview of the User Operator component

The User Operator manages Kafka users for a Kafka cluster by watching for `KafkaUser` OpenShift or Kubernetes 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 will create the user it describes
- if a `KafkaUser` is deleted, the User Operator will delete the user it describes
- if a `KafkaUser` is changed, the User Operator will update the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the OpenShift or Kubernetes resources. Unlike the Kafka topics which might be created by applications directly in Kafka, it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator, so this should not be needed.

The User Operator allows you to declare a `KafkaUser` as part of your application's deployment. When the user is created, the credentials will be 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 rights in the `KafkaUser` declaration.

2.8.2. Deploying the User Operator using the Cluster Operator

Prerequisites

- A running Cluster Operator
- A `Kafka` resource to be created or updated.

Procedure

1. Edit the `Kafka` resource ensuring it has a `Kafka.spec.entityOperator.userOperator` object that configures the User Operator how you want.
2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about the `Kafka.spec.entityOperator` object used to configure the User Operator when deployed by the Cluster Operator, see `EntityOperatorSpec` schema reference.

3. Deployment configuration

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

- Kafka clusters
- Kafka Connect clusters
- Kafka Connect clusters with *Source2Image* support
- Kafka Mirror Maker

3.1. Kafka cluster configuration

The full schema of the `Kafka` resource is described in the `Kafka` schema reference. All labels that are applied to the desired `Kafka` resource will also be applied to the OpenShift or Kubernetes resources making up the Kafka cluster. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.1.1. Kafka and Zookeeper storage

Kafka brokers and Zookeeper are stateful applications. They need to store data on disks. Strimzi allows you to configure the type of storage, which they want to use for Kafka and Zookeeper. Storage configuration is mandatory and has to be specified in every `Kafka` resource.

Storage can be configured using the `storage` property in following resources:

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`

Strimzi supports two types of storage:

- Ephemeral
- Persistent

The type of storage is specified in the `type` field.

Important	Once the Kafka cluster is deployed, the storage cannot be changed.

Ephemeral storage

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

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

An example of Ephemeral storage

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: ephemeral
    # ...
  zookeeper:
    # ...
    storage:
      type: ephemeral
    # ...
```

Persistent storage

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

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

`size` (required)
Defines the size of the persistent volume claim, for example, "1000Gi".

`class` (optional)
The OpenShift or Kubernetes Storage Class to use for dynamic volume provisioning.

`selector` (optional)
Allows selecting a specific persistent volume to use. It contains a `matchLabels` field which contains key:value pairs representing labels for selecting such a volume.

`delete-claim` (optional)
Boolean value which specifies if the Persistent Volume Claim has to be deleted when the cluster is undeployed. Default is `false`.

Warning	Resizing persistent storage for existing Strimzi clusters is not currently supported. You must decide the necessary storage size before deploying the cluster.

Example fragment of persistent storage configuration with 1000Gi `size`

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

The following example demonstrates the use of a storage class.

Example fragment of persistent storage configuration with specific Storage Class

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

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

Example fragment of persistent storage configuration with selector

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

Persistent Volume Claim naming

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

`data-cluster-name-kafka-idx`

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

`data-cluster-name-zookeeper-idx`

Persistent Volume Claim for the volume used for storing data for the Zookeeper node pod `idx`.

Additional resources

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

3.1.2. Replicas

Kafka cluster can run with many brokers and Kafka brokers can run with various numbers of nodes. The number of brokers used for the Kafka cluster is defined in the Kafka resource. The best number of brokers for your cluster has to be determined based on your specific use case.

Configuring the number of broker nodes

Number of Kafka broker nodes is configured using the `replicas` property in `Kafka.spec.kafka`.

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
```

```
  kafka:
    # ...
    replicas: 3
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.3. Kafka broker configuration

Strimzi allows you to customize the configuration of Apache Kafka brokers. You can specify and configure most of the options listed in Apache Kafka documentation.

The only options which cannot be configured are those related to the following areas:

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

These options are automatically configured by Strimzi.

Kafka broker configuration

Kafka broker can be configured using the `config` property in `Kafka.spec.kafka`.

This property should contain the Kafka broker configuration options as keys. The values could be in one of the following JSON types:

- String
- Number
- Boolean

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

- `listeners`
- `advertised.`
- `broker.`
- `listener.`
- `host.name`
- `port`
- `inter.broker.listener.name`
- `sasl.`
- `ssl.`
- `security.`
- `password.`
- `principal.builder.class`
- `log.dir`

- `zookeeper.connect`
- `zookeeper.set.acl`
- `authorizer.`
- `super.user`

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

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

An example showing Kafka broker configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    config:
      num.partitions: 1
      num.recovery.threads.per.data.dir: 1
      default.replication.factor: 3
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 1
      log.retention.hours: 168
      log.segment.bytes: 1073741824
      log.retention.check.interval.ms: 300000
      num.network.threads: 3
      num.io.threads: 8
      socket.send.buffer.bytes: 102400
      socket.receive.buffer.bytes: 102400
      socket.request.max.bytes: 104857600
      group.initial.rebalance.delay.ms: 0
    # ...
```

Configuring Kafka brokers

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `config` property in the `Kafka` resource specifying the cluster deployment. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    config:
      default.replication.factor: 3
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 1
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

3.1.4. Kafka broker listeners

Strimzi allows users to configure the listeners which will be enabled in Kafka brokers. Two types of listeners are supported:

- Plain listener on port 9092 (without encryption)
- TLS listener on port 9093 (with encryption)

Mutual TLS authentication for clients

Mutual TLS authentication

Mutual authentication or two-way authentication is when both the server and the client present certificates. Strimzi can configure Kafka to use TLS (Transport Layer Security) to provide encrypted communication between Kafka brokers and clients either with or without mutual authentication. When you configure mutual authentication, the broker authenticates the client and the client authenticates the broker. Mutual TLS authentication is always used for the communication between Kafka brokers and Zookeeper pods.

Note	In many common uses of TLS (such as the HTTPS protocol used between a web browser and a web server) the authentication is not mutual: Only one party to the communication gets proof of the identity of the other party.

TLS authentication is more commonly one-way, where only one party authenticates to another. For example, when the HTTPS protocol is used between a web browser and a web server, the authentication is not usually mutual and only the server gets proof of the identity of the browser.

When to use mutual TLS authentication for clients

Mutual TLS authentication is recommended for authenticating Kafka clients when:

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

SCRAM-SHA authentication

SCRAM (Salted Challenge Response Authentication Mechanism) is an authentication protocol that can establish mutual authentication using passwords. Strimzi can configure Kafka to use SASL SCRMA-SHA-512 to provide authentication on both unencrypted and TLS-encrypted client connections. TLS authentication is always used internally between Kafka brokers and Zookeeper nodes. When used with a TLS client connection, the TLS protocol provides encryption, but is not used for authentication.

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

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

Supported SCRAM credentials

Strimzi supports SCRMA-SHA-512 only. When a `KafkaUser.spec.authentication.type` is configured with `scram-sha-512` the User Operator will generate a random 12 character password consisting of upper and lowercase ASCII letters and numbers.

When to use SCRAM-SHA authentication for clients

SCRAM-SHA is recommended for authenticating Kafka clients when:

- The client supports authentication using SCRAM-SHA-512
- It is necessary to use passwords rather than the TLS certificates
- When you want to have authentication for unencrypted communication

Kafka listeners

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

- `plain`
- `tls`
- `external`

When none of these properties are defined, the listener will be disabled.

An example of `listeners` property with all listeners enabled

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

An example of `listeners` property with only the plain listener enabled

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

External listener

The external listener is used to connect to a Kafka cluster from outside of an OpenShift or Kubernetes environment. Strimzi supports three types of external listeners:

- `route`
- `loadbalancer`
- `nodeport`

Exposing Kafka using OpenShift Routes
An external listener of type `route` exposes Kafka by using OpenShift `Routes` and the HAProxy router. A dedicated `Route` is created for every Kafka broker pod. An additional `Route` is created to serve as a Kafka bootstrap address. Kafka clients can use these `Routes` to connect to Kafka on port 443.

Note	`Routes` are available only on OpenShift. External listeners of type `route` cannot be used on Kubernetes.

When exposing Kafka using OpenShift `Routes`, TLS encryption is always used.

For more information on using `Routes` to access Kafka, see Accessing Kafka using OpenShift routes.

Exposing Kafka using loadbalancers
External listeners of type `loadbalancer` expose Kafka by using `Loadbalancer` type `Services`. A new loadbalancer service is created for every Kafka broker pod. An additional loadbalancer is created to serve as a Kafka *bootstrap* address. Loadbalancers listen to connections on port 9094.

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

For more information on using loadbalancers to access Kafka, see Accessing Kafka using loadbalancers routes.

Exposing Kafka using node ports
External listeners of type `nodeport` expose Kafka by using `NodePort` type `Services`. When exposing Kafka in this way, Kafka clients connect directly to the nodes of OpenShift or Kubernetes. You must enable access to the ports on the OpenShift or Kubernetes nodes for each client (for example, in firewalls or security groups). Each Kafka broker pod is then accessible on a separate port. Additional `NodePort` type `Service` is created to serve as a Kafka bootstrap address.

When configuring the advertised addresses for the Kafka broker pods, Strimzi uses the address of the node on which the given pod is running. When selecting the node address, the different address types are used with the following priority:

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

5. InternalIP

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

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

For more information on using node ports to access Kafka, see Accessing Kafka using node ports routes.

Listener authentication

The listener sub-properties can also contain additional configuration. Both listeners support the `authentication` property. This is used to specify an authentication mechanism specific to that listener:

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

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

An example where the plain listener is configured for SCRAM-SHA authentication and the `tls` listener with mutual TLS authentication

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

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

Configuring Kafka listeners

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `listeners` property in the `Kafka.spec.kafka` resource.

An example configuration of the plain (unencrypted) listener without authentication:

+

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      plain: {}
    # ...
  zookeeper:
    # ...
```

1. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

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

Accessing Kafka using OpenShift routes

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type `route`.

   An example configuration with an external listener configured to use `Routes`:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       listeners:
         external:
           type: route
           # ...
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

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

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

   ```
   oc get routes _cluster-name_-kafka-bootstrap -o=jsonpath='{.status.ingress[0].host}{'
   ```

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

4. Extract the public certificate of the broker certification authority

   ```
   oc extract secret/_cluster-name_-cluster-ca-cert --keys=ca.crt --to=- > ca.crt
   ```

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

Additional resources

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

Accessing Kafka using loadbalancers routes

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type `loadbalancer`.

   An example configuration with an external listener configured to use loadbalancers:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       listeners:
         external:
           type: loadbalancer
           tls: true
           # ...
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3. Find the hostname of the bootstrap loadbalancer.

   On Kubernetes this can be done using `kubectl get`:

   ```
   kubectl get service cluster-name-kafka-external-bootstrap -o=jsonpath='{.status.loadB
   ```

   On OpenShift this can be done using `oc get`:

   ```
   oc get service cluster-name-kafka-external-bootstrap -o=jsonpath='{.status.loadBalanc
   ```

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

   On Kubernetes this can be done using `kubectl get`:

   ```
   kubectl get service cluster-name-kafka-external-bootstrap -o=jsonpath='{.status.loadB
   ```

   On OpenShift this can be done using `oc get`:

   ```
   oc get service cluster-name-kafka-external-bootstrap -o=jsonpath='{.status.loadBalanc
   ```

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

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

   On Kubernetes this can be done using `kubectl get`:

   ```
   kubectl get secret cluster-name-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base6
   ```

   On OpenShift this can be done using `oc extract`:

   ```
   oc extract secret/cluster-name-cluster-ca-cert --keys=ca.crt --to=- > ca.crt
   ```

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

Additional resources

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

Accessing Kafka using node ports routes

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type `nodeport`.

   An example configuration with an external listener configured to use node ports:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       listeners:
         external:
           type: nodeport
           tls: true
   ```

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

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3. Find the port number of the bootstrap service.

   On Kubernetes this can be done using `kubectl get`:

   ```
   kubectl get service cluster-name-kafka-external-bootstrap -o=jsonpath='{.spec.ports[0
   ```

   On OpenShift this can be done using `oc get`:

   ```
   oc get service cluster-name-kafka-external-bootstrap -o=jsonpath='{.spec.ports[0].nod
   ```

   The port should be used in the Kafka *bootstrap* address.

4. Find the address of the OpenShift or Kubernetes node.

   On Kubernetes this can be done using `kubectl get`:

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

   On OpenShift this can be done using `oc get`:

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

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

   a. ExternalDNS

   b. ExternalIP

   c. Hostname

   d. InternalDNS

   e. InternalIP

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

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

   On Kubernetes this can be done using `kubectl get`:

   ```
   kubectl get secret cluster-name-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base6
   ```

   On OpenShift this can be done using `oc extract`:

   ```
   oc extract secret/cluster-name-cluster-ca-cert --keys=ca.crt --to=- > ca.crt
   ```

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

Additional resources

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

3.1.5. Authentication and Authorization

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

Authentication

Authentication is configured as part of the listener configuration in the `authentication` property. When the `authentication` property is missing, no authentication will be enabled on given listener. The authentication mechanism which will be used is defined by the `type` field.

The supported authentication mechanisms are:

- TLS client authentication
- SASL SCRAM-SHA-512

TLS client authentication

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

An example of `authentication` with type `tls`

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

Configuring authentication in Kafka brokers

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `listeners` property in the `Kafka.spec.kafka` resource. Add the `authentication` field to the listeners where you want to enable authentication. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       listeners:
         tls:
           authentication:
             type: tls
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

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

Authorization

Authorization can be configured using the `authorization` property in the `Kafka.spec.kafka` resource. When the `authorization` property is missing, no authorization will be enabled. When authorization is enabled it will be applied for all enabled listeners. The authorization method is defined by the `type` field.

Currently, the only supported authorization method is the Simple authorization.

Simple authorization

Simple authorization is using the `SimpleAclAuthorizer` plugin. `SimpleAclAuthorizer` is the default authorization plugin which is part of Apache Kafka. To enable simple authorization, the `type` field should be set to `simple`.

An example of Simple authorization

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

Configuring authorization in Kafka brokers

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    authorization:
      type: simple
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the supported authorization methods, see authorization reference.
- For more information about the schema for `Kafka`, see `Kafka` schema reference.

3.1.6. Replicas

Zookeeper clusters or ensembles usually run with an odd number of nodes and always requires the majority of the nodes to be available in order to maintain a quorum. Maintaining a quorum is important because when the Zookeeper cluster loses a quorum, it will stop responding to clients. As a result, a Zookeeper cluster without a quorum will cause the Kafka brokers to stop working as well. This is why having a stable and highly available Zookeeper cluster is very important for Strimzi.

A Zookeeper cluster is usually deployed with three, five, or seven nodes.

Three nodes
Zookeeper cluster consisting of three nodes requires at least two nodes to be up and running in order to maintain the quorum. It can tolerate only one node being unavailable.

Five nodes
Zookeeper cluster consisting of five nodes requires at least three nodes to be up and running in order to maintain the quorum. It can tolerate two nodes being unavailable.

Seven nodes
Zookeeper cluster consisting of seven nodes requires at least four nodes to be up and running in order to maintain the quorum. It can tolerate three nodes being unavailable.

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

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

decide for the number of nodes to use.

Number of Zookeeper nodes

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

An example showing replicas configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    replicas: 3
    # ...
```

Changing number of replicas

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
     zookeeper:
       # ...
       replicas: 3
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.7. Zookeeper configuration

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

The only options which cannot be configured are those related to the following areas:

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

These options are automatically configured by Strimzi.

Zookeeper configuration

Zookeeper nodes can be configured using the `config` property in `Kafka.spec.zookeeper`. This property should contain the Zookeeper configuration options as keys. The values could be in one of the following JSON types:

- String
- Number
- Boolean

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

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

When one of the forbidden options is present in the `config` property, it will be ignored and a warning message will be printed to the Custer Operator log file. All other options will be passed to Zookeeper.

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

Selected options have default values:

- `timeTick` with default value `2000`
- `initLimit` with default value `5`
- `syncLimit` with default value `2`
- `autopurge.purgeInterval` with default value `1`

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

An example showing Zookeeper configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
    # ...
```

Configuring Zookeeper

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `config` property in the `Kafka` resource specifying the cluster deployment. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
```

```
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.8. Entity Operator

The Entity Operator is responsible for managing different entities in a running Kafka cluster. The currently supported entities are:

Kafka topics
managed by the Topic Operator.

Kafka users
managed by the User Operator

Both Topic and User Operators can be deployed on their own. But the easiest way to deploy them is together with the Kafka cluster as part of the Entity Operator. The Entity Operator can include either one or both of them depending on the configuration. They will be automatically configured to manage the topics and users of the Kafka cluster with which they are deployed.

For more information about Topic Operator, see Topic Operator. For more information about how to use Topic Operator to create or delete topics, see Using the Topic Operator.

Configuration

The Entity Operator can be configured using the `entityOperator` property in `Kafka.spec`

The `entityOperator` property supports several sub-properties:

- `tlsSidecar`
- `affinity`
- `tolerations`
- `topicOperator`
- `userOperator`

The `tlsSidecar` property can be used to configure the TLS sidecar container which is used to communicate with Zookeeper. For more details about configuring the TLS sidecar, see TLS sidecar.

The `affinity` and `tolerations` properties can be used to configure how OpenShift or Kubernetes schedules the Entity Operator pod. For more details about pod scheduling, see Configuring pod scheduling.

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

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

Example of basic configuration enabling both operators

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

When both `topicOperator` and `userOperator` properties are missing, the Entity Operator will be not deployed.

Topic Operator

Topic Operator deployment can be configured using additional options inside the `topicOperator` object. Following options are supported:

`watchedNamespace`
The OpenShift or Kubernetes namespace in which the topic operator watches for `KafkaTopics`. Default is the namespace where the Kafka cluster is deployed.

`reconciliationIntervalSeconds`
The interval between periodic reconciliations in seconds. Default is 90.

`zookeeperSessionTimeoutSeconds`
The Zookeeper session timeout in seconds. Default is 20 seconds.

`topicMetadataMaxAttempts`
The number of attempts for getting topics metadata from Kafka. The time between each attempt is defined as an exponential back-off. You might want to increase this value when topic creation could take more time due to its many partitions or replicas. Default is `6`.

`image`
The `image` property can be used to configure the container image which will be used. For more details about configuring custom container images, see Container images.

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

`logging`
The `logging` property configures the logging of the Topic Operator For more details about logging configuration, see Logging.

Example of Topic Operator configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

User Operator

User Operator deployment can be configured using additional options inside the `userOperator` object. Following options are supported:

`watchedNamespace`
The OpenShift or Kubernetes namespace in which the topic operator watches for `KafkaUsers`. Default is the namespace where the Kafka cluster is deployed.

`reconciliationIntervalSeconds`
The interval between periodic reconciliations in seconds. Default is 120.

`zookeeperSessionTimeoutSeconds`
The Zookeeper session timeout in seconds. Default is 6 seconds.

`image`
The `image` property can be used to configure the container image which will be used. For more details about configuring custom container images, see Container images.

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

`logging`

The `logging` property configures the logging of the User Operator. For more details about logging configuration, see Logging.

Example of Topic Operator configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-user-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

Configuring Entity Operator

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
    userOperator:
      watchedNamespace: my-user-namespace
      reconciliationIntervalSeconds: 60
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.9. CPU and memory resources

For every deployed container, Strimzi allows you to specify the resources which should be reserved for it and the maximum resources that can be consumed by it. Strimzi supports two types of resources:

- Memory
- CPU

Strimzi is using the OpenShift or Kubernetes syntax for specifying CPU and memory resources.

Resource limits and requests

Resource limits and requests can be configured using the `resources` property in following resources:

- `Kafka.spec.kafka`
- `Kafka.spec.kafka.tlsSidecar`

- `Kafka.spec.zookeeper`
- `Kafka.spec.zookeeper.tlsSidecar`
- `Kafka.spec.entityOperator.topicOperator`
- `Kafka.spec.entityOperator.userOperator`
- `Kafka.spec.entityOperator.tlsSidecar`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

Resource requests

Requests specify the resources that will be reserved for a given container. Reserving the resources will ensure that they are always available.

Important	If the resource request is for more than the available free resources in the OpenShift or Kubernetes cluster, the pod will not be scheduled.

Resource requests can be specified in the `request` property. The resource requests currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource request configuration

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

It is also possible to specify a resource request just for one of the resources:

An example showing resource request configuration with memory request only

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

Or:

An example showing resource request configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

Resource limits

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

Resource limits can be specified in the `limits` property. The resource limits currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource limits configuration

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

It is also possible to specify the resource limit just for one of the resources:

An example showing resource limit configuration with memory request only

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

Or:

An example showing resource limits configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

Supported CPU formats

CPU requests and limits are supported in the following formats:

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

An example of using different CPU units

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

Note	The amount of computing power of 1 CPU core might differ depending on the platform where the OpenShift or Kubernetes is deployed.

For more details about the CPU specification, see the Meaning of CPU website.

Supported memory formats

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

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

An example of using different memory units

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

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

Additional resources

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

Configuring resource requests and limits

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the schema, see `Resources` schema reference.

3.1.10. Logging

Logging enables you to diagnose error and performance issues of Strimzi. For the logging, various logger implementations are used. Kafka and Zookeeper use `log4j` logger and Topic Operator, User Operator, and other components use `log4j2` logger.

This section provides information about different loggers and describes how to configure log levels.

You can set the log levels by specifying the loggers and their levels directly (inline) or by using a custom (external) config map.

Using inline logging setting

Procedure

1. Edit the YAML file to specify the loggers and their level for the required components. For example:

   ```
   apiVersion: {KafkaApiVersion}
   kind: Kafka
   spec:
     kafka:
       # ...
       logging:
         type: inline
         loggers:
           logger.name: "INFO"
       # ...
   ```

   In the above example, the log level is set to INFO. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF. For more information about the log levels, see link: log4j manual.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using external ConfigMap for logging setting

Procedure

1. Edit the YAML file to specify the name of the `ConfigMap` which should be used for the required components. For example:

```
apiVersion: {KafkaApiVersion}
kind: Kafka
spec:
  kafka:
    # ...
    logging:
      type: external
      name: customConfigMap
    # ...
```

   Remember to place your custom ConfigMap under `log4j.properties` eventually `log4j2.properties` key.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Loggers

Strimzi consists of several components. Each component has its own loggers and is configurable. This section provides information about loggers of various components.

Components and their loggers are listed below.

- Kafka
  - `kafka.root.logger.level`
  - `log4j.logger.org.I0Itec.zkclient.ZkClient`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.kafka`
  - `log4j.logger.org.apache.kafka`
  - `log4j.logger.kafka.request.logger`
  - `log4j.logger.kafka.network.Processor`
  - `log4j.logger.kafka.server.KafkaApis`
  - `log4j.logger.kafka.network.RequestChannel$`
  - `log4j.logger.kafka.controller`
  - `log4j.logger.kafka.log.LogCleaner`
  - `log4j.logger.state.change.logger`
  - `log4j.logger.kafka.authorizer.logger`
- Zookeeper
  - `zookeeper.root.logger`
- Kafka Connect and Kafka Connect with Source2Image support
  - `connect.root.logger.level`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.org.I0Itec.zkclient`
  - `log4j.logger.org.reflections`
- Kafka Mirror Maker
  - `mirrormaker.root.logger`

- Topic Operator
  - `rootLogger.level`
- User Operator
  - `rootLogger.level`

3.1.11. Kafka rack awareness

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

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

Configuring rack awareness in Kafka brokers

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

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    rack:
      topologyKey: failure-domain.beta.kubernetes.io/zone
    # ...
```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional Resources

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

3.1.12. Healthchecks

Healthchecks are periodical tests which verify that the application's health. When the Healthcheck fails, OpenShift or Kubernetes can assume that the application is not healthy and attempt to fix it. OpenShift or Kubernetes supports two types of Healthcheck probes:

- Liveness probes
- Readiness probes

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

Users can configure selected options for liveness and readiness probes

Healthcheck configurations

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

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

Both `livenessProbe` and `readinessProbe` support two additional options:

- `initialDelaySeconds`
- `timeoutSeconds`

The `initialDelaySeconds` property defines the initial delay before the probe is tried for the first time. Default is 15 seconds.

The `timeoutSeconds` property defines timeout of the probe. Default is 5 seconds.

An example of liveness and readiness probe configuration

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

Configuring healthchecks

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       readinessProbe:
         initialDelaySeconds: 15
         timeoutSeconds: 5
       livenessProbe:
         initialDelaySeconds: 15
         timeoutSeconds: 5
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.13. Prometheus metrics

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

For more information about configuring Prometheus and Grafana, see Metrics.

Metrics configuration

Prometheus metrics can be enabled by configuring the `metrics` property in following resources:

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

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

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

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

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

Configuring Prometheus metrics

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
```

```
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.14. JVM Options

Apache Kafka and Apache Zookeeper are running inside of a Java Virtual Machine (JVM). JVM has many configuration options to optimize the performance for different platforms and architectures. Strimzi allows configuring some of these options.

JVM configuration

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

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

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

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

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

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

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

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

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

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

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

Example fragment configuring `-Xmx` and `-Xms`

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

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

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

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

Example fragment configuring `-server`

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

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

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

Example showing the use of the `-XX` object

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

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

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+Explicit
```

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

Configuring JVM options

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jvmOptions:
      "-Xmx": "8g"
      "-Xms": "8g"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.15. Container images

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

Container image configurations

Container image which should be used for given components can be specified using the `image` property in:

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

The `image` specified in the component-specific custom resource will be used during deployment. If the `image` field is missing, the `image` specified in the Cluster Operator configuration will be used. If the `image` name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka brokers:
  1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka:latest` container image.
- For Kafka broker TLS sidecar:
  1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka-stunnel:latest` container image.

- For Zookeeper nodes:
    1. Container image specified in the `STRIMZI_DEFAULT_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/zookeeper:latest` container image.
- For Zookeeper node TLS sidecar:
    1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/zookeeper-stunnel:latest` container image.
- For Topic Operator:
    1. Container image specified in the `STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration. .** `strimzi/topic-operator:latest` container image.
- For User Operator:
    1. Container image specified in the `STRIMZI_DEFAULT_USER_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/user-operator:latest` container image.
- For Entity Operator TLS sidecar:
    1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/entity-operator-stunnel:latest` container image.
- For Kafka Connect:
    1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/kafka-connect:latest` container image.
- For Kafka Connect with Source2image support:
    1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/kafka-connect-s2i:latest` container image.

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

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

Configuring container images

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.16. TLS sidecar

Sidecar is a container which is running in a pod and serves an auxiliary purpose. The purpose of the TLS sidecar is to encrypt or decrypt the communication between Strimzi components and Zookeeper since Zookeeper does not support TLS encryption natively. Zookeeper does not support TLS encryption natively. Therefore Strimzi uses the sidecar to add the TLS support.

The TLS sidecar is currrently being used in:

- Kafka brokers
- Zookeeper
- Entity Operator

TLS sidecar configuration

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`

The TLS sidecar supports three additional options:

- `image`
- `resources`
- `logLevel`

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

The `image` property can be used to configure the container image which will be used. For more details about configuring custom container images, see Container images.

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

- emerg
- alert
- crit
- err
- warning
- notice

- info
- debug

The default value is *notice*.

Example of TLS sidecar configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    tlsSidecar:
      image: my-org/my-image:latest
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
      logLevel: debug
    # ...
  zookeeper:
    # ...
```

Configuring TLS sidecar

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       tlsSidecar:
         resources:
           requests:
             cpu: 200m
             memory: 64Mi
           limits:
             cpu: 500m
             memory: 128Mi
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.17. Configuring pod scheduling

Important	When two application are scheduled to the same OpenShift or Kubernetes node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids

	sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

Scheduling pods based on other applications

Avoid critical applications to share the node

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Scheduling pods to specific nodes

Node scheduling

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

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring node affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Label the nodes where Strimzi components should be scheduled.

   On Kubernetes this can be done using `kubectl label`:

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

   On OpenShift this can be done using `oc label`:

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

   Alternatively, some of the existing labels might be reused.

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
               - matchExpressions:
                 - key: node-type
                   operator: In
                   values:
                   - fast-network
       # ...
     zookeeper:
       # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using dedicated nodes

Dedicated nodes

Cluster administrators can mark selected OpenShift or Kubernetes nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Tolerations

Tolerations ca be configured using the `tolerations` property in following resources:

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The format of the `tolerations` property follows the OpenShift or Kubernetes specification. For more details, see the Kubernetes taints and tolerations.

Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated
2. Make sure there are no workloads scheduled on these nodes
3. Set the taints on the selected nodes

   On Kubernetes this can be done using `kubectl taint`:

```
kubectl taint node your-node dedicated=Kafka:NoSchedule
```

On OpenShift this can be done using `oc adm taint`:

```
oc adm taint node your-node dedicated=Kafka:NoSchedule
```

4. Additionally, add a label to the selected nodes as well.

   On Kubernetes this can be done using `kubectl label`:

   ```
   kubectl label node your-node dedicated=Kafka
   ```

   On OpenShift this can be done using `oc label`:

   ```
   oc label node your-node dedicated=Kafka
   ```

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       tolerations:
         - key: "dedicated"
           operator: "Equal"
           value: "Kafka"
           effect: "NoSchedule"
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
             - matchExpressions:
               - key: dedicated
                 operator: In
                 values:
                 - Kafka
       # ...
     zookeeper:
       # ...
   ```

6. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.1.18. Performing a rolling update of a Kafka cluster

This procedure describes how to manually trigger a rolling update of an existing Kafka cluster by using an OpenShift or Kubernetes annotation.

Prerequisites

- A running Kafka cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the `StatefulSet` that controls the Kafka pods you want to manually update.

   For example, if your Kafka cluster is named *my-cluster*, the corresponding `StatefulSet` is named *my-cluster-kafka*.

2. Annotate a `StatefulSet` resource in OpenShift or Kubernetes.

   On Kubernetes, use `kubectl annotate`:

   ```
   kubectl annotate statefulset cluster-name-kafka operator.strimzi.io/manual-rolling-up
   ```

On OpenShift, use `oc annotate`:

```
oc annotate statefulset cluster-name-kafka operator.strimzi.io/manual-rolling-update=
```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of all pods within the annotated `StatefulSet` is triggered, as long as the annotation was detected by the reconciliation process. Once the rolling update of all the pods is complete, the annotation is removed from the `StatefulSet`.

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about deploying the Kafka cluster, see Deploying the Kafka cluster to OpenShift and Deploying the Kafka cluster to Kubernetes.

3.1.19. Performing a rolling update of a Zookeeper cluster

This procedure describes how to manually trigger a rolling update of an existing Zookeeper cluster by using an OpenShift or Kubernetes annotation.

Prerequisites

- A running Zookeeper cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the `StatefulSet` that controls the Zookeeper pods you want to manually update.

   For example, if your Kafka cluster is named *my-cluster*, the corresponding `StatefulSet` is named *my-cluster-zookeeper*.

2. Annotate a `StatefulSet` resource in OpenShift or Kubernetes.

   On Kubernetes, use `kubectl annotate`:

   ```
   kubectl annotate statefulset cluster-name-zookeeper operator.strimzi.io/manual-rollir
   ```

   On OpenShift, use `oc annotate`:

   ```
   oc annotate statefulset cluster-name-zookeeper operator.strimzi.io/manual-rolling-upc
   ```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of all pods within the annotated `StatefulSet` is triggered, as long as the annotation was detected by the reconciliation process. Once the rolling update of all the pods is complete, the annotation is removed from the `StatefulSet`.

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about deploying the Zookeeper cluster, see Deploying the Kafka cluster to OpenShift.

3.1.20. Deleting Kafka nodes manually

This procedure describes how to delete an existing Kafka node by using an OpenShift or Kubernetes annotation. Deleting a Kafka node consists of deleting both the `Pod` on which the Kafka broker is running and the related `PersistentVolumeClaim` (if the cluster was deployed with persistent storage). After deletion, the `Pod` and its related `PersistentVolumeClaim` are recreated automatically.

Warning	Deleting a `PersistentVolumeClaim` can cause permanent data loss. The following procedure should only be performed if you have encountered storage issues.

Prerequisites

- A running Kafka cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the `Pod` that you want to delete.

For example, if the cluster is named *cluster-name*, the pods are named *cluster-name*-kafka-*index*, where *index* starts at zero and ends at the total number of replicas.

1. Annotate the `Pod` resource in OpenShift or Kubernetes.

   On Kubernetes use `kubectl annotate`:

   ```
   kubectl annotate pod cluster-name-kafka-index operator.strimzi.io/delete-pod-and-pvc=
   ```

   On OpenShift use `oc annotate`:

   ```
   oc annotate pod cluster-name-kafka-index operator.strimzi.io/delete-pod-and-pvc=true
   ```

2. Wait for the next reconciliation, when the annotated pod with the underlying persistent volume claim will be deleted and then recreated.

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about deploying the Kafka cluster, see Deploying the Kafka cluster to OpenShift and Deploying the Kafka cluster to Kubernetes.

3.1.21. Deleting Zookeeper nodes manually

This procedure describes how to delete an existing Zookeeper node by using an OpenShift or Kubernetes annotation. Deleting a Zookeeper node consists of deleting both the `Pod` on which Zookeeper is running and the related `PersistentVolumeClaim` (if the cluster was deployed with persistent storage). After deletion, the `Pod` and its related `PersistentVolumeClaim` are recreated automatically.

Warning	Deleting a `PersistentVolumeClaim` can cause permanent data loss. The following procedure should only be performed if you have encountered storage issues.

Prerequisites

- A running Zookeeper cluster.
- A running Cluster Operator.

Procedure

1. Find the name of the `Pod` that you want to delete.

For example, if the cluster is named *cluster-name*, the pods are named *cluster-name*-zookeeper-*index*, where *index* starts at zero and ends at the total number of replicas.

1. Annotate the `Pod` resource in OpenShift or Kubernetes.

   On Kubernetes use `kubectl annotate`:

   ```
   kubectl annotate pod cluster-name-zookeeper-index operator.strimzi.io/delete-pod-and-
   ```

   On OpenShift use `oc annotate`:

   ```
   oc annotate pod cluster-name-zookeeper-index operator.strimzi.io/delete-pod-and-pvc=t
   ```

2. Wait for the next reconciliation, when the annotated pod with the underlying persistent volume claim will be deleted and then recreated.

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about deploying the Zookeeper cluster, see Deploying the Kafka cluster to OpenShift and Deploying the Kafka cluster to Kubernetes.

3.1.22. List of resources created as part of Kafka cluster

The following resources will created by the Cluster Operator in the OpenShift or Kubernetes cluster:

`cluster-name-kafka`
StatefulSet which is in charge of managing the Kafka broker pods.

`cluster-name-kafka-brokers`

Service needed to have DNS resolve the Kafka broker pods IP addresses directly.

`cluster-name-kafka-bootstrap`
Service can be used as bootstrap servers for Kafka clients.

`cluster-name-kafka-external-bootstrap`
Bootstrap service for clients connecting from outside of the OpenShift or Kubernetes cluster. This resource will be created only when external listener is enabled.

`cluster-name-kafka-pod-id`
Service used to route traffic from outside of the OpenShift or Kubernetes cluster to individual pods. This resource will be created only when external listener is enabled.

`cluster-name-kafka-external-bootstrap`
Bootstrap route for clients connecting from outside of the OpenShift or Kubernetes cluster. This resource will be created only when external listener is enabled and set to type `route`.

`cluster-name-kafka-pod-id`
Route for traffic from outside of the OpenShift or Kubernetes cluster to individual pods. This resource will be created only when external listener is enabled and set to type `route`.

`cluster-name-kafka-config`
ConfigMap which contains the Kafka ancillary configuration and is mounted as a volume by the Kafka broker pods.

`cluster-name-kafka-brokers`
Secret with Kafka broker keys.

`cluster-name-kafka`
Service account used by the Kafka brokers.

`strimzi-namespace-name-cluster-name-kafka-init`
Cluster role binding used by the Kafka brokers.

`cluster-name-zookeeper`
StatefulSet which is in charge of managing the Zookeeper node pods.

`cluster-name-zookeeper-nodes`
Service needed to have DNS resolve the Zookeeper pods IP addresses directly.

`cluster-name-zookeeper-client`
Service used by Kafka brokers to connect to Zookeeper nodes as clients.

`cluster-name-zookeeper-config`
ConfigMap which contains the Zookeeper ancillary configuration and is mounted as a volume by the Zookeeper node pods.

`cluster-name-zookeeper-nodes`
Secret with Zookeeper node keys.

`cluster-name-entity-operator`
Deployment with Topic and User Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

`cluster-name-entity-topic-operator-config`
Configmap with ancillary configuration for Topic Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

`cluster-name-entity-user-operator-config`
Configmap with ancillary configuration for User Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

`cluster-name-entity-operator-certs`
Secret with Entitiy operators keys for communication with Kafka and Zookeeper. This resource will be created only if Cluster Operator deployed Entity Operator.

`cluster-name-entity-operator`
Service account used by the Entity Operator.

`strimzi-cluster-name-topic-operator`
Role binding used by the Entity Operator.

`strimzi-cluster-name-user-operator`
Role binding used by the Entity Operator.

`cluster-name-cluster-ca`

Secret with the Cluster CA used to encrypt the cluster communication.

`cluster-name-cluster-ca-cert`
Secret with the Cluster CA public key. This key can be used to verify the identity of the Kafka brokers.

`cluster-name-clients-ca`
Secret with the Clients CA used to encrypt the communication between Kafka brokers and Kafka clients.

`cluster-name-clients-ca-cert`
Secret with the Clients CA public key. This key can be used to verify the identity of the Kafka brokers.

`data-cluster-name-kafka-idx`
Persistent Volume Claim for the volume used for storing data for the Kafka broker pod `idx`. This resource will be created only if persistent storage is selected for provisioning persistent volumes to store data.

`data-cluster-name-zookeeper-idx`
Persistent Volume Claim for the volume used for storing data for the Zookeeper node pod `idx`. This resource will be created only if persistent storage is selected for provisioning persistent volumes to store data.

3.2. Kafka Connect cluster configuration

The full schema of the `KafkaConnect` resource is described in the `KafkaConnect` schema reference. All labels that are applied to the desired `KafkaConnect` resource will also be applied to the OpenShift or Kubernetes resources making up the Kafka Connect cluster. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.2.1. Replicas

Kafka Connect clusters can run with a different number of nodes. The number of nodes is defined in the `KafkaConnect` and `KafkaConnectS2I` resources. Running Kafka Connect cluster with multiple nodes can provide better availability and scalability. However, when running Kafka Connect on OpenShift or Kubernetes it is not absolutely necessary to run multiple nodes of Kafka Connect for high availability. When the node where Kafka Connect is deployed to crashes, OpenShift or Kubernetes will automatically take care of rescheduling the Kafka Connect pod to a different node. However, running Kafka Connect with multiple nodes can provide faster failover times, because the other nodes will be already up and running.

Configuring the number of nodes

Number of Kafka Connect nodes can be configured using the `replicas` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`.

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `replicas` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  replicas: 3
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.2. Bootstrap servers

Kafka Connect cluster always works together with a Kafka cluster. The Kafka cluster is specified in the form of a list of bootstrap servers. On OpenShift or Kubernetes, the list must ideally contain the Kafka cluster bootstrap service which is named `cluster-name-kafka-bootstrap` and a port of 9092 for plain traffic or 9093 for encrypted traffic.

The list of bootstrap servers can be configured in the `bootstrapServers` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. The servers should be a comma-separated list containing one or more Kafka brokers or a service pointing to Kafka brokers specified as a `hostname:_port_` pairs.

When using Kafka Connect with a Kafka cluster not managed by Strimzi, you can specify the bootstrap servers list according to the configuration of a given cluster.

Configuring bootstrap servers

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `bootstrapServers` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.3. Connecting to Kafka brokers using TLS

By default, Kafka Connect will try to connect to Kafka brokers using a plain text connection. If you would prefer to use TLS additional configuration will be necessary.

TLS support in Kafka Connect

TLS support is configured in the `tls` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. The `tls` property contains a list of secrets with key names under which the certificates are stored. The certificates should be stored in X509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-secret
        certificate: ca.crt
      - secretName: my-other-secret
        certificate: certificate.crt
  # ...
```

When multiple certificates are stored in the same secret, it can be listed multiple times.

An example showing TLS configuration with multiple certificates from the same secret

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-secret
        certificate: ca.crt
      - secretName: my-secret
        certificate: ca2.crt
  # ...
```

Configuring TLS in Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Find out the name of the secret with the certificate which should be used for TLS Server Authentication and the key under which the certificate is stored in the secret. If such secret does not exist yet, prepare the certificate in a file and create the secret.

   On Kubernetes this can be done using `kubectl create`:

   ```
   kubectl create secret generic my-secret --from-file=my-file.crt
   ```

   On OpenShift this can be done using `oc create`:

   ```
   oc create secret generic my-secret --from-file=my-file.crt
   ```

2. Edit the `tls` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnect
   metadata:
     name: my-connect
   spec:
     # ...
     tls:
       trustedCertificates:
         - secretName: my-cluster-cluster-cert
           certificate: ca.crt
     # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.4. Connecting to Kafka brokers with Authentication

By default, Kafka Connect will try to connect to Kafka brokers without any authentication. Authentication can be enabled in the `KafkaConnect` and `KafkaConnectS2I` resources.

Authentication support in Kafka Connect

Authentication can be configured in the `authentication` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. The `authentication` property specifies the type of the authentication mechanisms which should be used and additional configuration details depending on the mechanism. The currently supported authentication types are:

- TLS client authentication
- SASL based authentication using SCRAM-SHA-512 mechanism

TLS Client Authentication

To use the TLS client authentication, set the `type` property to the value `tls`. TLS client authentication is using TLS certificate to authenticate. The certificate has to be specified in the `certificateAndKey` property. It is always loaded from an OpenShift or Kubernetes secret. Inside the secret, it has to be stored in the X509 format under two different keys: for public and private keys.

Note	TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Connect see Connecting to Kafka brokers using TLS.

An example showing TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  # ...
```

SCRAM-SHA-512 authentication

To use the authentication using the SCRAM-SHA-512 SASL mechanism, set the `type` property to the value `scram-sha-512`. SCRAM-SHA-512 uses a username and password to authenticate. Specify the username in the `username` property. Specify the password as a link to a `Secret` containing the password in the `passwordSecret` property. It has to specify the name of the `Secret` containing the password and the name of the key under which the password is stored inside the `Secret`.

An example showing SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: password
  # ...
```

Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Find out the name of the `Secret` with the public and private keys which should be used for TLS Client Authentication and the keys under which they are stored in the `Secret`. If such a `Secret` does not exist yet, prepare the keys in a file and create the `Secret`.

   On Kubernetes this can be done using `kubectl create`:

   ```
   kubectl create secret generic my-secret --from-file=my-public.crt --from-file=my-priv
   ```

   On OpenShift this can be done using `oc create`:

   ```
   oc create secret generic my-secret --from-file=my-public.crt --from-file=my-private.k
   ```

2. Edit the `authentication` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: my-public.crt
      key: my-private.key
  # ...
```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Configuring SCRAM-SHA-512 authentication in Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator
- Username of the user which should be used for authentication

Procedure

1. Find out the name of the `Secret` with the password which should be used for authentication and the key under which the password is stored in the `Secret`. If such a `Secret` does not exist yet, prepare a file with the password and create the `Secret`.

   On Kubernetes this can be done using `kubectl create`:

   ```
   echo -n 'password' > my-password.txt
   kubectl create secret generic my-secret --from-file=my-password.txt
   ```

   On OpenShift this can be done using `oc create`:

   ```
   echo -n '1f2d1e2e67df' > my-password.txt
   oc create secret generic my-secret --from-file=my-password.txt
   ```

2. Edit the `authentication` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnect
   metadata:
     name: my-connect
   spec:
     # ...
     authentication:
       type: scram-sha-512
       username: _my-username_
       passwordSecret:
         secretName: _my-secret_
         password: _my-password.txt_
     # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.5. Kafka Connect configuration

Strimzi allows you to customize the configuration of Apache Kafka Connect nodes by editing most of the options listed in Apache Kafka documentation.

The only options which cannot be configured are those related to the following areas:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Listener / REST interface configuration
- Plugin path configuration

These options are automatically configured by Strimzi.

Kafka Connect configuration

Kafka Connect can be configured using the `config` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. This property should contain the Kafka Connect configuration options as keys. The values could be in one of the following JSON types:

- String
- Number
- Boolean

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

- `ssl.`
- `sasl.`
- `security.`
- `listeners`
- `plugin.path`
- `rest.`
- `bootstrap.servers`

When one of the forbidden options is present in the `config` property, it will be ignored and a warning message will be printed to the Custer Operator log file. All other options will be passed to Kafka Connect.

Important	The Cluster Operator does not validate keys or values in the provided `config` object. When an invalid configuration is provided, the Kafka Connect cluster might not start or might become unstable. In such cases, the configuration in the `KafkaConnect.spec.config` or `KafkaConnectS2I.spec.config` object should be fixed and the cluster operator will roll out the new configuration to all Kafka Connect nodes.

Selected options have default values:

- `group.id` with default value `connect-cluster`
- `offset.storage.topic` with default value `connect-cluster-offsets`
- `config.storage.topic` with default value `connect-cluster-configs`
- `status.storage.topic` with default value `connect-cluster-status`
- `key.converter` with default value `org.apache.kafka.connect.json.JsonConverter`
- `value.converter` with default value `org.apache.kafka.connect.json.JsonConverter`
- `internal.key.converter` with default value `org.apache.kafka.connect.json.JsonConverter`
- `internal.value.converter` with default value `org.apache.kafka.connect.json.JsonConverter`

- `internal.key.converter.schemas.enable` with default value `false`
- `internal.value.converter.schemas.enable` with default value `false`

These options will be automatically configured in case they are not present in the `KafkaConnect.spec.config` or `KafkaConnectS2I.spec.config` properties.

An example showing Kafka Connect configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    internal.key.converter: org.apache.kafka.connect.json.JsonConverter
    internal.value.converter: org.apache.kafka.connect.json.JsonConverter
    internal.key.converter.schemas.enable: false
    internal.value.converter.schemas.enable: false
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  # ...
```

Configuring Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `config` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnect
   metadata:
     name: my-connect
   spec:
     # ...
     config:
       group.id: my-connect-cluster
       offset.storage.topic: my-connect-cluster-offsets
       config.storage.topic: my-connect-cluster-configs
       status.storage.topic: my-connect-cluster-status
       key.converter: org.apache.kafka.connect.json.JsonConverter
       value.converter: org.apache.kafka.connect.json.JsonConverter
       key.converter.schemas.enable: true
       value.converter.schemas.enable: true
       internal.key.converter: org.apache.kafka.connect.json.JsonConverter
       internal.value.converter: org.apache.kafka.connect.json.JsonConverter
       internal.key.converter.schemas.enable: false
       internal.value.converter.schemas.enable: false
       config.storage.replication.factor: 3
       offset.storage.replication.factor: 3
       status.storage.replication.factor: 3
     # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.6. CPU and memory resources

For every deployed container, Strimzi allows you to specify the resources which should be reserved for it and the maximum resources that can be consumed by it. Strimzi supports two types of resources:

- Memory
- CPU

Strimzi is using the OpenShift or Kubernetes syntax for specifying CPU and memory resources.

Resource limits and requests

Resource limits and requests can be configured using the `resources` property in following resources:

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

Resource requests

Requests specify the resources that will be reserved for a given container. Reserving the resources will ensure that they are always available.

Important	If the resource request is for more than the available free resources in the OpenShift or Kubernetes cluster, the pod will not be scheduled.

Resource requests can be specified in the `request` property. The resource requests currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource request configuration

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

It is also possible to specify a resource request just for one of the resources:

An example showing resource request configuration with memory request only

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

Or:

An example showing resource request configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

Resource limits

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

Resource limits can be specified in the `limits` property. The resource limits currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource limits configuration

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

It is also possible to specify the resource limit just for one of the resources:

An example showing resource limit configuration with memory request only

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

Or:

An example showing resource limits configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

Supported CPU formats

CPU requests and limits are supported in the following formats:

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

An example of using different CPU units

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

Note	The amount of computing power of 1 CPU core might differ depending on the platform where the OpenShift or Kubernetes is deployed.

For more details about the CPU specification, see the Meaning of CPU website.

Supported memory formats

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

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

An example of using different memory units

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

```
    memory: 2Gi
# ...
```

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

Additional resources

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

Configuring resource requests and limits

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the schema, see `Resources` schema reference.

3.2.7. Logging

Logging enables you to diagnose error and performance issues of Strimzi. For the logging, various logger implementations are used. Kafka and Zookeeper use `log4j` logger and Topic Operator, User Operator, and other components use `log4j2` logger.

This section provides information about different loggers and describes how to configure log levels.

You can set the log levels by specifying the loggers and their levels directly (inline) or by using a custom (external) config map.

Using inline logging setting

Procedure

1. Edit the YAML file to specify the loggers and their level for the required components. For example:

```
apiVersion: {KafkaApiVersion}
kind: Kafka
spec:
  kafka:
    # ...
    logging:
      type: inline
      loggers:
```

```
      logger.name: "INFO"
    # ...
```

In the above example, the log level is set to INFO. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF. For more information about the log levels, see link: log4j manual.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using external ConfigMap for logging setting

Procedure

1. Edit the YAML file to specify the name of the `ConfigMap` which should be used for the required components. For example:

   ```
   apiVersion: {KafkaApiVersion}
   kind: Kafka
   spec:
     kafka:
       # ...
       logging:
         type: external
         name: customConfigMap
       # ...
   ```

   Remember to place your custom ConfigMap under `log4j.properties` eventually `log4j2.properties` key.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Loggers

Strimzi consists of several components. Each component has its own loggers and is configurable. This section provides information about loggers of various components.

Components and their loggers are listed below.

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

- `log4j.logger.kafka.authorizer.logger`

- Zookeeper
  - `zookeeper.root.logger`
- Kafka Connect and Kafka Connect with Source2Image support
  - `connect.root.logger.level`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.org.I0Itec.zkclient`
  - `log4j.logger.org.reflections`
- Kafka Mirror Maker
  - `mirrormaker.root.logger`
- Topic Operator
  - `rootLogger.level`
- User Operator
  - `rootLogger.level`

3.2.8. Healthchecks

Healthchecks are periodical tests which verify that the application's health. When the Healthcheck fails, OpenShift or Kubernetes can assume that the application is not healthy and attempt to fix it. OpenShift or Kubernetes supports two types of Healthcheck probes:

- Liveness probes
- Readiness probes

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

Users can configure selected options for liveness and readiness probes

Healthcheck configurations

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

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

Both `livenessProbe` and `readinessProbe` support two additional options:

- `initialDelaySeconds`
- `timeoutSeconds`

The `initialDelaySeconds` property defines the initial delay before the probe is tried for the first time. Default is 15 seconds.

The `timeoutSeconds` property defines timeout of the probe. Default is 5 seconds.

An example of liveness and readiness probe configuration

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

Configuring healthchecks

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    readinessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.9. Prometheus metrics

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

For more information about configuring Prometheus and Grafana, see Metrics.

Metrics configuration

Prometheus metrics can be enabled by configuring the `metrics` property in following resources:

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

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

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

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

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

Configuring Prometheus metrics

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.10. JVM Options

Apache Kafka and Apache Zookeeper are running inside of a Java Virtual Machine (JVM). JVM has many configuration options to optimize the performance for different platforms and architectures. Strimzi allows configuring some of these options.

JVM configuration

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

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

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

-Xms and -Xmx

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

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

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

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

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

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

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

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

Example fragment configuring `-Xmx` and `-Xms`

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

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

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

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

Example fragment configuring `-server`

```
# ...
jvmOptions:
```

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

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

-XX

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

Example showing the use of the `-XX` object

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

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

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+Explicit
```

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

Configuring JVM options

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       jvmOptions:
         "-Xmx": "8g"
         "-Xms": "8g"
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply` :

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

   On OpenShift this can be done using `oc apply` :

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

3.2.11. Container images

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

Container image configurations

Container image which should be used for given components can be specified using the `image` property in:

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

The `image` specified in the component-specific custom resource will be used during deployment. If the `image` field is missing, the `image` specified in the Cluster Operator configuration will be used. If the `image` name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka brokers:
  1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka:latest` container image.
- For Kafka broker TLS sidecar:
  1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka-stunnel:latest` container image.
- For Zookeeper nodes:
  1. Container image specified in the `STRIMZI_DEFAULT_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/zookeeper:latest` container image.
- For Zookeeper node TLS sidecar:
  1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/zookeeper-stunnel:latest` container image.
- For Topic Operator:
  1. Container image specified in the `STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration. .** `strimzi/topic-operator:latest` container image.
- For User Operator:
  1. Container image specified in the `STRIMZI_DEFAULT_USER_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/user-operator:latest` container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/entity-operator-stunnel:latest` container image.
- For Kafka Connect:
  1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka-connect:latest` container image.

- For Kafka Connect with Source2image support:
  1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka-connect-s2i:latest` container image.

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

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

Configuring container images

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       image: my-org/my-image:latest
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.12. Configuring pod scheduling

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

Scheduling pods based on other applications

Avoid critical applications to share the node

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Scheduling pods to specific nodes

Node scheduling

The OpenShift or Kubernetes cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while other might be optimized for storage (fast local SSDs) or network. Using different nodes helps to optimize both costs and performance. To achieve

the best possible performance, it is important to allow scheduling of Strimzi components to use the right nodes.

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring node affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Label the nodes where Strimzi components should be scheduled.

   On Kubernetes this can be done using `kubectl label`:

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

   On OpenShift this can be done using `oc label`:

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

   Alternatively, some of the existing labels might be reused.

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
               - matchExpressions:
                 - key: node-type
                   operator: In
                   values:
                   - fast-network
       # ...
     zookeeper:
       # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using dedicated nodes

Dedicated nodes

Cluster administrators can mark selected OpenShift or Kubernetes nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Tolerations

Tolerations ca be configured using the `tolerations` property in following resources:

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The format of the `tolerations` property follows the OpenShift or Kubernetes specification. For more details, see the Kubernetes taints and tolerations.

Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated
2. Make sure there are no workloads scheduled on these nodes
3. Set the taints on the selected nodes

   On Kubernetes this can be done using `kubectl taint`:

   ```
   kubectl taint node your-node dedicated=Kafka:NoSchedule
   ```

   On OpenShift this can be done using `oc adm taint`:

   ```
   oc adm taint node your-node dedicated=Kafka:NoSchedule
   ```

4. Additionally, add a label to the selected nodes as well.

   On Kubernetes this can be done using `kubectl label`:

   ```
   kubectl label node your-node dedicated=Kafka
   ```

   On OpenShift this can be done using `oc label`:

   ```
   oc label node your-node dedicated=Kafka
   ```

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       tolerations:
         - key: "dedicated"
           operator: "Equal"
           value: "Kafka"
           effect: "NoSchedule"
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
             - matchExpressions:
               - key: dedicated
                 operator: In
                 values:
                 - Kafka
       # ...
     zookeeper:
       # ...
   ```

6. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.2.13. List of resources created as part of Kafka Connect cluster

The following resources will created by the Cluster Operator in the OpenShift or Kubernetes cluster:

connect-cluster-name-connect
Deployment which is in charge to create the Kafka Connect worker node pods.

connect-cluster-name-connect-api
Service which exposes the REST interface for managing the Kafka Connect cluster.

connect-cluster-name-config
ConfigMap which contains the Kafka Connect ancillary configuration and is mounted as a volume by the Kafka broker pods.

3.3. Kafka Connect cluster with Source2Image support

The full schema of the `KafkaConnectS2I` resource is described in the `KafkaConnectS2I` schema reference. All labels that are applied to the desired `KafkaConnectS2I` resource will also be applied to the OpenShift or Kubernetes resources making up the Kafka Connect cluster with Source2Image support. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.3.1. Replicas

Kafka Connect clusters can run with a different number of nodes. The number of nodes is defined in the `KafkaConnect` and `KafkaConnectS2I` resources. Running Kafka Connect cluster with multiple nodes can provide better availability and scalability. However, when running Kafka Connect on OpenShift or Kubernetes it is not absolutely necessary to run multiple nodes of Kafka Connect for high availability.

When the node where Kafka Connect is deployed to crashes, OpenShift or Kubernetes will automatically take care of rescheduling the Kafka Connect pod to a different node. However, running Kafka Connect with multiple nodes can provide faster failover times, because the other nodes will be already up and running.

Configuring the number of nodes

Number of Kafka Connect nodes can be configured using the `replicas` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`.

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `replicas` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  replicas: 3
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.2. Bootstrap servers

Kafka Connect cluster always works together with a Kafka cluster. The Kafka cluster is specified in the form of a list of bootstrap servers. On OpenShift or Kubernetes, the list must ideally contain the Kafka cluster bootstrap service which is named `cluster-name-kafka-bootstrap` and a port of 9092 for plain traffic or 9093 for encrypted traffic.

The list of bootstrap servers can be configured in the `bootstrapServers` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. The servers should be a comma-separated list containing one or more Kafka brokers or a service pointing to Kafka brokers specified as a `hostname:_port_` pairs.

When using Kafka Connect with a Kafka cluster not managed by Strimzi, you can specify the bootstrap servers list according to the configuration of a given cluster.

Configuring bootstrap servers

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `bootstrapServers` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  # ...
```

2. Create or update the resource.

On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

3.3.3. Connecting to Kafka brokers using TLS

By default, Kafka Connect will try to connect to Kafka brokers using a plain text connection. If you would prefer to use TLS additional configuration will be necessary.

TLS support in Kafka Connect

TLS support is configured in the `tls` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. The `tls` property contains a list of secrets with key names under which the certificates are stored. The certificates should be stored in X509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-secret
        certificate: ca.crt
      - secretName: my-other-secret
        certificate: certificate.crt
  # ...
```

When multiple certificates are stored in the same secret, it can be listed multiple times.

An example showing TLS configuration with multiple certificates from the same secret

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-secret
        certificate: ca.crt
      - secretName: my-secret
        certificate: ca2.crt
  # ...
```

Configuring TLS in Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Find out the name of the secret with the certificate which should be used for TLS Server Authentication and the key under which the certificate is stored in the secret. If such secret does not exist yet, prepare the certificate in a file and create the secret.

   On Kubernetes this can be done using `kubectl create`:

   ```
   kubectl create secret generic my-secret --from-file=my-file.crt
   ```

   On OpenShift this can be done using `oc create`:

   ```
   oc create secret generic my-secret --from-file=my-file.crt
   ```

2. Edit the `tls` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        certificate: ca.crt
  # ...
```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.4. Connecting to Kafka brokers with Authentication

By default, Kafka Connect will try to connect to Kafka brokers without any authentication. Authentication can be enabled in the `KafkaConnect` and `KafkaConnectS2I` resources.

Authentication support in Kafka Connect

Authentication can be configured in the `authentication` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. The `authentication` property specifies the type of the authentication mechanisms which should be used and additional configuration details depending on the mechanism. The currently supported authentication types are:

- TLS client authentication
- SASL based authentication using SCRAM-SHA-512 mechanism

TLS Client Authentication

To use the TLS client authentication, set the `type` property to the value `tls`. TLS client authentication is using TLS certificate to authenticate. The certificate has to be specified in the `certificateAndKey` property. It is always loaded from an OpenShift or Kubernetes secret. Inside the secret, it has to be stored in the X509 format under two different keys: for public and private keys.

Note	TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Connect see Connecting to Kafka brokers using TLS.

An example showing TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  # ...
```

SCRAM-SHA-512 authentication

To use the authentication using the SCRAM-SHA-512 SASL mechanism, set the `type` property to the value `scram-sha-512`. SCRAM-SHA-512 uses a username and password to authenticate. Specify the username in the `username` property. Specify the password as a link to a `Secret` containing the password in the `passwordSecret` property. It has to specify the name of the `Secret` containing the password and the name of the key under which the password is stored inside the `Secret`.

An example showing SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: password
  # ...
```

Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Find out the name of the `Secret` with the public and private keys which should be used for TLS Client Authentication and the keys under which they are stored in the `Secret`. If such a `Secret` does not exist yet, prepare the keys in a file and create the `Secret`.

   On Kubernetes this can be done using `kubectl create`:

   ```
   kubectl create secret generic my-secret --from-file=my-public.crt --from-file=my-priv
   ```

   On OpenShift this can be done using `oc create`:

   ```
   oc create secret generic my-secret --from-file=my-public.crt --from-file=my-private.k
   ```

2. Edit the `authentication` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnect
   metadata:
     name: my-connect
   spec:
     # ...
     authentication:
       type: tls
       certificateAndKey:
         secretName: my-secret
         certificate: my-public.crt
         key: my-private.key
     # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Configuring SCRAM-SHA-512 authentication in Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator
- Username of the user which should be used for authentication

Procedure

1. Find out the name of the `Secret` with the password which should be used for authentication and the key under which the password is stored in the `Secret`. If such a `Secret` does not exist yet, prepare a file with the password and create the `Secret`.

   On Kubernetes this can be done using `kubectl create`:

   ```
   echo -n 'password' > my-password.txt
   kubectl create secret generic my-secret --from-file=my-password.txt
   ```

   On OpenShift this can be done using `oc create`:

   ```
   echo -n '1f2d1e2e67df' > my-password.txt
   oc create secret generic my-secret --from-file=my-password.txt
   ```

2. Edit the `authentication` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnect
   metadata:
     name: my-connect
   spec:
     # ...
     authentication:
       type: scram-sha-512
       username: _my-username_
       passwordSecret:
         secretName: _my-secret_
         password: _my-password.txt_
     # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.5. Kafka Connect configuration

Strimzi allows you to customize the configuration of Apache Kafka Connect nodes by editing most of the options listed in Apache Kafka documentation.

The only options which cannot be configured are those related to the following areas:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Listener / REST interface configuration
- Plugin path configuration

These options are automatically configured by Strimzi.

Kafka Connect configuration

Kafka Connect can be configured using the `config` property in `KafkaConnect.spec` and `KafkaConnectS2I.spec`. This property should contain the Kafka Connect configuration options as keys. The values could be in one of the following JSON types:

- String
- Number
- Boolean

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

- `ssl.`

- `sasl.`
- `security.`
- `listeners`
- `plugin.path`
- `rest.`
- `bootstrap.servers`

When one of the forbidden options is present in the `config` property, it will be ignored and a warning message will be printed to the Custer Operator log file. All other options will be passed to Kafka Connect.

Important	The Cluster Operator does not validate keys or values in the provided `config` object. When an invalid configuration is provided, the Kafka Connect cluster might not start or might become unstable. In such cases, the configuration in the `KafkaConnect.spec.config` or `KafkaConnectS2I.spec.config` object should be fixed and the cluster operator will roll out the new configuration to all Kafka Connect nodes.

Selected options have default values:

- `group.id` with default value `connect-cluster`
- `offset.storage.topic` with default value `connect-cluster-offsets`
- `config.storage.topic` with default value `connect-cluster-configs`
- `status.storage.topic` with default value `connect-cluster-status`
- `key.converter` with default value `org.apache.kafka.connect.json.JsonConverter`
- `value.converter` with default value `org.apache.kafka.connect.json.JsonConverter`
- `internal.key.converter` with default value `org.apache.kafka.connect.json.JsonConverter`
- `internal.value.converter` with default value `org.apache.kafka.connect.json.JsonConverter`
- `internal.key.converter.schemas.enable` with default value `false`
- `internal.value.converter.schemas.enable` with default value `false`

These options will be automatically configured in case they are not present in the `KafkaConnect.spec.config` or `KafkaConnectS2I.spec.config` properties.

An example showing Kafka Connect configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    internal.key.converter: org.apache.kafka.connect.json.JsonConverter
    internal.value.converter: org.apache.kafka.connect.json.JsonConverter
    internal.key.converter.schemas.enable: false
    internal.value.converter.schemas.enable: false
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  # ...
```

Configuring Kafka Connect

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `config` property in the `KafkaConnect` or `KafkaConnectS2I` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    internal.key.converter: org.apache.kafka.connect.json.JsonConverter
    internal.value.converter: org.apache.kafka.connect.json.JsonConverter
    internal.key.converter.schemas.enable: false
    internal.value.converter.schemas.enable: false
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.6. CPU and memory resources

For every deployed container, Strimzi allows you to specify the resources which should be reserved for it and the maximum resources that can be consumed by it. Strimzi supports two types of resources:

- Memory
- CPU

Strimzi is using the OpenShift or Kubernetes syntax for specifying CPU and memory resources.

Resource limits and requests

Resource limits and requests can be configured using the `resources` property in following resources:

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

Resource requests

Requests specify the resources that will be reserved for a given container. Reserving the resources will ensure that they are always available.

Important	If the resource request is for more than the available free resources in the OpenShift or Kubernetes cluster, the pod will not be scheduled.

Resource requests can be specified in the `request` property. The resource requests currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource request configuration

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

It is also possible to specify a resource request just for one of the resources:

An example showing resource request configuration with memory request only

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

Or:

An example showing resource request configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

Resource limits

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

Resource limits can be specified in the `limits` property. The resource limits currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource limits configuration

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

It is also possible to specify the resource limit just for one of the resources:

An example showing resource limit configuration with memory request only

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

Or:

An example showing resource limits configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

Supported CPU formats

CPU requests and limits are supported in the following formats:

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

An example of using different CPU units

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

Note	The amount of computing power of 1 CPU core might differ depending on the platform where the OpenShift or Kubernetes is deployed.

For more details about the CPU specification, see the Meaning of CPU website.

Supported memory formats

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

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

An example of using different memory units

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

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

Additional resources

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

Configuring resource requests and limits

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the schema, see `Resources` schema reference.

3.3.7. Logging

Logging enables you to diagnose error and performance issues of Strimzi. For the logging, various logger implementations are used. Kafka and Zookeeper use `log4j` logger and Topic Operator, User Operator, and other components use `log4j2` logger.

This section provides information about different loggers and describes how to configure log levels.

You can set the log levels by specifying the loggers and their levels directly (inline) or by using a custom (external) config map.

Using inline logging setting

Procedure

1. Edit the YAML file to specify the loggers and their level for the required components. For example:

   ```
   apiVersion: {KafkaApiVersion}
   kind: Kafka
   spec:
     kafka:
       # ...
       logging:
         type: inline
         loggers:
           logger.name: "INFO"
       # ...
   ```

   In the above example, the log level is set to INFO. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF. For more information about the log levels, see link: log4j manual.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using external ConfigMap for logging setting

Procedure

1. Edit the YAML file to specify the name of the `ConfigMap` which should be used for the required components. For example:

   ```
   apiVersion: {KafkaApiVersion}
   kind: Kafka
   spec:
     kafka:
       # ...
       logging:
         type: external
         name: customConfigMap
       # ...
   ```

   Remember to place your custom ConfigMap under `log4j.properties` eventually `log4j2.properties` key.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

Loggers

Strimzi consists of several components. Each component has its own loggers and is configurable. This section provides information about loggers of various components.

Components and their loggers are listed below.

- Kafka
  - `kafka.root.logger.level`
  - `log4j.logger.org.I0Itec.zkclient.ZkClient`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.kafka`
  - `log4j.logger.org.apache.kafka`
  - `log4j.logger.kafka.request.logger`
  - `log4j.logger.kafka.network.Processor`
  - `log4j.logger.kafka.server.KafkaApis`
  - `log4j.logger.kafka.network.RequestChannel$`
  - `log4j.logger.kafka.controller`
  - `log4j.logger.kafka.log.LogCleaner`
  - `log4j.logger.state.change.logger`
  - `log4j.logger.kafka.authorizer.logger`
- Zookeeper
  - `zookeeper.root.logger`
- Kafka Connect and Kafka Connect with Source2Image support
  - `connect.root.logger.level`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.org.I0Itec.zkclient`
  - `log4j.logger.org.reflections`
- Kafka Mirror Maker
  - `mirrormaker.root.logger`
- Topic Operator
  - `rootLogger.level`
- User Operator
  - `rootLogger.level`

3.3.8. Healthchecks

Healthchecks are periodical tests which verify that the application's health. When the Healthcheck fails, OpenShift or Kubernetes can assume that the application is not healthy and attempt to fix it. OpenShift or Kubernetes supports two types of Healthcheck probes:

- Liveness probes
- Readiness probes

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

Users can configure selected options for liveness and readiness probes

Healthcheck configurations

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

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

Both `livenessProbe` and `readinessProbe` support two additional options:

- `initialDelaySeconds`
- `timeoutSeconds`

The `initialDelaySeconds` property defines the initial delay before the probe is tried for the first time. Default is 15 seconds.

The `timeoutSeconds` property defines timeout of the probe. Default is 5 seconds.

An example of liveness and readiness probe configuration

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

Configuring healthchecks

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    readinessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.9. Prometheus metrics

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

For more information about configuring Prometheus and Grafana, see Metrics.

Metrics configuration

Prometheus metrics can be enabled by configuring the `metrics` property in following resources:

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

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

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

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

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

Configuring Prometheus metrics

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
```

```
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.10. JVM Options

Apache Kafka and Apache Zookeeper are running inside of a Java Virtual Machine (JVM). JVM has many configuration options to optimize the performance for different platforms and architectures. Strimzi allows configuring some of these options.

JVM configuration

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

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

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

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

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

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

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

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

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

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

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

Example fragment configuring `-Xmx` and `-Xms`

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

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

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

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

Example fragment configuring `-server`

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

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

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

Example showing the use of the `-XX` object

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

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

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+Explicit
```

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

Configuring JVM options

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jvmOptions:
      "-Xmx": "8g"
      "-Xms": "8g"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.11. Container images

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

Container image configurations

Container image which should be used for given components can be specified using the `image` property in:

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

The `image` specified in the component-specific custom resource will be used during deployment. If the `image` field is missing, the `image` specified in the Cluster Operator configuration will be used. If the `image` name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka brokers:
  1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka:latest` container image.
- For Kafka broker TLS sidecar:
  1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka-stunnel:latest` container image.
- For Zookeeper nodes:

1. Container image specified in the `STRIMZI_DEFAULT_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
2. `strimzi/zookeeper:latest` container image.

- For Zookeeper node TLS sidecar:
  1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/zookeeper-stunnel:latest` container image.
- For Topic Operator:
  1. Container image specified in the `STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration. .** `strimzi/topic-operator:latest` container image.
- For User Operator:
  1. Container image specified in the `STRIMZI_DEFAULT_USER_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/user-operator:latest` container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/entity-operator-stunnel:latest` container image.
- For Kafka Connect:
  1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka-connect:latest` container image.
- For Kafka Connect with Source2image support:
  1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE` environment variable from the Cluster Operator configuration.
  2. `strimzi/kafka-connect-s2i:latest` container image.

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

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

Configuring container images

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.3.12. Configuring pod scheduling

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

Scheduling pods based on other applications

Avoid critical applications to share the node

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   ```

```
spec:
  kafka:
    # ...
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Scheduling pods to specific nodes

Node scheduling

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

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring node affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Label the nodes where Strimzi components should be scheduled.

   On Kubernetes this can be done using `kubectl label`:

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

On OpenShift this can be done using `oc label`:

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

Alternatively, some of the existing labels might be reused.

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
              - key: node-type
                operator: In
                values:
                - fast-network
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using dedicated nodes

Dedicated nodes

Cluster administrators can mark selected OpenShift or Kubernetes nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Tolerations

Tolerations ca be configured using the `tolerations` property in following resources:

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The format of the `tolerations` property follows the OpenShift or Kubernetes specification. For more details, see the Kubernetes taints and tolerations.

Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated
2. Make sure there are no workloads scheduled on these nodes
3. Set the taints on the selected nodes

   On Kubernetes this can be done using `kubectl taint`:

   ```
   kubectl taint node your-node dedicated=Kafka:NoSchedule
   ```

   On OpenShift this can be done using `oc adm taint`:

   ```
   oc adm taint node your-node dedicated=Kafka:NoSchedule
   ```

4. Additionally, add a label to the selected nodes as well.

   On Kubernetes this can be done using `kubectl label`:

   ```
   kubectl label node your-node dedicated=Kafka
   ```

   On OpenShift this can be done using `oc label`:

   ```
   oc label node your-node dedicated=Kafka
   ```

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       tolerations:
         - key: "dedicated"
           operator: "Equal"
           value: "Kafka"
           effect: "NoSchedule"
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
             - matchExpressions:
               - key: dedicated
                 operator: In
                 values:
                 - Kafka
       # ...
     zookeeper:
       # ...
   ```

6. Create or update the resource.

On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

3.3.13. List of resources created as part of Kafka Connect cluster with Source2Image support

The following resources will created by the Cluster Operator in the OpenShift or Kubernetes cluster:

connect-cluster-name-connect-source
ImageStream which is used as the base image for the newly-built Docker images.

connect-cluster-name-connect
BuildConfig which is responsible for building the new Kafka Connect Docker images.

connect-cluster-name-connect
ImageStream where the newly built Docker images will be pushed.

connect-cluster-name-connect
DeploymentConfig which is in charge of creating the Kafka Connect worker node pods.

connect-cluster-name-connect-api
Service which exposes the REST interface for managing the Kafka Connect cluster.

connect-cluster-name-config
ConfigMap which contains the Kafka Connect ancillary configuration and is mounted as a volume by the Kafka broker pods.

3.3.14. Using OpenShift builds and S2I to create new images

OpenShift supports builds, which can be used together with the Source-to-Image (S2I) framework to create new container images. An OpenShift build takes a builder image with S2I support together with source code and binaries provided by the user and uses them to build a new container image. The newly created container image is stored in OpenShift's local container image repository and can be used in deployments. Strimzi provides a Kafka Connect builder image, which can be found on Docker Hub as `strimzi/kafka-connect-s2i:0.8.0` with this S2I support. It takes user-provided binaries (with plugins and connectors) and creates a new Kafka Connect image. This enhanced Kafka Connect image can be used with the Kafka Connect deployment.

The S2I deployment provided as an OpenShift template. It can be deployed from the template using the command-line or the OpenShift console.

Procedure

1. Create a Kafka Connect S2I cluster from the command-line

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

2. Once the cluster is deployed, a new build can be triggered from the command-line by creating a directory with Kafka Connect plugins:

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-3.4.2.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mongodb-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mongodb-driver-3.4.2.jar
│   ├── mongodb-driver-core-3.4.2.jar
│   └── README.md
├── debezium-connector-mysql
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mysql-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
```

```
│   ├── mysql-binlog-connector-java-0.13.0.jar
│   ├── mysql-connector-java-5.1.40.jar
│   ├── README.md
│   └── wkb-1.0.2.jar
└── debezium-connector-postgres
    ├── CHANGELOG.md
    ├── CONTRIBUTE.md
    ├── COPYRIGHT.txt
    ├── debezium-connector-postgres-0.7.1.jar
    ├── debezium-core-0.7.1.jar
    ├── LICENSE.txt
    ├── postgresql-42.0.0.jar
    ├── protobuf-java-2.6.1.jar
    └── README.md
```

3. Start a new image build using the prepared directory:

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

Note	The name of the build will be changed according to the cluster name of the deployed Kafka Connect cluster.

4. Once the build is finished, the new image will be used automatically by the Kafka Connect deployment.

3.4. Kafka Mirror Maker configuration

The full schema of the `KafkaMirrorMaker` resource is described in the `KafkaMirrorMaker` schema reference. All labels that apply to the desired `KafkaMirrorMaker` resource will also be applied to the OpenShift or Kubernetes resources making up Mirror Maker. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.4.1. Replicas

It is possible to run multiple Mirror Maker replicas. The number of replicas is defined in the `KafkaMirrorMaker` resource. You can run multiple Mirror Maker replicas to provide better availability and scalability. However, when running Kafka Mirror Maker on OpenShift or Kubernetes it is not absolutely necessary to run multiple replicas of the Kafka Mirror Maker for high availability. When the node where the Kafka Mirror Maker has deployed crashes, OpenShift or Kubernetes will automatically reschedule the Kafka Mirror Maker pod to a different node. However, running Kafka Mirror Maker with multiple replicas can provide faster failover times as the other nodes will be up and running.

Configuring the number of replicas

The number of Kafka Mirror Maker replicas can be configured using the `replicas` property in `KafkaMirrorMaker.spec`.

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `replicas` property in the `KafkaMirrorMaker` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  replicas: 3
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.2. Bootstrap servers

Kafka Mirror Maker always works together with two Kafka clusters (source and target). The source and the target Kafka clusters are specified in the form of two lists of comma-separated list of `<hostname>:<port>` pairs. The bootstrap server lists can refer to Kafka clusters which do not need to be deployed in the same OpenShift or Kubernetes cluster. They can even refer to any Kafka cluster not deployed by Strimzi or even deployed by Strimzi but on a different OpenShift or Kubernetes cluster and accessible from outside.

If on the same OpenShift or Kubernetes cluster, each list must ideally contain the Kafka cluster bootstrap service which is named `<cluster-name>-kafka-bootstrap` and a port of 9092 for plain traffic or 9093 for encrypted traffic. If deployed by Strimzi but on different OpenShift or Kubernetes clusters, the list content depends on the way used for exposing the clusters (routes, nodeports or loadbalancers).

The list of bootstrap servers can be configured in the `KafkaMirrorMaker.spec.consumer.bootstrapServers` and `KafkaMirrorMaker.spec.producer.bootstrapServers` properties. The servers should be a comma-separated list containing one or more Kafka brokers or a `Service` pointing to Kafka brokers specified as a `<hostname>:<port>` pairs.

When using Kafka Mirror Maker with a Kafka cluster not managed by Strimzi, you can specify the bootstrap servers list according to the configuration of the given cluster.

Configuring bootstrap servers

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `KafkaMirrorMaker.spec.consumer.bootstrapServers` and `KafkaMirrorMaker.spec.producer.bootstrapServers` properties. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    bootstrapServers: my-source-cluster-kafka-bootstrap:9092
  # ...
  producer:
    bootstrapServers: my-target-cluster-kafka-bootstrap:9092
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.3. Whitelist

You specify the list topics that the Kafka Mirror Maker has to mirror from the source to the target Kafka cluster in the KafkaMirrorMaker resource using the *whitelist* option. It allows any regular expression from the simplest case with a single topic name to complex patterns. For example, you can mirror topics A and B using "A|B" or all topics using "*". You can also pass multiple regular expressions separated by commas to the Kafka Mirror Maker.

Configuring the topics whitelist

Specify the list topics that have to be mirrored by the Kafka Mirror Maker from source to target Kafka cluster using the `whitelist` property in `KafkaMirrorMaker.spec`.

Prerequisites

- An OpenShift or Kubernetes cluster

- A running Cluster Operator

Procedure

1. Edit the `whitelist` property in the `KafkaMirrorMaker` resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  whitelist: "my-topic|other-topic"
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.4. Consumer group identifier

The Kafka Mirror Maker uses Kafka consumer to consume messages and it behaves like any other Kafka consumer client. It is in charge to consume the messages from the source Kafka cluster which will be mirrored to the target Kafka cluster. The consumer needs to be part of a *consumer group* for being assigned partitions.

Configuring the consumer group identifier

The consumer group identifier can be configured in the `KafkaMirrorMaker.spec.consumer.groupId` property.

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `KafkaMirrorMaker.spec.consumer.groupId` property. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    groupId: "my-group"
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.5. Number of consumer streams

You can increase the throughput in mirroring topics by increase the number of consumer threads. More consumer threads will belong to the same configured *consumer group*. The topic partitions will be assigned across these consumer threads which will consume messages in parallel.

Configuring the number of consumer streams

The number of consumer streams can be configured using the `KafkaMirrorMaker.spec.consumer.numStreams` property.

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `KafkaMirrorMaker.spec.consumer.numStreams` property. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    numStreams: 2
  # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.6. Connecting to Kafka brokers using TLS

By default, Kafka Mirror Maker will try to connect to Kafka brokers, in the source and target clusters, using a plain text connection. You must make additional configurations to use TLS.

TLS support in Kafka Mirror Maker

TLS support is configured in the `tls` sub-property of `consumer` and `producer` properties in `KafkaMirrorMaker.spec`. The `tls` property contains a list of secrets with key names under which the certificates are stored. The certificates should be stored in X.509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    tls:
      trustedCertificates:
        - secretName: my-source-secret
          certificate: ca.crt
        - secretName: my-other-source-secret
          certificate: certificate.crt
  # ...
  producer:
    tls:
      trustedCertificates:
        - secretName: my-target-secret
          certificate: ca.crt
        - secretName: my-other-target-secret
          certificate: certificate.crt
  # ...
```

When multiple certificates are stored in the same secret, it can be listed multiple times.

An example showing TLS configuration with multiple certificates from the same secret

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
```

```
    tls:
      trustedCertificates:
        - secretName: my-source-secret
          certificate: ca.crt
        - secretName: my-source-secret
          certificate: ca2.crt
  # ...
  producer:
    tls:
      trustedCertificates:
        - secretName: my-target-secret
          certificate: ca.crt
        - secretName: my-target-secret
          certificate: ca2.crt
  # ...
```

Configuring TLS encryption in Kafka Mirror Maker

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

As the Kafka Mirror Maker connects to two Kafka clusters (source and target), you can choose to configure TLS for one or both the clusters. The following steps describe how to configure TLS on the consumer side for connecting to the source Kafka cluster:

1. Find out the name of the secret with the certificate which should be used for TLS Server Authentication and the key under which the certificate is stored in the secret. If such secret does not exist yet, prepare the certificate in a file and create the secret.

   On Kubernetes this can be done using `kubectl create`:

   ```
   kubectl create secret generic <my-secret> --from-file=<my-file.crt>
   ```

   On OpenShift this can be done using `oc create`:

   ```
   oc create secret generic <my-secret> --from-file=<my-file.crt>
   ```

2. Edit the `KafkaMirrorMaker.spec.consumer.tls` property. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaMirrorMaker
   metadata:
     name: my-mirror-maker
   spec:
     # ...
     consumer:
       tls:
         trustedCertificates:
           - secretName: my-cluster-cluster-cert
             certificate: ca.crt
     # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Repeat the above steps for configuring TLS on the target Kafka cluster. In this case, the secret containing the certificate has to be configured in the `KafkaMirrorMaker.spec.producer.tls` property.

3.4.7. Connecting to Kafka brokers with Authentication

By default, Kafka Mirror Maker will try to connect to Kafka brokers without any authentication. Authentication can be enabled in the `KafkaMirrorMaker` resource.

Authentication support in Kafka Mirror Maker

Authentication can be configured in the `KafkaMirrorMaker.spec.consumer.authentication` and `KafkaMirrorMaker.spec.producer.authentication` properties. The `authentication` property specifies the type of the authentication method which should be used and additional configuration details depending on the mechanism. The currently supported authentication types are:

- TLS client authentication
- SASL based authentication using SCRAM-SHA-512 mechanism

TLS Client Authentication

To use the TLS client authentication, set the `type` property to the value `tls`. The TLS client authentication uses TLS certificate to authenticate. The certificate has to be specified in the certificateAndKey property. It is always loaded from an OpenShift or Kubernetes secret. Inside the secret, it has to be stored in the X.509 format separately as public and private keys.

Note	TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Mirror Maker see Connecting to Kafka brokers using TLS.

An example showing TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    authentication:
      type: tls
      certificateAndKey:
        secretName: my-source-secret
        certificate: public.crt
        key: private.key
  # ...
  producer:
    authentication:
      type: tls
      certificateAndKey:
        secretName: my-target-secret
        certificate: public.crt
        key: private.key
  # ...
```

SCRAM-SHA-512 authentication

To use the authentication using the SCRAM-SHA-512 SASL mechanism, set the `type` property to the value `scram-sha-512`. It is possible to use it only if the broker listener, clients are connecting to, is configured to use it. SCRAM-SHA-512 uses a username and password to authenticate. Specify the username in the `username` property. Specify the password as a link to a `Secret` containing the password in the `passwordSecret` property. It has to specify the name of the `Secret` containing the password and the name of the key under which the password is stored inside the `Secret`.

An example showing SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    authentication:
      type: scram-sha-512
      username: my-source-user
      passwordSecret:
        secretName: my-source-user
        password: password
  # ...
  producer:
    authentication:
      type: scram-sha-512
      username: my-producer-user
      passwordSecret:
        secretName: my-producer-user
```

```
      password: password
  # ...
```

Configuring TLS client authentication in Kafka Mirror Maker

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator with a `tls` listener with `tls` authentication enabled

Procedure

As the Kafka Mirror Maker connects to two Kafka clusters (source and target), you can choose to configure TLS client authentication for one or both the clusters. The following steps describe how to configure TLS client authentication on the consumer side for connecting to the source Kafka cluster:

1. Find out the name of the `Secret` with the public and private keys which should be used for TLS Client Authentication and the keys under which they are stored in the `Secret`. If such a `Secret` does not exist yet, prepare the keys in a file and create the `Secret`.

   On Kubernetes this can be done using `kubectl create`:

   ```
   kubectl create secret generic <my-secret> --from-file=<my-public.crt> --from-file=<my
   ```

   On OpenShift this can be done using `oc create`:

   ```
   oc create secret generic <my-secret> --from-file=<my-public.crt> --from-file=<my-priv
   ```

2. Edit the `KafkaMirrorMaker.spec.consumer.authentication` property. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaMirrorMaker
   metadata:
     name: my-mirror-maker
   spec:
     # ...
     consumer:
       authentication:
         type: tls
         certificateAndKey:
           secretName: my-secret
           certificate: my-public.crt
           key: my-private.key
     # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Repeat the above steps for configuring TLS client authentication on the target Kafka cluster. In this case, the secret containing the certificate has to be configured in the `KafkaMirrorMaker.spec.producer.authentication` property.

Configuring SCRAM-SHA-512 authentication in Kafka Mirror Maker

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator with a `listener` configured for SCRAM-SHA-512 authentication
- Username to be used for authentication

Procedure

As the Kafka Mirror Maker connects to two Kafka clusters (source and target), you can choose to configure SCRAM-SHA-512 authentication for one or both the clusters. The following steps describe how to configure SCRAM-SHA-512 authentication on the consumer side for connecting to the source Kafka cluster:

1. Find out the name of the `Secret` with the password which should be used for authentication and the key under which the password is stored in the `Secret`. If such a `Secret` does not exist yet, prepare a file with the password and create the `Secret`.

   On Kubernetes this can be done using `kubectl create`:

   ```
   echo -n '<password>' > <my-password.txt>
   kubectl create secret generic <my-secret> --from-file=<my-password.txt>
   ```

   On OpenShift this can be done using `oc create`:

   ```
   echo -n '1f2d1e2e67df' > <my-password.txt>
   oc create secret generic <my-secret> --from-file=<my-password.txt>
   ```

2. Edit the `KafkaMirrorMaker.spec.consumer.authentication` property. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaMirrorMaker
   metadata:
     name: my-mirror-maker
   spec:
     # ...
     consumer:
       authentication:
         type: scram-sha-512
         username: _<my-username>_
         passwordSecret:
           secretName: _<my-secret>_
           password: _<my-password.txt>_
     # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Repeat the above steps for configuring SCRAM-SHA-512 authentication on the target Kafka cluster. In this case, the secret containing the certificate has to be configured in the `KafkaMirrorMaker.spec.producer.authentication` property.

3.4.8. Kafka Mirror Maker configuration

Strimzi allows you to customize the configuration of the Kafka Mirror Maker by editing most of the options for the related consumer and producer. Producer options are listed in Apache Kafka documentation. Consumer options are listed in Apache Kafka documentation.

The only options which cannot be configured are those related to the following areas:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Consumer group identifier

These options are automatically configured by Strimzi.

Kafka Mirror Maker configuration

Kafka Mirror Maker can be configured using the `config` sub-property in `KafkaMirrorMaker.spec.consumer` and `KafkaMirrorMaker.spec.producer`. This property should contain the Kafka Mirror Maker consumer and producer configuration options as keys. The values could be in one of the following JSON types:

- String
- Number
- Boolean

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

- `ssl.`
- `sasl.`
- `security.`
- `bootstrap.servers`
- `group.id`

When one of the forbidden options is present in the `config` property, it will be ignored and a warning message will be printed to the Custer Operator log file. All other options will be passed to Kafka Mirror Maker.

Important	The Cluster Operator does not validate keys or values in the provided `config` object. When an invalid configuration is provided, the Kafka Mirror Maker might not start or might become unstable. In such cases, the configuration in the `KafkaMirrorMaker.spec.consumer.config` or `KafkaMirrorMaker.spec.producer.config` object should be fixed and the cluster operator will roll out the new configuration for Kafka Mirror Maker.

An example showing Kafka Mirror Maker configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirroMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    config:
      max.poll.records: 100
      receive.buffer.bytes: 32768
  producer:
    config:
      compression.type: gzip
      batch.size: 8192
  # ...
```

Configuring Kafka Mirror Maker

Prerequisites

- Two OpenShift or Kubernetes clusters (source and target)
- A running Cluster Operator

Procedure

1. Edit the `KafkaMirrorMaker.spec.consumer.config` and `KafkaMirrorMaker.spec.producer.config` properties. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaMirroMaker
   metadata:
     name: my-mirror-maker
   spec:
     # ...
     consumer:
       config:
         max.poll.records: 100
         receive.buffer.bytes: 32768
     producer:
       config:
         compression.type: gzip
         batch.size: 8192
     # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

3.4.9. CPU and memory resources

For every deployed container, Strimzi allows you to specify the resources which should be reserved for it and the maximum resources that can be consumed by it. Strimzi supports two types of resources:

- Memory
- CPU

Strimzi is using the OpenShift or Kubernetes syntax for specifying CPU and memory resources.

Resource limits and requests

Resource limits and requests can be configured using the `resources` property in following resources:

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

Resource requests

Requests specify the resources that will be reserved for a given container. Reserving the resources will ensure that they are always available.

Important	If the resource request is for more than the available free resources in the OpenShift or Kubernetes cluster, the pod will not be scheduled.

Resource requests can be specified in the `request` property. The resource requests currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource request configuration

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

It is also possible to specify a resource request just for one of the resources:

An example showing resource request configuration with memory request only

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

Or:

An example showing resource request configuration with CPU request only

```
# ...
resources:
  requests:
```

```
    cpu: 12
# ...
```

Resource limits

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

Resource limits can be specified in the `limits` property. The resource limits currently supported by Strimzi are memory and CPU. Memory is specified under the property `memory`. CPU is specified under the property `cpu`.

An example showing resource limits configuration

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

It is also possible to specify the resource limit just for one of the resources:

An example showing resource limit configuration with memory request only

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

Or:

An example showing resource limits configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

Supported CPU formats

CPU requests and limits are supported in the following formats:

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

An example of using different CPU units

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

Note	The amount of computing power of 1 CPU core might differ depending on the platform where the OpenShift or Kubernetes is deployed.

For more details about the CPU specification, see the Meaning of CPU website.

Supported memory formats

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

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

An example of using different memory units

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

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

Additional resources

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

Configuring resource requests and limits

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the schema, see `Resources` schema reference.

3.4.10. Logging

Logging enables you to diagnose error and performance issues of Strimzi. For the logging, various logger implementations are used. Kafka and Zookeeper use `log4j` logger and Topic Operator, User Operator, and other components use `log4j2` logger.

This section provides information about different loggers and describes how to configure log levels.

You can set the log levels by specifying the loggers and their levels directly (inline) or by using a custom (external) config map.

Using inline logging setting

Procedure

1. Edit the YAML file to specify the loggers and their level for the required components. For example:

```
apiVersion: {KafkaApiVersion}
kind: Kafka
spec:
  kafka:
```

```
    # ...
    logging:
      type: inline
      loggers:
       logger.name: "INFO"
    # ...
```

In the above example, the log level is set to INFO. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF. For more information about the log levels, see link: log4j manual.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using external ConfigMap for logging setting

Procedure

1. Edit the YAML file to specify the name of the `ConfigMap` which should be used for the required components. For example:

   ```
   apiVersion: {KafkaApiVersion}
   kind: Kafka
   spec:
     kafka:
       # ...
       logging:
         type: external
         name: customConfigMap
       # ...
   ```

   Remember to place your custom ConfigMap under `log4j.properties` eventually `log4j2.properties` key.

2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Loggers

Strimzi consists of several components. Each component has its own loggers and is configurable. This section provides information about loggers of various components.

Components and their loggers are listed below.

- Kafka
  - `kafka.root.logger.level`
  - `log4j.logger.org.I0Itec.zkclient.ZkClient`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.kafka`
  - `log4j.logger.org.apache.kafka`
  - `log4j.logger.kafka.request.logger`
  - `log4j.logger.kafka.network.Processor`
  - `log4j.logger.kafka.server.KafkaApis`
  - `log4j.logger.kafka.network.RequestChannel$`

  - `log4j.logger.kafka.controller`
  - `log4j.logger.kafka.log.LogCleaner`
  - `log4j.logger.state.change.logger`
  - `log4j.logger.kafka.authorizer.logger`
- Zookeeper
  - `zookeeper.root.logger`
- Kafka Connect and Kafka Connect with Source2Image support
  - `connect.root.logger.level`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.org.I0Itec.zkclient`
  - `log4j.logger.org.reflections`
- Kafka Mirror Maker
  - `mirrormaker.root.logger`
- Topic Operator
  - `rootLogger.level`
- User Operator
  - `rootLogger.level`

3.4.11. Prometheus metrics

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

For more information about configuring Prometheus and Grafana, see Metrics.

Metrics configuration

Prometheus metrics can be enabled by configuring the `metrics` property in following resources:

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

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

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

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

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
```

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

Configuring Prometheus metrics

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.12. JVM Options

Apache Kafka and Apache Zookeeper are running inside of a Java Virtual Machine (JVM). JVM has many configuration options to optimize the performance for different platforms and architectures. Strimzi allows configuring some of these options.

JVM configuration

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

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

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

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

Note	The units accepted by JVM settings such as `-Xmx` and `-Xms` are those accepted by the JDK `java` binary in the corresponding image. Accordingly, `1g` or `1G` means

	1,073,741,824 bytes, and `Gi` is not a valid unit suffix. This is in contrast to the units used for memory requests and limits, which follow the OpenShift or Kubernetes convention where `1G` means 1,000,000,000 bytes, and `1Gi` means 1,073,741,824 bytes

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

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

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

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

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

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

Example fragment configuring `-Xmx` and `-Xms`

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

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

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

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

Example fragment configuring `-server`

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

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

-XX

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

Example showing the use of the `-XX` object

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

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

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+Explicit
```

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

Configuring JVM options

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       jvmOptions:
         "-Xmx": "8g"
         "-Xms": "8g"
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply` :

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

   On OpenShift this can be done using `oc apply` :

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

3.4.13. Container images

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

Container image configurations

Container image which should be used for given components can be specified using the `image` property in:

- `Kafka.spec.kafka`
- `Kafka.spec.kafka.tlsSidecar`

- `Kafka.spec.zookeeper`
- `Kafka.spec.zookeeper.tlsSidecar`
- `Kafka.spec.entityOperator.topicOperator`
- `Kafka.spec.entityOperator.userOperator`
- `Kafka.spec.entityOperator.tlsSidecar`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The `image` specified in the component-specific custom resource will be used during deployment. If the `image` field is missing, the `image` specified in the Cluster Operator configuration will be used. If the `image` name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka brokers:
    1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/kafka:latest` container image.
- For Kafka broker TLS sidecar:
    1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/kafka-stunnel:latest` container image.
- For Zookeeper nodes:
    1. Container image specified in the `STRIMZI_DEFAULT_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/zookeeper:latest` container image.
- For Zookeeper node TLS sidecar:
    1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/zookeeper-stunnel:latest` container image.
- For Topic Operator:
    1. Container image specified in the `STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration. .** `strimzi/topic-operator:latest` container image.
- For User Operator:
    1. Container image specified in the `STRIMZI_DEFAULT_USER_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/user-operator:latest` container image.
- For Entity Operator TLS sidecar:
    1. Container image specified in the `STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/entity-operator-stunnel:latest` container image.
- For Kafka Connect:
    1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/kafka-connect:latest` container image.
- For Kafka Connect with Source2image support:
    1. Container image specified in the `STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE` environment variable from the Cluster Operator configuration.
    2. `strimzi/kafka-connect-s2i:latest` container image.

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

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

Configuring container images

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   metadata:
     name: my-cluster
   spec:
     kafka:
       # ...
       image: my-org/my-image:latest
       # ...
     zookeeper:
       # ...
   ```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.14. Configuring pod scheduling

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

Scheduling pods based on other applications

Avoid critical applications to share the node

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Scheduling pods to specific nodes

Node scheduling

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

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

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Configuring node affinity in Kafka components

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Label the nodes where Strimzi components should be scheduled.

   On Kubernetes this can be done using `kubectl label`:

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

   On OpenShift this can be done using `oc label`:

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

   Alternatively, some of the existing labels might be reused.

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
               - matchExpressions:
                 - key: node-type
                   operator: In
                   values:
                   - fast-network
       # ...
     zookeeper:
       # ...
   ```

3. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Using dedicated nodes

Dedicated nodes

Cluster administrators can mark selected OpenShift or Kubernetes nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

Affinity

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

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

Tolerations

Tolerations ca be configured using the `tolerations` property in following resources:

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

The format of the `tolerations` property follows the OpenShift or Kubernetes specification. For more details, see the Kubernetes taints and tolerations.

Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift or Kubernetes cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated
2. Make sure there are no workloads scheduled on these nodes
3. Set the taints on the selected nodes

   On Kubernetes this can be done using `kubectl taint`:

   ```
   kubectl taint node your-node dedicated=Kafka:NoSchedule
   ```

   On OpenShift this can be done using `oc adm taint`:

   ```
   oc adm taint node your-node dedicated=Kafka:NoSchedule
   ```

4. Additionally, add a label to the selected nodes as well.

   On Kubernetes this can be done using `kubectl label`:

   ```
   kubectl label node your-node dedicated=Kafka
   ```

On OpenShift this can be done using `oc label`:

```
oc label node your-node dedicated=Kafka
```

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    tolerations:
      - key: "dedicated"
        operator: "Equal"
        value: "Kafka"
        effect: "NoSchedule"
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: dedicated
              operator: In
              values:
              - Kafka
    # ...
  zookeeper:
    # ...
```

6. Create or update the resource.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3.4.15. List of resources created as part of Kafka Mirror Maker

The following resources will created by the Cluster Operator in the OpenShift or Kubernetes cluster:

<*mirror-maker-name*>-mirror-maker
Deployment which is in charge to create the Kafka Mirror Maker pods.

<*mirror-maker-name*>-config
ConfigMap which contains the Kafka Mirror Maker ancillary configuration and is mounted as a volume by the Kafka broker pods.

4. Operators

4.1. Cluster Operator

4.1.1. Overview of the Cluster Operator component

The Cluster Operator is in charge of deploying a Kafka cluster alongside a Zookeeper ensemble. As part of the Kafka cluster, it can also deploy the topic operator which provides operator-style topic management via `KafkaTopic` custom resources. The Cluster Operator is also able to deploy a Kafka Connect cluster which connects to an existing Kafka cluster. On OpenShift such a cluster can be deployed using the Source2Image feature, providing an easy way of including more connectors.

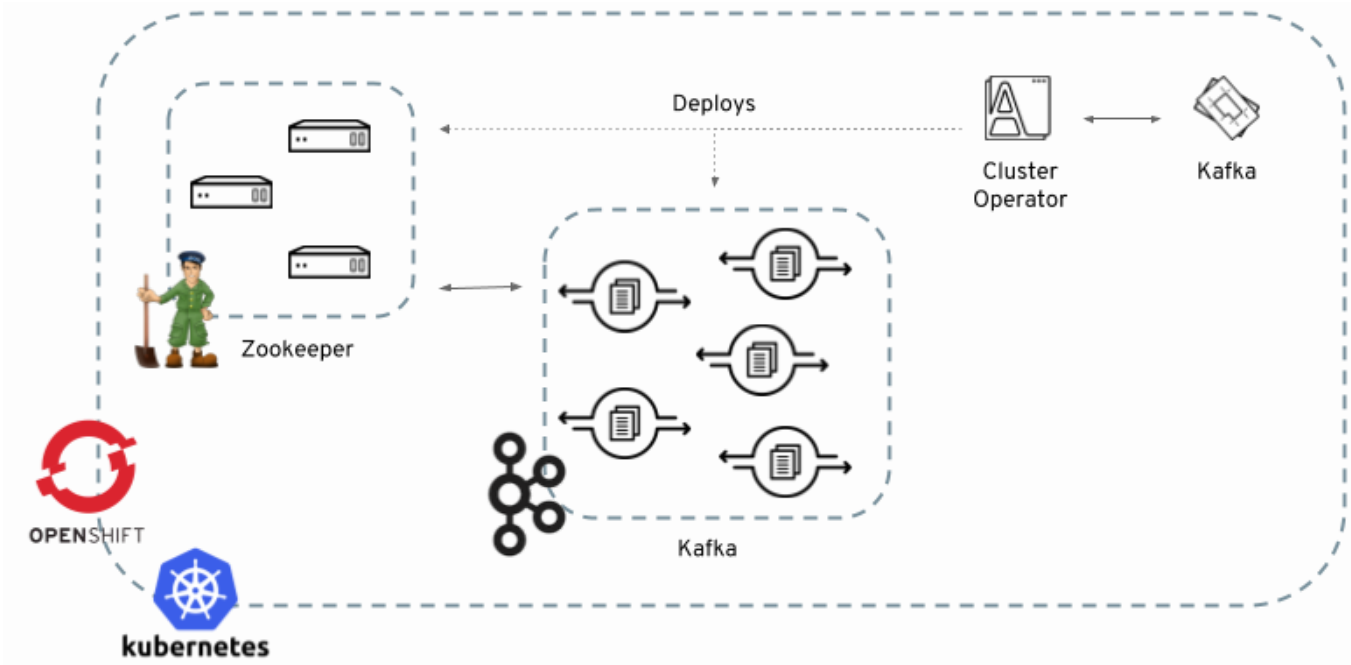


Figure 2. Example Architecture diagram of the Cluster Operator.

When the Cluster Operator is up, it starts to *watch* for certain OpenShift or Kubernetes resources containing the desired Kafka or Kafka Connect cluster configuration. By default, it watches only in the same namespace or project where it is installed. The Cluster Operator can be configured to watch for more OpenShift projects or Kubernetes namespaces. Cluster Operator watches the following resources:

- A `Kafka` resource for the Kafka cluster.
- A `KafkaConnect` resource for the Kafka Connect cluster.
- A `KafkaConnectS2I` resource for the Kafka Connect cluster with Source2Image support.

When a new `Kafka`, `KafkaConnect`, or `KafkaConnectS2I` resource is created in the OpenShift or Kubernetes cluster, the operator gets the cluster description from the desired resource and starts creating a new Kafka or Kafka Connect cluster by creating the necessary other OpenShift or Kubernetes resources, such as StatefulSets, Services, ConfigMaps, and so on.

Every time the desired resource is updated by the user, the operator performs corresponding updates on the OpenShift or Kubernetes resources which make up the Kafka or Kafka Connect cluster. Resources are either patched or deleted and then re-created in order to make the Kafka or Kafka Connect cluster reflect the state of the desired cluster resource. This might cause a rolling update which might lead to service disruption.

Finally, when the desired resource is deleted, the operator starts to undeploy the cluster and delete all the related OpenShift or Kubernetes resources.

4.1.2. Deploying the Cluster Operator to Kubernetes

Prerequisites

- Modify the installation files according to the namespace the Cluster Operator is going to be installed in.

  On Linux, use:

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

  On MacOS, use:

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

Procedure

1. Deploy the Cluster Operator

```
kubectl apply -f install/cluster-operator -n _my-namespace_
```

4.1.3. Deploying the Cluster Operator to OpenShift

Prerequisites

- A user with `cluster-admin` role needs to be used, for example, `system:admin`.
- Modify the installation files according to the namespace the Cluster Operator is going to be installed in.

  On Linux, use:

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

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-project/' install/cluster-operator/*RoleBinc
```

Procedure

1. Deploy the Cluster Operator

```
oc apply -f install/cluster-operator -n _my-project_
oc apply -f examples/templates/cluster-operator -n _my-project_
```

4.1.4. Deploying the Cluster Operator to watch multiple namespaces

Prerequisites

- Edit the installation files according to the OpenShift project or Kubernetes namespace the Cluster Operator is going to be installed in.

  On Linux, use:

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

  On MacOS, use:

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

Procedure

1. Edit the file `install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml` and in the environment variable `STRIMZI_NAMESPACE` list all the OpenShift projects or Kubernetes namespaces where Cluster Operator should watch for resources. For example:

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
      - name: strimzi-cluster-operator
        image: strimzi/cluster-operator:latest
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: myproject,myproject2,myproject3
```

2. For all namespaces or projects which should be watched by the Cluster Operator, install the `RoleBindings`. Replace the *`my-namespace`* or *`my-project`* with the OpenShift project or Kubernetes namespace used in the previous step.

   On Kubernetes this can be done using `kubectl apply`:

```
kubectl apply -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.ya
kubectl apply -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-er
kubectl apply -f install/cluster-operator/032-RoleBinding-strimzi-cluster-operator-tc
```

   On OpenShift this can be done using `oc apply`:

```
oc apply -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -r
oc apply -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-
oc apply -f install/cluster-operator/032-RoleBinding-strimzi-cluster-operator-topic-c
```

3. Deploy the Cluster Operator

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

4.1.5. Deploying the Cluster Operator using Helm Chart

Prerequisites

- Helm client has to be installed on the local machine.
- Helm has to be installed in the OpenShift or Kubernetes cluster.

Procedure

1. Add the Strimzi Helm Chart repository:

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

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

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

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

```
helm ls
```

Additional resources

- For more information about Helm, see the Helm website.

4.1.6. Reconciliation

Although the operator reacts to all notifications about the desired cluster resources received from the OpenShift or Kubernetes cluster, if the operator is not running, or if a notification is not received for any reason, the desired resources will get out of sync with the state of the running OpenShift or Kubernetes cluster.

In order to handle failovers properly, a periodic reconciliation process is executed by the Cluster Operator so that it can compare the state of the desired resources with the current cluster deployments in order to have a consistent state across all of them. You can set the time interval for the periodic reconciliations using the `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` variable.

4.1.7. Cluster Operator Configuration

The Cluster Operator can be configured through the following supported environment variables:

`STRIMZI_NAMESPACE`
Required. A comma-separated list of namespaces that the operator should operate in. The Cluster Operator deployment might use the Kubernetes Downward API to set this automatically to the namespace the Cluster Operator is deployed in. See the example below:

```
env:
  - name: STRIMZI_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

`STRIMZI_FULL_RECONCILIATION_INTERVAL_MS`
Optional, default: 120000 ms. The interval between periodic reconciliations, in milliseconds.

`STRIMZI_LOG_LEVEL`
Optional, default `INFO`. The level for printing logging messages. The value can be set to: `ERROR`, `WARNING`, `INFO`, `DEBUG`, and `TRACE`.

`STRIMZI_OPERATION_TIMEOUT_MS`
Optional, default: 300000 ms. The timeout for internal operations, in milliseconds. This value should be increased when using Strimzi on clusters where regular OpenShift or Kubernetes operations take longer than usual (because of slow downloading of Docker images, for example).

`STRIMZI_DEFAULT_KAFKA_IMAGE`
Optional, default `strimzi/kafka:latest`. The image name to use as the default when deploying Kafka, if no image is specified as the `Kafka.spec.kafka.image` in the Container images.

`STRIMZI_DEFAULT_KAFKA_INIT_IMAGE`
Optional, default `strimzi/kafka-init:latest`. The image name to use as default for the init container started before the broker for initial configuration work (that is, rack support), if no image is specified as the `kafka-init-image` in the Container images.

`STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE`
Optional, default `strimzi/kafka-stunnel:latest`. The image name to use as the default when deploying the sidecar container which provides TLS support for Kafka, if no image is specified as the `Kafka.spec.kafka.tlsSidecar.image` in the Container images.

`STRIMZI_DEFAULT_ZOOKEEPER_IMAGE`
Optional, default `strimzi/zookeeper:latest`. The image name to use as the default when deploying Zookeeper, if no image is specified as the `Kafka.spec.zookeeper.image` in the Container images.

`STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE`
Optional, default `strimzi/zookeeper-stunnel:latest`. The image name to use as the default when deploying the sidecar container which provides TLS support for Zookeeper, if no image is specified as the `Kafka.spec.zookeeper.tlsSidecar.image` in the Container images.

`STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE`
Optional, default `strimzi/kafka-connect:latest`. The image name to use as the default when deploying Kafka Connect, if no image is specified as the `image` in the Kafka Connect cluster ConfigMap

`STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE`
Optional, default `strimzi/kafka-connect-s2i:latest`. The image name to use as the default when deploying Kafka Connect S2I, if no image is specified as the `image` in the cluster ConfigMap.

`STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE`
Optional, default `strimzi/topic-operator:latest`. The image name to use as the default when deploying the topic operator, if no image is specified as the `Kafka.spec.entityOperator.topicOperator.image` in the Container images of the `Kafka` resource.

`STRIMZI_DEFAULT_USER_OPERATOR_IMAGE`
Optional, default `strimzi/user-operator:latest`. The image name to use as the default when deploying the user operator, if no image is specified as the `Kafka.spec.entityOperator.userOperator.image` in the Container images of the `Kafka` resource.

`STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE`
Optional, default `strimzi/entity-operator-stunnel:latest`. The image name to use as the default when deploying the sidecar container which provides TLS support for the Entity Operator, if no image is specified as the `Kafka.spec.entityOperator.tlsSidecar.image` in the Container images.

4.1.8. Role-Based Access Control (RBAC)

Provisioning Role-Based Access Control (RBAC) for the Cluster Operator

For the Cluster Operator to function it needs permission within the OpenShift or Kubernetes cluster to interact with resources such as `Kafka`, `KafkaConnect`, and so on, as well as the managed resources, such as `ConfigMaps`, `Pods`, `Deployments`, `StatefulSets`, `Services`, and so on. Such permission is described in terms of OpenShift or Kubernetes role-based access control (RBAC) resources:

- `ServiceAccount`,
- `Role` and `ClusterRole`,
- `RoleBinding` and `ClusterRoleBinding`.

In addition to running under its own `ServiceAccount` with a `ClusterRoleBinding`, the Cluster Operator manages some RBAC resources for the components that need access to OpenShift or Kubernetes resources.

OpenShift or Kubernetes also includes privilege escalation protections that prevent components operating under one `ServiceAccount` from granting other `ServiceAccounts` privileges that the granting `ServiceAccount` does not have. Because the Cluster Operator must be able to create the `ClusterRoleBindings`, and `RoleBindings` needed by resources it manages, the Cluster Operator must also have those same privileges.

Delegated privileges

When the Cluster Operator deploys resources for a desired `Kafka` resource it also creates `ServiceAccounts`, `RoleBindings`, and `ClusterRoleBindings`, as follows:

- The Kafka broker pods use a `ServiceAccount` called *`cluster-name`*`-kafka`
  - When the rack feature is used, the `strimzi-`*`cluster-name`*`-kafka-init` `ClusterRoleBinding` is used to grant this `ServiceAccount` access to the nodes within the cluster via a `ClusterRole` called `strimzi-kafka-broker`
  - When the rack feature is not used no binding is created.

- The Zookeeper pods use the default `ServiceAccount`, as they do not need access to the OpenShift or Kubernetes resources.
- The Topic Operator pod uses a `ServiceAccount` called `cluster-name-topic-operator`
  - The Topic Operator produces OpenShift or Kubernetes events with status information, so the `ServiceAccount` is bound to a `ClusterRole` called `strimzi-topic-operator` which grants this access via the `strimzi-topic-operator-role-binding` `RoleBinding`.

The pods for `KafkaConnect` and `KafkaConnectS2I` resources use the default `ServiceAccount`, as they do not require access to the OpenShift or Kubernetes resources.

`ServiceAccount`

The Cluster Operator is best run using a `ServiceAccount`:

Example `ServiceAccount` for the Cluster Operator

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
```

The `Deployment` of the operator then needs to specify this in its `spec.template.spec.serviceAccountName`:

Partial example of `Deployment` for the Cluster Operator

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: strimzi-cluster-operator
    spec:
      # ...
```

Note line 12, where the the `strimzi-cluster-operator` `ServiceAccount` is specified as the `serviceAccountName`.

`ClusterRoles`

The Cluster Operator needs to operate using `ClusterRoles` that gives access to the necessary resources. Depending on the OpenShift or Kubernetes cluster setup, a cluster administrator might be needed to create the `ClusterRoles`.

Note	Cluster administrator rights are only needed for the creation of the `ClusterRoles`. The Cluster Operator will not run under the cluster admin account.

The `ClusterRoles` follow the *principle of least privilege* and contain only those privileges needed by the Cluster Operator to operate Kafka, Kafka Connect, and Zookeeper clusters. The first set of assigned privileges allow the Cluster Operator to manage OpenShift or Kubernetes resources such as `StatefulSets`, `Deployments`, `Pods`, and `ConfigMaps`.

Cluster Operator uses ClusterRoles to grant permission at the namespace-scoped resources level and cluster-scoped resources level:

`ClusterRole` with namespaced resources for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-namespaced
  labels:
    app: strimzi
rules:
- apiGroups:
```

```
  - ""
  resources:
  - serviceaccounts
  verbs:
  - get
  - create
  - delete
  - patch
  - update
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - rolebindings
  verbs:
  - get
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - kafka.strimzi.io
  resources:
  - kafkas
  - kafkaconnects
  - kafkaconnects2is
  - kafkamirrormakers
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
  - watch
  - delete
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - get
  - list
```

```
  - watch
- apiGroups:
  - extensions
  resources:
  - deployments
  - deployments/scale
  - replicasets
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - apps
  resources:
  - deployments
  - deployments/scale
  - deployments/status
  - statefulsets
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - create
- apiGroups:
  - extensions
  resources:
  - replicationcontrollers
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - apps.openshift.io
  resources:
  - deploymentconfigs
  - deploymentconfigs/scale
  - deploymentconfigs/status
  - deploymentconfigs/finalizers
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - build.openshift.io
  resources:
  - buildconfigs
  - builds
  verbs:
  - create
  - delete
  - get
  - list
  - patch
```

```
  - watch
  - update
- apiGroups:
  - image.openshift.io
  resources:
  - imagestreams
  - imagestreams/status
  verbs:
  - create
  - delete
  - get
  - list
  - watch
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - replicationcontrollers
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - create
  - delete
  - patch
  - update
- apiGroups:
  - extensions
  resources:
  - networkpolicies
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - route.openshift.io
  resources:
  - routes
  verbs:
  - get
  - list
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - persistentvolumeclaims
  verbs:
  - get
  - list
  - create
  - delete
  - patch
  - update
```

The second includes the permissions needed for cluster-scoped resources.

`ClusterRole` with cluster-scoped resources for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-global
  labels:
    app: strimzi
rules:
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  verbs:
  - get
  - create
  - delete
  - patch
  - update
```

The `strimzi-kafka-broker` `ClusterRole` represents the access needed by the init container in Kafka pods that is used for the rack feature. As described in the Delegated privileges section, this role is also needed by the Cluster Operator in order to be able to delegate this access.

`ClusterRole` for the Cluster Operator allowing it to delegate access to OpenShift or Kubernetes nodes to the Kafka broker pods

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-kafka-broker
  labels:
    app: strimzi
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  verbs:
  - get
```

The `strimzi-topic-operator` `ClusterRole` represents the access needed by the Topic Operator. As described in the Delegated privileges section, this role is also needed by the Cluster Operator in order to be able to delegate this access.

`ClusterRole` for the Cluster Operator allowing it to delegate access to events to the Topic Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-entity-operator
  labels:
    app: strimzi
rules:
- apiGroups:
  - kafka.strimzi.io
  resources:
  - kafkatopics
  verbs:
  - get
  - list
  - watch
  - create
  - patch
  - update
  - delete
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - create
- apiGroups:
  - kafka.strimzi.io
  resources:
  - kafkausers
```

```
  verbs:
  - get
  - list
  - watch
  - create
  - patch
  - update
  - delete
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - create
  - patch
  - update
  - delete
```

`ClusterRoleBindings`

The operator needs `ClusterRoleBindings` and `RoleBindings` which associates its `ClusterRole` with its `ServiceAccount`: `ClusterRoleBindings` are needed for `ClusterRoles` containing cluster-scoped resources.

Example `ClusterRoleBinding` for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-cluster-operator-global
  apiGroup: rbac.authorization.k8s.io
```

`ClusterRoleBindings` are also needed for the `ClusterRoles` needed for delegation:

Examples `RoleBinding` for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator-kafka-broker-delegation
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-broker
  apiGroup: rbac.authorization.k8s.io
```

`ClusterRoles` containing only namespaced resources are bound using `RoleBindings` only.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
```

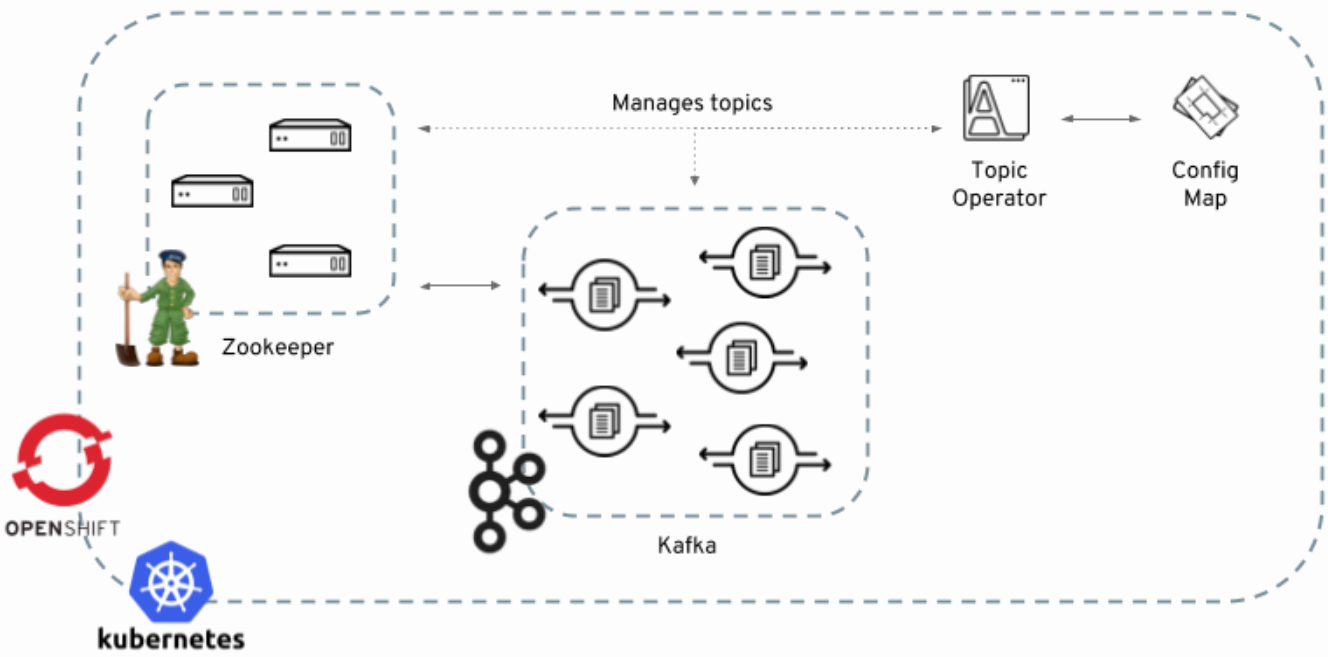
```
  kind: ClusterRole
  name: strimzi-cluster-operator-namespaced
  apiGroup: rbac.authorization.k8s.io
```

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-entity-operator-delegation
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-entity-operator
  apiGroup: rbac.authorization.k8s.io
```

4.2. Topic Operator

4.2.1. Overview of the Topic Operator component

The Topic Operator provides a way of managing topics in a Kafka cluster via OpenShift or Kubernetes resources.



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

Specifically:

- if a `KafkaTopic` is created, the operator will create the topic it describes
- if a `KafkaTopic` is deleted, the operator will delete the topic it describes
- if a `KafkaTopic` is changed, the operator will update the topic it describes

And also, in the other direction:

- if a topic is created within the Kafka cluster, the operator will create a `KafkaTopic` describing it
- if a topic is deleted from the Kafka cluster, the operator will create the `KafkaTopic` describing it
- if a topic in the Kafka cluster is changed, the operator will update the `KafkaTopic` describing it

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

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

For more details about creating, modifying and deleting topics, see Using the Topic Operator.

4.2.2. Understanding the Topic Operator

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

Complicating this, the Topic Operator might not always be able to observe changes at each end in real time (for example, the operator might be down).

To resolve this, the operator maintains its own private copy of the information about each topic. When a change happens either in the Kafka cluster, or in OpenShift or Kubernetes, it looks at both the state of the other system and at its private copy in order to determine what needs to change to keep everything in sync. The same thing happens whenever the operator starts, and periodically while it is running.

For example, suppose the Topic Operator is not running, and a `KafkaTopic` `my-topic` gets created. When the operator starts it will lack a private copy of "my-topic", so it can infer that the `KafkaTopic` has been created since it was last running. The operator will create the topic corresponding to "my-topic" and also store a private copy of the metadata for "my-topic".

The private copy allows the operator to cope with scenarios where the topic configuration gets changed both in Kafka and in OpenShift or Kubernetes, so long as the changes are not incompatible (for example, both changing the same topic config key, but to different values). In the case of incompatible changes, the Kafka configuration wins, and the `KafkaTopic` will be updated to reflect that.

The private copy is held in the same ZooKeeper ensemble used by Kafka itself. This mitigates availability concerns, because if ZooKeeper is not running then Kafka itself cannot run, so the operator will be no less available than it would even if it was stateless.

4.2.3. Deploying the Topic Operator using the Cluster Operator

Prerequisites

- A running Cluster Operator
- A `Kafka` resource to be created or updated

Procedure

1. Topic Operator can be included in the Entity Operator. Edit the `Kafka` resource ensuring it has a `Kafka.spec.entityOperator` object that configures the Entity Operator.
2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about deploying the Entity Operator, see Entity Operator.
- For more information about the `Kafka.spec.entityOperator` object used to configure the Topic Operator when deployed by the Cluster Operator, see `EntityOperatorSpec` schema reference.

4.2.4. Configuring the Topic Operator with resource requests and limits

Prerequisites

- A running Cluster Operator

Procedure

1. Edit the `Kafka` resource specifying in the `Kafka.spec.entityOperator.topicOperator.resources` property the resource requests and limits you want the Topic Operator to have.

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     # kafka and zookeeper sections...
     topicOperator:
       resources:
         request:
           cpu: "1"
           memory: 500Mi
         limit:
           cpu: "1"
           memory: 500Mi
   ```

2. Create or update the `Kafka` resource.

On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the schema of the resources object, see `Resources` schema reference.

4.2.5. Deploying the standalone Topic Operator

Deploying the Topic Operator as a standalone component is more complicated than installing it using the Cluster Operator, but is more flexible. For instance is can operate *with* any Kafka cluster, not necessarily one deployed by the Cluster Operator.

Prerequisites

- An existing Kafka cluster for the Topic Operator to connect to.

Procedure

1. Edit the `install/topic-operator/05-Deployment-strimzi-topic-operator.yaml` resource. You will need to change the following
   a. The `STRIMZI_KAFKA_BOOTSTRAP_SERVERS` environment variable in `Deployment.spec.template.spec.containers[0].env` should be set to a list of bootstrap brokers in your Kafka cluster, given as a comma-separated list of *`hostname:port`* pairs.
   b. The `STRIMZI_ZOOKEEPER_CONNECT` environment variable in `Deployment.spec.template.spec.containers[0].env` should be set to a list of the Zookeeper nodes, given as a comma-separated list of *`hostname:port`* pairs. This should be the same Zookeeper cluster that your Kafka cluster is using.
   c. The `STRIMZI_NAMESPACE` environment variable in `Deployment.spec.template.spec.containers[0].env` should be set to the OpenShift or Kubernetes namespace in which you want the operator to watch for `KafkaTopic` resources.
2. Deploy the Cluster Operator.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

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

   On Kubernetes this can be done using `kubectl describe`:

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

   On OpenShift this can be done using `oc describe`:

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

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

Note	This could take some time if you have a slow connection to the OpenShift or Kubernetes and the images have not been downloaded before.

Additional resources

- For more information about the environment variables used to configure the Topic Operator, see Topic Operator environment.
- For more information about getting the Cluster Operator to deploy the Topic Operator for you, see Deploying the Topic Operator using the Cluster Operator.

4.2.6. Topic Operator environment

When deployed standalone the Topic Operator can be configured using environment variables.

Note	The Topic Operator should be configured using the `Kafka.spec.entityOperator.topicOperator` property when deployed by the Cluster Operator.

`STRIMZI_RESOURCE_LABELS`
The label selector used to identify `KafkaTopics` to be managed by the operator.

`STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS`
The Zookeeper session timeout, in milliseconds. For example, `10000`. Default: `20000` (20 seconds).

`STRIMZI_KAFKA_BOOTSTRAP_SERVERS`
The list of Kafka bootstrap servers. This variable is mandatory.

`STRIMZI_ZOOKEEPER_CONNECT`
The Zookeeper connection information. This variable is mandatory.

`STRIMZI_FULL_RECONCILIATION_INTERVAL_MS`
The interval between periodic reconciliations, in milliseconds.

`STRIMZI_TOPIC_METADATA_MAX_ATTEMPTS`
The number of attempts for getting topics metadata from Kafka. The time between each attempt is defined as an exponential back-off. You might want to increase this value when topic creation could take more time due to its larger size (that is, many partitions/replicas). Default `6`.

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

`STRIMZI_TLS_ENABLED`
For enabling the TLS support so encrypting the communication with Kafka brokers. Default `true`.

`STRIMZI_TRUSTSTORE_LOCATION`
The path to the truststore containing certificates for enabling TLS based communication. This variable is mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.

`STRIMZI_TRUSTSTORE_PASSWORD`
The password for accessing the truststore defined by `STRIMZI_TRUSTSTORE_LOCATION`. This variable is mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.

`STRIMZI_KEYSTORE_LOCATION`
The path to the keystore containing private keys for enabling TLS based communication. This variable is mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.

`STRIMZI_KEYSTORE_PASSWORD`
The password for accessing the keystore defined by `STRIMZI_KEYSTORE_LOCATION`. This variable is mandatory only if TLS is enabled through `STRIMZI_TLS_ENABLED`.

4.3. User Operator

The User Operator provides a way of managing Kafka users via OpenShift or Kubernetes resources.

4.3.1. Overview of the User Operator component

The User Operator manages Kafka users for a Kafka cluster by watching for `KafkaUser` OpenShift or Kubernetes 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 will create the user it describes
- if a `KafkaUser` is deleted, the User Operator will delete the user it describes
- if a `KafkaUser` is changed, the User Operator will update the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the OpenShift or Kubernetes resources. Unlike the Kafka topics which might be created by applications directly in Kafka, it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator, so this should not be needed.

The User Operator allows you to declare a `KafkaUser` as part of your application's deployment. When the user is created, the credentials will be 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 rights in the `KafkaUser` declaration.

4.3.2. Deploying the User Operator using the Cluster Operator

Prerequisites

- A running Cluster Operator
- A `Kafka` resource to be created or updated.

Procedure

1. Edit the `Kafka` resource ensuring it has a `Kafka.spec.entityOperator.userOperator` object that configures the User Operator how you want.
2. Create or update the Kafka resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about the `Kafka.spec.entityOperator` object used to configure the User Operator when deployed by the Cluster Operator, see `EntityOperatorSpec` schema reference.

4.3.3. Deploying the standalone User Operator

Deploying the User Operator as a standalone component is more complicated than installing it using the Cluster Operator, but is more flexible. For instance it can operate *with* any Kafka cluster, not only the one deployed by the Cluster Operator.

Prerequisites

- An existing Kafka cluster for the User Operator to connect to.

Procedure

1. Edit the `install/user-operator/05-Deployment-strimzi-user-operator.yaml` resource. You will need to change the following

   a. The `STRIMZI_CA_NAME` environment variable in `Deployment.spec.template.spec.containers[0].env` should be set to point to an OpenShift or Kubernetes `Secret` which should contain the Certificate Authority for signing new user certificates for TLS Client Authentication. The `Secret` should contain the public key of the Certificate Authority under the key `clients-ca.crt` and the private key under `clients-ca.key`.

   b. The `STRIMZI_ZOOKEEPER_CONNECT` environment variable in `Deployment.spec.template.spec.containers[0].env` should be set to a list of the Zookeeper nodes, given as a comma-separated list of *`hostname:port`* pairs. This should be the same Zookeeper cluster that your Kafka cluster is using.

   c. The `STRIMZI_NAMESPACE` environment variable in `Deployment.spec.template.spec.containers[0].env` should be set to the OpenShift or Kubernetes namespace in which you want the operator to watch for `KafkaUser` resources.

2. Deploy the Cluster Operator.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

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

   On Kubernetes this can be done using `kubectl describe`:

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

   On OpenShift this can be done using `oc describe`:

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

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

Note	This could take some time if you have a slow connection to the OpenShift or Kubernetes and the images have not been downloaded before.

Additional resources

- For more information about getting the Cluster Operator to deploy the User Operator for you, see Deploying the User Operator using the Cluster Operator.

5. Using the Topic Operator

5.1. Topic Operator usage recommendations

- Be consistent and always operate on `KafkaTopic` resources or always operate on topics directly. Avoid routinely using both methods for a given topic.
- When creating a `KafkaTopic` resource:
  - Remember that the name cannot be changed later.
  - Choose a name for the `KafkaTopic` resource that reflects the name of the topic it describes.
  - Ideally the `KafkaTopic.metadata.name` should be the same as its `spec.topicName`. To do this, the topic name will have to be a valid Kubernetes resource name.
- When creating a topic:
  - Remember that the name cannot be changed later.
  - It is best to use a name that is a valid Kubernetes resource name, otherwise the operator will have to modify the name when creating the corresponding `KafkaTopic`.

5.2. Creating a topic

This procedure describes how to create a Kafka topic using a `KafkaTopic` OpenShift or Kubernetes resource.

Prerequisites

- A running Kafka cluster.
- A running Topic Operator.

Procedure

1. Prepare a file containing the `KafkaTopic` to be created

   An example `KafkaTopic`

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaTopic
   metadata:
     name: orders
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     partitions: 10
     replicas: 2
   ```

Note	It is recommended to use a topic name that is a valid OpenShift or Kubernetes resource name. Doing this means that it is not necessary to set the `KafkaTopic.spec.topicName` property. In any case the `KafkaTopic.spec.topicName` cannot be changed after creation.

Note	The `KafkaTopic.spec.partitions` cannot be decreased.

2. Create the `KafkaTopic` resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the schema for `KafkaTopics`, see `KafkaTopic` schema reference.
- For more information about deploying a Kafka cluster using the Cluster Operator, see Cluster Operator.
- For more information about deploying the Topic Operator using the Cluster Operator, see Deploying the Topic Operator using the Cluster Operator.
- For more information about deploying the standalone Topic Operator, see Deploying the standalone Topic Operator.

5.3. Changing a topic

This procedure describes how to change the configuration of an existing Kafka topic by using a `KafkaTopic` OpenShift or Kubernetes resource.

Prerequisites

- A running Kafka cluster.
- A running Topic Operator.
- An existing `KafkaTopic` to be changed.

Procedure

1. Prepare a file containing the desired `KafkaTopic`

   An example `KafkaTopic`

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaTopic
   metadata:
     name: orders
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     partitions: 16
     replicas: 2
   ```

Tip	You can get the current version of the resource using `oc get kafkatopic orders -o yaml`.

Note	Changing topic names using the `KafkaTopic.spec.topicName` variable and decreasing partition size using the `KafkaTopic.spec.partitions` variable is not supported by Kafka.

Caution	Increasing `spec.partitions` for topics with keys will change how records are partitioned, which can be particularly problematic when the topic uses *semantic partitioning*.

2. Update the `KafkaTopic` resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

Additional resources

- For more information about the schema for `KafkaTopics`, see `KafkaTopic` schema reference.
- For more information about deploying a Kafka cluster, see Cluster Operator.
- For more information about deploying the Topic Operator using the Cluster Operator, see Deploying the Topic Operator using the Cluster Operator.

- For more information about creating a topic using the Topic Operator, see Creating a topic.

5.4. Deleting a topic

This procedure describes how to delete a Kafka topic using a `KafkaTopic` OpenShift or Kubernetes resource.

Prerequisites

- A running Kafka cluster.
- A running Topic Operator.
- An existing `KafkaTopic` to be deleted.

Procedure

1. Delete the `KafkaTopic` resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl`:

   ```
   kubectl delete kafkatopic your-topic-name
   ```

   On OpenShift this can be done using `oc`:

   ```
   oc delete kafkatopic your-topic-name
   ```

Note	Whether the topic can actually be deleted depends on the value of the `delete.topic.enable` Kafka broker configuration, specified in the `Kafka.spec.kafka.config` property.

Additional resources

- For more information about deploying a Kafka cluster using the Cluster Operator, see Cluster Operator.
- For more information about deploying the Topic Operator using the Cluster Operator, see Deploying the Topic Operator using the Cluster Operator.
- For more information about creating a topic using the Topic Operator, see Creating a topic.

6. Using the User Operator

The User Operator provides a way of managing Kafka users via OpenShift or Kubernetes resources.

6.1. Overview of the User Operator component

The User Operator manages Kafka users for a Kafka cluster by watching for `KafkaUser` OpenShift or Kubernetes 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 will create the user it describes
- if a `KafkaUser` is deleted, the User Operator will delete the user it describes
- if a `KafkaUser` is changed, the User Operator will update the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the OpenShift or Kubernetes resources. Unlike the Kafka topics which might be created by applications directly in Kafka, it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator, so this should not be needed.

The User Operator allows you to declare a `KafkaUser` as part of your application's deployment. When the user is created, the credentials will be 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 rights in the `KafkaUser` declaration.

6.2. Mutual TLS authentication for clients

6.2.1. Mutual TLS authentication

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

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

Note	In many common uses of TLS (such as the HTTPS protocol used between a web browser and a web server) the authentication is not mutual: Only one party to the communication gets proof of the identity of the other party.

TLS authentication is more commonly one-way, where only one party authenticates to another. For example, when the HTTPS protocol is used between a web browser and a web server, the authentication is not usually mutual and only the server gets proof of the identity of the browser.

6.2.2. When to use mutual TLS authentication for clients

Mutual TLS authentication is recommended for authenticating Kafka clients when:

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

6.3. Creating a Kafka user with mutual TLS authentication

Prerequisites

- A running Kafka cluster configured with a listener using TLS authentication.
- A running User Operator.

Procedure

1. Prepare a YAML file containing the `KafkaUser` to be created.

   An example `KafkaUser`

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaUser
   metadata:
     name: my-user
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     authentication:
       type: tls
     authorization:
       type: simple
       acls:
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Read
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Describe
         - resource:
             type: group
             name: my-group
             patternType: literal
           operation: Read
   ```

2. Create the `KafkaUser` resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3. Use the credentials from the secret `my-user` in your application

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about configuring a listener that authenticates using TLS see Kafka broker listeners.
- For more information about deploying the Entity Operator, see Entity Operator.
- For more information about the `KafkaUser` object, see `KafkaUser` schema reference.

6.4. SCRAM-SHA authentication

SCRAM (Salted Challenge Response Authentication Mechanism) is an authentication protocol that can establish mutual authentication using passwords. Strimzi can configure Kafka to use SASL SCRMA-SHA-512 to provide authentication on both unencrypted and TLS-encrypted client connections. TLS authentication is always used internally between Kafka brokers and Zookeeper nodes. When used with a TLS client connection, the TLS protocol provides encryption, but is not used for authentication.

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

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

6.4.1. Supported SCRAM credentials

Strimzi supports SCRMA-SHA-512 only. When a `KafkaUser.spec.authentication.type` is configured with `scram-sha-512` the User Operator will generate a random 12 character password consisting of upper and lowercase ASCII letters and numbers.

6.4.2. When to use SCRAM-SHA authentication for clients

SCRAM-SHA is recommended for authenticating Kafka clients when:

- The client supports authentication using SCRAM-SHA-512
- It is necessary to use passwords rather than the TLS certificates
- When you want to have authentication for unencrypted communication

6.5. Creating a Kafka user with SCRAM SHA authentication

Prerequisites

- A running Kafka cluster configured with a listener using SCRAM SHA authentication.
- A running User Operator.

Procedure

1. Prepare a YAML file containing the `KafkaUser` to be created.

   An example `KafkaUser`

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Read
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Describe
      - resource:
          type: group
```

```
          name: my-group
          patternType: literal
        operation: Read
```

2. Create the `KafkaUser` resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

   On OpenShift this can be done using `oc apply`:

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

3. Use the credentials from the secret `my-user` in your application

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about configuring a listener that authenticates using SCRAM SHA see Kafka broker listeners.
- For more information about deploying the Entity Operator, see Entity Operator.
- For more information about the `KafkaUser` object, see `KafkaUser` schema reference.

6.6. Editing a Kafka user

This procedure describes how to change the configuration of an existing Kafka user by using a `KafkaUser` OpenShift or Kubernetes resource.

Prerequisites

- A running Kafka cluster.
- A running User Operator.
- An existing `KafkaUser` to be changed

Procedure

1. Prepare a YAML file containing the desired `KafkaUser`.

   An example `KafkaUser`

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaUser
   metadata:
     name: my-user
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     authentication:
       type: tls
     authorization:
       type: simple
       acls:
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Read
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Describe
         - resource:
             type: group
             name: my-group
             patternType: literal
           operation: Read
   ```

2. Update the `KafkaUser` resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl apply`:

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

On OpenShift this can be done using `oc apply`:

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

3. Use the updated credentials from the `my-user` secret in your application.

Additional resources

- For more information about deploying the Cluster Operator, see Cluster Operator.
- For more information about deploying the Entity Operator, see Entity Operator.
- For more information about the `KafkaUser` object, see `KafkaUser` schema reference.

6.7. Deleting a Kafka user

This procedure describes how to delete a Kafka user created with `KafkaUser` OpenShift or Kubernetes resource.

Prerequisites

- A running Kafka cluster.
- A running User Operator.
- An existing `KafkaUser` to be deleted.

Procedure

1. Delete the `KafkaUser` resource in OpenShift or Kubernetes.

   On Kubernetes this can be done using `kubectl`:

   ```
   kubectl delete kafkauser your-user-name
   ```

   On OpenShift this can be done using `oc`:

   ```
   oc delete kafkauser your-user-name
   ```

   Additional resources

   - For more information about deploying the Cluster Operator, see Cluster Operator.
   - For more information about the `KafkaUser` object, see `KafkaUser` schema reference.

6.8. Kafka User resource

The `KafkaUser` resource is used to declare a user with its authentication mechanism, authorization mechanism, and access rights.

6.8.1. Authentication

Authentication is configured using the `authentication` property in `KafkaUser.spec`. The authentication mechanism enabled for this user will be specified using the `type` field. Currently, the only supported authentication mechanisms are the TLS Client Authentication mechanism and the SCRAM-SHA-512 mechanism.

When no authentication mechanism is specified, User Operator will not create the user or its credentials.

TLS Client Authentication

To use TLS client authentication, set the `type` field to `tls`.

An example of `KafkaUser` with enabled TLS Client Authentication

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  # ...
```

When the user is created by the User Operator, it will create a new secret with the same name as the `KafkaUser` resource. The secret will contain a public and private key which should be used for the TLS Client Authentication. Bundled with them will be the public key of the client certification authority which was used to sign the user certificate. All keys will be in X509 format.

An example of the `Secret` with user credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: # Public key of the Clients CA
  user.crt: # Public key of the user
  user.key: # Private key of the user
```

SCRAM-SHA-512 Authentication

To use SCRAM-SHA-512 authentication mechanism, set the `type` field to `scram-sha-512`.

An example of `KafkaUser` with enabled SCRAM-SHA-512 authentication

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  # ...
```

When the user is created by the User Operator, the User Operator will create a new secret with the same name as the `KafkaUser` resource. The secret will contain the generated password.

An example of the `Secret` with user credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  password: # Generated password
```

6.8.2. Authorization

Authorization is configured using the `authorization` property in `KafkaUser.spec`. The authorization type enabled for this user will be specified using the `type` field. Currently, the only supported authorization type is the Simple authorization.

When no authorization is specified, the User Operator will not provision any access rights for the user.

Simple Authorization

To use Simple Authorization, set the `type` property to `simple`. Simple authorization is using the `SimpleAclAuthorizer` plugin. `SimpleAclAuthorizer` is the default authorization plugin which is part of Apache Kafka. Simple Authorization allows you to specify list of ACL rules in the `acls` property.

The `acls` property should contain a list of `AclRule` objects. `AclRule` specifies the access rights whcih will be granted to the user. The `AclRule` object contains following properties:

`type`
Specifies the type of the ACL rule. The type can be either `allow` or `deny`. The `type` field is optional and when not specified, the ACL rule will be treated as `allow` rule.

`operation`
Specifies the operation which will be allowed or denied. Following operations are supported:

- Read
- Write
- Delete

- Alter
- Describe
- All
- IdempotentWrite
- ClusterAction
- Create
- AlterConfigs
- DescribeConfigs

Note	Not every operation can be combined with every resource.

`host`
Specifies a remote host from which is the rule allowed or denied. Use `*` to allow or deny the operation from all hosts. The `host` field is optional and when not specified, the value `*` will be used as default.

`resource`
Specifies the resource for which does the rule apply. Simple Authorization supports 3 different resource types:

- Topics
- Consumer Groups
- Clusters

  The resource type can be specified in the `type` property. Use `topic` for Topics, `group` for Consumer Groups and `cluster` for clusters.

  Topic and Group resources additionally allow to specify the name of the resource for which the rule applies. The name can be specified in the `name` property. The name can be either specified as literal or as a prefix. To specify the name as literal, set the `patternType` property to the value `literal`. Literal names will be taken exactly as they are specified in the `name` field. To specify the name as a prefix, set the `patternType` property to the value `prefix`. Prefix type names will use the value from the `name` only a prefix and will apply the rule to all resources with names starting with the value. The cluster type resources have no name.

For more details about `SimpleAclAuthorizer`, its ACL rules and the allowed combinations of resources and operations, see Authorization and ACLs.

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

An example `KafkaUser`

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Read
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Describe
      - resource:
          type: group
          name: my-group
          patternType: prefix
        operation: Read
```

6.8.3. Additional resources

- For more information about the `KafkaUser` object, see `KafkaUser` schema reference.
- For more information about the TLS Client Authentication, see Mutual TLS authentication for clients.
- For more information about the SASL SCRAM-SHA-512 authentication, see SCRAM-SHA authentication.

7. Security

Strimzi supports encrypted communication between the Kafka and Strimzi components using the TLS protocol. Communication between Kafka brokers (interbroker communication), between Zookeeper nodes (internodal communication), and between these and the Strimzi operators is always encrypted. Communication between Kafka clients and Kafka brokers is encrypted according to how the cluster is configured. For the Kafka and Strimzi components, TLS certificates are also used for authentication.

The Cluster Operator automatically sets up TLS certificates to enable encryption and authentication within your cluster. It also sets up other TLS certificates if you want to enable encryption or TLS authentication between Kafka brokers and clients.

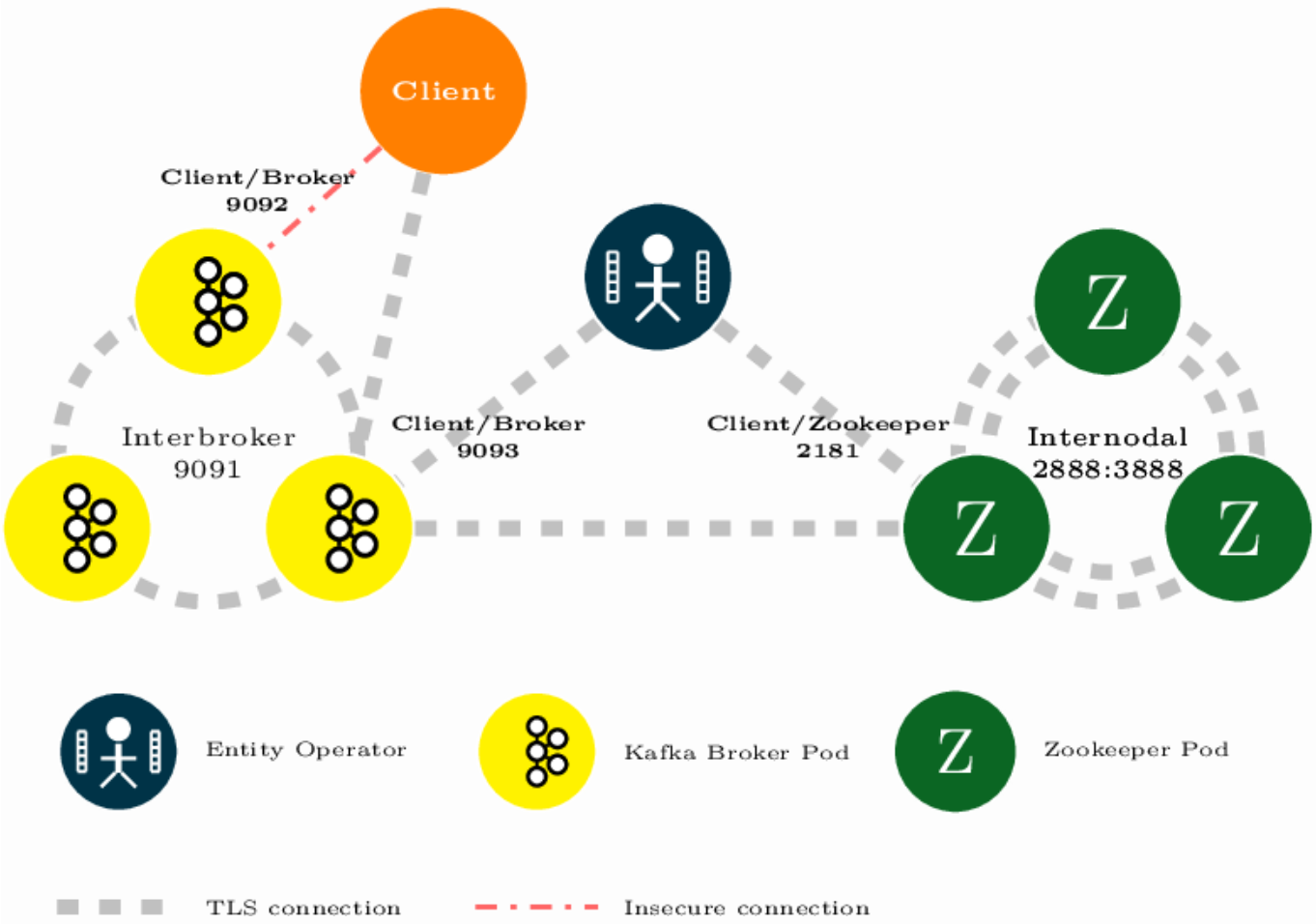


Figure 3. Example architecture diagram of the communication secured by TLS.

7.1. Certificate Authorities

To support encryption, each {ProductName] component needs its own private keys and public key certificates. All component certificates are signed by a Certificate Authority (CA) called the *cluster CA*.

Similarly, each Kafka client application connecting using TLS client authentication needs private keys and certificates. The *clients CA* is used to sign the certificates for the Kafka clients.

7.1.1. CA certificates

Each CA has a self-signed public key certificate.

Kafka brokers are configured to trust certificates signed by either the clients CA or the cluster CA. Components to which clients do not need to connect, such as Zookeeper, only trust certificates signed by the cluster CA. Client applications that perform mutual TLS authentication have to trust the certificates signed by the cluster CA.

By default, Strimzi generates and renews CA certificates automatically. You can configure the management of CA certificates in the `Kafka.spec.clusterCa` and `Kafka.spec.clientsCa` objects.

7.2. Certificates and `Secrets`

Strimzi stores CA, component and Kafka client private keys and certificates in `Secrets`. All keys are 2048 bits in size.

CA certificate validity periods, expressed as a number of days after certificate generation, can be configured in `Kafka.spec.clusterCa.validityDays` and `Kafka.spec.clusterCa.validityDays`.

7.2.1. Cluster CA `Secrets`

Table 2. Cluster CA `Secrets` managed by the Cluster Operator in *<cluster>*

`Secret` name	Field within `Secret`	Description
`<cluster>-cluster-ca`	`ca.key`	The current private key for the cluster CA.
`<cluster>-cluster-ca-cert`	`ca.crt`	The current certificate for the cluster CA.
	`ca-<date>.crt`	Former (but not yet expired) certificate for the cluster CA. `<date>` is the date the certificate will expire.
`<cluster>-kafka-brokers`	`<cluster>-kafka-<num>.crt`	Certificate for Kafka broker pod *<num>*. Signed by a current or former cluster CA private key in `<cluster>-cluster-ca`.
	`<cluster>-kafka-<num>.key`	Private key for Kafka broker pod *<num>*.
`<cluster>-zookeeper-nodes`	`<cluster>-zookeeper-<num>.crt`	Certificate for Zookeeper node *<num>*. Signed by a current or former cluster CA private key in `<cluster>-cluster-ca`.
	`<cluster>-zookeeper-<num>.key`	Private key for Zookeeper pod *<num>*.
`<cluster>-entity-operator-certs`	`entity-operator_.crt`	Certificate for TLS communication between the Entity Operator and Kafka or Zookeeper. Signed by a current or former cluster CA private key in `<cluster>-cluster-ca`.
	`entity-operator.key`	Private key for TLS communication between the Entity Operator and Kafka or Zookeeper

The CA certificates in `<cluster>-cluster-ca-cert` must be trusted by Kafka client applications so that they validate the Kafka broker certificates when connecting to Kafka brokers over TLS.

Note	Only `<cluster>-cluster-ca-cert` needs to be used by clients. All other `Secrets` in the table above only need to be accessed by the Strimzi components. You can enforce this using OpenShift or Kubernetes role-based access controls if necessary.

7.2.2. Client CA `Secrets`

Table 3. Clients CA `Secrets` managed by the Cluster Operator in *<cluster>*

`Secret` name	Field within `Secret`	Description
`<cluster>-clients-ca`	`ca.key`	The current private key for the clients CA.
`<cluster>-clients-ca-cert`	`ca.crt`	The current certificate for the clients CA.
	`ca-<date>.crt`	Former (but not yet expired) certificate for the clients CA. `<date>` is the date the certificate will expire.

The certificates in `<cluster>-clients-ca-cert` are those which the Kafka brokers trust.

Note	`<cluster>-cluster-ca` is used to sign certificates of client applications. It needs to be

	accessible to the Strimzi components and for administrative access if you are intending to issue application certificates without using the User Operator. You can enforce this using OpenShift or Kubernetes role-based access controls if necessary.

7.2.3. User `Secrets`

Table 4. `Secrets` managed by the User Operator

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

7.3. Installing your own CA certificates

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

Prerequisites

- The Cluster Operator is running.
- A `Kafka` resource within OpenShift or Kubernetes
- Your own X.509 certificates and keys in PEM format for the cluster CA or clisters CA. For example, these could be generated by `openssl`, using a command such as:

```
openssl req -x509 -new -days <validity> --nodes -out ca.crt -keyout ca.key
```

Procedure

1. Edit the `Kafka` resource for your cluster, configuring either the `Kafka.spec.clusterCa` or the `Kafka.spec.clientsCa` object to *not* use generated CAs:

   Example fragment `Kafka` resource configuring the cluster CA to use certificates you supply for yourself

   ```
   kind: Kafka
   version: v1alpha1
   spec:
     # ...
     clusterCa:
       generateCertificateAuthority: false
   ```

   This will prevent the Cluster Operator from generating a new CA. It will not disable an existing generated CA.

2. Put your CA certificate in the corresponding `Secret` (*`<cluster>`*`-cluster-ca-cert` for the cluster CA or *`<cluster>`*`-client-ca-cert` for the clients CA):

   On Kubernetes, run the following commands:

   ```
   # Delete any existing secret (ignore "Not Exists" errors)
   kubectl delete secret <ca-cert-secret>
   # Create the new one
   kubectl create secret generic <ca-cert-secret> --from-file=ca.crt=<ca-cert-file>
   ```

   On OpenShift, run the following commands:

   ```
   # Delete any existing secret (ignore "Not Exists" errors)
   oc delete secret <ca-cert-secret>
   # Create the new one
   oc create secret generic <ca-cert-secret> --from-file=ca.crt=<ca-cert-file>
   ```

3. Put your CA key in the corresponding `Secret` (*`<cluster>`*`-cluster-ca` for the cluster CA or *`<cluster>`*`-client-ca` for the clients CA)

   On Kubernetes, run the following commands:

   ```
   # Delete the existing secret
   kubectl delete secret <ca-key-secret>
   ```

```
# Create the new one
kubectl create secret generic <ca-key-secret> --from-file=ca.key=<ca-key-file>
```

On OpenShift, run the following commands:

```
# Delete the existing secret
oc delete secret <ca-key-secret>
# Create the new one
oc create secret generic <ca-key-secret> --from-file=ca.key=<ca-key-file>
```

Additional resources

- For the procedure for renewing CA certificates you have previousy installed, see Renewing your own CA certificates

7.4. Certificate renewal

The cluster CA and clients CA certificates are only valid for a limited time period, known as the validity period. This is usually defined as a number of days since the certificate was generated. For auto-generated CA certificates, you can configure the validity period in `Kafka.spec.clusterCa.validityDays` and `Kafka.spec.clientsCa.validityDays`. The default validity period for both certificates is 365 days. Manually-installed CA certificates should have their own validity period defined.

When a CA certificate expires, the certificates that it has signed will fail validation, even if they were previously valid. This means that, when replacing a CA certificate, you must also replace all other certificates signed by it. When the replacement of a CA certificate is in progress, it is necessary for peers to trust certificates signed by either the old or the new CA. This ensures the continued operation of the cluster.

To allow the renewal of CA certificates without a loss of service, the Cluster Operator will initiate certificate renewal before the old CA certificates expire. You can configure the renewal period in `Kafka.spec.clusterCa.renewalDays` and `Kafka.spec.clientsCa.renewalDays` (both default to 30 days). The renewal period is measured backwards, from the expiry date of the current certificate.

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

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

7.4.1. Renewal process with generated CAs

The Cluster Operator performs the following process to renew CA certificates:

1. Generate a new CA key and certificate. The new private key replaces the old private key in the corresponding `Secret`. The new certificate is given the name `ca.crt` within the corresponding `Secret` and the old certificate is renamed `ca-<expiry-date>.crt`.
2. Restart Zookeeper nodes so that they will trust the new CA certificate.
3. Restart Kafka brokers so that they will trust the new CA certificate.
4. Restart the Topic and User Operators so that they will trust the new CA certificate.
5. Generate new client certificates (for Zookeeper nodes, Kafka brokers, and the entity operator) signed by the new CA.
6. Perform the same restarts so that clients are using certificates signed by the new CA certificate.

At the end of the renewal period the Cluster Operator will remove the now expired CA certificates (those named `ca-<expiry-date>.crt`) from the corresponding `Secret` and perform a further round of restarts.

7.4.2. Renewal process with your own CA certificates

At the start of the renewal period the Cluster Operator will start logging at the `WARN` level that new CA certificates and keys are needed. Once you have provided the new certificates and keys, the Cluster Operator performs a further set of restarts within the Kafka cluster for which the warning was issued.

7.4.3. Client applications

The Cluster Operator is not aware of all the client applications using the Kafka cluster.

Important	Depending on how your applications are configured, you might need take action to ensure they continue working after certificate renewal.

Consider the following important points to ensure that client applications continue working.

- When they connect to the cluster, client applications must trust *all* the cluster CA certificates published in <*cluster*>-cluster-ca-certs.
- When using the User Operator to provision client certificates, client applications must use the current `user.crt` and `user.key` published in their `<user>` `Secret` when they connect to the cluster. For workloads running inside the same OpenShift or Kubernetes cluster this can be achieved by mounting the secrets as a volume and having the client Pods construct their key- and truststores from the current state of the `Secrets`. For more details on this procedure, see Configuring internal clients to trust the cluster CA.
- When renewing client certificates, if you are provisioning client certificates and keys manually, you must generate new client certificates and ensure the new certificates are used by clients within the renewal period. Failure to do this by the end of the renewal period could result in client applications being unable to connect.

7.5. Renewing your own CA certificates

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

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster in which you previously installed your own CA certificates and private keys.
- New cluster and clients X.509 certificates and keys in PEM format. These could be generated using `openssl` using a command such as:

```
openssl req -x509 -new -days <validity> --nodes -out ca.crt -keyout ca.key
```

Procedure

1. Establish what CA certificates already exist in the `Secret`:

   On Kubernetes this can be done using the following commands:

   ```
   kubectl describe secret <ca-cert-secret>
   ```

   On OpenShift this can be done using the following commands:

   ```
   oc describe secret <ca-cert-secret>
   ```

2. Prepare a directory containing the existing CA certificates in the secret.

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

   On Kubernetes for each certificate <*ca-certificate*> from the previous step, run:

   ```
   # Fetch the existing secret
   kubectl get secret <ca-cert-secret> -o 'jsonpath={.data.<ca-certificate>}' | base64 -
   ```

   On OpenShift for each certificate <*ca-certificate*> from the previous step, run:

   ```
   # Fetch the existing secret
   oc get secret <ca-cert-secret> -o 'jsonpath={.data.<ca-certificate>}' | base64 -d > <
   ```

3. Rename the old `ca.crt` file to `ca_<date>_.crt`, where <*date*> is the certificate expiry date in the format <*year*>-<*month*>-<*day*>_*T*<hour>_-<*minute*>-_<second>_Z, for example `ca-2018-09-27T17-32-00Z.crt`.

   ```
   mv ca.crt ca-$(date -u -d$(openssl x509 -enddate -noout -in ca.crt | sed 's/.*=//') +
   ```

4. Copy the new CA certificate into the directory, naming it `ca.crt`

```
cp <path-to-new-cert> ca.crt
```

5. Replace the CA certificate `Secret` (`<cluster>-cluster-ca` or `<cluster>-clients-ca`).

   On OpenShift this can be done using the following commands:

   ```
   # Delete the existing secret
   kubectl delete secret <ca-cert-secret>
   # Re-create the secret with the new private key
   kubectl create secret generic <ca-cert-secret> --from-file=.
   ```

   On OpenShift this can be done using the following commands:

   ```
   # Delete the existing secret
   oc delete secret <ca-cert-secret>
   # Re-create the secret with the new private key
   oc create secret generic <ca-cert-secret> --from-file=.
   ```

   You can now delete the directory you created:

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

6. Replace the CA key `Secret` (`<cluster>-cluster-ca` or `<cluster>-clients-ca`).

   On Kubernetes this can be done using the following commands:

   ```
   # Delete the existing secret
   kubectl delete secret <ca-key-secret>
   # Re-create the secret with the new private key
   kubectl create secret generic <ca-key-secret> --from-file=ca.key=<ca-key-file>
   ```

   On OpenShift this can be done using the following commands:

   ```
   # Delete the existing secret
   oc delete secret <ca-key-secret>
   # Re-create the secret with the new private key
   oc create secret generic <ca-key-secret> --from-file=ca.key=<ca-key-file>
   ```

7.6. TLS connections

7.6.1. Zookeeper communication

Zookeeper does not support TLS itself. By deploying an `stunnel` sidecar within every Zookeeper pod, the Cluster Operator is able to provide data encryption and authentication between Zookeeper nodes in a cluster. Zookeeper communicates only with the `stunnel` sidecar over the loopback interface. The `stunnel` sidecar then proxies all Zookeeper traffic, TLS decrypting data upon entry into a Zookeeper pod and TLS encrypting data upon departure from a Zookeeper pod.

This TLS encrypting `stunnel` proxy is instantiated from the `spec.zookeeper.stunnelImage` specified in the Kafka resource.

7.6.2. Kafka interbroker communication

Communication between Kafka brokers is done through the `REPLICATION` listener on port 9091, which is encrypted by default.

Communication between Kafka brokers and Zookeeper nodes uses an `stunnel` sidecar, as described above.

7.6.3. Topic and User Operators

```
Like the Cluster Operator, the Topic and User Operators each use an `stunnel` sidecar wh
The Topic Operator connects to Kafka brokers on port 9091.
```

7.6.4. Kafka Client connections

Encrypted communication between Kafka brokers and clients running within the same OpenShift or Kubernetes cluster is provided through the `CLIENTTLS` listener on port 9093.

Encrypted communication between Kafka brokers and clients running outside the same OpenShift or Kubernetes cluster is provided through the `EXTERNAL` listener on port 9094.

Note	You can use the `CLIENT` listener on port 9092 for unencrypted communication with brokers.

7.7. Configuring internal clients to trust the cluster CA

This procedure describes how to configure a Kafka client that resides inside the OpenShift or Kubernetes cluster — connecting to the `tls` listener on port 9093 — to trust the cluster CA certificate.

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

Prerequisites

- The Cluster Operator is running.
- A `Kafka` resource within the OpenShift or Kubernetes cluster.
- A Kafka client application inside the OpenShift or Kubernetes cluster which will connect using TLS and needs to trust the cluster CA certificate.

Procedure

1. When defining the client `Pod`
2. The Kafka client has to be configured to trust certificates signed by this CA. For the Java-based Kafka Producer, Consumer, and Streams APIs, you can do this by importing the CA certificate into the JVM's truststore using the following `keytool` command:

```
keytool -keystore client.truststore.jks -alias CARoot -import -file ca.crt
```

3. To configure the Kafka client, specify the following properties:
   - `security.protocol: SSL` when using TLS for encryption (with or without TLS authentication), or `security.protocol: SASL_SSL` when using SCRAM-SHA authentication over TLS.
   - `ssl.truststore.location`: the truststore location where the certificates were imported.
   - `ssl.truststore.password`: the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.

Additional resources

- For the procedure for configuring external clients to trust the cluster CA, see Configuring external clients to trust the cluster CA

7.8. Configuring external clients to trust the cluster CA

This procedure describes how to configure a Kafka client that resides outside the OpenShift or Kubernetes cluster – connecting to the `external` listener on port 9094 – to trust the cluster CA certificate.

You can use the same procedure to configure clients inside OpenShift or Kubernetes, which connect to the `tls` listener on port 9093, but it is usually more convenient to access the `Secrets` using a volume mount in the client `Pod`.

Follow this procedure when setting up the client and during the renewal period, when the old clients CA certificate is replaced.

Important	The `<cluster-name>-cluster-ca-cert` `Secret` will contain more than one CA certificate during CA certificate renewal. Clients must add *all* of them to their truststores.

Prerequisites

- The Cluster Operator is running.
- A `Kafka` resource within the OpenShift or Kubernetes cluster.
- A Kafka client application outside the OpenShift or Kubernetes cluster which will connect using TLS and needs to trust the cluster CA certificate.

Procedure

1. Extract the cluster CA certificates from the generated `<cluster-name>-cluster-ca-cert` `Secret`.

   On Kubernetes, run the following command to extract the certificates:

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

On OpenShift, run the following command to extract the certificates:

```
oc extract secret/<cluster-name>-cluster-ca-cert --keys ca.crt
```

Execute the same command for every `.crt` file contained in the `Secret`.

2. The Kafka client has to be configured to trust certificates signed by this CA. For the Java-based Kafka Producer, Consumer, and Streams APIs, you can do this by importing the CA certificates into the JVM's truststore using the following `keytool` command:

```
keytool -keystore client.truststore.jks -alias CARoot -import -file ca.crt
```

The same command should be executed for each of the `.crt` files extracted in the first step.

3. To configure the Kafka client, specify the following properties:
   - `security.protocol: SSL` when using TLS for encryption (with or without TLS authentication), or `security.protocol: SASL_SSL` when using SCRAM-SHA authentication over TLS.
   - `ssl.truststore.location`: the truststore location where the certificates were imported.
   - `ssl.truststore.password`: the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.

Additional resources

- For the procedure for configuring internal clients to trust the cluster CA, see Configuring internal clients to trust the cluster CA

Appendix A: Frequently Asked Questions

A.1. Cluster Operator

A.1.1. Log contains warnings about failing to acquire lock

For each cluster, the Cluster Operator always executes only one operation at a time. The Cluster Operator uses locks to make sure that there are never two parallel operations running for the same cluster. In case an operation requires more time to complete, other operations will wait until it is completed and the lock is released.

INFO
Examples of cluster operations are *cluster creation*, *rolling update*, *scale down* or *scale up* and so on.

If the wait for the lock takes too long, the operation times out and the following warning message will be printed to the log:

```
2018-03-04 17:09:24 WARNING AbstractClusterOperations:290 - Failed to acquire lock for k
```

Depending on the exact configuration of `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` and `STRIMZI_OPERATION_TIMEOUT_MS`, this warning message may appear regularly without indicating any problems. The operations which time out will be picked up by the next periodic reconciliation. It will try to acquire the lock again and execute.

Should this message appear periodically even in situations when there should be no other operations running for a given cluster, it might indicate that due to some error the lock was not properly released. In such cases it is recommended to restart the cluster operator.

Appendix B: Installing OpenShift or Kubernetes cluster

The easiest way to get started with OpenShift or Kubernetes is using the `Minikube`, `Minishift` or `oc cluster up` utilities. This section provides basic guidance on how to use them. More details are provided on the websites of the tools themselves.

B.1. Kubernetes

In order to interact with a Kubernetes cluster the `kubectl` utility needs to be installed.

The easiest way to get a running Kubernetes cluster is using `Minikube`. `Minikube` can be downloaded and installed from the Kubernetes website. Depending on the number of brokers you want to deploy inside the cluster and if you need Kafka Connect running as well, it could be worth running `Minikube` at least with 4 GB of RAM instead of the default 2 GB. Once installed, it can be started using:

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

B.2. OpenShift

In order to interact with an OpenShift cluster, the `oc` utility is needed.

An OpenShift cluster can be started in two different ways. The `oc` utility can start a cluster locally using the command:

```
oc cluster up
```

This command requires Docker to be installed. More information about this way can be found here.

Another option is to use `Minishift`. `Minishift` is an OpenShift installation within a VM. It can be downloaded and installed from the Minishift website. Depending on the number of brokers you want to deploy inside the cluster and if you need Kafka Connect running as well, it could be worth running `Minishift` at least with 4 GB of RAM instead of the default 2 GB. Once installed, `Minishift` can be started using the following command:

```
minishift start --memory 4GB
```

Appendix C: Custom Resource API Reference

C.1. `Kafka` schema reference

Field	Description
spec `KafkaSpec`	The specification of the Kafka and Zookeeper clusters, and Topic Operator.

C.2. `KafkaSpec` schema reference

Used in: `Kafka`

Field	Description
kafka `KafkaClusterSpec`	Configuration of the Kafka cluster.
zookeeper `ZookeeperClusterSpec`	Configuration of the Zookeeper cluster.
topicOperator `TopicOperatorSpec`	Configuration of the Topic Operator.
Field clientsCa `CertificateAuthority`	**Description** Configuration of the clients certificate authority.
clusterCa `CertificateAuthority`	Configuration of the cluster certificate authority.
entityOperator `EntityOperatorSpec`	Configuration of the Entity Operator.

C.3. `KafkaClusterSpec` schema reference

Used in: `KafkaSpec`

Field	Description
replicas integer	The number of pods in the cluster.
image string	The docker image for the pods.
storage `EphemeralStorage`, `PersistentClaimStorage`	Storage configuration (disk). Cannot be updated. The type depends on the value of the `storage.type` property within the given object, which must be one of [ephemeral, persistent-claim].
listeners `KafkaListeners`	Configures listeners of Kafka brokers.
authorization `KafkaAuthorizationSimple`	Authorization configuration for Kafka brokers. The type depends on the value of the `authorization.type` property within the given object, which must be one of [simple].
config map	The kafka broker config. Properties with the following prefixes cannot be set: listeners, advertised., broker., listener., host.name, port, inter.broker.listener.name, sasl., ssl., security., password., principal.builder.class, log.dir, zookeeper.connect, zookeeper.set.acl, authorizer., super.user.
rack `Rack`	Configuration of the `broker.rack` broker config.
brokerRackInitImage string	The image of the init container used for initializing the `broker.rack`.
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.
livenessProbe `Probe`	Pod liveness checking.

Field	Description
readinessProbe `Probe`	Pod readiness checking.
jvmOptions `JvmOptions`	JVM Options for pods.
resources `Resources`	Resource constraints (limits and requests).
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
logging `InlineLogging` , `ExternalLogging`	Logging configuration for Kafka. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].
tlsSidecar `TlsSidecar`	TLS sidecar configuration.

C.4. `EphemeralStorage` schema reference

Used in: `KafkaClusterSpec` , `ZookeeperClusterSpec`

The `type` property is a discriminator that distinguishes the use of the type `EphemeralStorage` from `PersistentClaimStorage` . It must have the value `ephemeral` for the type `EphemeralStorage` .

Field	Description
type string	Must be `ephemeral` .

C.5. `PersistentClaimStorage` schema reference

Used in: `KafkaClusterSpec` , `ZookeeperClusterSpec`

The `type` property is a discriminator that distinguishes the use of the type `PersistentClaimStorage` from `EphemeralStorage` . It must have the value `persistent-claim` for the type `PersistentClaimStorage` .

Field	Description
type string	Must be `persistent-claim` .
size string	When type=persistent-claim, defines the size of the persistent volume claim (i.e 1Gi). Mandatory when type=persistent-claim.

Field	Description
selector	Specifies a specific persistent volume to use. It contains a matchLabels field which defines an inner JSON object with key:value representing labels for selecting such a volume.
map	
deleteClaim	Specifies if the persistent volume claim has to be deleted when the cluster is un-deployed.
boolean	
class	The storage class to use for dynamic volume allocation.
string	

C.6. `KafkaListeners` schema reference

Used in: `KafkaClusterSpec`

Field	Description
plain	Configures plain listener on port 9092.
`KafkaListenerPlain`	
tls	Configures TLS listener on port 9093.
`KafkaListenerTls`	
external	Configures external listener on port 9094. The type depends on the value of the `external.type` property within the given object, which must be one of [route, loadbalancer, nodeport].
`KafkaListenerExternalRoute`, `KafkaListenerExternalLoadBalancer`, `KafkaListenerExternalNodePort`	

C.7. `KafkaListenerPlain` schema reference

Used in: `KafkaListeners`

Field	Description
authentication	Authentication configuration for this listener. Since this listener does not use TLS transport you cannot configure an authentication with `type: tls`. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
`KafkaListenerAuthenticationTls`, `KafkaListenerAuthenticationScramSha512`	

C.8. `KafkaListenerAuthenticationTls` schema reference

Used in: `KafkaListenerExternalLoadBalancer`, `KafkaListenerExternalNodePort`, `KafkaListenerExternalRoute`, `KafkaListenerPlain`, `KafkaListenerTls`

The `type` property is a discriminator that distinguishes the use of the type `KafkaListenerAuthenticationTls` from `KafkaListenerAuthenticationScramSha512`. It must have the value `tls` for the type `KafkaListenerAuthenticationTls`.

Field	Description

Field	Description
type string	Must be `tls`.

C.9. `KafkaListenerAuthenticationScramSha512` schema reference

Used in: `KafkaListenerExternalLoadBalancer`, `KafkaListenerExternalNodePort`, `KafkaListenerExternalRoute`, `KafkaListenerPlain`, `KafkaListenerTls`

The `type` property is a discriminator that distinguishes the use of the type `KafkaListenerAuthenticationScramSha512` from `KafkaListenerAuthenticationTls`. It must have the value `scram-sha-512` for the type `KafkaListenerAuthenticationScramSha512`.

Field	Description
type string	Must be `scram-sha-512`.

C.10. `KafkaListenerTls` schema reference

Used in: `KafkaListeners`

Field	Description
authentication `KafkaListenerAuthenticationTls`, `KafkaListenerAuthenticationScramSha512`	Authentication configuration for this listener. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].

C.11. `KafkaListenerExternalRoute` schema reference

Used in: `KafkaListeners`

The `type` property is a discriminator that distinguishes the use of the type `KafkaListenerExternalRoute` from `KafkaListenerExternalLoadBalancer`, `KafkaListenerExternalNodePort`. It must have the value `route` for the type `KafkaListenerExternalRoute`.

Field	Description
type string	Must be `route`.
Field	**Description**
authentication `KafkaListenerAuthenticationTls`, `KafkaListenerAuthenticationScramSha512`	Authentication configuration for Kafka brokers. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].

C.12. `KafkaListenerExternalLoadBalancer` schema reference

Used in: `KafkaListeners`

The `type` property is a discriminator that distinguishes the use of the type `KafkaListenerExternalLoadBalancer` from `KafkaListenerExternalRoute`, `KafkaListenerExternalNodePort`. It must have the value `loadbalancer` for the type `KafkaListenerExternalLoadBalancer`.

Field	Description
type	Must be `loadbalancer`.
string	
authentication	Authentication configuration for Kafka brokers. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
`KafkaListenerAuthenticationTls`, `KafkaListenerAuthenticationScramSha512`	
tls	Enables TLS encryption on the listener. By default set to `true` for enabled TLS encryption.
boolean	

C.13. `KafkaListenerExternalNodePort` schema reference

Used in: `KafkaListeners`

The `type` property is a discriminator that distinguishes the use of the type `KafkaListenerExternalNodePort` from `KafkaListenerExternalRoute`, `KafkaListenerExternalLoadBalancer`. It must have the value `nodeport` for the type `KafkaListenerExternalNodePort`.

Field	Description
type	Must be `nodeport`.
string	
authentication	Authentication configuration for Kafka brokers. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
`KafkaListenerAuthenticationTls`, `KafkaListenerAuthenticationScramSha512`	
tls	Enables TLS encryption on the listener. By default set to `true` for enabled TLS encryption.
boolean	

C.14. `KafkaAuthorizationSimple` schema reference

Used in: `KafkaClusterSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaAuthorizationSimple` from other subtypes which may be added in the future. It must have the value `simple` for the type `KafkaAuthorizationSimple`.

Field	Description
type	Must be `simple`.
string	
superUsers	List of super users. Should contain list of user principals which should get unlimited access rights.
string array	

C.15. `Rack` schema reference

Used in: `KafkaClusterSpec`

Field	Description
topologyKey string	A key that matches labels assigned to the OpenShift or Kubernetes cluster nodes. The value of the label is used to set the broker's `broker.rack` config.

C.16. `Probe` schema reference

Used in: `KafkaClusterSpec`, `KafkaConnectS2ISpec`, `KafkaConnectSpec`, `ZookeeperClusterSpec`

Field	Description
initialDelaySeconds integer	The initial delay before first the health is first checked.
timeoutSeconds integer	The timeout for each attempted health check.

C.17. `JvmOptions` schema reference

Used in: `KafkaClusterSpec`, `KafkaConnectS2ISpec`, `KafkaConnectSpec`, `KafkaMirrorMakerSpec`, `ZookeeperClusterSpec`

Field	Description
-XX map	A map of -XX options to the JVM.
-Xms string	-Xms option to to the JVM.
-Xmx string	-Xmx option to to the JVM.

C.18. `Resources` schema reference

Used in: `EntityTopicOperatorSpec`, `EntityUserOperatorSpec`, `KafkaClusterSpec`, `KafkaConnectS2ISpec`, `KafkaConnectSpec`, `KafkaMirrorMakerSpec`, `TlsSidecar`, `TopicOperatorSpec`, `ZookeeperClusterSpec`

Field	Description
limits `CpuMemory`	Resource limits applied at runtime.
requests `CpuMemory`	Resource requests applied during pod scheduling.

C.19. `CpuMemory` schema reference

Used in: `Resources`

Field	Description
cpu	CPU.
string	
memory	Memory.
string	

C.20. `InlineLogging` schema reference

Used in: `EntityTopicOperatorSpec`, `EntityUserOperatorSpec`, `KafkaClusterSpec`, `KafkaConnectS2ISpec`, `KafkaConnectSpec`, `KafkaMirrorMakerSpec`, `TopicOperatorSpec`, `ZookeeperClusterSpec`

The `type` property is a discriminator that distinguishes the use of the type `InlineLogging` from `ExternalLogging`. It must have the value `inline` for the type `InlineLogging`.

Field	Description
type	Must be `inline`.
string	
loggers	A Map from logger name to logger level.
map	

C.21. `ExternalLogging` schema reference

Used in: `EntityTopicOperatorSpec`, `EntityUserOperatorSpec`, `KafkaClusterSpec`, `KafkaConnectS2ISpec`, `KafkaConnectSpec`, `KafkaMirrorMakerSpec`, `TopicOperatorSpec`, `ZookeeperClusterSpec`

The `type` property is a discriminator that distinguishes the use of the type `ExternalLogging` from `InlineLogging`. It must have the value `external` for the type `ExternalLogging`.

Field	Description
type	Must be `external`.
string	
name	The name of the `ConfigMap` from which to get the logging configuration.
string	

C.22. `TlsSidecar` schema reference

Used in: `EntityOperatorSpec`, `KafkaClusterSpec`, `TopicOperatorSpec`, `ZookeeperClusterSpec`

Field	Description
image	The docker image for the container.

Field	Description
string	
logLevel string (one of [emerg, debug, crit, err, alert, warning, notice, info])	The log level for the TLS sidecar.Default value is `notice`.
resources `Resources`	Resource constraints (limits and requests).

C.23. `ZookeeperClusterSpec` schema reference

Used in: `KafkaSpec`

Field	Description
replicas integer	The number of pods in the cluster.
image string	The docker image for the pods.
storage `EphemeralStorage`, `PersistentClaimStorage`	Storage configuration (disk). Cannot be updated. The type depends on the value of the `storage.type` property within the given object, which must be one of [ephemeral, persistent-claim].
config map	The zookeeper broker config. Properties with the following prefixes cannot be set: server., dataDir, dataLogDir, clientPort, authProvider, quorum.auth, requireClientAuthScheme.
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.
livenessProbe `Probe`	Pod liveness checking.
readinessProbe `Probe`	Pod readiness checking.

Field	Description
jvmOptions `JvmOptions`	JVM Options for pods.
resources `Resources`	Resource constraints (limits and requests).
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
logging `InlineLogging` , `ExternalLogging`	Logging configuration for Zookeeper. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].
tlsSidecar `TlsSidecar`	TLS sidecar configuration.

C.24. `TopicOperatorSpec` schema reference

Used in: `KafkaSpec`

Field	Description
watchedNamespace string	The namespace the Topic Operator should watch.
image string	The image to use for the Topic Operator.
reconciliationIntervalSeconds integer	Interval between periodic reconciliations.
zookeeperSessionTimeoutSeconds integer	Timeout for the Zookeeper session.
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
resources `Resources`	Resource constraints (limits and requests).
topicMetadataMaxAttempts	The number of attempts at getting topic metadata.

Field	Description
integer	
tlsSidecar `TlsSidecar`	TLS sidecar configuration.
logging `InlineLogging`, `ExternalLogging`	Logging configuration. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].

C.25. `CertificateAuthority` schema reference

Used in: `KafkaSpec`

Configuration of how TLS certificates are used within the cluster.This applies to certificates used for both internal communication within the cluster and to certificates used for client access via `Kafka.spec.kafka.listeners.tls`.

Field	Description
generateCertificateAuthority boolean	If true then Certificate Authority certificates will be generated automatically. Otherwise the user will need to provide a Secret with the CA certificate. Default is true.
validityDays integer	The number of days generated certificates should be valid for. Default is 365.
renewalDays integer	The number of days in the certificate renewal period. This is the number of days before the a certificate expires during which renewal actions may be performed.When `generateCertificateAuthority` is true, this will cause the generation of a new certificate. When `generateCertificateAuthority` is true, this will cause extra logging at WARN level about the pending certificate expiry. Default is 30.

C.26. `EntityOperatorSpec` schema reference

Used in: `KafkaSpec`

Field	Description
topicOperator `EntityTopicOperatorSpec`	Configuration of the Topic Operator.
userOperator `EntityUserOperatorSpec`	Configuration of the User Operator.
affinity	Pod affinity rules.See external documentation of core/v1 affinity.

Field	Description
Affinity	
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.
tlsSidecar `TlsSidecar`	TLS sidecar configuration.

C.27. `EntityTopicOperatorSpec` schema reference

Used in: `EntityOperatorSpec`

Field	Description
watchedNamespace string	The namespace the Topic Operator should watch.
image string	The image to use for the Topic Operator.
reconciliationIntervalSeconds integer	Interval between periodic reconciliations.
zookeeperSessionTimeoutSeconds integer	Timeout for the Zookeeper session.
resources `Resources`	Resource constraints (limits and requests).
topicMetadataMaxAttempts integer	The number of attempts at getting topic metadata.
logging `InlineLogging`, `ExternalLogging`	Logging configuration. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].

C.28. `EntityUserOperatorSpec` schema reference

Used in: `EntityOperatorSpec`

Field	Description
watchedNamespace string	The namespace the User Operator should watch.

Field	Description
image string	The image to use for the User Operator.
reconciliationIntervalSeconds integer	Interval between periodic reconciliations.
zookeeperSessionTimeoutSeconds integer	Timeout for the Zookeeper session.
resources `Resources`	Resource constraints (limits and requests).
logging `InlineLogging`, `ExternalLogging`	Logging configuration. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].

C.29. `KafkaConnect` schema reference

Field	Description
spec `KafkaConnectSpec`	The specification of the Kafka Connect deployment.

C.30. `KafkaConnectSpec` schema reference

Used in: `KafkaConnect`

Field	Description
replicas integer	The number of pods in the Kafka Connect group.
image string	The docker image for the pods.
livenessProbe `Probe`	Pod liveness checking.
readinessProbe `Probe`	Pod readiness checking.
jvmOptions	JVM Options for pods.

Field	Description
`JvmOptions`	
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.
logging `InlineLogging`, `ExternalLogging`	Logging configuration for Kafka Connect. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
authentication `KafkaConnectAuthenticationTls`, `KafkaConnectAuthenticationScramSha512`	Authentication configuration for Kafka Connect. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
bootstrapServers string	Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>*:*<port>* pairs.
config map	The Kafka Connect configuration. Properties with the following prefixes cannot be set: ssl., sasl., security., listeners, plugin.path, rest., bootstrap.servers.
resources `Resources`	Resource constraints (limits and requests).
tls `KafkaConnectTls`	TLS configuration.

C.31. `KafkaConnectAuthenticationTls` schema reference

Used in: `KafkaConnectS2ISpec`, `KafkaConnectSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaConnectAuthenticationTls` from `KafkaConnectAuthenticationScramSha512`. It must have the value `tls` for the type `KafkaConnectAuthenticationTls`.

Field	Description
certificateAndKey `CertAndKeySecretSource`	Certificate and private key pair for TLS authentication.

Field	Description
type string	Must be `tls`.

C.32. `CertAndKeySecretSource` schema reference

Used in: `KafkaConnectAuthenticationTls`, `KafkaMirrorMakerAuthenticationTls`

Field	Description
certificate string	The name of the file certificate in the Secret.
key string	The name of the private key in the Secret.
secretName string	The name of the Secret containing the certificate.

C.33. `KafkaConnectAuthenticationScramSha512` schema reference

Used in: `KafkaConnectS2ISpec`, `KafkaConnectSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaConnectAuthenticationScramSha512` from `KafkaConnectAuthenticationTls`. It must have the value `scram-sha-512` for the type `KafkaConnectAuthenticationScramSha512`.

Field	Description
passwordSecret `PasswordSecretSource`	Password used for the authentication.
type string	Must be `scram-sha-512`.
username string	Username used for the authentication.

Field	Description

C.34. `PasswordSecretSource` schema reference

Used in: `KafkaConnectAuthenticationScramSha512`, `KafkaMirrorMakerAuthenticationScramSha512`

Field	Description
password string	The name of the key in the Secret under which the password is stored.
secretName	The name of the Secret containing the password.

Field	Description
string	

C.35. `KafkaConnectTls` schema reference

Used in: `KafkaConnectS2ISpec`, `KafkaConnectSpec`

Field	Description
trustedCertificates	Trusted certificates for TLS connection.
`CertSecretSource` array	

C.36. `CertSecretSource` schema reference

Used in: `KafkaConnectTls`, `KafkaMirrorMakerTls`

Field	Description
certificate	The name of the file certificate in the Secret.
string	
secretName	The name of the Secret containing the certificate.
string	

C.37. `KafkaConnectS2I` schema reference

Field	Description
spec	The specification of the Kafka Connect deployment.
`KafkaConnectS2ISpec`	

C.38. `KafkaConnectS2ISpec` schema reference

Used in: `KafkaConnectS2I`

Field	Description
replicas	The number of pods in the Kafka Connect group.
integer	
image	The docker image for the pods.
string	
livenessProbe	Pod liveness checking.
`Probe`	
readinessProbe	Pod readiness checking.

Field	Description
`Probe`	
jvmOptions	JVM Options for pods.
`JvmOptions`	
affinity	Pod affinity rules.See external documentation of core/v1 affinity.
Affinity	
metrics	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
map	
authentication	Authentication configuration for Kafka Connect. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
`KafkaConnectAuthenticationTls`, `KafkaConnectAuthenticationScramSha512`	
bootstrapServers	Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>*:*<port>* pairs.
string	
config	The Kafka Connect configuration. Properties with the following prefixes cannot be set: ssl., sasl., security., listeners, plugin.path, rest., bootstrap.servers.
map	
insecureSourceRepository	When true this configures the source repository with the 'Local' reference policy and an import policy that accepts insecure source tags.
boolean	
logging	Logging configuration for Kafka Connect. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].
`InlineLogging`, `ExternalLogging`	
resources	Resource constraints (limits and requests).
`Resources`	
tls	TLS configuration.
`KafkaConnectTls`	
tolerations	Pod's tolerations.See external documentation of core/v1 tolerations.
Toleration array	

C.39. `KafkaTopic` schema reference

Field	Description

Field	Description
spec	The specification of the topic.
`KafkaTopicSpec`	

C.40. `KafkaTopicSpec` schema reference

Used in: `KafkaTopic`

Field	Description
partitions	The number of partitions the topic should have. This cannot be decreased after topic creation. It can be increased after topic creation, but it is important to understand the consequences that has, especially for topics with semantic partitioning. If unspecified this will default to the broker's `num.partitions` config.
integer	
replicas	The number of replicas the topic should have. If unspecified this will default to the broker's `default.replication.factor` config.
integer	
config	The topic configuration.
map	
topicName	The name of the topic. When absent this will default to the metadata.name of the topic. It is recommended to not set this unless the topic name is not a valid Kubernetes resource name.
string	

C.41. `KafkaUser` schema reference

Field	Description
spec	The specification of the user.
`KafkaUserSpec`	

C.42. `KafkaUserSpec` schema reference

Used in: `KafkaUser`

Field	Description
authentication	Authentication mechanism enabled for this Kafka user. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
`KafkaUserTlsClientAuthentication`, `KafkaUserScramSha512ClientAuthentication`	
authorization	Authorization rules for this Kafka user. The type depends on the value of the `authorization.type` property within the given object, which must be one of [simple].
`KafkaUserAuthorizationSimple`	

C.43. `KafkaUserTlsClientAuthentication` schema reference

Used in: `KafkaUserSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaUserTlsClientAuthentication` from `KafkaUserScramSha512ClientAuthentication`. It must have the value `tls` for the type `KafkaUserTlsClientAuthentication`.

Field	Description
type string	Must be `tls`.

C.44. `KafkaUserScramSha512ClientAuthentication` schema reference

Used in: `KafkaUserSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaUserScramSha512ClientAuthentication` from `KafkaUserTlsClientAuthentication`. It must have the value `scram-sha-512` for the type `KafkaUserScramSha512ClientAuthentication`.

Field	Description
type string	Must be `scram-sha-512`.

C.45. `KafkaUserAuthorizationSimple` schema reference

Used in: `KafkaUserSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaUserAuthorizationSimple` from other subtypes which may be added in the future. It must have the value `simple` for the type `KafkaUserAuthorizationSimple`.

Field	Description
type string	Must be `simple`.
acls `AclRule` array	List of ACL rules which should be applied to this user.

C.46. `AclRule` schema reference

Used in: `KafkaUserAuthorizationSimple`

Field	Description
host string	The host from which the action described in the ACL rule is allowed or denied.
operation string (one of [Read, Write, Delete, Alter, Describe, All, IdempotentWrite, ClusterAction, Create, AlterConfigs, DescribeConfigs])	Operation which will be allowed or denied. Supported operations are: Read, Write, Create, Delete, Alter, Describe, ClusterAction, AlterConfigs, DescribeConfigs, IdempotentWrite and All.

Field	Description
resource `AclRuleTopicResource`, `AclRuleGroupResource`, `AclRuleClusterResource`	Indicates the resource for which given ACL rule applies. The type depends on the value of the `resource.type` property within the given object, which must be one of [topic, group, cluster].
type string (one of [allow, deny])	The type of the rule.Currently the only supported type is `allow`.ACL rules with type `allow` are used to allow user to execute the specified operations. Default value is `allow`.

C.47. `AclRuleTopicResource` schema reference

Used in: `AclRule`

The `type` property is a discriminator that distinguishes the use of the type `AclRuleTopicResource` from `AclRuleGroupResource`, `AclRuleClusterResource`. It must have the value `topic` for the type `AclRuleTopicResource`.

Field	Description
type string	Must be `topic`.
name string	Name of resource for which given ACL rule applies. Can be combined with `patternType` field to use prefix pattern.
patternType string (one of [prefix, literal])	Describes the pattern used in the resource field. The supported types are `literal` and `prefix`. With `literal` pattern type, the resource field will be used as a definition of a full topic name. With `prefix` pattern type, the resource name will be used only as a prefix. Default value is `literal`.

C.48. `AclRuleGroupResource` schema reference

Used in: `AclRule`

The `type` property is a discriminator that distinguishes the use of the type `AclRuleGroupResource` from `AclRuleTopicResource`, `AclRuleClusterResource`. It must have the value `group` for the type `AclRuleGroupResource`.

Field	Description
type string	Must be `group`.
name string	Name of resource for which given ACL rule applies. Can be combined with `patternType` field to use prefix pattern.
patternType	Describes the pattern used in the resource field. The supported types are `literal` and `prefix`. With `literal` pattern type, the resource field will be used as a definition of a

Field	Description
string (one of [prefix, literal])	full topic name. With `prefix` pattern type, the resource name will be used only as a prefix. Default value is `literal`.

C.49. `AclRuleClusterResource` schema reference

Used in: `AclRule`

The `type` property is a discriminator that distinguishes the use of the type `AclRuleClusterResource` from `AclRuleTopicResource`, `AclRuleGroupResource`. It must have the value `cluster` for the type `AclRuleClusterResource`.

Field	Description
type	Must be `cluster`.
string	

C.50. `KafkaMirrorMaker` schema reference

Field	Description
spec	The specification of the mirror maker.
`KafkaMirrorMakerSpec`	

C.51. `KafkaMirrorMakerSpec` schema reference

Used in: `KafkaMirrorMaker`

Field	Description
replicas	The number of pods in the `Deployment`.
integer	
image	The docker image for the pods.
string	
whitelist	List of topics which are included for mirroring. This option allows any regular expression using Java-style regular expressions.Mirroring two topics named A and B can be achieved by using the whitelist `'A|B'`. Or, as a special case, you can mirror all topics using the whitelist '*'. Multiple regular expressions separated by commas can be specified as well.
string	
consumer	Configuration of source cluster.
`KafkaMirrorMakerConsumerSpec`	
producer	Configuration of target cluster.
`KafkaMirrorMakerProducerSpec`	

Field	Description
resources `Resources`	Resource constraints (limits and requests).
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.
jvmOptions `JvmOptions`	JVM Options for pods.
logging `InlineLogging`, `ExternalLogging`	Logging configuration for Mirror Maker. The type depends on the value of the `logging.type` property within the given object, which must be one of [inline, external].
metrics map	The Prometheus JMX Exporter configuration. See JMX Exporter documentation for details of the structure of this configuration.

C.52. `KafkaMirrorMakerConsumerSpec` schema reference

Used in: `KafkaMirrorMakerSpec`

Field	Description
numStreams integer	Specifies the number of consumer stream threads to create.
groupId string	A unique string that identifies the consumer group this consumer belongs to.
bootstrapServers string	A list of host:port pairs to use for establishing the initial connection to the Kafka cluster.
authentication `KafkaMirrorMakerAuthenticationTls`, `KafkaMirrorMakerAuthenticationScramSha512`	Authentication configuration for connecting to the cluster. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
config map	The mirror maker consumer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, group.id, sasl., security.
tls	TLS configuration for connecting to the cluster.

Field	Description
`KafkaMirrorMakerTls`	

C.53. `KafkaMirrorMakerAuthenticationTls` schema reference

Used in: `KafkaMirrorMakerConsumerSpec`, `KafkaMirrorMakerProducerSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaMirrorMakerAuthenticationTls` from `KafkaMirrorMakerAuthenticationScramSha512`. It must have the value `tls` for the type `KafkaMirrorMakerAuthenticationTls`.

Field	Description
certificateAndKey `CertAndKeySecretSource`	Reference to the `Secret` which holds the certificate and private key pair.
type string	Must be `tls`.

C.54. `KafkaMirrorMakerAuthenticationScramSha512` schema reference

Used in: `KafkaMirrorMakerConsumerSpec`, `KafkaMirrorMakerProducerSpec`

The `type` property is a discriminator that distinguishes the use of the type `KafkaMirrorMakerAuthenticationScramSha512` from `KafkaMirrorMakerAuthenticationTls`. It must have the value `scram-sha-512` for the type `KafkaMirrorMakerAuthenticationScramSha512`.

Field	Description
passwordSecret `PasswordSecretSource`	Reference to the `Secret` which holds the password.
type string	Must be `scram-sha-512`.
username string	Username used for the authentication.

C.55. `KafkaMirrorMakerTls` schema reference

Used in: `KafkaMirrorMakerConsumerSpec`, `KafkaMirrorMakerProducerSpec`

Field	Description
trustedCertificates `CertSecretSource` array	Trusted certificates for TLS connection.

C.56. `KafkaMirrorMakerProducerSpec` schema reference

Used in: `KafkaMirrorMakerSpec`

Field	Description

Field	Description
bootstrapServers string	A list of host:port pairs to use for establishing the initial connection to the Kafka cluster.
authentication `KafkaMirrorMakerAuthenticationTls`, `KafkaMirrorMakerAuthenticationScramSha512`	Authentication configuration for connecting to the cluster. The type depends on the value of the `authentication.type` property within the given object, which must be one of [tls, scram-sha-512].
config map	The mirror maker producer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, sasl., security.
tls `KafkaMirrorMakerTls`	TLS configuration for connecting to the cluster.

Appendix D: Metrics

This section describes how to monitor Strimzi Kafka and ZooKeeper clusters using Grafana dashboards. In order to run the example dashboards you must configure Prometheus server and add the appropriate Prometheus JMX Exporter rules to your Kafka cluster resource.

Warning	The resources referenced in this section serve as a good starting point for setting up monitoring, but they are provided as an example only. If you require further support on configuration and running Prometheus or Grafana in production then please reach out to their respective communities.

When adding Prometheus and Grafana servers to an Apache Kafka deployment using `minikube` or `minishift`, the memory available to the virtual machine should be increased (to 4 GB of RAM, for example, instead of the default 2 GB). Information on how to increase the default amount of memory can be found in the following section Installing OpenShift or Kubernetes cluster.

D.1. Kafka Metrics Configuration

Strimzi uses the Prometheus JMX Exporter to export JMX metrics from Kafka and ZooKeeper to a Prometheus HTTP metrics endpoint that is scraped by Prometheus server. The Grafana dashboard relies on the Kafka and ZooKeeper Prometheus JMX Exporter relabeling rules defined in the example `Kafka` resource configuration in `kafka-metrics.yaml`. Copy this configuration to your own `Kafka` resource definition, or run this example, in order to use the provided Grafana dashboards.

D.1.1. Deploying on OpenShift

To deploy the example Kafka cluster the following command should be executed:

```
oc apply -f https://raw.githubusercontent.com/strimzi/strimzi-kafka-operator/0.8.0/metrics/examples/kafka/kafka-metrics.yaml
```

D.1.2. Deploying on Kubernetes

To deploy the example Kafka cluster the following command should be executed:

```
kubectl apply -f https://raw.githubusercontent.com/strimzi/strimzi-kafka-operator/0.8.0/metrics/examples/kafka/kafka-metrics.yaml
```

D.2. Prometheus

The provided Prometheus `kubernetes.yaml` YAML file describes all the resources required by Prometheus in order to effectively monitor a Strimzi Kafka & ZooKeeper cluster. These resources lack important production configuration to run a healthy and highly available Prometheus server. They should only be used to demonstrate this Grafana dashboard example.

The following resources are defined:

- A `ClusterRole` that grants permissions to read Prometheus health endpoints of the Kubernetes system, including cAdvisor and kubelet for container metrics. The Prometheus server configuration uses the Kubernetes service discovery feature in order to discover the pods in the cluster from which it gets metrics. In order to have this feature working, it is necessary for the service account used for running the Prometheus service pod to have access to the API server to get the pod list.
- A `ServiceAccount` for the Prometheus pods to run under.
- A `ClusterRoleBinding` which binds the aforementioned `ClusterRole` to the `ServiceAccount`.
- A `Deployment` to manage the actual Prometheus server pod.
- A `ConfigMap` to manage the configuration of Prometheus Server.
- A `Service` to provide an easy to reference hostname for other services to connect to Prometheus server (such as Grafana).

D.2.1. Deploying on OpenShift

To deploy all these resources you can run the following. Note that this file creates a `ClusterRoleBinding` in the `myproject` namespace. If you're not using this namespace then download the resource file locally and update it.

```
oc login -u system:admin
oc apply -f https://raw.githubusercontent.com/strimzi/strimzi-kafka-operator/0.8.0/metrics/examples/prometheus/kubernetes.yaml
```

D.2.2. Deploying on Kubernetes

To deploy all these resources you can run the following. Note that this file creates a `ClusterRoleBinding` in the `myproject` namespace. If you're not using this namespace then download the resource file locally and update it.

```
kubectl apply -f https://raw.githubusercontent.com/strimzi/strimzi-kafka-operator/0.8.0/metrics/examples/prometheus/kubernetes.yaml
```

D.3. Grafana

A Grafana server is necessary to get a visualisation of the Prometheus metrics. The source for the Grafana docker image used can be found in the `./metrics/examples/grafana/grafana-openshift` directory.

D.3.1. Deploying on OpenShift

To deploy Grafana the following commands should be executed:

```
oc apply -f https://raw.githubusercontent.com/strimzi/strimzi-kafka-operator/0.8.0/metrics/examples/grafana/kubernetes.yaml
```

D.3.2. Deploying on Kubernetes

To deploy Grafana the following commands should be executed:

```
kubectl apply -f https://raw.githubusercontent.com/strimzi/strimzi-kafka-operator/0.8.0/metrics/examples/grafana/kubernetes.yaml
```

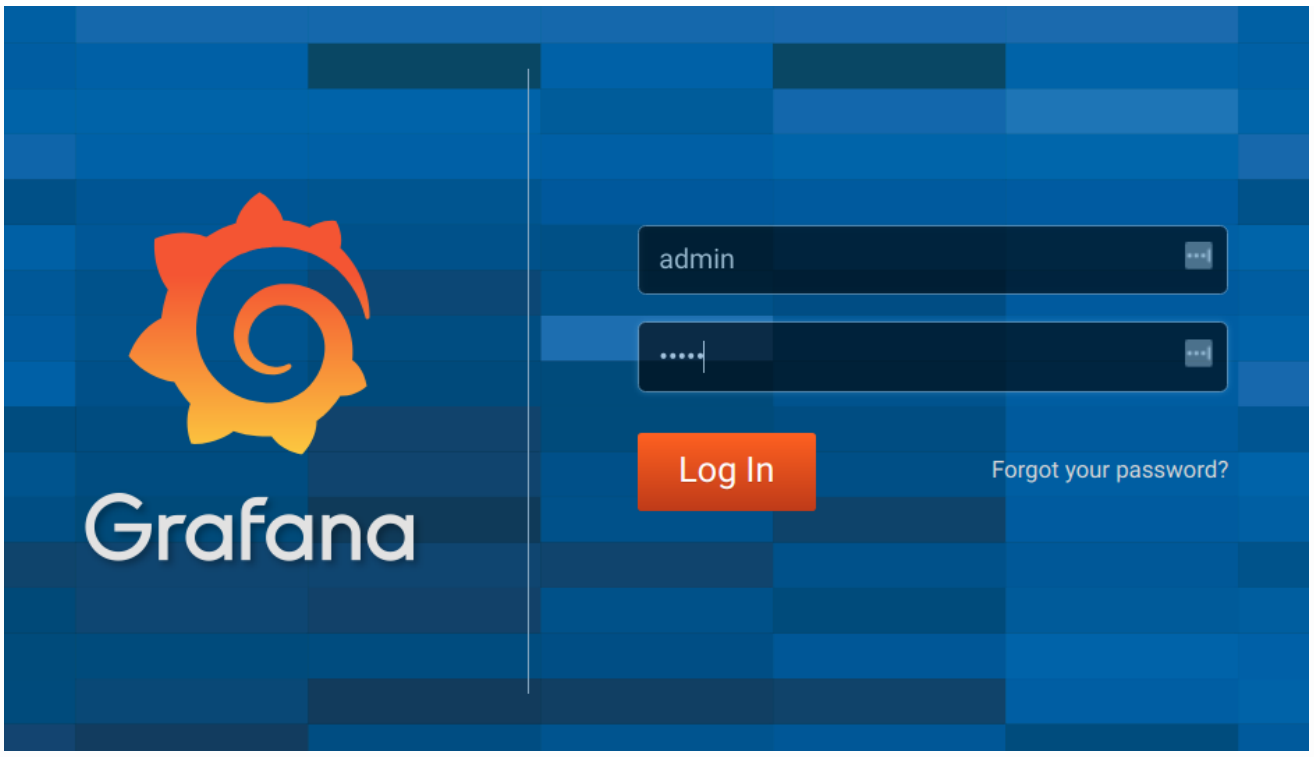
D.4. Grafana dashboard

As an example, and in order to visualize the exported metrics in Grafana, two sample dashboards are provided `strimzi-kafka.json` and `strimzi-zookeeper.json`. These dashboards represent a good starting point for key metrics to monitor Kafka and ZooKeeper clusters, but depending on your infrastructure you may need to update or add to them. Please note that they are not representative of all the metrics available. No alerting rules are defined.

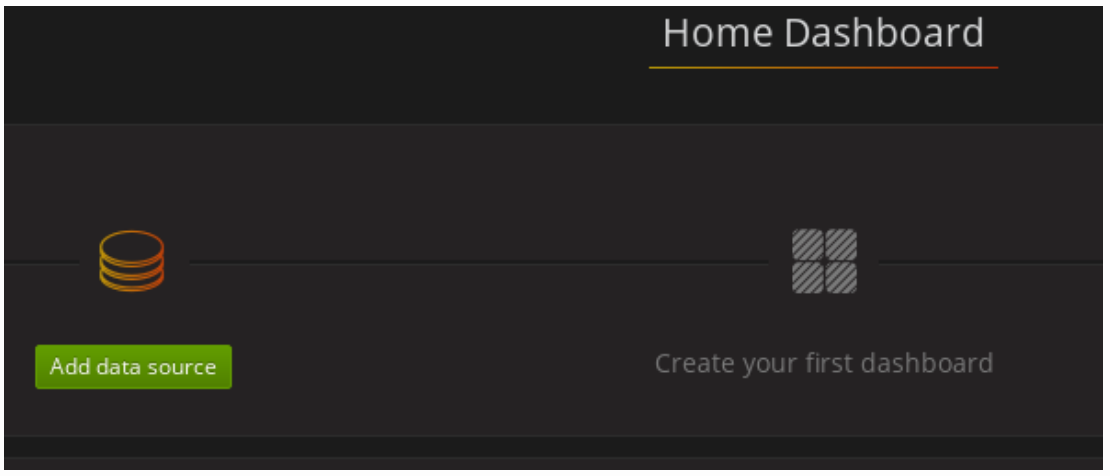
The Grafana Prometheus data source, and the above dashboards, can be set up in Grafana by following these steps.

Note	For accessing the dashboard, you can use the `port-forward` command for forwarding traffic from the Grafana pod to the host. For example, you can access the Grafana UI by running `oc port-forward grafana-1-fbl7s 3000:3000` (or using `kubectl` instead of `oc`) and then pointing a browser to `http://localhost:3000`.

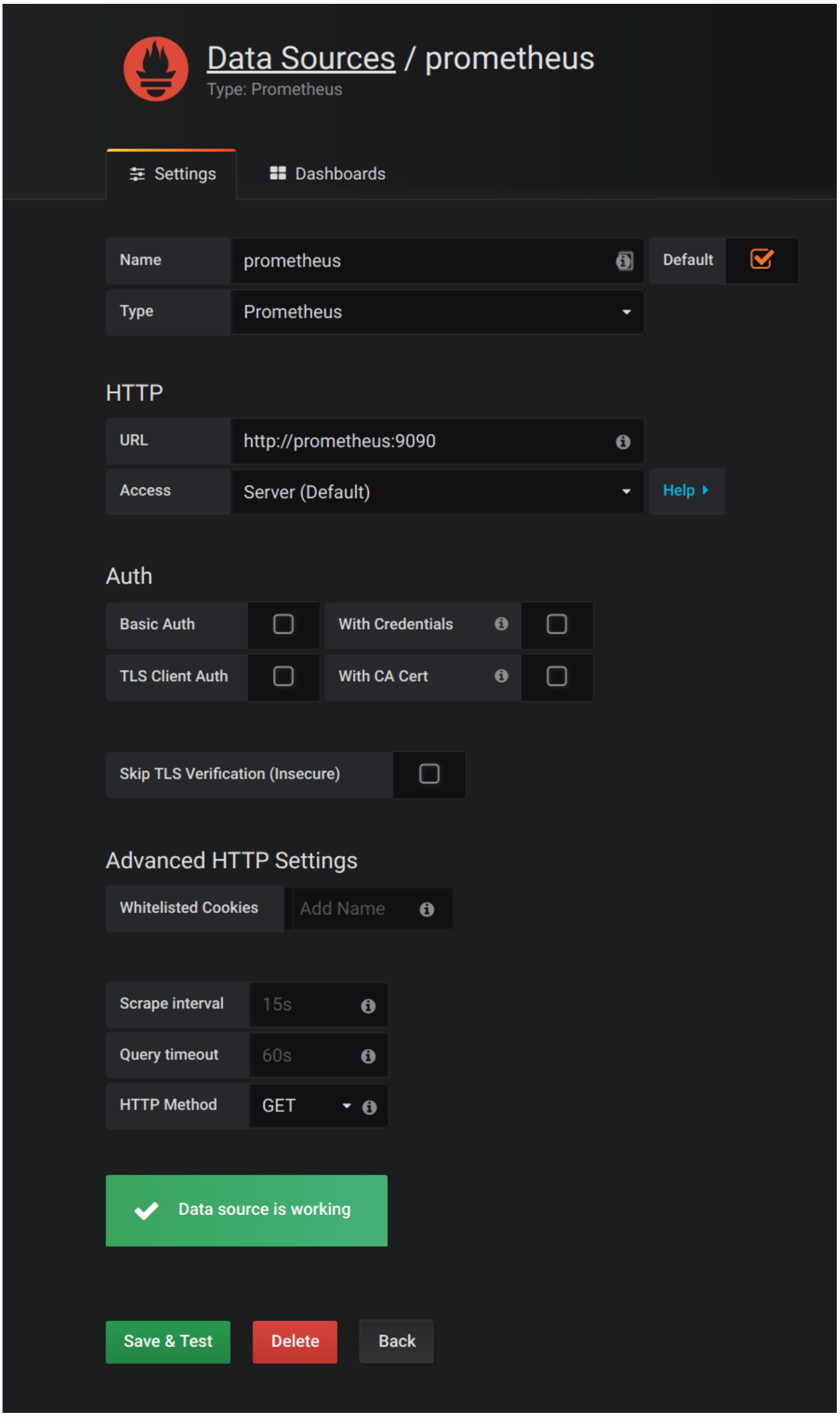
1. Access to the Grafana UI using `admin/admin` credentials. On the following view you can choose to skip resetting the admin password, or set it to a password you desire.



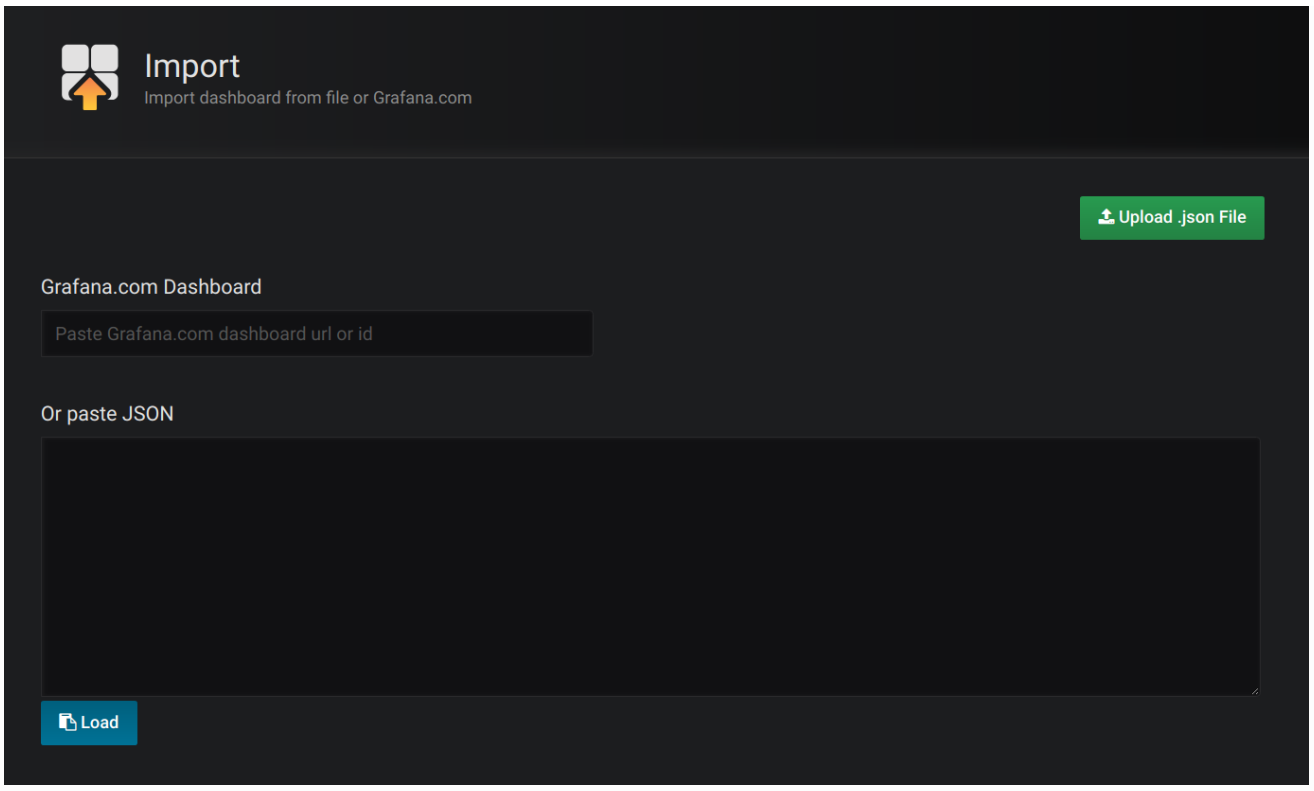
2. Click on the "Add data source" button from the Grafana home in order to add Prometheus as data source.



3. Fill in the information about the Prometheus data source, specifying a name and "Prometheus" as type. In the URL field, the connection string to the Prometheus server (that is, `http://prometheus:9090`) should be specified. After "Add" is clicked, Grafana will test the connection to the data source.

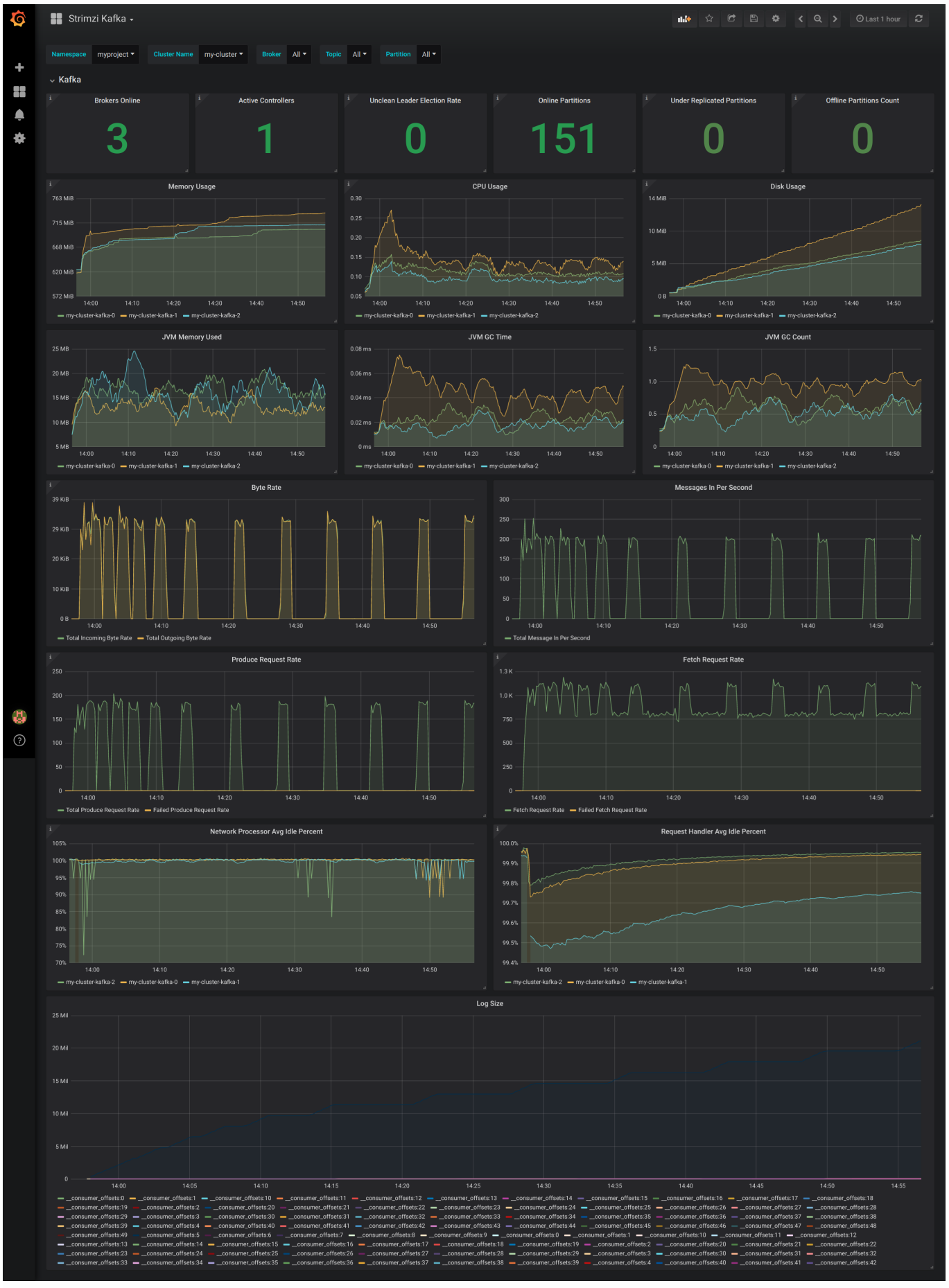


4. From the top left menu, click on "Dashboards" and then "Import" to open the "Import Dashboard" window where the provided `strimzi-kafka.json` and `strimzi-zookeeper.json` files can be imported or their content pasted.

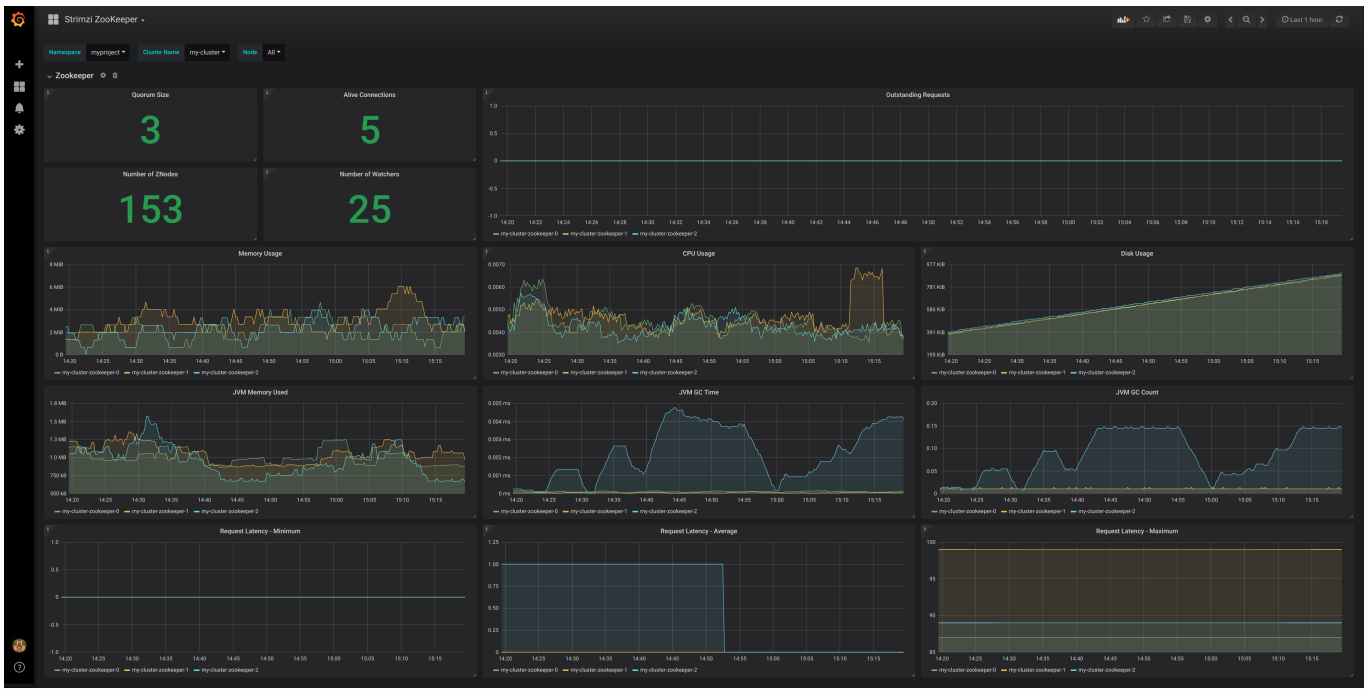


5. After importing the dashboards, the Grafana dashboard homepage will now list two dashboards for you to choose from. After your Prometheus server has been collecting metrics for a Strimzi cluster for some time you should see a populated dashboard such as the examples list below.

D.4.1. Kafka Dashboard



D.4.2. ZooKeeper Dashboard



D.4.3. Metrics References

To learn more about what metrics are available to monitor for Kafka, ZooKeeper, and Kubernetes in general, please review the following resources.

- Apache Kafka Monitoring - A list of JMX metrics exposed by Apache Kafka. It includes a description, JMX mbean name, and in some cases a suggestion on what is a normal value returned.
- ZooKeeper JMX - A list of JMX metrics exposed by Apache ZooKeeper.
- Prometheus - Monitoring Docker Container Metrics using cAdvisor - cAdvisor (short for container Advisor) analyzes and exposes resource usage (such as CPU, Memory, and Disk) and performance data from running containers within pods on Kubernetes. cAdvisor is bundled along with the kubelet binary so that it is automatically available within Kubernetes clusters. This reference describes how to monitor cAdvisor metrics in various ways using Prometheus.
  - cAdvisor Metrics - A full list of cAdvisor metrics as exposed through Prometheus.