

Deploying and Managing Strimzi

Table of Contents

1. Deployment overview	1
1.1. Strimzi custom resources	1
1.1.1. Strimzi custom resource example	1
1.1.2. Performing <code>kubectl</code> operations on custom resources	4
1.1.3. Strimzi custom resource status information	6
1.1.4. Finding the status of a custom resource	9
1.2. Strimzi operators	10
1.2.1. Watching Strimzi resources in Kubernetes namespaces	10
1.2.2. Managing RBAC resources	11
1.2.3. Managing pod resources	14
1.3. Using the Kafka Bridge to connect with a Kafka cluster	14
1.4. Seamless FIPS support	14
1.4.1. NIST validation	15
1.4.2. Minimum password length	15
1.5. Document Conventions	15
1.6. Additional resources	16
2. Strimzi installation methods	17
3. What is deployed with Strimzi	18
3.1. Order of deployment	18
4. Preparing for your Strimzi deployment	19
4.1. Deployment prerequisites	19
4.2. Operator deployment best practices	20
4.3. Downloading Strimzi release artifacts	20
4.4. Pushing container images to your own registry	21
4.5. Designating Strimzi administrators	21
5. Using Kafka in KRaft mode	23
5.1. KRaft limitations	24
5.2. Migrating to KRaft mode	25
5.2.1. Performing a rollback on the migration	29
6. Deploying Strimzi using installation artifacts	31
6.1. Basic deployment path	31
6.2. Deploying the Cluster Operator	32
6.2.1. Specifying the namespaces the Cluster Operator watches	32
6.2.2. Deploying the Cluster Operator to watch a single namespace	33
6.2.3. Deploying the Cluster Operator to watch multiple namespaces	34
6.2.4. Deploying the Cluster Operator to watch all namespaces	36
6.3. Deploying Kafka	37
6.3.1. Deploying a Kafka cluster in KRaft mode	38

6.3.2. Deploying a ZooKeeper-based Kafka cluster	40
6.3.3. Deploying the Topic Operator using the Cluster Operator	42
6.3.4. Deploying the User Operator using the Cluster Operator	43
6.3.5. Connecting to ZooKeeper from a terminal	44
6.3.6. List of Kafka cluster resources	45
6.4. Deploying Kafka Connect	50
6.4.1. Deploying Kafka Connect to your Kubernetes cluster	50
6.4.2. List of Kafka Connect cluster resources	51
6.5. Adding Kafka Connect connectors	52
6.5.1. Building a new container image with connector plugins automatically	53
6.5.2. Building a new container image with connector plugins from the Kafka Connect base image	54
6.5.3. Deploying KafkaConnector resources	56
6.5.4. Exposing the Kafka Connect API	60
6.5.5. Limiting access to the Kafka Connect API	62
6.5.6. Switching from using the Kafka Connect API to using KafkaConnector custom resources	64
6.6. Deploying Kafka MirrorMaker	64
6.6.1. Deploying Kafka MirrorMaker to your Kubernetes cluster	64
6.6.2. List of Kafka MirrorMaker 2 cluster resources	66
6.6.3. List of Kafka MirrorMaker cluster resources	66
6.7. Deploying Kafka Bridge	67
6.7.1. Deploying Kafka Bridge to your Kubernetes cluster	67
6.7.2. Exposing the Kafka Bridge service to your local machine	68
6.7.3. Accessing the Kafka Bridge outside of Kubernetes	68
6.7.4. List of Kafka Bridge cluster resources	69
6.8. Alternative standalone deployment options for Strimzi operators	69
6.8.1. Deploying the standalone Topic Operator	69
6.8.2. Deploying the standalone User Operator	74
7. Deploying Strimzi from OperatorHub.io	78
8. Deploying Strimzi using Helm	79
9. Feature gates	80
9.1. Graduated feature gates (GA)	80
9.1.1. ControlPlaneListener feature gate	80
9.1.2. ServiceAccountPatching feature gate	80
9.1.3. UseStrimziPodSets feature gate	81
9.1.4. StableConnectIdentities feature gate	81
9.1.5. KafkaNodePools feature gate	81
9.1.6. UnidirectionalTopicOperator feature gate	82
9.1.7. UseKRaft feature gate	82
9.2. Stable feature gates (Beta)	82

9.3. Early access feature gates (Alpha)	82
9.3.1. ContinueReconciliationOnManualRollingUpdateFailure feature gate	82
9.4. Enabling feature gates	83
9.5. Feature gate releases	83
10. Configuring a deployment	86
10.1. Using example configuration files	87
10.2. Configuring Kafka in KRaft mode	88
10.2.1. Setting throughput and storage limits on brokers	92
10.2.2. Deleting Kafka nodes using annotations	94
10.3. Configuring Kafka with ZooKeeper	94
10.3.1. Default ZooKeeper configuration values	98
10.3.2. Deleting ZooKeeper nodes using annotations	99
10.4. Configuring node pools	99
10.4.1. Assigning IDs to node pools for scaling operations	101
10.4.2. Impact on racks when moving nodes from node pools	104
10.4.3. Adding nodes to a node pool	104
10.4.4. Removing nodes from a node pool	106
10.4.5. Moving nodes between node pools	107
10.4.6. Changing node pool roles	109
10.4.7. Transitioning to separate broker and controller roles	109
10.4.8. Transitioning to dual-role nodes	112
10.4.9. Managing storage using node pools	115
10.4.10. Managing storage affinity using node pools	117
10.4.11. Migrating existing Kafka clusters to use Kafka node pools	120
10.5. Configuring the Entity Operator	122
10.5.1. Configuring the Topic Operator	123
10.5.2. Configuring the User Operator	124
10.6. Configuring the Cluster Operator	125
10.6.1. Restricting access to the Cluster Operator using network policy	130
10.6.2. Setting periodic reconciliation of custom resources	131
10.6.3. Pausing reconciliation of custom resources using annotations	131
10.6.4. Running multiple Cluster Operator replicas with leader election	133
10.6.5. Configuring Cluster Operator HTTP proxy settings	137
10.6.6. Disabling FIPS mode using Cluster Operator configuration	138
10.7. Configuring Kafka Connect	139
10.7.1. Configuring Kafka Connect for multiple instances	143
10.7.2. Configuring Kafka Connect user authorization	144
10.7.3. Manually stopping or pausing Kafka Connect connectors	146
10.7.4. Manually restarting Kafka Connect connectors	147
10.7.5. Manually restarting Kafka Connect connector tasks	147
10.8. Configuring Kafka MirrorMaker 2	148

10.8.1. Configuring active/active or active/passive modes	155
10.8.2. Configuring MirrorMaker 2 for multiple instances	156
10.8.3. Configuring MirrorMaker 2 connectors	157
10.8.4. Configuring MirrorMaker 2 connector producers and consumers	163
10.8.5. Specifying a maximum number of data replication tasks	165
10.8.6. Synchronizing ACL rules for remote topics	166
10.8.7. Securing a Kafka MirrorMaker 2 deployment	167
10.8.8. Manually stopping or pausing MirrorMaker 2 connectors	175
10.8.9. Manually restarting MirrorMaker 2 connectors	176
10.8.10. Manually restarting MirrorMaker 2 connector tasks	177
10.9. Configuring Kafka MirrorMaker (deprecated)	177
10.10. Configuring the Kafka Bridge	181
10.11. Configuring Kafka and ZooKeeper storage	183
10.11.1. Data storage considerations	184
10.11.2. Ephemeral storage	185
10.11.3. Persistent storage	186
10.11.4. Resizing persistent volumes	190
10.11.5. JBOD storage	191
10.11.6. Adding volumes to JBOD storage	194
10.11.7. Removing volumes from JBOD storage	195
10.11.8. Tiered storage (early access)	196
10.12. Configuring CPU and memory resource limits and requests	197
10.13. Restrictions on Kubernetes labels	197
10.14. Configuring pod scheduling	198
10.14.1. Specifying affinity, tolerations, and topology spread constraints	198
10.14.2. Configuring pod anti-affinity to schedule each Kafka broker on a different worker node	199
10.14.3. Configuring pod anti-affinity in Kafka components	201
10.14.4. Configuring node affinity in Kafka components	202
10.14.5. Setting up dedicated nodes and scheduling pods on them	203
10.15. Configuring logging levels	204
10.15.1. Logging options for Kafka components and operators	205
10.15.2. Creating a ConfigMap for logging	206
10.15.3. Configuring Cluster Operator logging	207
10.15.4. Adding logging filters to Strimzi operators	207
10.15.5. Lock acquisition warnings for cluster operations	211
10.16. Using ConfigMaps to add configuration	211
10.16.1. Naming custom ConfigMaps	212
10.17. Loading configuration values from external sources	213
10.17.1. Enabling configuration providers	213
10.17.2. Loading configuration values from secrets or config maps	215

10.17.3. Loading configuration values from environment variables	217
10.17.4. Loading configuration values from a file within a directory	219
10.17.5. Loading configuration values from multiple files within a directory	221
10.18. Customizing Kubernetes resources	223
10.18.1. Customizing the image pull policy	225
10.18.2. Applying a termination grace period	225
11. Using the Topic Operator to manage Kafka topics	227
11.1. Topic management	227
11.2. Topic naming conventions	227
11.3. Handling changes to topics	228
11.3.1. Downgrading to a Strimzi version that uses internal topics to store topic metadata	229
11.3.2. Downgrading to a Strimzi version that uses ZooKeeper to store topic metadata	229
11.3.3. Automatic creation of topics	229
11.4. Configuring Kafka topics	229
11.5. Configuring topics for replication and number of partitions	232
11.6. Managing KafkaTopic resources without impacting Kafka topics	233
11.7. Enabling topic management for existing Kafka topics	234
11.8. Deleting managed topics	236
11.9. Removing finalizers on topics	237
11.10. Considerations when disabling topic deletion	237
11.11. Tuning request batches for topic operations	238
12. Using the User Operator to manage Kafka users	239
12.1. Configuring Kafka users	239
13. Setting up client access to a Kafka cluster	243
13.1. Deploying example clients	243
13.2. Configuring listeners to connect to Kafka	244
13.3. Listener naming conventions	245
13.4. Accessing Kafka using node ports	246
13.5. Accessing Kafka using loadbalancers	249
13.6. Accessing Kafka using an Ingress NGINX Controller for Kubernetes	252
13.7. Accessing Kafka using OpenShift routes	255
13.8. Discovering connection details for clients	258
14. Securing access to a Kafka cluster	261
14.1. Configuring client authentication on listeners	261
14.1.1. mTLS authentication	264
14.1.2. SCRAM-SHA-512 authentication	265
14.1.3. Restricting access to listeners with network policies	265
14.1.4. Using custom listener certificates for TLS encryption	266
14.1.5. Specifying SANs for custom listener certificates	269
14.2. Configuring authorized access to Kafka	270
14.2.1. Designating super users	271

14.3. Configuring user (client-side) security mechanisms	272
14.3.1. Associating users with Kafka clusters	272
14.3.2. Configuring user authentication	272
14.3.3. Configuring user authorization	276
14.3.4. Configuring user quotas	277
14.4. Example: Setting up secure client access	278
14.4.1. Securing Kafka brokers	279
14.4.2. Securing user access to Kafka	281
14.5. Troubleshooting TLS hostname verification with node ports	284
15. Enabling OAuth 2.0 token-based access	285
15.1. Configuring an OAuth 2.0 authorization server	285
15.2. Using OAuth 2.0 token-based authentication	285
15.2.1. Configuring OAuth 2.0 authentication on listeners	286
15.2.2. Configuring OAuth 2.0 on client applications	293
15.2.3. OAuth 2.0 client authentication flows	297
15.2.4. Re-authenticating sessions	301
15.2.5. Example: Enabling OAuth 2.0 authentication	302
15.3. Using OAuth 2.0 token-based authorization	307
15.3.1. Example: Enabling OAuth 2.0 authorization	307
15.4. Setting up permissions in Keycloak	310
15.4.1. Kafka and Keycloak authorization models	311
15.4.2. Mapping authorization models	312
15.4.3. Permissions for common Kafka operations	314
15.4.4. Example: Setting up Keycloak Authorization Services	317
16. Managing TLS certificates	332
16.1. Internal cluster CA and clients CA	335
16.2. Secrets generated by the operators	335
16.2.1. TLS authentication using keys and certificates in PEM or PKCS #12 format	336
16.2.2. Secrets generated by the Cluster Operator	337
16.2.3. Cluster CA secrets	338
16.2.4. Clients CA secrets	341
16.2.5. User secrets generated by the User Operator	341
16.2.6. Adding labels and annotations to cluster CA secrets	341
16.2.7. Disabling <code>ownerReference</code> in the CA secrets	342
16.3. Certificate renewal and validity periods	343
16.3.1. Renewing automatically generated CA Certificates	344
16.3.2. Renewing client certificates	344
16.3.3. Scheduling maintenance time windows	345
16.3.4. Manually renewing Cluster Operator-managed CA certificates	346
16.3.5. Manually recovering from expired Cluster Operator-managed CA certificates	347
16.3.6. Replacing private keys used by Cluster Operator-managed CA certificates	349

16.4. Configuring internal clients to trust the cluster CA	350
16.5. Configuring external clients to trust the cluster CA	352
16.6. Using your own CA certificates and private keys	353
16.6.1. Installing your own CA certificates and private keys	354
16.6.2. Renewing your own CA certificates	356
16.6.3. Renewing or replacing CA certificates and private keys with your own	358
17. Applying security context to Strimzi pods and containers	365
17.1. How to configure security context	365
17.1.1. Template configuration for security context	366
17.1.2. Baseline Provider for pod security	366
17.1.3. Restricted Provider for pod security	367
17.2. Enabling the Restricted Provider for the Cluster Operator	368
17.3. Implementing a custom pod security provider	369
17.4. Handling of security context by Kubernetes platform	370
18. Scaling clusters by adding or removing brokers	371
18.1. Skipping checks on scale-down operations	372
19. Using Cruise Control for cluster rebalancing	373
19.1. Cruise Control components and features	373
19.2. Optimization goals overview	374
19.2.1. Goals order of priority	374
19.2.2. Goals configuration in Strimzi custom resources	375
19.2.3. Hard and soft optimization goals	376
19.2.4. Main optimization goals	377
19.2.5. Default optimization goals	378
19.2.6. User-provided optimization goals	379
19.3. Optimization proposals overview	379
19.3.1. Rebalancing modes	380
19.3.2. The results of an optimization proposal	380
19.3.3. Manually approving or rejecting an optimization proposal	381
19.3.4. Automatically approving an optimization proposal	383
19.3.5. Optimization proposal summary properties	383
19.3.6. Broker load properties	385
19.3.7. Cached optimization proposal	386
19.4. Rebalance performance tuning overview	386
19.4.1. Partition reassignment commands	386
19.4.2. Replica movement strategies	387
19.4.3. Intra-broker disk balancing	387
19.4.4. Rebalance tuning options	388
19.5. Configuring and deploying Cruise Control with Kafka	390
19.6. Generating optimization proposals	393
19.7. Approving an optimization proposal	398

19.8. Stopping a cluster rebalance	400
19.9. Fixing problems with a KafkaRebalance resource	400
20. Using Cruise Control to modify topic replication factor	402
21. Using the partition reassignment tool	405
21.1. Partition reassignment tool overview	405
21.1.1. Generating a partition reassignment plan	406
21.1.2. Specifying topics in a partition reassignment JSON file	406
21.1.3. Reassigning partitions between JBOD volumes	408
21.1.4. Throttling partition reassignment	408
21.2. Generating a reassignment JSON file to reassign partitions	409
21.3. Using the partition reassignment tool to reassign partitions after adding brokers	414
21.4. Using the partition reassignment tool to reassign partitions before removing brokers	416
21.5. Using the partition reassignment tool to modify topic replication factor	419
22. Introducing metrics	423
22.1. Monitoring consumer lag with Kafka Exporter	424
22.2. Monitoring Cruise Control operations	425
22.2.1. Monitoring balancedness scores	426
22.2.2. Setting up alerts for anomaly detection	426
22.3. Example metrics files	426
22.3.1. Example Prometheus metrics configuration	427
22.3.2. Example Prometheus rules for alert notifications	428
22.3.3. Example Grafana dashboards	429
22.4. Using Prometheus with Strimzi	430
22.4.1. Enabling Prometheus metrics through configuration	430
22.4.2. Setting up Prometheus	435
22.4.3. Deploying Alertmanager	438
22.5. Enabling the example Grafana dashboards	439
23. Introducing distributed tracing	442
23.1. Tracing options	442
23.2. Environment variables for tracing	443
23.3. Setting up distributed tracing	444
23.3.1. Prerequisites	444
23.3.2. Enabling tracing in MirrorMaker, Kafka Connect, and Kafka Bridge resources	444
23.3.3. Initializing tracing for Kafka clients	447
23.3.4. Instrumenting producers and consumers for tracing	448
23.3.5. Instrumenting Kafka Streams applications for tracing	450
23.3.6. Introducing a different OpenTelemetry tracing system	452
23.3.7. Specifying custom span names for OpenTelemetry	453
24. Evicting pods with the Strimzi Drain Cleaner	455
24.1. Downloading the Strimzi Drain Cleaner deployment files	456
24.2. Deploying the Strimzi Drain Cleaner using installation files	456

24.3. Deploying the Strimzi Drain Cleaner using Helm	459
24.4. Using the Strimzi Drain Cleaner	460
24.5. Adding or renewing the TLS certificates used by the Strimzi Drain Cleaner	462
24.6. Watching the TLS certificates used by the Strimzi Drain Cleaner	465
25. Managing rolling updates	467
25.1. Performing a rolling update using a pod management annotation	467
25.2. Performing a rolling update using a pod annotation	468
26. Finding information on Kafka restarts	470
26.1. Reasons for a restart event	470
26.2. Restart event filters	471
26.3. Checking Kafka restarts	472
27. Upgrading Strimzi	474
27.1. Required upgrade sequence	474
27.2. Strimzi upgrade paths	475
27.2.1. Support for Kafka versions when upgrading	475
27.2.2. Upgrading from a Strimzi version earlier than 0.22	475
27.2.3. Kafka version and image mappings	476
27.3. Strategies for upgrading clients	476
27.4. Upgrading Kubernetes with minimal downtime	477
27.4.1. Rolling pods using Drain Cleaner	478
27.4.2. Rolling pods manually (alternative to Drain Cleaner)	478
27.5. Upgrading the Cluster Operator	479
27.5.1. Upgrading the Cluster Operator using installation files	479
27.5.2. Upgrading the Cluster Operator using the OperatorHub	480
27.5.3. Upgrading the Cluster Operator using a Helm chart	481
27.5.4. Upgrading the Cluster Operator returns Kafka version error	481
27.5.5. Upgrading from a Strimzi version using the Bidirectional Topic Operator	481
27.6. Upgrading KRaft-based Kafka clusters	482
27.7. Upgrading Kafka when using ZooKeeper	485
27.7.1. Updating Kafka versions	485
27.7.2. Upgrading clients with older message formats	487
27.7.3. Upgrading ZooKeeper-based Kafka clusters	487
27.8. Checking the status of an upgrade	491
27.9. Switching to FIPS mode when upgrading Strimzi	491
28. Downgrading Strimzi	493
28.1. Downgrading the Cluster Operator to a previous version	493
28.2. Downgrading KRaft-based Kafka clusters and client applications	494
28.3. Downgrading Kafka when using ZooKeeper	496
28.3.1. Kafka version compatibility for downgrades	496
28.3.2. Downgrading ZooKeeper-based Kafka clusters and client applications	497
29. Uninstalling Strimzi	500

29.1. Uninstalling Strimzi using the CLI	500
29.2. Uninstalling Strimzi from OperatorHub.io	501
30. Cluster recovery from persistent volumes	503
30.1. Cluster recovery scenarios	503
30.1.1. Recovering from namespace deletion	503
30.1.2. Recovering from cluster loss	504
30.2. Recovering a deleted Kafka cluster	504
31. Tuning Kafka configuration	509
31.1. Tools that help with tuning	509
31.2. Managed broker configurations	509
31.3. Kafka broker configuration tuning	510
31.3.1. Basic broker configuration	510
31.3.2. Replicating topics for high availability	510
31.3.3. Internal topic settings for transactions and commits	511
31.3.4. Improving request handling throughput by increasing I/O threads	512
31.3.5. Increasing bandwidth for high latency connections	513
31.3.6. Managing Kafka logs with delete and compact policies	514
31.3.7. Managing efficient disk utilization for compaction	518
31.3.8. Controlling the log flush of message data	519
31.3.9. Partition rebalancing for availability	519
31.3.10. Unclean leader election	521
31.3.11. Avoiding unnecessary consumer group rebalances	521
31.4. Kafka producer configuration tuning	521
31.4.1. Basic producer configuration	522
31.4.2. Data durability	523
31.4.3. Ordered delivery	524
31.4.4. Reliability guarantees	525
31.4.5. Optimizing producers for throughput and latency	525
31.5. Kafka consumer configuration tuning	527
31.5.1. Basic consumer configuration	528
31.5.2. Scaling data consumption using consumer groups	529
31.5.3. Choosing the right partition assignment strategy	529
31.5.4. Message ordering guarantees	530
31.5.5. Optimizing consumers for throughput and latency	530
31.5.6. Avoiding data loss or duplication when committing offsets	531
31.5.7. Recovering from failure to avoid data loss	533
31.5.8. Managing offset policy	533
31.5.9. Minimizing the impact of rebalances	534
31.6. Handling high volumes of messages	535
31.6.1. Configuring Kafka Connect for high-volume messages	537
31.6.2. Configuring MirrorMaker 2 for high-volume messages	539

31.6.3. Checking the MirrorMaker 2 message flow	540
31.7. Handling large message sizes	540
31.7.1. Configuring Kafka components to handle larger messages	540
31.7.2. Producer-side compression	543
31.7.3. Reference-based messaging	544
31.7.4. Reference-based messaging flow	544

Chapter 1. Deployment overview

Strimzi simplifies the process of running [Apache Kafka](#) in a Kubernetes cluster.

This guide provides instructions for deploying and managing Strimzi. Deployment options and steps are covered using the example installation files included with Strimzi. While the guide highlights important configuration considerations, it does not cover all available options. For a deeper understanding of the Kafka component configuration options, refer to the [Strimzi Custom Resource API Reference](#).

In addition to deployment instructions, the guide offers pre- and post-deployment guidance. It covers setting up and securing client access to your Kafka cluster. Furthermore, it explores additional deployment options such as metrics integration, distributed tracing, and cluster management tools like Cruise Control and the Strimzi Drain Cleaner. You'll also find recommendations on managing Strimzi and fine-tuning Kafka configuration for optimal performance.

Upgrade instructions are provided for both Strimzi and Kafka, to help keep your deployment up to date.

Strimzi is designed to be compatible with all types of Kubernetes clusters, irrespective of their distribution. Whether your deployment involves public or private clouds, or if you are setting up a local development environment, the instructions in this guide are applicable in all cases.

1.1. Strimzi custom resources

The deployment of Kafka components onto a Kubernetes cluster using Strimzi is highly configurable through the use of custom resources. These resources are created as instances of APIs introduced by Custom Resource Definitions (CRDs), which extend Kubernetes resources.

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

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

1.1.1. Strimzi custom resource example

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

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

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

NOTE Access to manage custom resources is limited to Strimzi administrators. For more

information, see [Designating Strimzi administrators](#).

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

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

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

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

Kafka topic CRD

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

```
# ...
```

- ① The metadata for the topic CRD, its name and a label to identify the CRD.
- ② The specification for this CRD, including the group (domain) name, the plural name and the supported schema version, which are used in the URL to access the API of the topic. The other names are used to identify instance resources in the CLI. For example, `kubectl get kafkatopic my-topic` or `kubectl get kafkatopics`.
- ③ The shortname can be used in CLI commands. For example, `kubectl get kt` can be used as an abbreviation instead of `kubectl get kafkatopic`.
- ④ The information presented when using a `get` command on the custom resource.
- ⑤ The current status of the CRD as described in the [schema reference](#) for the resource.
- ⑥ openAPI3Schema validation provides validation for the creation of topic custom resources. For example, a topic requires at least one partition and one replica.

NOTE

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

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

Kafka topic custom resource

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

- ① The `kind` and `apiVersion` identify the CRD of which the custom resource is an instance.
- ② A label, applicable only to `KafkaTopic` and `KafkaUser` resources, that defines the name of the Kafka cluster (which is same as the name of the `Kafka` resource) to which a topic or user belongs.
- ③ The spec shows the number of partitions and replicas for the topic as well as the configuration parameters for the topic itself. In this example, the retention period for a message to remain in

the topic and the segment file size for the log are specified.

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

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

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

Additional resources

- [Extend the Kubernetes API with CustomResourceDefinitions](#)
- [Example configuration files provided with Strimzi](#)

1.1.2. Performing `kubectl` operations on custom resources

You can use `kubectl` commands to retrieve information and perform other operations on Strimzi custom resources. Use `kubectl` commands, such as `get`, `describe`, `edit`, or `delete`, to perform operations on resource types. For example, `kubectl get kafkatopics` retrieves a list of all Kafka topics and `kubectl get kafkas` retrieves all deployed Kafka clusters.

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

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

Listing Kafka clusters

```
kubectl get k

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

Table 1. Long and short names for each Strimzi resource

Strimzi resource	Long name	Short name
Kafka	kafka	k
Kafka Node Pool	kafkanodepool	knp
Kafka Topic	kafkatopic	kt
Kafka User	kafkauser	ku
Kafka Connect	kafkaconnect	kc
Kafka Connector	kafkaconnector	kctr
Kafka Mirror Maker	kafkamirrormaker	kmm

Strimzi resource	Long name	Short name
Kafka Mirror Maker 2	kafkamirrormaker2	kmm2
Kafka Bridge	kafkabridge	kb
Kafka Rebalance	kafkarebalance	kr

Resource categories

Categories of custom resources can also be used in `kubectl` commands.

All Strimzi custom resources belong to the category `strimzi`, so you can use `strimzi` to get all the Strimzi resources with one command.

For example, running `kubectl get strimzi` lists all Strimzi custom resources in a given namespace.

Listing all custom resources

```
kubectl get strimzi

NAME                                     DESIRED KAFKA REPLICAS DESIRED ZK REPLICAS
kafka.kafka.strimzi.io/my-cluster          3                      3

NAME                                     PARTITIONS REPLICATION FACTOR
kafkatopic.kafka.strimzi.io/kafka-apps    3                      3

NAME                                     AUTHENTICATION AUTHORIZATION
kafkauser.kafka.strimzi.io/my-user        tls                     simple
```

The `kubectl get strimzi -o name` command returns all resource types and resource names. The `-o name` option fetches the output in the `type/name` format

Listing all resource types and names

```
kubectl get strimzi -o name

kafka.kafka.strimzi.io/my-cluster
kafkatopic.kafka.strimzi.io/kafka-apps
kafkauser.kafka.strimzi.io/my-user
```

You can combine this `strimzi` command with other commands. For example, you can pass it into a `kubectl delete` command to delete all resources in a single command.

Deleting all custom resources

```
kubectl delete $(kubectl get strimzi -o name)

kafka.kafka.strimzi.io "my-cluster" deleted
kafkatopic.kafka.strimzi.io "kafka-apps" deleted
kafkauser.kafka.strimzi.io "my-user" deleted
```

Deleting all resources in a single operation might be useful, for example, when you are testing new Strimzi features.

Querying the status of sub-resources

There are other values you can pass to the `-o` option. For example, by using `-o yaml` you get the output in YAML format. Using `-o json` will return it as JSON.

You can see all the options in `kubectl get --help`.

One of the most useful options is the [JSONPath support](#), which allows you to pass JSONPath expressions to query the Kubernetes API. A JSONPath expression can extract or navigate specific parts of any resource.

For example, you can use the JSONPath expression `{.status.listeners[?(@.name=="tls")].bootstrapServers}` to get the bootstrap address from the status of the Kafka custom resource and use it in your Kafka clients.

Here, the command retrieves the `bootstrapServers` value of the listener named `tls`:

Retrieving the bootstrap address

```
kubectl get kafka my-cluster  
-o=jsonpath='{.status.listeners[?(@.name=="tls")].bootstrapServers}{"\n"}'  
  
my-cluster-kafka-bootstrap.myproject.svc:9093
```

By changing the name condition you can also get the address of the other Kafka listeners.

You can use `jsonpath` to extract any other property or group of properties from any custom resource.

1.1.3. Strimzi custom resource status information

Status properties provide status information for certain custom resources.

The following table lists the custom resources that provide status information (when deployed) and the schemas that define the status properties.

For more information on the schemas, see the [Strimzi Custom Resource API Reference](#).

Table 2. Custom resources that provide status information

Strimzi resource	Schema reference	Publishes status information on...
Kafka	<code>KafkaStatus</code> schema reference	The Kafka cluster, its listeners, and node pools
KafkaNodePool	<code>KafkaNodePoolStatus</code> schema reference	The nodes in the node pool, their roles, and the associated Kafka cluster

Strimzi resource	Schema reference	Publishes status information on...
KafkaTopic	KafkaTopicStatus schema reference	Kafka topics in the Kafka cluster
KafkaUser	KafkaUserStatus schema reference	Kafka users in the Kafka cluster
KafkaConnect	KafkaConnectStatus schema reference	The Kafka Connect cluster and connector plugins
KafkaConnector	KafkaConnectorStatus schema reference	KafkaConnector resources
KafkaMirrorMaker2	KafkaMirrorMaker2Status schema reference	The Kafka MirrorMaker 2 cluster and internal connectors
KafkaMirrorMaker	KafkaMirrorMakerStatus schema reference	The Kafka MirrorMaker cluster
KafkaBridge	KafkaBridgeStatus schema reference	The Kafka Bridge
KafkaRebalance	KafkaRebalance schema reference	The status and results of a rebalance
StrimziPodSet	StrimziPodSetStatus schema reference	The number of pods: being managed, using the current version, and in a ready state

The `status` property of a resource provides information on the state of the resource. The `status.conditions` and `status.observedGeneration` properties are common to all resources.

`status.conditions`

Status conditions describe the *current state* of a resource. Status condition properties are useful for tracking progress related to the resource achieving its *desired state*, as defined by the configuration specified in its `spec`. Status condition properties provide the time and reason the state of the resource changed, and details of events preventing or delaying the operator from realizing the desired state.

`status.observedGeneration`

Last observed generation denotes the latest reconciliation of the resource by the Cluster Operator. If the value of `observedGeneration` is different from the value of `metadata.generation` (the current version of the deployment), the operator has not yet processed the latest update to the resource. If these values are the same, the status information reflects the most recent changes to the resource.

The `status` properties also provide resource-specific information. For example, `KafkaStatus` provides information on listener addresses, and the ID of the Kafka cluster.

`KafkaStatus` also provides information on the Kafka and Strimzi versions being used. You can check the values of `operatorLastSuccessfulVersion` and `kafkaVersion` to determine whether an upgrade of Strimzi or Kafka has completed

Strimzi creates and maintains the status of custom resources, periodically evaluating the current state of the custom resource and updating its status accordingly. When performing an update on a custom resource using `kubectl edit`, for example, its `status` is not editable. Moreover, changing the `status` would not affect the configuration of the Kafka cluster.

Here we see the `status` properties for a `Kafka` custom resource.

Kafka custom resource status

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
spec:
  # ...
status:
  clusterId: XP9FP2P-RByvEy0W4cOEUA ①
  conditions: ②
    - lastTransitionTime: '2023-01-20T17:56:29.396588Z'
      status: 'True'
      type: Ready ③
  kafkaMetadataState: KRaft ④
  kafkaVersion: 3.7.1 ⑤
  kafkaNodePools: ⑥
    - name: broker
    - name: controller
  listeners: ⑦
    - addresses:
        - host: my-cluster-kafka-bootstrap.prm-project.svc
          port: 9092
        bootstrapServers: 'my-cluster-kafka-bootstrap.prm-project.svc:9092'
        name: plain
    - addresses:
        - host: my-cluster-kafka-bootstrap.prm-project.svc
          port: 9093
        bootstrapServers: 'my-cluster-kafka-bootstrap.prm-project.svc:9093'
        certificates:
          - |
            -----BEGIN CERTIFICATE-----
            -----END CERTIFICATE-----
        name: tls
    - addresses:
        - host: >-
          2054284155.us-east-2.elb.amazonaws.com
          port: 9095
        bootstrapServers: >-
          2054284155.us-east-2.elb.amazonaws.com:9095
        certificates:
          - |
            -----BEGIN CERTIFICATE-----
```

```

-----END CERTIFICATE-----
name: external3
- addresses:
  - host: ip-10-0-172-202.us-east-2.compute.internal
    port: 31644
bootstrapServers: 'ip-10-0-172-202.us-east-2.compute.internal:31644'
certificates:
- |
-----BEGIN CERTIFICATE-----
```

```

-----END CERTIFICATE-----
name: external4
observedGeneration: 3 ⑧
operatorLastSuccessfulVersion: 0.42.0 ⑨
```

- ① The Kafka cluster ID.
- ② Status **conditions** describe the current state of the Kafka cluster.
- ③ The **Ready** condition indicates that the Cluster Operator considers the Kafka cluster able to handle traffic.
- ④ Kafka metadata state that shows the mechanism used (KRaft or ZooKeeper) to manage Kafka metadata and coordinate operations.
- ⑤ The version of Kafka being used by the Kafka cluster.
- ⑥ The node pools belonging to the Kafka cluster.
- ⑦ The **listeners** describe Kafka bootstrap addresses by type.
- ⑧ The **observedGeneration** value indicates the last reconciliation of the **Kafka** custom resource by the Cluster Operator.
- ⑨ The version of the operator that successfully completed the last reconciliation.

NOTE

The Kafka bootstrap addresses listed in the status do not signify that those endpoints or the Kafka cluster is in a **Ready** state.

1.1.4. Finding the status of a custom resource

Use **kubectl** with the **status** subresource of a custom resource to retrieve information about the resource.

Prerequisites

- A Kubernetes cluster.
- The Cluster Operator is running.

Procedure

- Specify the custom resource and use the **-o jsonpath** option to apply a standard JSONPath expression to select the **status** property:

```
kubectl get kafka <kafka_resource_name> -o jsonpath='{.status}' | jq
```

This expression returns all the status information for the specified custom resource. You can use dot notation, such as `status.listeners` or `status.observedGeneration`, to fine-tune the status information you wish to see.

Using the [jq command line JSON parser tool](#) makes it easier to read the output.

Additional resources

- For more information about using JSONPath, see [JSONPath support](#).

1.2. Strimzi operators

Strimzi operators are purpose-built with specialist operational knowledge to effectively manage Kafka on Kubernetes. Each operator performs a distinct function.

Cluster Operator

The Cluster Operator handles the deployment and management of Apache Kafka clusters on Kubernetes. It automates the setup of Kafka brokers, and other Kafka components and resources.

Topic Operator

The Topic Operator manages the creation, configuration, and deletion of topics within Kafka clusters.

User Operator

The User Operator manages Kafka users that require access to Kafka brokers.

When you deploy Strimzi, you first deploy the Cluster Operator. The Cluster Operator is then ready to handle the deployment of Kafka. You can also deploy the Topic Operator and User Operator using the Cluster Operator (recommended) or as standalone operators. You would use a standalone operator with a Kafka cluster that is not managed by the Cluster Operator.

The Topic Operator and User Operator are part of the Entity Operator. The Cluster Operator can deploy one or both operators based on the Entity Operator configuration.

IMPORTANT

To deploy the standalone operators, you need to set environment variables to connect to a Kafka cluster. These environment variables do not need to be set if you are deploying the operators using the Cluster Operator as they will be set by the Cluster Operator.

1.2.1. Watching Strimzi resources in Kubernetes namespaces

Operators watch and manage Strimzi resources in Kubernetes namespaces. The Cluster Operator can watch a single namespace, multiple namespaces, or all namespaces in a Kubernetes cluster. The Topic Operator and User Operator can watch a single namespace.

- The Cluster Operator watches for [Kafka](#) resources
- The Topic Operator watches for [KafkaTopic](#) resources
- The User Operator watches for [KafkaUser](#) resources

The Topic Operator and the User Operator can only watch a single Kafka cluster in a namespace. And they can only be connected to a single Kafka cluster.

If multiple Topic Operators watch the same namespace, name collisions and topic deletion can occur. This is because each Kafka cluster uses Kafka topics that have the same name (such as [_consumer_offsets](#)). Make sure that only one Topic Operator watches a given namespace.

When using multiple User Operators with a single namespace, a user with a given username can exist in more than one Kafka cluster.

If you deploy the Topic Operator and User Operator using the Cluster Operator, they watch the Kafka cluster deployed by the Cluster Operator by default. You can also specify a namespace using [watchedNamespace](#) in the operator configuration.

For a standalone deployment of each operator, you specify a namespace and connection to the Kafka cluster to watch in the configuration.

1.2.2. Managing RBAC resources

The Cluster Operator creates and manages role-based access control (RBAC) resources for Strimzi components that need access to Kubernetes resources.

For the Cluster Operator to function, it needs permission within the Kubernetes cluster to interact with Kafka resources, such as [Kafka](#) and [KafkaConnect](#), as well as managed resources like [ConfigMap](#), [Pod](#), [Deployment](#), and [Service](#).

Permission is specified through the following Kubernetes RBAC resources:

- [ServiceAccount](#)
- [Role](#) and [ClusterRole](#)
- [RoleBinding](#) and [ClusterRoleBinding](#)

Delegating privileges to Strimzi components

The Cluster Operator runs under a service account called [strimzi-cluster-operator](#), which is assigned cluster roles that give it permission to create the necessary RBAC resources for Strimzi components. Role bindings associate the cluster roles with the service account.

Kubernetes enforces [privilege escalation prevention](#), meaning the Cluster Operator cannot grant privileges it does not possess, nor can it grant such privileges in a namespace it cannot access. Consequently, the Cluster Operator must have the necessary privileges for all the components it orchestrates.

The Cluster Operator must be able to do the following:

- Enable the Topic Operator to manage [KafkaTopic](#) resources by creating [Role](#) and [RoleBinding](#) resources in the relevant namespace.
- Enable the User Operator to manage [KafkaUser](#) resources by creating [Role](#) and [RoleBinding](#) resources in the relevant namespace.
- Allow Strimzi to discover the failure domain of a [Node](#) by creating a [ClusterRoleBinding](#).

When using rack-aware partition assignment, broker pods need to access information about the [Node](#) they are running on, such as the Availability Zone in Amazon AWS. Since a [Node](#) is a cluster-scoped resource, this access must be granted through a [ClusterRoleBinding](#), not a namespace-scoped [RoleBinding](#).

The following sections describe the RBAC resources required by the Cluster Operator.

[ClusterRole](#) resources

The Cluster Operator uses [ClusterRole](#) resources to provide the necessary access to resources. Depending on the Kubernetes cluster setup, a cluster administrator might be needed to create the cluster roles.

NOTE

Cluster administrator rights are only needed for the creation of [ClusterRole](#) resources. The Cluster Operator will not run under a cluster admin account.

The RBAC resources follow the *principle of least privilege* and contain only those privileges needed by the Cluster Operator to operate the cluster of the Kafka component.

All cluster roles are required by the Cluster Operator in order to delegate privileges.

Table 3. ClusterRole resources

Name	Description
strimzi-cluster-operator-namespaced	Access rights for namespace-scoped resources used by the Cluster Operator to deploy and manage the operands.
strimzi-cluster-operator-global	Access rights for cluster-scoped resources used by the Cluster Operator to deploy and manage the operands.
strimzi-cluster-operator-leader-election	Access rights used by the Cluster Operator for leader election.
strimzi-cluster-operator-watched	Access rights used by the Cluster Operator to watch and manage the Strimzi custom resources.
strimzi-kafka-broker	Access rights to allow Kafka brokers to get the topology labels from Kubernetes worker nodes when rack-awareness is used.
strimzi-entity-operator	Access rights used by the Topic and User Operators to manage Kafka users and topics.

Name	Description
<code>strimzi-kafka-client</code>	Access rights to allow Kafka Connect, MirrorMaker (1 and 2), and Kafka Bridge to get the topology labels from Kubernetes worker nodes when rack-awareness is used.

ClusterRoleBinding resources

The Cluster Operator uses `ClusterRoleBinding` and `RoleBinding` resources to associate its `ClusterRole` with its `ServiceAccount`. Cluster role bindings are required by cluster roles containing cluster-scoped resources.

Table 4. ClusterRoleBinding resources

Name	Description
<code>strimzi-cluster-operator</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-global</code> cluster role.
<code>strimzi-cluster-operator-kafka-broker-delegation</code>	Grants the Cluster Operator the rights from the <code>strimzi-entity-operator</code> cluster role.
<code>strimzi-cluster-operator-kafka-client-delegation</code>	Grants the Cluster Operator the rights from the <code>strimzi-kafka-client</code> cluster role.

Table 5. RoleBinding resources

Name	Description
<code>strimzi-cluster-operator</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-namespaced</code> cluster role.
<code>strimzi-cluster-operator-leader-election</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-leader-election</code> cluster role.
<code>strimzi-cluster-operator-watched</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-watched</code> cluster role.
<code>strimzi-cluster-operator-entity-operator-delegation</code>	Grants the Cluster Operator the rights from the <code>strimzi-cluster-operator-entity-operator-delegation</code> cluster role.

ServiceAccount resources

The Cluster Operator runs using the `strimzi-cluster-operator ServiceAccount`. This service account grants it the privileges it requires to manage the operands. The Cluster Operator creates additional `ClusterRoleBinding` and `RoleBinding` resources to delegate some of these RBAC rights to the operands.

Each of the operands uses its own service account created by the Cluster Operator. This allows the Cluster Operator to follow the principle of least privilege and give the operands only the access rights that are really needed.

Table 6. ServiceAccount resources

Name	Used by
<cluster_name>-zookeeper	ZooKeeper pods
<cluster_name>-kafka	Kafka broker pods
<cluster_name>-entity-operator	Entity Operator
<cluster_name>-cruise-control	Cruise Control pods
<cluster_name>-kafka-exporter	Kafka Exporter pods
<cluster_name>-connect	Kafka Connect pods
<cluster_name>-mirror-maker	MirrorMaker pods
<cluster_name>-mirrormaker2	MirrorMaker 2 pods
<cluster_name>-bridge	Kafka Bridge pods

1.2.3. Managing pod resources

The [StrimziPodSet](#) custom resource is used by Strimzi to create and manage Kafka, Kafka Connect, and MirrorMaker 2 pods. If you are using ZooKeeper, ZooKeeper pods are also created and managed using [StrimziPodSet](#) resources.

You must not create, update, or delete [StrimziPodSet](#) resources. The [StrimziPodSet](#) custom resource is used internally and resources are managed solely by the Cluster Operator. As a consequence, the Cluster Operator must be running properly to avoid the possibility of pods not starting and Kafka clusters not being available.

NOTE Kubernetes [Deployment](#) resources are used for creating and managing the pods of other components: Kafka Bridge, Kafka Exporter, Cruise Control, (deprecated) MirrorMaker 1, User Operator and Topic Operator.

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

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

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

Additional resources

- For information on installing and using the Kafka Bridge, see [Using the Kafka Bridge](#).

1.4. Seamless FIPS support

Federal Information Processing Standards (FIPS) are standards for computer security and

interoperability. When running Strimzi on a FIPS-enabled Kubernetes cluster, the OpenJDK used in Strimzi container images automatically switches to FIPS mode. From version 0.33, Strimzi can run on FIPS-enabled Kubernetes clusters without any changes or special configuration. It uses only the FIPS-compliant security libraries from the OpenJDK.

IMPORTANT

If you are using FIPS-enabled Kubernetes clusters, you may experience higher memory consumption compared to regular Kubernetes clusters. To avoid any issues, we suggest increasing the memory request to at least 512Mi.

1.4.1. NIST validation

Strimzi is designed to use FIPS-validated cryptographic libraries for secure communication in a FIPS-enabled Kubernetes cluster. However, it's important to note that while Strimzi can leverage these libraries in a FIPS environment, the underlying Universal Base Images (UBI) used in Strimzi deployments may not inherently include NIST-validated binaries. This means that while Strimzi can leverage cryptographic libraries for FIPS, the specific binaries within the Strimzi container images might not have undergone NIST validation.

For more information about the NIST validation program and validated modules, see [Cryptographic Module Validation Program](#) on the NIST website.

1.4.2. Minimum password length

When running in the FIPS mode, SCRAM-SHA-512 passwords need to be at least 32 characters long. From Strimzi 0.33, the default password length in Strimzi User Operator is set to 32 characters as well. If you have a Kafka cluster with custom configuration that uses a password length that is less than 32 characters, you need to update your configuration. If you have any users with passwords shorter than 32 characters, you need to regenerate a password with the required length. You can do that, for example, by deleting the user secret and waiting for the User Operator to create a new password with the appropriate length.

Additional resources

- [Disabling FIPS mode using Cluster Operator configuration](#)
- [What are Federal Information Processing Standards \(FIPS\)](#)

1.5. Document Conventions

User-replaced values

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

For example, the following code shows that `<my_namespace>` must be replaced by the correct namespace name:

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

```
operator/*RoleBinding*.yaml
```

1.6. Additional resources

- [Strimzi Overview](#)
- [Strimzi Custom Resource API Reference](#)
- [Using the Kafka Bridge](#)

Chapter 2. Strimzi installation methods

You can install Strimzi on Kubernetes 1.23 and later in three ways.

Installation method	Description
Installation artifacts (YAML files)	<p>Download the release artifacts from the GitHub releases page.</p> <p>Download the <code>strimzi-<version>.zip</code> or <code>strimzi-<version>.tar.gz</code> archive file. The archive file contains installation artifacts and example configuration files.</p> <p>Deploy the YAML installation artifacts to your Kubernetes cluster using <code>kubectl</code>. You start by deploying the Cluster Operator from install/cluster-operator to a single namespace, multiple namespaces, or all namespaces.</p> <p>You can also use the <code>install/</code> artifacts to deploy the following:</p> <ul style="list-style-type: none">• Strimi administrator roles (<code>strimzi-admin</code>)• A standalone Topic Operator (<code>topic-operator</code>)• A standalone User Operator (<code>user-operator</code>)• Strimzi Drain Cleaner (<code>drain-cleaner</code>)
OperatorHub.io	Use the Strimzi Kafka operator in the OperatorHub.io to deploy the Cluster Operator. You then deploy Strimzi components using custom resources.
Helm chart	Use a Helm chart to deploy the Cluster Operator. You then deploy Strimzi components using custom resources.

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

Chapter 3. What is deployed with Strimzi

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

A typical deployment incorporating Kafka components might include:

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

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

3.1. Order of deployment

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

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

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

Chapter 4. Preparing for your Strimzi deployment

Prepare for a deployment of Strimzi by completing any necessary pre-deployment tasks. Take the necessary preparatory steps according to your specific requirements, such as the following:

- [Ensuring you have the necessary prerequisites before deploying Strimzi](#)
- [Downloading the Strimzi release artifacts to facilitate your deployment](#)
- [Pushing the Strimzi container images into your own registry \(if required\)](#)
- [Setting up admin roles to enable configuration of custom resources used in the deployment](#)

NOTE

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

4.1. Deployment prerequisites

To deploy Strimzi, you will need the following:

- A Kubernetes 1.23 and later cluster.
- The `kubectl` command-line tool is installed and configured to connect to the running cluster.

For more information on the tools available for running Kubernetes, see [Install Tools](#) in the Kubernetes documentation.

NOTE

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

oc and kubectl commands

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

In other words, instead of using:

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

when using OpenShift you can use:

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

NOTE

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

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

4.2. Operator deployment best practices

Potential issues can arise from installing more than one Strimzi operator in the same Kubernetes cluster, especially when using different versions. Each Strimzi operator manages a set of resources in a Kubernetes cluster. When you install multiple Strimzi operators, they may attempt to manage the same resources concurrently. This can lead to conflicts and unpredictable behavior within your cluster. Conflicts can still occur even if you deploy Strimzi operators in different namespaces within the same Kubernetes cluster. Although namespaces provide some degree of resource isolation, certain resources managed by the Strimzi operator, such as Custom Resource Definitions (CRDs) and roles, have a cluster-wide scope.

Additionally, installing multiple operators with different versions can result in compatibility issues between the operators and the Kafka clusters they manage. Different versions of Strimzi operators may introduce changes, bug fixes, or improvements that are not backward-compatible.

To avoid the issues associated with installing multiple Strimzi operators in a Kubernetes cluster, the following guidelines are recommended:

- Install the Strimzi operator in a separate namespace from the Kafka cluster and other Kafka components it manages, to ensure clear separation of resources and configurations.
- Use a single Strimzi operator to manage all your Kafka instances within a Kubernetes cluster.
- Update the Strimzi operator and the supported Kafka version as often as possible to reflect the latest features and enhancements.

By following these best practices and ensuring consistent updates for a single Strimzi operator, you can enhance the stability of managing Kafka instances in a Kubernetes cluster. This approach also enables you to make the most of Strimzi's latest features and capabilities.

4.3. Downloading Strimzi release artifacts

To use deployment files to install Strimzi, download and extract the files from the [GitHub releases page](#).

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

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

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

NOTE Strimzi container images are also available through the [Container Registry](#).

However, we recommend that you use the YAML files provided to deploy Strimzi.

4.4. Pushing container images to your own registry

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

If you do not have access to the [Container Registry](#) or want to use your own container repository:

1. Pull **all** container images listed here
2. Push them into your own registry
3. Update the image names in the YAML files used in deployment

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

Container image	Namespace/Repository	Description
Kafka	<ul style="list-style-type: none">• quay.io/stimzi/kafka:0.42.0-kafka-3.6.0• quay.io/stimzi/kafka:0.42.0-kafka-3.6.1• quay.io/stimzi/kafka:0.42.0-kafka-3.6.2• quay.io/stimzi/kafka:0.42.0-kafka-3.7.0• quay.io/stimzi/kafka:0.42.0-kafka-3.7.1	Stimzi image for running Kafka, including: <ul style="list-style-type: none">• Kafka Broker• Kafka Connect• Kafka MirrorMaker• ZooKeeper• Cruise Control
Operator	<ul style="list-style-type: none">• quay.io/stimzi/operator:0.4.2.0	Stimzi image for running the operators: <ul style="list-style-type: none">• Cluster Operator• Topic Operator• User Operator• Kafka Initializer
Kafka Bridge	<ul style="list-style-type: none">• quay.io/stimzi/kafka-bridge:0.29.0	Stimzi image for running the Kafka Bridge
Stimzi Drain Cleaner	<ul style="list-style-type: none">• quay.io/stimzi/drain-cleaner:1.1.0	Stimzi image for running the Stimzi Drain Cleaner

4.5. Designating Strimzi administrators

Stimzi provides custom resources for configuration of your deployment. By default, permission to view, create, edit, and delete these resources is limited to Kubernetes cluster administrators. Stimzi

provides two cluster roles that you can use to assign these rights to other users:

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

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

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

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

Prerequisites

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

Procedure

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

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

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

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

Chapter 5. Using Kafka in KRaft mode

KRaft (Kafka Raft metadata) mode replaces Kafka's dependency on ZooKeeper for cluster management. KRaft mode simplifies the deployment and management of Kafka clusters by bringing metadata management and coordination of clusters into Kafka.

Kafka in KRaft mode is designed to offer enhanced reliability, scalability, and throughput. Metadata operations become more efficient as they are directly integrated. And by removing the need to maintain a ZooKeeper cluster, there's also a reduction in the operational and security overhead.

To deploy a Kafka cluster in KRaft mode, you must use [Kafka](#) and [KafkaNodePool](#) custom resources. The [Kafka](#) resource using KRaft mode must also have the annotations `strimzi.io/kraft: enabled` and `strimzi.io/node-pools: enabled`. For more details and examples, see [Deploying a Kafka cluster in KRaft mode](#).

Through [node pool configuration using KafkaNodePool resources](#), nodes are assigned the role of broker, controller, or both:

- **Controller** nodes operate in the control plane to manage cluster metadata and the state of the cluster using a Raft-based consensus protocol.
- **Broker** nodes operate in the data plane to manage the streaming of messages, receiving and storing data in topic partitions.
- **Dual-role** nodes fulfill the responsibilities of controllers and brokers.

Controllers use a metadata log, stored as a single-partition topic (`__cluster_metadata`) on every node, which records the state of the cluster. When requests are made to change the cluster configuration, an active (lead) controller manages updates to the metadata log, and follower controllers replicate these updates. The metadata log stores information on brokers, replicas, topics, and partitions, including the state of in-sync replicas and partition leadership. Kafka uses this metadata to coordinate changes and manage the cluster effectively.

Broker nodes act as observers, storing the metadata log passively to stay up-to-date with the cluster's state. Each node fetches updates to the log independently.

NOTE The KRaft metadata version used in the Kafka cluster must be supported by the Kafka version in use. Both versions are managed through the [Kafka](#) resource configuration. For more information, see [Configuring Kafka in KRaft mode](#).

In the following example, a Kafka cluster comprises a quorum of controller and broker nodes for fault tolerance and high availability.

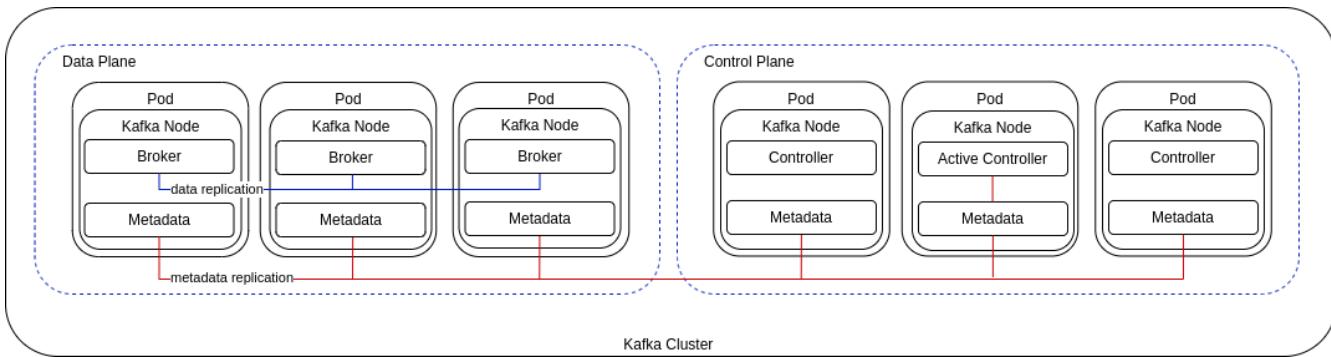


Figure 1. Example cluster with separate broker and controller nodes

In a typical production environment, use dedicated broker and controller nodes. However, you might want to use nodes in a dual-role configuration for development or testing.

You can use a combination of nodes that combine roles with nodes that perform a single role. In the following example, three nodes perform a dual role and two nodes act only as brokers.

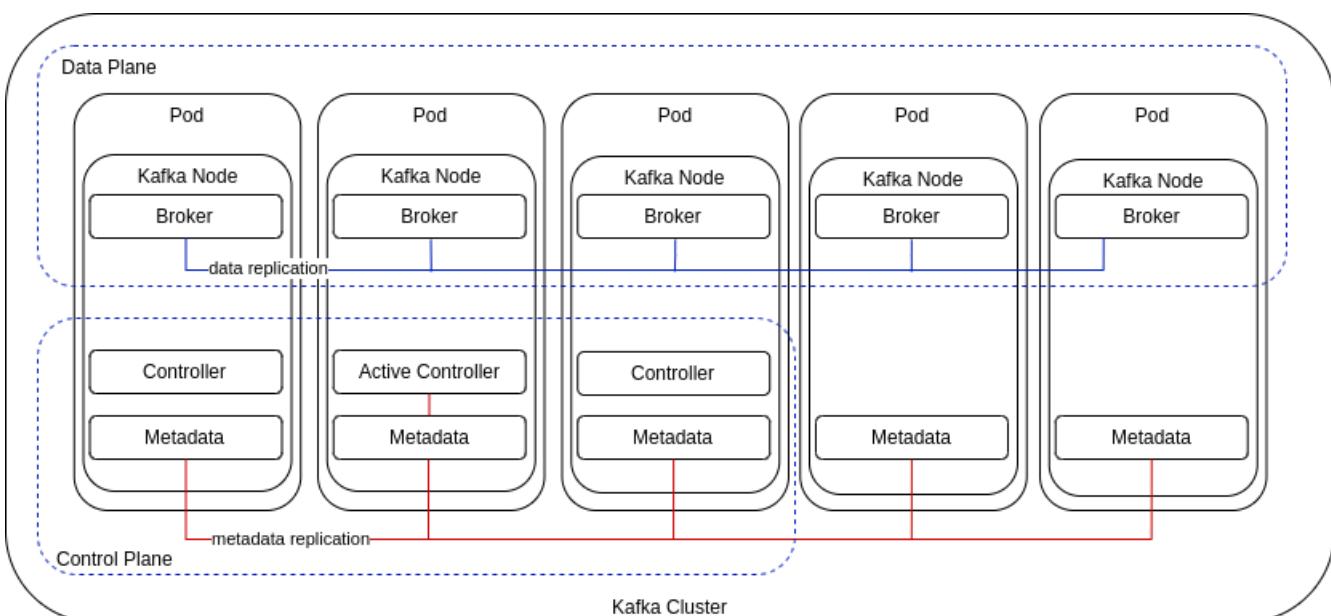


Figure 2. Example cluster with dual-role nodes and dedicated broker nodes

5.1. KRaft limitations

Currently, the KRaft mode in Strimzi has the following major limitations:

- JBOD storage is supported only with Apache Kafka 3.7.0 and higher. It is considered early-access in Apache Kafka 3.7.x. (Storage with `type: jbod` configuration can be used also with older Kafka versions, but the JBOD array can contain only one disk.)
- Scaling of KRaft controller nodes up or down is not supported.
- Unregistering Kafka nodes removed from the Kafka cluster.

NOTE

If you are using JBOD storage, you can [change the volume that stores the metadata log](#).

5.2. Migrating to KRaft mode

If you are using ZooKeeper for metadata management in your Kafka cluster, you can migrate to using Kafka in KRaft mode.

During the migration, you install a quorum of controller nodes as a node pool, which replaces ZooKeeper for management of your cluster. You enable KRaft migration in the cluster configuration by applying the `strimzi.io/kraft="migration"` annotation. After the migration is complete, you switch the brokers to using KRaft and the controllers out of migration mode using the `strimzi.io/kraft="enabled"` annotation.

Before starting the migration, verify that your environment can support Kafka in KRaft mode, as there are [a number of limitations](#). Note also, the following:

- Migration is only supported on dedicated controller nodes, not on nodes with dual roles as brokers and controllers.
- Throughout the migration process, ZooKeeper and controller nodes operate in parallel for a period, requiring sufficient compute resources in the cluster.
- **Once KRaft mode is enabled, rollback to ZooKeeper is not possible. Consider this carefully before proceeding with the migration.**

Prerequisites

- You must be using Strimzi 0.40 or newer with Kafka 3.7.0 or newer. If you are using an earlier version of Strimzi or Apache Kafka, upgrade before migrating to KRaft mode.
- Verify that the ZooKeeper-based deployment is operating without the following, as they are not supported in KRaft mode:
 - JBOD storage. While the `jbd` storage type can be used, the JBOD array must contain only one disk.
- The Cluster Operator that manages the Kafka cluster is running.
- The Kafka cluster deployment uses Kafka node pools.

If your ZooKeeper-based cluster is already using node pools, it is ready to migrate. If not, you can [migrate the cluster to use node pools](#). To migrate when the cluster is not using node pools, brokers must be contained in a `KafkaNodePool` resource configuration that is assigned a `broker` role and has the name `kafka`. Support for node pools is enabled in the `Kafka` resource configuration using the `strimzi.io/node-pools: enabled` annotation.

IMPORTANT

Using a single controller with ephemeral storage for migrating to KRaft will not work. During the migration, controller restart will cause loss of metadata synced from ZooKeeper (such as topics and ACLs). In general, migrating an ephemeral-based ZooKeeper cluster to KRaft is not recommended.

In this procedure, the Kafka cluster name is `my-cluster`, which is located in the `my-project` namespace. The name of the controller node pool created is `controller`. The node pool for the brokers is called `kafka`.

Procedure

1. For the Kafka cluster, create a node pool with a controller role.

The node pool adds a quorum of controller nodes to the cluster.

Example configuration for a controller node pool

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: controller
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 20Gi
        deleteClaim: false
  resources:
    requests:
      memory: 64Gi
      cpu: "8"
    limits:
      memory: 64Gi
      cpu: "12"
```

NOTE

For the migration, you cannot use a node pool of nodes that share the broker and controller roles.

2. Apply the new **KafkaNodePool** resource to create the controllers.

Errors related to using controllers in a ZooKeeper-based environment are expected in the Cluster Operator logs. The errors can block reconciliation. To prevent this, perform the next step immediately.

3. Enable KRaft migration in the **Kafka** resource by setting the **strimzi.io/kraft** annotation to **migration**:

```
kubectl annotate kafka my-cluster strimzi.io/kraft="migration" --overwrite
```

Enabling KRaft migration

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

```

kind: Kafka
metadata:
  name: my-cluster
  namespace: my-project
  annotations:
    strimzi.io/kraft: migration
# ...

```

Applying the annotation to the **Kafka** resource configuration starts the migration.

4. Check the controllers have started and the brokers have rolled:

```
kubectl get pods -n my-project
```

Output shows nodes in broker and controller node pools

NAME	READY	STATUS	RESTARTS
my-cluster-kafka-0	1/1	Running	0
my-cluster-kafka-1	1/1	Running	0
my-cluster-kafka-2	1/1	Running	0
my-cluster-controller-3	1/1	Running	0
my-cluster-controller-4	1/1	Running	0
my-cluster-controller-5	1/1	Running	0
# ...			

5. Check the status of the migration:

```
kubectl get kafka my-cluster -n my-project -w
```

Updates to the metadata state

NAME	METADATA STATE
my-cluster	Zookeeper
my-cluster	KRaftMigration
my-cluster	KRaftDualWriting
my-cluster	KRaftPostMigration

METADATA STATE shows the mechanism used to manage Kafka metadata and coordinate operations. At the start of the migration this is **ZooKeeper**.

- **ZooKeeper** is the initial state when metadata is only stored in ZooKeeper.
- **KRaftMigration** is the state when the migration is in progress. The flag to enable ZooKeeper to KRaft migration (`zookeeper.metadata.migration.enable`) is added to the brokers and they are rolled to register with the controllers. The migration can take some time at this point depending on the number of topics and partitions in the cluster.
- **KRaftDualWriting** is the state when the Kafka cluster is working as a KRaft cluster, but

metadata are being stored in both Kafka and ZooKeeper. Brokers are rolled a second time to remove the flag to enable migration.

- **KRaftPostMigration** is the state when KRaft mode is enabled for brokers. Metadata are still being stored in both Kafka and ZooKeeper.

The migration status is also represented in the `status.kafkaMetadataState` property of the `Kafka` resource.

WARNING

You can [roll back to using ZooKeeper from this point](#). The next step is to enable KRaft. Rollback cannot be performed after enabling KRaft.

- When the metadata state has reached `KRaftPostMigration`, enable KRaft in the `Kafka` resource configuration by setting the `strimzi.io/kraft` annotation to `enabled`:

```
kubectl annotate kafka my-cluster strimzi.io/kraft="enabled" --overwrite
```

Enabling KRaft migration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: my-project
  annotations:
    strimzi.io/kraft: enabled
# ...
```

- Check the status of the move to full KRaft mode:

```
kubectl get kafka my-cluster -n my-project -w
```

Updates to the metadata state

NAME	... METADATA STATE
my-cluster	... Zookeeper
my-cluster	... KRaftMigration
my-cluster	... KRaftDualWriting
my-cluster	... KRaftPostMigration
my-cluster	... PreKRaft
my-cluster	... KRaft

- **PreKRaft** is the state when all ZooKeeper-related resources have been automatically deleted.
- **KRaft** is the final state (after the controllers have rolled) when the KRaft migration is finalized.

NOTE

Depending on how `deleteClaim` is configured for ZooKeeper, its Persistent

Volume Claims (PVCs) and persistent volumes (PVs) may not be deleted. `deleteClaim` specifies whether the PVC is deleted when the cluster is uninstalled. The default is `false`.

8. Remove any ZooKeeper-related configuration from the `Kafka` resource.

Remove the following section:

- `spec.zookeeper`

If present, you can also remove the following options from the `.spec.kafka.config` section:

- `log.message.format.version`
- `inter.broker.protocol.version`

Removing `log.message.format.version` and `inter.broker.protocol.version` causes the brokers and controllers to roll again. Removing ZooKeeper properties removes any warning messages related to ZooKeeper configuration being present in a KRaft-operated cluster.

5.2.1. Performing a rollback on the migration

Before the migration is finalized by enabling KRaft in the `Kafka` resource, and the state has moved to the `KRaft` state, you can perform a rollback operation as follows:

1. Apply the `strimzi.io/kraft="rollback"` annotation to the `Kafka` resource to roll back the brokers.

```
kubectl annotate kafka my-cluster strimzi.io/kraft="rollback" --overwrite
```

Rolling back KRaft migration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: my-project
  annotations:
    strimzi.io/kraft: rollback
# ...
```

The migration process must be in the `KRaftPostMigration` state to do this. The brokers are rolled back so that they can be connected to ZooKeeper again and the state returns to `KRaftDualWriting`.

2. Delete the controllers node pool:

```
kubectl delete KafkaNodePool controller -n my-project
```

3. Apply the `strimzi.io/kraft="disabled"` annotation to the `Kafka` resource to return the metadata state to `ZooKeeper`.

```
kubectl annotate kafka my-cluster strimzi.io/kraft="disabled" --overwrite
```

Switching back to using ZooKeeper

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: my-project
  annotations:
    strimzi.io/kraft: disabled
# ...
```

Chapter 6. Deploying Strimzi using installation artifacts

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

You can deploy Strimzi 0.42.0 on Kubernetes 1.23 and later.

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

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

NOTE

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

6.1. Basic deployment path

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

The basic deployment path is as follows:

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

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

6.2. Deploying the Cluster Operator

The first step for any deployment of Strimzi is to install the Cluster Operator, which is responsible for deploying and managing Kafka clusters within a Kubernetes cluster. A single command applies all the installation files in the `install/cluster-operator` folder: `kubectl apply -f ./install/cluster-operator`.

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

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

Cluster-scoped resources like `CustomResourceDefinition`, `ClusterRole`, and `ClusterRoleBinding` require administrator privileges for installation. Prior to installation, it's advisable to review the `ClusterRole` specifications to ensure they do not grant unnecessary privileges.

After installation, the Cluster Operator runs as a regular [Deployment](#) to watch for updates of Kafka resources. Any standard (non-admin) Kubernetes user with privileges to access the [Deployment](#) can configure it. A cluster administrator can also grant standard users the [privileges necessary to manage Strimzi custom resources](#).

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

6.2.1. Specifying the namespaces the Cluster Operator watches

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

- [A single selected namespace](#) (the same namespace containing the Cluster Operator)
- [Multiple selected namespaces](#)
- [All namespaces in the cluster](#)

Watching multiple selected namespaces has the most impact on performance due to increased processing overhead. To optimize performance for namespace monitoring, it is generally recommended to either watch a single namespace or monitor the entire cluster. Watching a single namespace allows for focused monitoring of namespace-specific resources, while monitoring all namespaces provides a comprehensive view of the cluster's resources across all namespaces.

The Cluster Operator watches for changes to the following resources:

- [Kafka](#) for the Kafka cluster.

- [KafkaConnect](#) for the Kafka Connect cluster.
- [KafkaConnector](#) for creating and managing connectors in a Kafka Connect cluster.
- [KafkaMirrorMaker](#) for the Kafka MirrorMaker instance.
- [KafkaMirrorMaker2](#) for the Kafka MirrorMaker 2 instance.
- [KafkaBridge](#) for the Kafka Bridge instance.
- [KafkaRebalance](#) for the Cruise Control optimization requests.

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

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

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

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

NOTE

While the Cluster Operator can watch one, multiple, or all namespaces in a Kubernetes cluster, the Topic Operator and User Operator watch for [KafkaTopic](#) and [KafkaUser](#) resources in a single namespace. For more information, see [Watching Strimzi resources in Kubernetes namespaces](#).

6.2.2. Deploying the Cluster Operator to watch a single namespace

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

Prerequisites

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

Procedure

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

For example, in this procedure the Cluster Operator is installed into the namespace [my-cluster-operator-namespace](#).

On Linux, use:

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

On MacOS, use:

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

2. Deploy the Cluster Operator:

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

3. Check the status of the deployment:

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

Output shows the deployment name and readiness

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

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

6.2.3. Deploying the Cluster Operator to watch multiple namespaces

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

Prerequisites

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

Procedure

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

For example, in this procedure the Cluster Operator is installed into the namespace [my-cluster-operator-namespace](#).

On Linux, use:

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

On MacOS, use:

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

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

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

```
apiVersion: apps/v1  
kind: Deployment  
spec:  
  # ...  
  template:  
    spec:  
      serviceAccountName: strimzi-cluster-operator  
      containers:  
        - name: strimzi-cluster-operator  
          image: quay.io/strimzi/operator:0.42.0  
          imagePullPolicy: IfNotPresent  
          env:  
            - name: STRIMZI_NAMESPACE  
              value: watched-namespace-1,watched-namespace-2,watched-namespace-3
```

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

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

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

4. Deploy the Cluster Operator:

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

5. Check the status of the deployment:

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

Output shows the deployment name and readiness

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

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

6.2.4. Deploying the Cluster Operator to watch all namespaces

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

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

Prerequisites

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

Procedure

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

For example, in this procedure the Cluster Operator is installed into the namespace [my-cluster-operator-namespace](#).

On Linux, use:

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

On MacOS, use:

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

2. Edit the [install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml](#) file to set the value of the [STRIMZI_NAMESPACE](#) environment variable to [*](#).

```
apiVersion: apps/v1  
kind: Deployment  
spec:  
  # ...  
  template:  
    spec:  
      # ...
```

```

serviceAccountName: strimzi-cluster-operator
containers:
- name: strimzi-cluster-operator
  image: quay.io/strimzi/operator:0.42.0
  imagePullPolicy: IfNotPresent
  env:
  - name: STRIMZI_NAMESPACE
    value: "*"
  # ...

```

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

```

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

```

4. Deploy the Cluster Operator to your Kubernetes cluster.

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

5. Check the status of the deployment:

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

Output shows the deployment name and readiness

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

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

6.3. Deploying Kafka

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

After you have deployed the Cluster Operator, use a [Kafka](#) resource to deploy the following components:

- A Kafka cluster that uses KRaft or ZooKeeper:
 - [KRaft-based Kafka cluster](#)
 - [ZooKeeper-based Kafka cluster](#)
- [Topic Operator](#)
- [User Operator](#)

Node pools are used in the deployment of a Kafka cluster in KRaft (Kafka Raft metadata) mode, and may be used for the deployment of a Kafka cluster with ZooKeeper. Node pools represent a distinct group of Kafka nodes within the Kafka cluster that share the same configuration. For each Kafka node in the node pool, any configuration not defined in node pool is inherited from the cluster configuration in the [Kafka](#) resource.

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

6.3.1. Deploying a Kafka cluster in KRaft mode

This procedure shows how to deploy a Kafka cluster in KRaft mode and associated node pools using the Cluster Operator.

The deployment uses a YAML file to provide the specification to create a [Kafka](#) resource and [KafkaNodePool](#) resources.

Strimzi provides the following [example deployment files](#) that you can use to create a Kafka cluster that uses node pools:

[kafka/kraft/kafka-with-dual-role-nodes.yaml](#)

Deploys a Kafka cluster with one pool of nodes that share the broker and controller roles.

[kafka/kraft/kafka.yaml](#)

Deploys a persistent Kafka cluster with one pool of controller nodes and one pool of broker nodes.

[kafka/kraft/kafka-ephemeral.yaml](#)

Deploys an ephemeral Kafka cluster with one pool of controller nodes and one pool of broker nodes.

[kafka/kraft/kafka-single-node.yaml](#)

Deploys a Kafka cluster with a single node.

[kafka/kraft/kafka-jbod.yaml](#)

Deploys a Kafka cluster with multiple volumes in each broker node.

In this procedure, we use the example deployment file that deploys a Kafka cluster with one pool of nodes that share the broker and controller roles.

The `Kafka` resource configuration for each example includes the `strimzi.io/node-pools: enabled` annotation, which is required when using node pools. `Kafka` resources using KRaft mode must also have the annotation `strimzi.io/kraft: enabled`.

The example YAML files specify the latest supported Kafka version and KRaft metadata version used by the Kafka cluster.

NOTE

You can perform the steps outlined here to deploy a new Kafka cluster with `KafkaNodePool` resources or [migrate your existing Kafka cluster](#).

Prerequisites

- [The Cluster Operator must be deployed](#).

Before you begin

By default, the example deployment files specify `my-cluster` as the Kafka cluster name. The name cannot be changed after the cluster has been deployed. To change the cluster name before you deploy the cluster, edit the `Kafka.metadata.name` property of the `Kafka` resource in the relevant YAML file.

Procedure

1. Deploy a KRaft-based Kafka cluster.

To deploy a Kafka cluster with a single node pool that uses dual-role nodes:

```
kubectl apply -f examples/kafka/kraft/kafka-with-dual-role-nodes.yaml
```

2. Check the status of the deployment:

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

Output shows the node pool names and readiness

NAME	READY	STATUS	RESTARTS
my-cluster-entity-operator	3/3	Running	0
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-4	1/1	Running	0

- `my-cluster` is the name of the Kafka cluster.
- `pool-a` is the name of the node pool.

A sequential index number starting with `0` identifies each Kafka pod created. If you are using ZooKeeper, you'll also see the ZooKeeper pods.

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

Information on the deployment is also shown in the status of the **KafkaNodePool** resource, including a list of IDs for nodes in the pool.

NOTE

Node IDs are assigned sequentially starting at 0 (zero) across all node pools within a cluster. This means that node IDs might not run sequentially within a specific node pool. If there are gaps in the sequence of node IDs across the cluster, the next node to be added is assigned an ID that fills the gap. When scaling down, the node with the highest node ID within a pool is removed.

Additional resources

- [Kafka cluster configuration](#)
- [Node pool configuration](#)

6.3.2. Deploying a ZooKeeper-based Kafka cluster

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

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

Strimzi provides the following [example deployment files](#) to create a Kafka cluster that uses ZooKeeper for cluster management:

kafka-persistent.yaml

Deploys a persistent cluster with three ZooKeeper and three Kafka nodes.

kafka-jbod.yaml

Deploys a persistent cluster with three ZooKeeper and three Kafka nodes (each using multiple persistent volumes).

kafka-persistent-single.yaml

Deploys a persistent cluster with a single ZooKeeper node and a single Kafka node.

kafka-ephemeral.yaml

Deploys an ephemeral cluster with three ZooKeeper and three Kafka nodes.

kafka-ephemeral-single.yaml

Deploys an ephemeral cluster with three ZooKeeper nodes and a single Kafka node.

To deploy a Kafka cluster that uses node pools, the following example YAML file provides the specification to create a **Kafka** resource and **KafkaNodePool** resources:

kafka/kafka-with-node-pools.yaml

Deploys ZooKeeper with 3 nodes, and 2 different pools of Kafka brokers. Each of the pools has 3 brokers. The pools in the example use different storage configuration.

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

The example YAML files specify the latest supported Kafka version and inter-broker protocol version.

NOTE

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

Prerequisites

- [The Cluster Operator must be deployed.](#)

Before you begin

By default, the example deployment files specify `my-cluster` as the Kafka cluster name. The name cannot be changed after the cluster has been deployed. To change the cluster name before you deploy the cluster, edit the `Kafka.metadata.name` property of the `Kafka` resource in the relevant YAML file.

Procedure

1. Deploy a ZooKeeper-based Kafka cluster.

- To deploy an ephemeral cluster:

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

- To deploy a persistent cluster:

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

2. Check the status of the deployment:

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

Output shows the pod names and readiness

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

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

A sequential index number starting with `0` identifies each Kafka and ZooKeeper pod created.

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

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

Additional resources

- [Kafka cluster configuration](#)
- [Node pool configuration](#)

6.3.3. Deploying the Topic Operator using the Cluster Operator

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

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

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

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

For more information about configuring the `entityOperator` and `topicOperator` properties, see [Configuring the Entity Operator](#).

Prerequisites

- [The Cluster Operator must be deployed.](#)

Procedure

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

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

2. Configure the Topic Operator `spec` using the properties described in the [EntityTopicOperatorSpec schema reference](#).

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

3. Create or update the resource:

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

4. Check the status of the deployment:

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

Output shows the pod name and readiness

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

my-cluster is the name of the Kafka cluster.

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

6.3.4. Deploying the User Operator using the Cluster Operator

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

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

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

For more information about configuring the **entityOperator** and **userOperator** properties, see [Configuring the Entity Operator](#).

Prerequisites

- The Cluster Operator must be deployed.

Procedure

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

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

```
spec:  
  #...  
  entityOperator:  
    topicOperator: {}  
    userOperator: {}
```

- Configure the User Operator `spec` using the properties described in [EntityUserOperatorSpec schema reference](#).

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

- Create or update the resource:

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

- Check the status of the deployment:

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

Output shows the pod name and readiness

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

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

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

6.3.5. Connecting to ZooKeeper from a terminal

ZooKeeper services are secured with encryption and authentication and are not intended to be used by external applications that are not part of Strimzi.

However, if you want to use CLI tools that require a connection to ZooKeeper, you can use a terminal inside a ZooKeeper pod and connect to `localhost:12181` as the ZooKeeper address.

Prerequisites

- A Kubernetes cluster is available.
- A Kafka cluster is running.
- The Cluster Operator is running.

Procedure

- Open the terminal using the Kubernetes console or run the `exec` command from your CLI.

For example:

```
kubectl exec -ti my-cluster-zookeeper-0 -- bin/zookeeper-shell.sh localhost:12181  
ls /
```

Be sure to use `localhost:12181`.

6.3.6. List of Kafka cluster resources

The following resources are created by the Cluster Operator in the Kubernetes cluster.

Shared resources

`<kafka_cluster_name>-cluster-ca`

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

`<kafka_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.

`<kafka_cluster_name>-clients-ca`

Secret with the Clients CA private key used to sign user certificates

`<kafka_cluster_name>-clients-ca-cert`

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

`<kafka_cluster_name>-cluster-operator-certs`

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

ZooKeeper nodes

`<kafka_cluster_name>-zookeeper`

Name given to the following ZooKeeper resources:

- StrimziPodSet for managing the ZooKeeper node pods.
- Service account used by the ZooKeeper nodes.
- PodDisruptionBudget configured for the ZooKeeper nodes.

`<kafka_cluster_name>-zookeeper-<pod_id>`

Pods created by the StrimziPodSet.

`<kafka_cluster_name>-zookeeper-nodes`

Headless Service needed to have DNS resolve the ZooKeeper pods IP addresses directly.

`<kafka_cluster_name>-zookeeper-client`

Service used by Kafka brokers to connect to ZooKeeper nodes as clients.

<kafka_cluster_name>-zookeeper-config

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

<kafka_cluster_name>-zookeeper-nodes

Secret with ZooKeeper node keys.

<kafka_cluster_name>-network-policy-zookeeper

Network policy managing access to the ZooKeeper services.

data-<kafka_cluster_name>-zookeeper-<pod_id>

Persistent Volume Claim for the volume used for storing data for a specific ZooKeeper node. This resource will be created only if persistent storage is selected for provisioning persistent volumes to store data.

Kafka brokers

<kafka_cluster_name>-kafka

Name given to the following Kafka resources:

- StrimziPodSet for managing the Kafka broker pods.
- Service account used by the Kafka pods.
- PodDisruptionBudget configured for the Kafka brokers.

<kafka_cluster_name>-kafka-<pod_id>

Name given to the following Kafka resources:

- Pods created by the StrimziPodSet.
- ConfigMaps with Kafka broker configuration.

<kafka_cluster_name>-kafka-brokers

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

<kafka_cluster_name>-kafka-bootstrap

Service can be used as bootstrap servers for Kafka clients connecting from within the Kubernetes cluster.

<kafka_cluster_name>-kafka-external-bootstrap

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

<kafka_cluster_name>-kafka-<pod_id>

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

<kafka_cluster_name>-kafka-external-bootstrap

Bootstrap route for clients connecting from outside the Kubernetes cluster. This resource is

created only when an external listener is enabled and set to type `route`. The old route name will be used for backwards compatibility when the listener name is `external` and port is `9094`.

`<kafka_cluster_name>-kafka-<pod_id>`

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

`<kafka_cluster_name>-kafka-<listener_name>-bootstrap`

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

`<kafka_cluster_name>-kafka-<listener_name>-<pod_id>`

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

`<kafka_cluster_name>-kafka-<listener_name>-bootstrap`

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

`<kafka_cluster_name>-kafka-<listener_name>-<pod_id>`

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

`<kafka_cluster_name>-kafka-config`

ConfigMap containing the Kafka ancillary configuration, which is mounted as a volume by the broker pods when the `UseStrimziPodSets` feature gate is disabled.

`<kafka_cluster_name>-kafka-brokers`

Secret with Kafka broker keys.

`<kafka_cluster_name>-network-policy-kafka`

Network policy managing access to the Kafka services.

`strimzi-namespace-name-<kafka_cluster_name>-kafka-init`

Cluster role binding used by the Kafka brokers.

`<kafka_cluster_name>-jmx`

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

`data-<kafka_cluster_name>-kafka-<pod_id>`

Persistent Volume Claim for the volume used for storing data for a specific Kafka broker. This resource is created only if persistent storage is selected for provisioning persistent volumes to

store data.

`data-<id>-<kafka_cluster_name>-kafka-<pod_id>`

Persistent Volume Claim for the volume `id` used for storing data for a specific Kafka broker. This resource is created only if persistent storage is selected for JBOD volumes when provisioning persistent volumes to store data.

Kafka node pools

If you are using Kafka node pools, the resources created apply to the nodes managed in the node pools whether they are operating as brokers, controllers, or both. The naming convention includes the name of the Kafka cluster and the node pool: `<kafka_cluster_name>-<pool_name>`.

`<kafka_cluster_name>-<pool_name>`

Name given to the StrimziPodSet for managing the Kafka node pool.

`<kafka_cluster_name>-<pool_name>-<pod_id>`

Name given to the following Kafka node pool resources:

- Pods created by the StrimziPodSet.
- ConfigMaps with Kafka node configuration.

`data-<kafka_cluster_name>-<pool_name>-<pod_id>`

Persistent Volume Claim for the volume used for storing data for a specific node. This resource is created only if persistent storage is selected for provisioning persistent volumes to store data.

`data-<id>-<kafka_cluster_name>-<pool_name>-<pod_id>`

Persistent Volume Claim for the volume `id` used for storing data for a specific node. This resource is created only if persistent storage is selected for JBOD volumes when provisioning persistent volumes to store data.

Entity Operator

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

`<kafka_cluster_name>-entity-operator`

Name given to the following Entity Operator resources:

- Deployment with Topic and User Operators.
- Service account used by the Entity Operator.
- Network policy managing access to the Entity Operator metrics.

`<kafka_cluster_name>-entity-operator-<random_string>`

Pod created by the Entity Operator deployment.

`<kafka_cluster_name>-entity-topic-operator-config`

ConfigMap with ancillary configuration for Topic Operators.

`<kafka_cluster_name>-entity-user-operator-config`

ConfigMap with ancillary configuration for User Operators.

<kafka_cluster_name>-entity-topic-operator-certs

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

<kafka_cluster_name>-entity-user-operator-certs

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

strimzi-<kafka_cluster_name>-entity-topic-operator

Role binding used by the Entity Topic Operator.

strimzi-<kafka_cluster_name>-entity-user-operator

Role binding used by the Entity User Operator.

Kafka Exporter

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

<kafka_cluster_name>-kafka-exporter

Name given to the following Kafka Exporter resources:

- Deployment with Kafka Exporter.
- Service used to collect consumer lag metrics.
- Service account used by the Kafka Exporter.
- Network policy managing access to the Kafka Exporter metrics.

<kafka_cluster_name>-kafka-exporter-<random_string>

Pod created by the Kafka Exporter deployment.

Cruise Control

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

<kafka_cluster_name>-cruise-control

Name given to the following Cruise Control resources:

- Deployment with Cruise Control.
- Service used to communicate with Cruise Control.
- Service account used by the Cruise Control.

<kafka_cluster_name>-cruise-control-<random_string>

Pod created by the Cruise Control deployment.

<kafka_cluster_name>-cruise-control-config

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

<kafka_cluster_name>-cruise-control-certs

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

<kafka_cluster_name>-network-policy-cruise-control

Network policy managing access to the Cruise Control service.

6.4. Deploying Kafka Connect

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

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

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

The Cluster Operator manages Kafka Connect clusters deployed using the [KafkaConnect](#) resource and connectors created using the [KafkaConnector](#) resource.

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

- [Deploy a Kafka Connect cluster](#)
- [Add connectors to integrate with other systems](#)

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

6.4.1. Deploying Kafka Connect to your Kubernetes cluster

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

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

The deployment uses a YAML file to provide the specification to create a [KafkaConnect](#) resource.

Strimzi provides [example configuration files](#). In this procedure, we use the following example file:

- [examples/connect/kafka-connect.yaml](#)

IMPORTANT If deploying Kafka Connect clusters to run in parallel, each instance must use unique names for internal Kafka Connect topics. To do this, [configure each Kafka Connect instance to replace the defaults](#).

Prerequisites

- The Cluster Operator must be deployed.

Procedure

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

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

2. Check the status of the deployment:

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

Output shows the deployment name and readiness

NAME	READY	STATUS	RESTARTS
my-connect-cluster-connect-<pod_id>	1/1	Running	0

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

A pod ID identifies each pod created.

With the default deployment, you create a single Kafka Connect pod.

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

Additional resources

[Kafka Connect cluster configuration](#)

6.4.2. List of Kafka Connect cluster resources

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

<connect_cluster_name>-connect

Name given to the following Kafka Connect resources:

- StrimziPodSet that creates the Kafka Connect worker node pods.
- Headless service that provides stable DNS names to the Kafka Connect pods.
- Service account used by the Kafka Connect pods.
- Pod disruption budget configured for the Kafka Connect worker nodes.
- Network policy managing access to the Kafka Connect REST API.

<connect_cluster_name>-connect-<pod_id>

Pods created by the Kafka Connect StrimziPodSet.

<connect_cluster_name>-connect-api

Service which exposes the REST interface for managing the Kafka Connect cluster.

<connect_cluster_name>-connect-config

ConfigMap which contains the Kafka Connect ancillary configuration and is mounted as a volume by the Kafka Connect pods.

strimzi-<namespace-name>-<connect_cluster_name>-connect-init

Cluster role binding used by the Kafka Connect cluster.

<connect_cluster_name>-connect-build

Pod used to build a new container image with additional connector plugins (only when Kafka Connect Build feature is used).

<connect_cluster_name>-connect-dockerfile

ConfigMap with the Dockerfile generated to build the new container image with additional connector plugins (only when the Kafka Connect build feature is used).

6.5. Adding Kafka Connect connectors

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

Source connector

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

Sink connector

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

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

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

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

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

- [Using Strimzi's KafkaConnector custom resource](#)

- [Using the Kafka Connect API](#)

You can also create new connector instances using these options.

6.5.1. Building a new container image with connector plugins automatically

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

Prerequisites

- [The Cluster Operator must be deployed.](#)
- A container registry.

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

Procedure

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ①
  #...
  build:
    output: ②
    type: docker
    image: my-registry.io/my-org/my-connect-cluster:latest
    pushSecret: my-registry-credentials
    plugins: ③
      - name: connector-1
        artifacts:
          - type: tgz
            url: <url_to_download_connector_1_artifact>
            sha512sum: <SHA-512_checksum_of_connector_1_artifact>
      - name: connector-2
        artifacts:
          - type: jar
            url: <url_to_download_connector_2_artifact>
            sha512sum: <SHA-512_checksum_of_connector_2_artifact>
  #...
```

① The specification for the Kafka Connect cluster.

- ② (Required) Configuration of the container registry where new images are pushed.
- ③ (Required) List of connector plugins and their artifacts to add to the new container image.
Each plugin must be configured with at least one [artifact](#).

2. Create or update the resource:

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

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

Rebuilding the container image with new artifacts

A new container image is built automatically when you change the base image ([.spec.image](#)) or change the connector plugin artifacts configuration ([.spec.build.plugins](#)).

To pull an upgraded base image or to download the latest connector plugin artifacts without changing the [KafkaConnect](#) resource, you can trigger a rebuild of the container image associated with the Kafka Connect cluster by applying the annotation [strimzi.io/force-rebuild=true](#) to the Kafka Connect [StrimziPodSet](#) resource.

The annotation triggers the rebuilding process, fetching any new artifacts for plugins specified in the [KafkaConnect](#) custom resource and incorporating them into the container image. The rebuild includes downloads of new plugin artifacts without versions.

Additional resources

- [Kafka Connect Build schema reference](#)

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

Create a custom Docker image with connector plugins from the Kafka Connect base image. Add the custom image to the [/opt/kafka/plugins](#) directory.

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

At startup, the Strimzi version of Kafka Connect loads any third-party connector plugins contained in the [/opt/kafka/plugins](#) directory.

Prerequisites

- [The Cluster Operator must be deployed.](#)

Procedure

1. Create a new [Dockerfile](#) using [quay.io/strimzi/kafka:0.42.0-kafka-3.7.1](#) as the base image:

```
FROM quay.io/strimzi/kafka:0.42.0-kafka-3.7.1
```

```
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

Example plugins file

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

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

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

2. Build the container image.

3. Push your custom image to your container registry.

4. Point to the new container image.

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

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

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

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

① The specification for the Kafka Connect cluster.

② The docker image for Kafka Connect pods.

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

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

Additional resources

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

6.5.3. Deploying KafkaConnector resources

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

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

In the configuration shown in this procedure, the `autoRestart` feature is enabled (`enabled: true`) for automatic restarts of failed connectors and tasks. You can also annotate the `KafkaConnector` resource to `restart a connector` or `restart a connector task` manually.

Example connectors

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

Strimzi provides an [example KafkaConnector configuration file \(examples/connect/source-connector.yaml\)](#) for the example file connector plugins, which creates the following connector instances as `KafkaConnector` resources:

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

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

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

Prerequisites

- A Kafka Connect deployment
- The Cluster Operator is running

Procedure

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

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

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

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

Example KafkaConnector source connector configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector ①
  labels:
    strimzi.io/cluster: my-connect-cluster ②
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector ③
  tasksMax: 2 ④
  autoRestart: ⑤
  enabled: true
  config: ⑥
    file: "/opt/kafka/LICENSE" ⑦
    topic: my-topic ⑧
  # ...
```

- ① Name of the **KafkaConnector** resource, which is used as the name of the connector. Use any name that is valid for a Kubernetes resource.
- ② Name of the Kafka Connect cluster to create the connector instance in. Connectors must be deployed to the same namespace as the Kafka Connect cluster they link to.
- ③ Full name of the connector class. This should be present in the image being used by the Kafka Connect cluster.
- ④ Maximum number of Kafka Connect tasks that the connector can create.
- ⑤ Enables automatic restarts of failed connectors and tasks. By default, the number of restarts is indefinite, but you can set a maximum on the number of automatic restarts using the `maxRestarts` property.
- ⑥ **Connector configuration** as key-value pairs.
- ⑦ Location of the external data file. In this example, we're configuring the `FileStreamSourceConnector` to read from the `/opt/kafka/LICENSE` file.
- ⑧ Kafka topic to publish the source data to.

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

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

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

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

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

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

```

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

```

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

② [Connector configuration](#) as key-value pairs.

③ Temporary file to publish the source data to.

④ Kafka topic to read the source data from.

7. Create the sink [KafkaConnector](#) in your Kubernetes cluster:

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

8. Check that the connector resources were created:

```

kubectl get kctr --selector strimzi.io/cluster=<my_connect_cluster> -o name
my-source-connector
my-sink-connector

```

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

9. In the container, execute [kafka-console-consumer.sh](#) to read the messages that were written to the topic by the source connector:

```

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

```

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

Source and sink connector configuration options

The connector configuration is defined in the `spec.config` property of the [KafkaConnector](#) resource.

The [FileStreamSourceConnector](#) and [FileStreamSinkConnector](#) classes support the same configuration options as the Kafka Connect REST API. Other connectors support different configuration options.

Table 7. Configuration options for the `FileStreamSource` connector class

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

Table 8. Configuration options for `FileStreamSinkConnector` class

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

6.5.4. Exposing the Kafka Connect API

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

The operations supported by the Kafka Connect REST API are described in the [Apache Kafka Connect API documentation](#).

NOTE

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

You can add the connector configuration as a JSON object.

Example curl request to add connector configuration

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
```

```

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

```

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

- [LoadBalancer](#) or [NodePort](#) type services
- [Ingress](#) resources (Kubernetes only)
- OpenShift routes (OpenShift only)

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

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

Selector configuration for the service

```

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

```

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

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

You must also create a [NetworkPolicy](#) that allows HTTP requests from external clients.

Example NetworkPolicy to allow requests to the Kafka Connect API

```

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

```

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

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

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

6.5.5. Limiting access to the Kafka Connect API

It is crucial to restrict access to the Kafka Connect API only to trusted users to prevent unauthorized actions and potential security issues. The Kafka Connect API provides extensive capabilities for altering connector configurations, which makes it all the more important to take security precautions. Someone with access to the Kafka Connect API could potentially obtain sensitive information that an administrator may assume is secure.

The Kafka Connect REST API can be accessed by anyone who has authenticated access to the Kubernetes cluster and knows the endpoint URL, which includes the hostname/IP address and port number.

For example, suppose an organization uses a Kafka Connect cluster and connectors to stream sensitive data from a customer database to a central database. The administrator uses a configuration provider plugin to store sensitive information related to connecting to the customer database and the central database, such as database connection details and authentication credentials. The configuration provider protects this sensitive information from being exposed to unauthorized users. However, someone who has access to the Kafka Connect API can still obtain access to the customer database without the consent of the administrator. They can do this by setting up a fake database and configuring a connector to connect to it. They then modify the connector configuration to point to the customer database, but instead of sending the data to the central database, they send it to the fake database. By configuring the connector to connect to the fake database, the login details and credentials for connecting to the customer database are intercepted, even though they are stored securely in the configuration provider.

If you are using the [KafkaConnector](#) custom resources, then by default the Kubernetes RBAC rules permit only Kubernetes cluster administrators to make changes to connectors. You can also [designate non-cluster administrators to manage Strimzi resources](#). With [KafkaConnector](#) resources enabled in your Kafka Connect configuration, changes made directly using the Kafka Connect REST API are reverted by the Cluster Operator. If you are not using the [KafkaConnector](#) resource, the default RBAC rules do not limit access to the Kafka Connect API. If you want to limit direct access to the Kafka Connect REST API using Kubernetes RBAC, you need to enable and use the [KafkaConnector](#) resources.

For improved security, we recommend configuring the following properties for the Kafka Connect API:

org.apache.kafka.disallowed.login.modules

(Kafka 3.4 or later) Set the `org.apache.kafka.disallowed.login.modules` Java system property to prevent the use of insecure login modules. For example, specifying `com.sun.security.auth.module.JndiLoginModule` prevents the use of the Kafka `JndiLoginModule`.

Example configuration for disallowing login modules

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  jvmOptions:
    javaSystemProperties:
      - name: org.apache.kafka.disallowed.login.modules
        value: com.sun.security.auth.module.JndiLoginModule,
          org.apache.kafka.common.security.kerberos.KerberosLoginModule
  # ...
```

Only allow trusted login modules and follow the latest advice from Kafka for the version you are using. As a best practice, you should explicitly disallow insecure login modules in your Kafka Connect configuration by using the `org.apache.kafka.disallowed.login.modules` system property.

connector.client.config.override.policy

Set the `connector.client.config.override.policy` property to `None` to prevent connector configurations from overriding the Kafka Connect configuration and the consumers and producers it uses.

Example configuration to specify connector override policy

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    connector.client.config.override.policy: None
  # ...
```

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

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

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

WARNING

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

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

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

6.6. Deploying Kafka MirrorMaker

Kafka MirrorMaker replicates data between two or more Kafka clusters, within or across data centers. This process is called mirroring to avoid confusion with the concept of Kafka partition replication. MirrorMaker consumes messages from a source cluster and republishes those messages to a target cluster.

Data replication across clusters supports scenarios that require the following:

- Recovery of data in the event of a system failure
- Consolidation of data from multiple source clusters for centralized analysis
- Restriction of data access to a specific cluster
- Provision of data at a specific location to improve latency

6.6.1. Deploying Kafka MirrorMaker to your Kubernetes cluster

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

The deployment uses a YAML file to provide the specification to create a `KafkaMirrorMaker` or `KafkaMirrorMaker2` resource depending on the version of MirrorMaker deployed. MirrorMaker 2 is based on Kafka Connect and uses its configuration properties.

IMPORTANT

Kafka MirrorMaker 1 (referred to as just *MirrorMaker* in the documentation)

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

Strimzi provides [example configuration files](#). In this procedure, we use the following example files:

- [examples/mirror-maker/kafka-mirror-maker.yaml](#)
- [examples/mirror-maker/kafka-mirror-maker-2.yaml](#)

IMPORTANT

If deploying MirrorMaker 2 clusters to run in parallel, using the same target Kafka cluster, each instance must use unique names for internal Kafka Connect topics. To do this, [configure each MirrorMaker 2 instance to replace the defaults](#).

Prerequisites

- [The Cluster Operator must be deployed.](#)

Procedure

1. Deploy Kafka MirrorMaker to your Kubernetes cluster:

For MirrorMaker:

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

For MirrorMaker 2:

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

2. Check the status of the deployment:

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

Output shows the deployment name and readiness

NAME	READY	STATUS	RESTARTS
my-mirror-maker-mirror-maker-<pod_id>	1/1	Running	1
my-mm2-cluster-mirrormaker2-<pod_id>	1/1	Running	1

[my-mirror-maker](#) is the name of the Kafka MirrorMaker cluster. [my-mm2-cluster](#) is the name of the Kafka MirrorMaker 2 cluster.

A pod ID identifies each pod created.

With the default deployment, you install a single MirrorMaker or MirrorMaker 2 pod.

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

Additional resources

- [Kafka MirrorMaker cluster configuration](#)

6.6.2. List of Kafka MirrorMaker 2 cluster resources

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

<mirrormaker2_cluster_name>-mirrormaker2

Name given to the following MirrorMaker 2 resources:

- StrimziPodSet that creates the MirrorMaker 2 worker node pods.
- Headless service that provides stable DNS names to the MirrorMaker 2 pods.
- Service account used by the MirrorMaker 2 pods.
- Pod disruption budget configured for the MirrorMaker 2 worker nodes.
- Network Policy managing access to the MirrorMaker 2 REST API.

<mirrormaker2_cluster_name>-mirrormaker2-<pod_id>

Pods created by the MirrorMaker 2 StrimziPodSet.

<mirrormaker2_cluster_name>-mirrormaker2-api

Service which exposes the REST interface for managing the MirrorMaker 2 cluster.

<mirrormaker2_cluster_name>-mirrormaker2-config

ConfigMap which contains the MirrorMaker 2 ancillary configuration and is mounted as a volume by the MirrorMaker 2 pods.

strimzi-<namespace-name>-<mirrormaker2_cluster_name>-mirrormaker2-init

Cluster role binding used by the MirrorMaker 2 cluster.

6.6.3. List of Kafka MirrorMaker cluster resources

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

<mirrormaker_cluster_name>-mirror-maker

Name given to the following MirrorMaker resources:

- Deployment which is responsible for creating the MirrorMaker pods.
- Service account used by the MirrorMaker nodes.
- Pod Disruption Budget configured for the MirrorMaker worker nodes.

<mirrormaker_cluster_name>-mirror-maker-config

ConfigMap which contains ancillary configuration for MirrorMaker, and is mounted as a volume

by the MirrorMaker pods.

6.7. Deploying Kafka Bridge

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

6.7.1. Deploying Kafka Bridge to your Kubernetes cluster

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

The deployment uses a YAML file to provide the specification to create a [KafkaBridge](#) resource.

Strimzi provides [example configuration files](#). In this procedure, we use the following example file:

- [examples/bridge/kafka-bridge.yaml](#)

Prerequisites

- [The Cluster Operator must be deployed.](#)

Procedure

1. Deploy Kafka Bridge to your Kubernetes cluster:

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

2. Check the status of the deployment:

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

Output shows the deployment name and readiness

NAME	READY	STATUS	RESTARTS
my-bridge-bridge-<pod_id>	1/1	Running	0

[my-bridge](#) is the name of the Kafka Bridge cluster.

A pod ID identifies each pod created.

With the default deployment, you install a single Kafka Bridge pod.

[READY](#) shows the number of replicas that are ready/expected. The deployment is successful when the [STATUS](#) displays as [Running](#).

Additional resources

- [Kafka Bridge cluster configuration](#)
- [Using the Kafka Bridge](#)

6.7.2. Exposing the Kafka Bridge service to your local machine

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

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

Procedure

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

```
kubectl get pods -o name  
  
pod/kafka-consumer  
# ...  
pod/my-bridge-bridge-<pod_id>
```

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

```
kubectl port-forward pod/my-bridge-bridge-<pod_id> 8080:8080 &
```

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

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

6.7.3. Accessing the Kafka Bridge outside of Kubernetes

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

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

- **LoadBalancer** or **NodePort** type services
- **Ingress** resources (Kubernetes only)
- OpenShift routes (OpenShift only)

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

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

```
#...
```

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

6.7.4. List of Kafka Bridge cluster resources

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

<bridge_cluster_name>-bridge

Deployment which is in charge to create the Kafka Bridge worker node pods.

<bridge_cluster_name>-bridge-service

Service which exposes the REST interface of the Kafka Bridge cluster.

<bridge_cluster_name>-bridge-config

ConfigMap which contains the Kafka Bridge ancillary configuration and is mounted as a volume by the Kafka broker pods.

<bridge_cluster_name>-bridge

Pod Disruption Budget configured for the Kafka Bridge worker nodes.

6.8. Alternative standalone deployment options for Strimzi operators

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

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

6.8.1. Deploying the standalone Topic Operator

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

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

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

Prerequisites

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

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

Procedure

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

Example standalone Topic Operator deployment configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-topic-operator
  labels:
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-topic-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE ①
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS ②
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_RESOURCE_LABELS ③
              value: "strimzi.io/cluster=my-cluster"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS ④
              value: "120000"
            - name: STRIMZI_LOG_LEVEL ⑤
              value: INFO
            - name: STRIMZI_TLS_ENABLED ⑥
              value: "false"
            - name: STRIMZI_JAVA_OPTS ⑦
              value: "-Xmx=512M -Xms=256M"
            - name: STRIMZI_JAVA_SYSTEM_PROPERTIES ⑧
              value: "-Djavax.net.debug=verbose -DpropertyName=value"
            - name: STRIMZI_PUBLIC_CA ⑨
              value: "false"
            - name: STRIMZI_TLS_AUTH_ENABLED ⑩
              value: "false"
```

```

- name: STRIMZI_SASL_ENABLED ⑪
  value: "false"
- name: STRIMZI_SASL_USERNAME ⑫
  value: "admin"
- name: STRIMZI_SASL_PASSWORD ⑬
  value: "password"
- name: STRIMZI_SASL_MECHANISM ⑭
  value: "scram-sha-512"
- name: STRIMZI_SECURITY_PROTOCOL ⑮
  value: "SSL"
- name: STRIMZI_USE_FINALIZERS
  value: "false" ⑯

```

- ① The Kubernetes namespace for the Topic Operator to watch for `KafkaTopic` resources. Specify the namespace of the Kafka cluster.
- ② The host and port pair of the bootstrap broker address to discover and connect to all brokers in the Kafka cluster. Use a comma-separated list to specify two or three broker addresses in case a server is down.
- ③ The label to identify the `KafkaTopic` resources managed by the Topic Operator. This does not have to be the name of the Kafka cluster. It can be the label assigned to the `KafkaTopic` resource. If you deploy more than one Topic Operator, the labels must be unique for each. That is, the operators cannot manage the same resources.
- ④ The interval between periodic reconciliations, in milliseconds. The default is `120000` (2 minutes).
- ⑤ The level for printing logging messages. You can set the level to `ERROR`, `WARNING`, `INFO`, `DEBUG`, or `TRACE`.
- ⑥ Enables TLS support for encrypted communication with the Kafka brokers.
- ⑦ (Optional) The Java options used by the JVM running the Topic Operator.
- ⑧ (Optional) The debugging (`-D`) options set for the Topic Operator.
- ⑨ (Optional) Skips the generation of trust store certificates if TLS is enabled through `STRIMZI_TLS_ENABLED`. If this environment variable is enabled, the brokers must use a public trusted certificate authority for their TLS certificates. The default is `false`.
- ⑩ (Optional) Generates key store certificates for mTLS authentication. Setting this to `false` disables client authentication with mTLS to the Kafka brokers. The default is `true`.
- ⑪ (Optional) Enables SASL support for client authentication when connecting to Kafka brokers. The default is `false`.
- ⑫ (Optional) The SASL username for client authentication. Mandatory only if SASL is enabled through `STRIMZI_SASL_ENABLED`.
- ⑬ (Optional) The SASL password for client authentication. Mandatory only if SASL is enabled through `STRIMZI_SASL_ENABLED`.
- ⑭ (Optional) The SASL mechanism for client authentication. Mandatory only if SASL is enabled through `STRIMZI_SASL_ENABLED`. You can set the value to `plain`, `scram-sha-256`, or `scram-sha-512`.
- ⑮ (Optional) The security protocol used for communication with Kafka brokers. The default

value is "PLAINTEXT". You can set the value to `PLAINTEXT`, `SSL`, `SASL_PLAINTEXT`, or `SASL_SSL`.

- ⑯ Set `STRIMZI_USE_FINALIZERS` to `false` if you do not want to use finalizers to control [topic deletion](#).
2. If you want to connect to Kafka brokers that are using certificates from a public certificate authority, set `STRIMZI_PUBLIC_CA` to `true`. Set this property to `true`, for example, if you are using Amazon AWS MSK service.
 3. If you enabled mTLS with the `STRIMZI_TLS_ENABLED` environment variable, specify the keystore and truststore used to authenticate connection to the Kafka cluster.

Example mTLS configuration

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

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

② The password for accessing the truststore.

③ The keystore contains the private key for mTLS authentication.

④ The password for accessing the keystore.

4. If you need to configure custom SASL authentication, you can define the necessary authentication properties using the `STRIMZI_SASL_CUSTOM_CONFIG_JSON` environment variable for the standalone operator. For example, this configuration may be used for accessing a Kafka cluster in a cloud provider with a custom login module like the [Amazon MSK Library for AWS Identity and Access Management \(aws-msk_iam-auth\)](#).

The property `STRIMZI_ALTERABLE_TOPIC_CONFIG` defaults to `ALL`, allowing all `.spec.config` properties to be set in the `KafkaTopic` resource. If this setting is not suitable for a managed Kafka service, do as follows:

- If only a subset of properties is configurable, list them as comma-separated values.
- If no properties are to be configured, use `NONE`, which is equivalent to an empty property list.

NOTE

Only Kafka configuration properties starting with `sasl.` can be set with the `STRIMZI_SASL_CUSTOM_CONFIG_JSON` environment variable.

Example custom SASL configuration

```
# ....
env:
  - name: STRIMZI_SASL_ENABLED
    value: "true"
  - name: STRIMZI_SECURITY_PROTOCOL
    value: SASL_SSL
  - name: STRIMZI_SKIP_CLUSTER_CONFIG REVIEW ①
    value: "true"
  - name: STRIMZI_ALTERABLE_TOPIC_CONFIG ②
    value: compression.type, max.message.bytes,
message.timestamp.difference.max.ms, message.timestamp.type, retention.bytes,
retention.ms
  - name: STRIMZI_SASL_CUSTOM_CONFIG_JSON ③
    value: |
      {
        "sasl.mechanism": "AWS_MSK_IAM",
        "sasl.jaas.config": "software.amazon.msk.auth.iam.IAMLoginModule
required;",
        "sasl.client.callback.handler.class":
"software.amazon.msk.auth.iam.IAMClientCallbackHandler"
      }
  - name: STRIMZI_PUBLIC_CA
    value: "true"
  - name: STRIMZI_TRUSTSTORE_LOCATION
    value: /etc/pki/java/cacerts
  - name: STRIMZI_TRUSTSTORE_PASSWORD
    value: changeit
  - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS
    value: my-kafka-cluster-.kafka-serverless.us-east-1.amazonaws.com:9098
# ...
```

① Disables cluster configuration lookup for managed Kafka services that don't allow topic configuration changes.

② Defines the topic configuration properties that can be updated based on the limitations set by managed Kafka services.

③ Specifies the SASL properties to be set in JSON format. Only properties starting with `sasl`. are allowed.

Example Dockerfile with external jars

```
FROM strimzi/operator:latest

USER root

RUN mkdir -p ${STRIMZI_HOME}/external-libs
RUN chmod +rx ${STRIMZI_HOME}/external-libs

COPY ./aws-msk-iam-auth-and-dependencies/* ${STRIMZI_HOME}/external-libs/
ENV JAVA_CLASSPATH=${STRIMZI_HOME}/external-libs/*
```

USER 1001

5. Apply the changes to the [Deployment](#) configuration to deploy the Topic Operator.
6. Check the status of the deployment:

```
kubectl get deployments
```

Output shows the deployment name and readiness

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

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

6.8.2. Deploying the standalone User Operator

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

A standalone deployment can operate with any Kafka cluster.

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

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

Prerequisites

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

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

Procedure

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

Example standalone User Operator deployment configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-user-operator
  labels:
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-user-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE ①
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS ②
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_CA_CERT_NAME ③
              value: my-cluster-clients-ca-cert
            - name: STRIMZI_CA_KEY_NAME ④
              value: my-cluster-clients-ca
            - name: STRIMZI_LABELS ⑤
              value: "strimzi.io/cluster=my-cluster"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS ⑥
              value: "120000"
            - name: STRIMZI_WORK_QUEUE_SIZE ⑦
              value: 10000
            - name: STRIMZI_CONTROLLER_THREAD_POOL_SIZE ⑧
              value: 10
            - name: STRIMZI_USER_OPERATIONS_THREAD_POOL_SIZE ⑨
              value: 4
            - name: STRIMZI_LOG_LEVEL ⑩
              value: INFO
            - name: STRIMZI_GC_LOG_ENABLED ⑪
              value: "true"
            - name: STRIMZI_CA_VALIDITY ⑫
              value: "365"
            - name: STRIMZI_CA_RENEWAL ⑬
              value: "30"
            - name: STRIMZI_JAVA_OPTS ⑭
              value: "-Xmx=512M -Xms=256M"
            - name: STRIMZI_JAVA_SYSTEM_PROPERTIES ⑮
              value: "-Djavax.net.debug=verbose -DpropertyName=value"
            - name: STRIMZI_SECRET_PREFIX ⑯
```

```

        value: "kafka-"
    - name: STRIMZI_ACLS_ADMIN_API_SUPPORTED ⑯
        value: "true"
    - name: STRIMZI_MAINTENANCE_TIME_WINDOWS ⑰
        value: '* * 8-10 * * ?;* * 14-15 * * ?'
    - name: STRIMZI_KAFKA_ADMIN_CLIENT_CONFIGURATION ⑱
        value: |
            default.api.timeout.ms=120000
            request.timeout.ms=60000

```

- ① The Kubernetes namespace for the User Operator to watch for `KafkaUser` resources. Only one namespace can be specified.
- ② The host and port pair of the bootstrap broker address to discover and connect to all brokers in the Kafka cluster. Use a comma-separated list to specify two or three broker addresses in case a server is down.
- ③ The Kubernetes `Secret` that contains the public key (`ca.crt`) value of the CA (certificate authority) that signs new user certificates for mTLS authentication.
- ④ The Kubernetes `Secret` that contains the private key (`ca.key`) value of the CA that signs new user certificates for mTLS authentication.
- ⑤ The label to identify the `KafkaUser` resources managed by the User Operator. This does not have to be the name of the Kafka cluster. It can be the label assigned to the `KafkaUser` resource. If you deploy more than one User Operator, the labels must be unique for each. That is, the operators cannot manage the same resources.
- ⑥ The interval between periodic reconciliations, in milliseconds. The default is `120000` (2 minutes).
- ⑦ The size of the controller event queue. The size of the queue should be at least as big as the maximal amount of users you expect the User Operator to operate. The default is `1024`.
- ⑧ The size of the worker pool for reconciling the users. Bigger pool might require more resources, but it will also handle more `KafkaUser` resources. The default is `50`.
- ⑨ The size of the worker pool for Kafka Admin API and Kubernetes operations. Bigger pool might require more resources, but it will also handle more `KafkaUser` resources. The default is `4`.
- ⑩ The level for printing logging messages. You can set the level to `ERROR`, `WARNING`, `INFO`, `DEBUG`, or `TRACE`.
- ⑪ Enables garbage collection (GC) logging. The default is `true`.
- ⑫ The validity period for the CA. The default is `365` days.
- ⑬ The renewal period for the CA. The renewal period is measured backwards from the expiry date of the current certificate. The default is `30` days to initiate certificate renewal before the old certificates expire.
- ⑭ (Optional) The Java options used by the JVM running the User Operator
- ⑮ (Optional) The debugging (`-D`) options set for the User Operator
- ⑯ (Optional) Prefix for the names of Kubernetes secrets created by the User Operator.

- ⑯ (Optional) Indicates whether the Kafka cluster supports management of authorization ACL rules using the Kafka Admin API. When set to `false`, the User Operator will reject all resources with `simple` authorization ACL rules. This helps to avoid unnecessary exceptions in the Kafka cluster logs. The default is `true`.
- ⑰ (Optional) Semi-colon separated list of Cron Expressions defining the maintenance time windows during which the expiring user certificates will be renewed.
- ⑲ (Optional) Configuration options for configuring the Kafka Admin client used by the User Operator in the properties format.

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

Example mTLS configuration

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

- ① The Kubernetes `Secret` that contains the public key (`ca.crt`) value of the CA that signs Kafka broker certificates.
- ② The Kubernetes `Secret` that contains the certificate public key (`entity-operator.crt`) and private key (`entity-operator.key`) that is used for mTLS authentication against the Kafka cluster.

3. Deploy the User Operator.

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

4. Check the status of the deployment:

```
kubectl get deployments
```

Output shows the deployment name and readiness

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

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

Chapter 7. Deploying Strimzi from OperatorHub.io

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

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

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

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

For information on upgrades, see [Upgrading Strimzi](#).

WARNING

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

Chapter 8. Deploying Strimzi using Helm

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

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

For information on upgrades, see [Upgrading Strimzi](#).

Prerequisites

- The Helm client must be installed on a local machine.

Procedure

1. Install the Strimzi Cluster Operator using the Helm command line tool:

```
helm install strimzi-cluster-operator oci://quay.io/strimzi-helm/strimzi-kafka-operator
```

Alternatively, you can use parameter values to install a specific version of the Cluster Operator or specify any changes to the default configuration.

Example configuration that installs a specific version of the Cluster Operator and changes the number of replicas

```
helm install strimzi-cluster-operator --set replicas=2 --version 0.35.0  
oci://quay.io/strimzi-helm/strimzi-kafka-operator
```

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

```
helm ls
```

3. [Deploy Kafka and other Kafka components](#) using custom resources.

Chapter 9. Feature gates

Strimzi operators use feature gates to enable or disable specific features and functions. Enabling a feature gate alters the behavior of the associated operator, introducing the corresponding feature to your Strimzi deployment.

The purpose of feature gates is to facilitate the trial and testing of a feature before it is fully adopted. The state (enabled or disabled) of a feature gate may vary by default, depending on its maturity level.

As a feature gate graduates and reaches General Availability (GA), it transitions to an enabled state by default and becomes a permanent part of the Strimzi deployment. A feature gate at the GA stage cannot be disabled.

The supported feature gates are applicable to all Strimzi operators. While a particular feature gate might be used by one operator and ignored by the others, it can still be configured in all operators. When deploying the User Operator and Topic Operator within the context of the `Kafka` custom resource, the Cluster Operator automatically propagates the feature gates configuration to them. When the User Operator and Topic Operator are deployed standalone, without a Cluster Operator available to configure the feature gates, they must be directly configured within their deployments.

9.1. Graduated feature gates (GA)

Graduated feature gates have reached General Availability (GA) and are permanently enabled features.

9.1.1. ControlPlaneListener feature gate

The [ControlPlaneListener](#) feature gate separates listeners for data replication and coordination:

- Connections between the Kafka controller and brokers use an internal *control plane listener* on port 9090.
- Replication of data between brokers, as well as internal connections from Strimzi operators, Cruise Control, or the Kafka Exporter use a *replication listener* on port 9091.

IMPORTANT

With the [ControlPlaneListener](#) feature gate permanently enabled, direct upgrades or downgrades between Strimzi 0.22 and earlier and Strimzi 0.32 and newer are not possible. You must first upgrade or downgrade through one of the Strimzi versions in-between, disable the [ControlPlaneListener](#) feature gate, and then downgrade or upgrade (with the feature gate enabled) to the target version.

9.1.2. ServiceAccountPatching feature gate

The [ServiceAccountPatching](#) feature gate ensures that the Cluster Operator always reconciles service accounts and updates them when needed. For example, when you change service account labels or annotations using the [template](#) property of a custom resource, the operator automatically updates

them on the existing service account resources.

9.1.3. UseStrimziPodSets feature gate

The `UseStrimziPodSets` feature gate introduced the `StrimziPodSet` custom resource for managing Kafka and ZooKeeper pods, replacing the use of Kubernetes `StatefulSet` resources.

IMPORTANT

With the `UseStrimziPodSets` feature gate permanently enabled, direct downgrades from Strimzi 0.35 and newer to Strimzi 0.27 or earlier are not possible. You must first downgrade through one of the Strimzi versions in-between, disable the `UseStrimziPodSets` feature gate, and then downgrade to Strimzi 0.27 or earlier.

9.1.4. StableConnectIdentities feature gate

The `StableConnectIdentities` feature gate introduced the `StrimziPodSet` custom resource for managing Kafka Connect and Kafka MirrorMaker 2 pods, replacing the use of Kubernetes `Deployment` resources.

`StrimziPodSet` resources give the pods stable names and stable addresses, which do not change during rolling upgrades, replacing the use of Kubernetes `Deployment` resources.

IMPORTANT

With the `StableConnectIdentities` feature gate permanently enabled, direct downgrades from Strimzi 0.39 and newer to Strimzi 0.33 or earlier are not possible. You must first downgrade through one of the Strimzi versions in-between, disable the `StableConnectIdentities` feature gate, and then downgrade to Strimzi 0.33 or earlier.

9.1.5. KafkaNodePools feature gate

The `KafkaNodePools` feature gate introduced a new `KafkaNodePool` custom resource that enables the configuration of different *pools* of Apache Kafka nodes.

A node pool refers to a distinct group of Kafka nodes within a Kafka cluster. Each pool has its own unique configuration, which includes mandatory settings such as the number of replicas, storage configuration, and a list of assigned roles. You can assign the `controller` role, `broker` role, or both roles to all nodes in the pool using the `.spec.roles` property. When used with a ZooKeeper-based Apache Kafka cluster, it must be set to the `broker` role. When used with a KRaft-based Apache Kafka cluster, it can be set to `broker`, `controller`, or both.

In addition, a node pool can have its own configuration of resource requests and limits, Java JVM options, and resource templates. Configuration options not set in the `KafkaNodePool` resource are inherited from the `Kafka` custom resource.

The `KafkaNodePool` resources use a `strimzi.io/cluster` label to indicate to which Kafka cluster they belong. The label must be set to the name of the `Kafka` custom resource. The `Kafka` resource configuration must also include the `strimzi.io/node-pools: enabled` annotation, which is required when using node pools.

Examples of the [KafkaNodePool](#) resources can be found in the [example configuration files](#) provided by Strimzi.

Downgrading from KafkaNodePools

If your cluster already uses [KafkaNodePool](#) custom resources, and you wish to downgrade to an older version of Strimzi that does not support them or with the [KafkaNodePools](#) feature gate disabled, you must first migrate from [KafkaNodePool](#) custom resources to managing Kafka nodes using only [Kafka](#) custom resources. For more information, see the instructions for [reversing a migration to node pools](#).

9.1.6. UnidirectionalTopicOperator feature gate

The [UnidirectionalTopicOperator](#) feature gate introduced a unidirectional topic management mode for creating Kafka topics using the [KafkaTopic](#) resource. Unidirectional mode is compatible with using KRaft for cluster management. With unidirectional mode, you create Kafka topics using the [KafkaTopic](#) resource, which are then managed by the Topic Operator. Any configuration changes to a topic outside the [KafkaTopic](#) resource are reverted. For more information on topic management, see [Topic management](#).

9.1.7. UseKRaft feature gate

The [UseKRaft](#) feature gate introduced the KRaft (Kafka Raft metadata) mode for running Apache Kafka clusters without ZooKeeper. ZooKeeper and KRaft are mechanisms used to manage metadata and coordinate operations in Kafka clusters. KRaft mode eliminates the need for an external coordination service like ZooKeeper. In KRaft mode, Kafka nodes take on the roles of brokers, controllers, or both. They collectively manage the metadata, which is replicated across partitions. Controllers are responsible for coordinating operations and maintaining the cluster's state. For more information on using KRaft, see [Using Kafka in KRaft mode](#).

9.2. Stable feature gates (Beta)

Stable feature gates have reached a beta level of maturity, and are generally enabled by default for all users. Stable feature gates are production-ready, but they can still be disabled. Currently, there are no beta level feature gates.

9.3. Early access feature gates (Alpha)

Early access feature gates have not yet reached the beta stage, and are disabled by default. An early access feature gate provides an opportunity for assessment before its functionality is permanently incorporated into Strimzi.

9.3.1. ContinueReconciliationOnManualRollingUpdateFailure feature gate

The [ContinueReconciliationOnManualRollingUpdateFailure](#) feature gate has a default state of *disabled*.

The [ContinueReconciliationOnManualRollingUpdateFailure](#) feature gate allows the Cluster Operator to continue a reconciliation if the manual rolling update of the operands fails. It applies to the following operands that support manual rolling updates using the [strimzi.io/manual-rolling-](#)

`update` annotation:

- ZooKeeper
- Kafka
- Kafka Connect
- Kafka MirrorMaker 2

Continuing the reconciliation after a manual rolling update failure allows the operator to recover from various situations that might prevent the update from succeeding. For example, a missing Persistent Volume Claim (PVC) or Persistent Volume (PV) might cause the manual rolling update to fail. However, the PVCs and PVs are created only in a later stage of the reconciliation. By continuing the reconciliation after this failure, the process can recreate the missing PVC or PV and recover.

The `ContinueReconciliationOnManualRollingUpdateFailure` feature gate is used by the Cluster Operator. It is ignored by the User and Topic Operators.

Enabling the ContinueReconciliationOnManualRollingUpdateFailure feature gate

To enable the `ContinueReconciliationOnManualRollingUpdateFailure` feature gate, specify `+ContinueReconciliationOnManualRollingUpdateFailure` in the `STRIMZI_FEATURE_GATES` environment variable in the Cluster Operator configuration.

9.4. Enabling feature gates

To modify a feature gate's default state, use the `STRIMZI_FEATURE_GATES` environment variable in the operator's configuration. You can modify multiple feature gates using this single environment variable. Specify a comma-separated list of feature gate names and prefixes. A `+` prefix enables the feature gate and a `-` prefix disables it.

Example feature gate configuration that enables FeatureGate1 and disables FeatureGate2

```
env:  
  - name: STRIMZI_FEATURE_GATES  
    value: +FeatureGate1,-FeatureGate2
```

9.5. Feature gate releases

Feature gates have three stages of maturity:

- Alpha — typically disabled by default
- Beta — typically enabled by default
- General Availability (GA) — typically always enabled

Alpha stage features might be experimental or unstable, subject to change, or not sufficiently tested for production use. Beta stage features are well tested and their functionality is not likely to change. GA stage features are stable and should not change in the future. Alpha and beta stage features are removed if they do not prove to be useful.

- The `ControlPlaneListener` feature gate moved to GA stage in Strimzi 0.32. It is now permanently enabled and cannot be disabled.
- The `ServiceAccountPatching` feature gate moved to GA stage in Strimzi 0.30. It is now permanently enabled and cannot be disabled.
- The `UseStimziPodSets` feature gate moved to GA stage in Strimzi 0.35 and the support for StatefulSets is completely removed. It is now permanently enabled and cannot be disabled.
- The `StableConnectIdentities` feature gate moved to GA stage in Strimzi 0.39. It is now permanently enabled and cannot be disabled.
- The `KafkaNodePools` feature gate moved to GA stage in Strimzi 0.41. It is now permanently enabled and cannot be disabled. To use the Kafka Node Pool resources, you still need to use the `stimzi.io/node-pools: enabled` annotation on the `Kafka` custom resources.
- The `UnidirectionalTopicOperator` feature gate moved to GA stage in Strimzi 0.41. It is now permanently enabled and cannot be disabled.
- The `UseKRaft` feature gate moved to GA stage in Strimzi 0.42. It is now permanently enabled and cannot be disabled. To use KRaft (ZooKeeper-less Apache Kafka), you still need to use the `stimzi.io/kraft: enabled` annotation on the `Kafka` custom resources or migrate from an existing ZooKeeper-based cluster.
- The `ContinueReconciliationOnManualRollingUpdateFailure` feature was introduced in Strimzi 0.41 and is disabled by default.

NOTE

Feature gates might be removed when they reach GA. This means that the feature was incorporated into the Strimzi core features and can no longer be disabled.

Table 9. Feature gates and the Strimzi versions when they moved to alpha, beta, or GA

Feature gate	Alpha	Beta	GA
<code>ControlPlaneListener</code>	0.23	0.27	0.32
<code>ServiceAccountPatching</code>	0.24	0.27	0.30
<code>UseStimziPodSets</code>	0.28	0.30	0.35
<code>UseKRaft</code>	0.29	0.40	0.42
<code>StableConnectIdentitie s</code>	0.34	0.37	0.39
<code>KafkaNodePools</code>	0.36	0.39	0.41
<code>UnidirectionalTopicOpe rator</code>	0.36	0.39	0.41
<code>ContinueReconciliation OnManualRollingUpdateF ailure</code>	0.41	0.43 (planned)	n/a

If a feature gate is enabled, you may need to disable it before upgrading or downgrading from a specific Strimzi version (or first upgrade / downgrade to a version of Strimzi where it can be disabled). The following table shows which feature gates you need to disable when upgrading or downgrading Strimzi versions.

Table 10. Feature gates to disable when upgrading or downgrading Strimzi

Disable Feature gate	Upgrading from Strimzi version	Downgrading to Strimzi version
ControlPlaneListener	0.22 and earlier	0.22 and earlier
UseStrimziPodSets	-	0.27 and earlier
StableConnectIdentities	-	0.33 and earlier

Chapter 10. Configuring a deployment

Configure and manage a Strimzi deployment to your precise needs using Strimzi custom resources. Strimzi provides example custom resources with each release, allowing you to configure and create instances of supported Kafka components. Fine-tune your deployment by configuring custom resources to include additional features according to your specific requirements.

Use custom resources to configure and create instances of the following components:

- Kafka clusters
- Kafka Connect clusters
- Kafka MirrorMaker
- Kafka Bridge
- Cruise Control

You can use configuration to manage your instances or modify your deployment to introduce additional features. New features are sometimes introduced through feature gates, which are controlled through operator configuration.

The [Strimzi Custom Resource API Reference](#) describes the properties you can use in your configuration.

Important Kafka configuration options

Through configuration of the [Kafka](#) resource, you can introduce the following:

- Data storage
- Rack awareness
- Listeners for authenticated client access to the Kafka cluster
- Topic Operator for managing Kafka topics
- User Operator for managing Kafka users (clients)
- Cruise Control for cluster rebalancing
- Kafka Exporter for collecting lag metrics

Use [KafkaNodePool](#) resources to configure distinct groups of nodes within a Kafka cluster.

Common configuration

Common configuration is configured independently for each component, such as the following:

- Metrics configuration
- Healthchecks and liveness probes
- Resource limits and requests (CPU/Memory)
- Logging frequency
- JVM options for maximum and minimum memory allocation

Config maps to centralize configuration

For specific areas of configuration, namely metrics, logging, and external configuration for Kafka Connect connectors, you can also use [ConfigMap](#) resources. By using a [ConfigMap](#) resource to incorporate configuration, you centralize maintenance. You can also use configuration providers to load configuration from external sources, which we recommend for supplying the credentials for Kafka Connect connector configuration.

TLS certificate management

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

Applying changes to a custom resource configuration file

You add configuration to a custom resource using `spec` properties. After adding the configuration, you can use `kubectl` to apply the changes to a custom resource configuration file:

Applying changes to a resource configuration file

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

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

10.1. Using example configuration files

Further enhance your deployment by incorporating additional supported configuration. Example configuration files are provided with the downloadable release artifacts from the [GitHub releases page](#). You can also access the example files directly from the [examples directory](#).

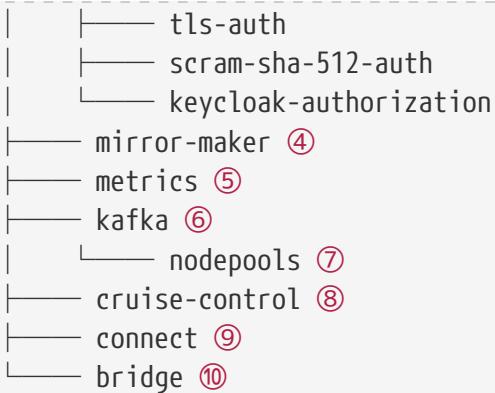
The example files include only the essential properties and values for custom resources by default. You can download and apply the examples using the `kubectl` command-line tool. The examples can serve as a starting point when building your own Kafka component configuration for deployment.

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

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

Example configuration files provided with Strimzi

```
examples
├── user ①
├── topic ②
└── security ③
```



- ① [KafkaUser](#) custom resource configuration, which is managed by the User Operator.
- ② [KafkaTopic](#) custom resource configuration, which is managed by Topic Operator.
- ③ Authentication and authorization configuration for Kafka components. Includes example configuration for TLS and SCRAM-SHA-512 authentication. The Keycloak example includes [Kafka](#) custom resource configuration and a Keycloak realm specification. You can use the example to try Keycloak authorization services. There is also an example with enabled [oauth](#) authentication and [keycloak](#) authorization metrics.
- ④ [Kafka](#) custom resource configuration for a deployment of Mirror Maker. Includes example configuration for replication policy and synchronization frequency.
- ⑤ [Metrics configuration](#), including Prometheus installation and Grafana dashboard files.
- ⑥ [Kafka](#) custom resource configuration for a deployment of Kafka. Includes example configuration for an ephemeral or persistent single or multi-node deployment.
- ⑦ [KafkaNodePool](#) configuration for Kafka nodes in a Kafka cluster. Includes example configuration for nodes in clusters that use KRaft (Kafka Raft metadata) mode or ZooKeeper.
- ⑧ [Kafka](#) custom resource with a deployment configuration for Cruise Control. Includes [KafkaRebalance](#) custom resources to generate optimization proposals from Cruise Control, with example configurations to use the default or user optimization goals.
- ⑨ [KafkaConnect](#) and [KafkaConnector](#) custom resource configuration for a deployment of Kafka Connect. Includes example configurations for a single or multi-node deployment.
- ⑩ [KafkaBridge](#) custom resource configuration for a deployment of Kafka Bridge.

10.2. Configuring Kafka in KRaft mode

Update the `spec` properties of the [Kafka](#) custom resource to configure your deployment of Kafka in KRaft mode.

As well as configuring Kafka, you can add configuration for Strimzi operators.

The KRaft metadata version (`.spec.kafka.metadataVersion`) must be a version supported by the Kafka version (`spec.kafka.version`). If the metadata version is not set in the configuration, the Cluster Operator updates the version to the default for the Kafka version used.

NOTE

The oldest supported metadata version is 3.3. Using a metadata version that is older than the Kafka version might cause some features to be disabled.

Kafka clusters operating in KRaft mode also use node pools. The following must be specified in the [node pool configuration](#):

- Roles assigned to each node within the Kafka cluster
- Number of replica nodes used
- Storage specification for the nodes

Other optional properties may also be set in node pools.

For a deeper understanding of the Kafka cluster configuration options, refer to the [Strimzi Custom Resource API Reference](#).

Example Kafka custom resource configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    version: 3.7.1 ①
    metadataVersion: 3.7 ②
    logging: ③
      type: inline
      loggers:
        kafka.root.logger.level: INFO
    resources: ④
      requests:
        memory: 64Gi
        cpu: "8"
      limits:
        memory: 64Gi
        cpu: "12"
    readinessProbe: ⑤
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    jvmOptions: ⑥
      -Xms: 8192m
      -Xmx: 8192m
    image: my-org/my-image:latest ⑦
    listeners: ⑧
      - name: plain ⑨
        port: 9092 ⑩
        type: internal ⑪
        tls: false ⑫
        configuration:
          useServiceDnsDomain: true ⑬
```

```

- name: tls
  port: 9093
  type: internal
  tls: true
  authentication: ⑯
    type: tls
- name: external1 ⑯
  port: 9094
  type: route
  tls: true
  configuration:
    brokerCertChainAndKey: ⑯
      secretName: my-secret
      certificate: my-certificate.crt
      key: my-key.key
  authorization: ⑰
    type: simple
  config: ⑱
    auto.create.topics.enable: "false"
    offsets.topic.replication.factor: 3
    transaction.state.log.replication.factor: 3
    transaction.state.log.min_isr: 2
    default.replication.factor: 3
    min.insync.replicas: 2
  rack: ⑲
    topologyKey: topology.kubernetes.io/zone
  metricsConfig: ⑳
    type: jmxPrometheusExporter
    valueFrom:
      configMapKeyRef:
        name: my-config-map
        key: my-key
  entityOperator:
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalMs: 60000
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
  resources:
    requests:
      memory: 512Mi
      cpu: "1"
    limits:
      memory: 512Mi
      cpu: "1"
  userOperator:
    watchedNamespace: my-topic-namespace
    reconciliationIntervalMs: 60000
    logging:

```

```

type: inline
loggers:
  rootLogger.level: INFO
resources:
  requests:
    memory: 512Mi
    cpu: "1"
  limits:
    memory: 512Mi
    cpu: "1"
kafkaExporter:
  # ...
cruiseControl:
  # ...

```

- ① Kafka version, which can be changed to a supported version by following the upgrade procedure.
- ② Kafka metadata version, which can be changed to a supported version by following the upgrade procedure.
- ③ Kafka loggers and log levels added directly (`inline`) or indirectly (`external`) through a ConfigMap. A custom Log4j configuration must be placed under the `log4j.properties` key in the ConfigMap. For the Kafka `kafka.root.logger.level` logger, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
- ④ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑤ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
- ⑥ JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka.
- ⑦ ADVANCED OPTION: Container image configuration, which is recommended only in special situations.
- ⑧ Listeners configure how clients connect to the Kafka cluster via bootstrap addresses. Listeners are configured as *internal* or *external* listeners for connection from inside or outside the Kubernetes cluster.
- ⑨ Name to identify the listener. Must be unique within the Kafka cluster.
- ⑩ Port number used by the listener inside Kafka. The port number has to be unique within a given Kafka cluster. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX. Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.
- ⑪ Listener type specified as `internal` or `cluster-ip` (to expose Kafka using per-broker `ClusterIP` services), or for external listeners, as `route` (OpenShift only), `loadbalancer`, `nodeport` or `ingress` (Kubernetes only).
- ⑫ Enables or disables TLS encryption for each listener. For `route` and `ingress` type listeners, TLS encryption must always be enabled by setting it to `true`.

- ⑯ Defines whether the fully-qualified DNS names including the cluster service suffix (usually `.cluster.local`) are assigned.
- ⑰ Listener authentication mechanism specified as mTLS, SCRAM-SHA-512, or token-based OAuth 2.0.
- ⑱ External listener configuration specifies how the Kafka cluster is exposed outside Kubernetes, such as through a `route`, `loadbalancer` or `nodeport`.
- ⑲ Optional configuration for a Kafka listener certificate managed by an external CA (certificate authority). The `brokerCertChainAndKey` specifies a `Secret` that contains a server certificate and a private key. You can configure Kafka listener certificates on any listener with enabled TLS encryption.
- ⑳ Authorization enables simple, OAUTH 2.0, or OPA authorization on the Kafka broker. Simple authorization uses the `AclAuthorizer` and `StandardAuthorizer` Kafka plugins.
- ㉑ Broker configuration. Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by Strimzi.
- ㉒ Rack awareness configuration to spread replicas across different racks, data centers, or availability zones. The `topologyKey` must match a node label containing the rack ID. The example used in this configuration specifies a zone using the standard `topology.kubernetes.io/zone` label.
- ㉓ Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter (the default metrics exporter).

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

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

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

Specified User Operator loggers and log levels.

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

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

10.2.1. Setting throughput and storage limits on brokers

Use the *Kafka Static Quota* plugin to set throughput and storage limits on brokers in your Kafka cluster. You enable the plugin and set limits by configuring the `quotas` section of the `Kafka` resource. You can set a byte-rate threshold and storage quotas to put limits on the clients interacting with your brokers.

NOTE Only one quota plugin can be used in Kafka. By enabling this plugin, the built-in Kafka quotas plugin is automatically disabled. This means that you won't see per-

client quota metrics, but only aggregated quota metrics.

You can set byte-rate thresholds for producer and consumer bandwidth. The total limit is distributed across all clients accessing the broker. For example, you can set a byte-rate threshold of 40 MBps for producers. If two producers are running, they are each limited to a throughput of 20 MBps.

Storage quotas enforce the throttling of Kafka producers based on Kafka disk storage utilization when it reaches the specified limit. You can specify the limit in bytes or percentage of available disk space. The limit applies to every disk individually. It prevents disks filling up too quickly and exceeding their capacity. Full disks can lead to issues that are hard to rectify.

Prerequisites

- The Cluster Operator that manages the Kafka cluster is running.

Procedure

1. Add the plugin configuration to the `quotas` section of the `Kafka` resource.

The plugin configuration is shown in this example configuration.

Example Kafka Static Quota plugin configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    quotas:
      type: strimzi
      producerByteRate: 1000000 ①
      consumerByteRate: 1000000 ②
      minAvailableBytesPerVolume: 500000000000 ③
      excludedPrincipals: ④
        - my-user
```

① Sets the producer byte-rate threshold. 1 MBps in this example.

② Sets the consumer byte-rate threshold. 1 MBps in this example.

③ Sets the available bytes limit for storage. 500 GB in this example.

④ Sets the list of excluded users that are removed from the quota. `my-user` in this example.

2. Apply the changes to the `Kafka` configuration.

NOTE

`minAvailableBytesPerVolume` and `minAvailableRatioPerVolume` are mutually exclusive. This means that only one of these parameters should be configured.

Additional resources

- [QuotasPluginStrimzi schema reference](#)
- [KafkaUserQuotas schema reference](#)

10.2.2. Deleting Kafka nodes using annotations

This procedure describes how to delete an existing Kafka node by using a 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 and the availability of your cluster cannot be guaranteed. The following procedure should only be performed if you have encountered storage issues.

Prerequisites

- A running Cluster Operator

Procedure

1. Find the name of the [Pod](#) that you want to delete.

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

2. Use `kubectl annotate` to annotate the [Pod](#) resource in Kubernetes:

```
kubectl annotate pod <cluster_name>-kafka-<index_number> strimzi.io/delete-pod-and-pvc="true"
```

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

10.3. Configuring Kafka with ZooKeeper

Update the `spec` properties of the [Kafka](#) custom resource to configure your deployment of Kafka with ZooKeeper.

As well as configuring Kafka, you can add configuration for ZooKeeper and the Strimzi operators. The configuration options for Kafka and the Strimzi operators are the same as when using Kafka in KRaft mode. For descriptions of the properties, see [Configuring Kafka in KRaft mode](#).

The inter-broker protocol version (`inter.broker.protocol.version`) must be a version supported by the Kafka version (`spec.kafka.version`). If the inter-broker protocol version is not set in the configuration, the Cluster Operator updates the version to the default for the Kafka version used.

If you are also using node pools, the following must be specified in the [node pool configuration](#):

- Roles assigned to each node within the Kafka cluster

- Number of replica nodes used
- Storage specification for the nodes

If set in the node pool configuration, the equivalent configuration in the [Kafka](#) resource, such as `spec.kafka.replicas`, is not required. Other optional properties may also be set in node pools.

For a deeper understanding of the ZooKeeper cluster configuration options, refer to the [Strimzi Custom Resource API Reference](#).

Example Kafka custom resource configuration when using ZooKeeper

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    version: 3.7.1
    logging:
      type: inline
      loggers:
        kafka.root.logger.level: INFO
    resources:
      requests:
        memory: 64Gi
        cpu: "8"
      limits:
        memory: 64Gi
        cpu: "12"
    readinessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    jvmOptions:
      -Xms: 8192m
      -Xmx: 8192m
    image: my-org/my-image:latest
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
        configuration:
          useServiceDnsDomain: true
      - name: tls
        port: 9093
        type: internal
        tls: true
```

```

authentication:
  type: tls
  - name: external1
    port: 9094
    type: route
    tls: true
  configuration:
    brokerCertChainAndKey:
      secretName: my-secret
      certificate: my-certificate.crt
      key: my-key.key
  authorization:
    type: simple
  config:
    auto.create.topics.enable: "false"
    offsets.topic.replication.factor: 3
    transaction.state.log.replication.factor: 3
    transaction.state.log.min_isr: 2
    default.replication.factor: 3
    min.insync.replicas: 2
    inter.broker.protocol.version: "3.7"
  storage:
    type: persistent-claim
    size: 10000Gi
  rack:
    topologyKey: topology.kubernetes.io/zone
  metricsConfig:
    type: jmxPrometheusExporter
    valueFrom:
      configMapKeyRef:
        name: my-config-map
        key: my-key
    # ...
zookeeper: ①
  replicas: 3 ②
  logging: ③
    type: inline
    loggers:
      zookeeper.root.logger: INFO
resources: ④
  requests:
    memory: 8Gi
    cpu: "2"
  limits:
    memory: 8Gi
    cpu: "2"
jvmOptions: ⑤
  -Xms: 4096m
  -Xmx: 4096m
storage: ⑥
  type: persistent-claim

```

```

size: 1000Gi
metricsConfig: ⑦
  type: jmxPrometheusExporter
valueFrom:
  configMapKeyRef: ⑧
    name: my-config-map
    key: my-key
# ...
entityOperator:
topicOperator:
  watchedNamespace: my-topic-namespace
  reconciliationIntervalSeconds: 60
logging:
  type: inline
  loggers:
    rootLogger.level: INFO
resources:
  requests:
    memory: 512Mi
    cpu: "1"
  limits:
    memory: 512Mi
    cpu: "1"
userOperator:
  watchedNamespace: my-topic-namespace
  reconciliationIntervalSeconds: 60
logging:
  type: inline
  loggers:
    rootLogger.level: INFO
resources:
  requests:
    memory: 512Mi
    cpu: "1"
  limits:
    memory: 512Mi
    cpu: "1"
kafkaExporter:
# ...
cruiseControl:
# ...

```

- ① ZooKeeper-specific configuration contains properties similar to the Kafka configuration.
- ② The number of ZooKeeper nodes. ZooKeeper clusters or ensembles usually run with an odd number of nodes, typically three, five, or seven. The majority of nodes must be available in order to maintain an effective quorum. If the ZooKeeper cluster loses its quorum, it will stop responding to clients and the Kafka brokers will stop working. Having a stable and highly available ZooKeeper cluster is crucial for Strimzi.
- ③ ZooKeeper loggers and log levels.

- ④ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑤ JVM configuration options to optimize performance for the Virtual Machine (VM) running ZooKeeper.
- ⑥ Storage size for persistent volumes may be increased and additional volumes may be added to JBOD storage.
- ⑦ Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter (the default metrics exporter).
- ⑧ Rules for exporting metrics in Prometheus format to a Grafana dashboard through the Prometheus JMX Exporter, which are enabled by referencing a ConfigMap containing configuration for the Prometheus JMX exporter. You can enable metrics without further configuration using a reference to a ConfigMap containing an empty file under `metricsConfig.valueFrom.configMapKeyRef.key`.

10.3.1. Default ZooKeeper configuration values

When deploying ZooKeeper with Strimzi, some of the default configuration set by Strimzi differs from the standard ZooKeeper defaults. This is because Strimzi sets a number of ZooKeeper properties with values that are optimized for running ZooKeeper within a Kubernetes environment.

The default configuration for key ZooKeeper properties in Strimzi is as follows:

Table 11. Default ZooKeeper Properties in Strimzi

Property	Default value	Description
<code>tickTime</code>	2000	The length of a single tick in milliseconds, which determines the length of a session timeout.
<code>initLimit</code>	5	The maximum number of ticks that a follower is allowed to fall behind the leader in a ZooKeeper cluster.
<code>syncLimit</code>	2	The maximum number of ticks that a follower is allowed to be out of sync with the leader in a ZooKeeper cluster.
<code>autopurge.purgeInterval</code>	1	Enables the <code>autopurge</code> feature and sets the time interval in hours for purging the server-side ZooKeeper transaction log.
<code>admin.enableServer</code>	false	Flag to disable the ZooKeeper admin server. The admin server is not used by Strimzi.

IMPORTANT

Modifying these default values as `zookeeper.config` in the `Kafka` custom resource may impact the behavior and performance of your ZooKeeper cluster.

10.3.2. Deleting ZooKeeper nodes using annotations

This procedure describes how to delete an existing ZooKeeper node by using a 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 and the availability of your cluster cannot be guaranteed. The following procedure should only be performed if you have encountered storage issues.

Prerequisites

- A running Cluster Operator

Procedure

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

ZooKeeper pods are named `<cluster_name>-zookeeper-<index_number>`, where `<index_number>` starts at zero and ends at the total number of replicas minus one. For example, `my-cluster-zookeeper-0`.

2. Use `kubectl annotate` to annotate the **Pod** resource in Kubernetes:

```
kubectl annotate pod <cluster_name>-zookeeper-<index_number> strimzi.io/delete-pod-and-pvc="true"
```

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

10.4. Configuring node pools

Update the `spec` properties of the **KafkaNodePool** custom resource to configure a node pool deployment.

A node pool refers to a distinct group of Kafka nodes within a Kafka cluster. Each pool has its own unique configuration, which includes mandatory settings for the number of replicas, roles, and storage allocation.

Optionally, you can also specify values for the following properties:

- `resources` to specify memory and cpu requests and limits
- `template` to specify custom configuration for pods and other Kubernetes resources
- `jvmOptions` to specify custom JVM configuration for heap size, runtime and other options

The relationship between **Kafka** and **KafkaNodePool** resources is as follows:

- **Kafka** resources represent the configuration for all nodes in a Kafka cluster.

- `KafkaNodePool` resources represent the configuration for nodes only in the node pool.

If a configuration property is not specified in `KafkaNodePool`, it is inherited from the `Kafka` resource. Configuration specified in the `KafkaNodePool` resource takes precedence if set in both resources. For example, if both the node pool and Kafka configuration includes `jvmOptions`, the values specified in the node pool configuration are used. When `-Xmx: 1024m` is set in `KafkaNodePool.spec.jvmOptions` and `-Xms: 512m` is set in `Kafka.spec.kafka.jvmOptions`, the node uses the value from its node pool configuration.

Properties from `Kafka` and `KafkaNodePool` schemas are not combined. To clarify, if `KafkaNodePool.spec.template` includes only `podSet.metadata.labels`, and `Kafka.spec.kafka.template` includes `podSet.metadata.annotations` and `pod.metadata.labels`, the template values from the Kafka configuration are ignored since there is a template value in the node pool configuration.

For a deeper understanding of the node pool configuration options, refer to the [Strimzi Custom Resource API Reference](#).

Example configuration for a node pool in a cluster using KRaft mode

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: kraft-dual-role ①
  labels:
    strimzi.io/cluster: my-cluster ②
spec:
  replicas: 3 ③
  roles: ④
    - controller
    - broker
  storage: ⑤
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
  resources: ⑥
    requests:
      memory: 64Gi
      cpu: "8"
    limits:
      memory: 64Gi
      cpu: "12"
```

① Unique name for the node pool.

② The Kafka cluster the node pool belongs to. A node pool can only belong to a single cluster.

③ Number of replicas for the nodes.

④ Roles for the nodes in the node pool. In this example, the nodes have dual roles as controllers

and brokers.

- ⑤ Storage specification for the nodes.
- ⑥ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.

NOTE

The configuration for the `Kafka` resource must be suitable for KRaft mode. Currently, KRaft mode has [a number of limitations](#).

Example configuration for a node pool in a cluster using ZooKeeper

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker ①
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
  resources:
    requests:
      memory: 64Gi
      cpu: "8"
    limits:
      memory: 64Gi
      cpu: "12"
```

① Roles for the nodes in the node pool, which can only be `broker` when using Kafka with ZooKeeper.

10.4.1. Assigning IDs to node pools for scaling operations

This procedure describes how to use annotations for advanced node ID handling by the Cluster Operator when performing scaling operations on node pools. You specify the node IDs to use, rather than the Cluster Operator using the next ID in sequence. Management of node IDs in this way gives greater control.

To add a range of IDs, you assign the following annotations to the `KafkaNodePool` resource:

- `strimzi.io/next-node-ids` to add a range of IDs that are used for new brokers
- `strimzi.io/remove-node-ids` to add a range of IDs for removing existing brokers

You can specify an array of individual node IDs, ID ranges, or a combination of both. For example, you can specify the following range of IDs: [0, 1, 2, 10-20, 30] for scaling up the Kafka node pool. This format allows you to specify a combination of individual node IDs (0, 1, 2, 30) as well as a range of IDs (10-20).

In a typical scenario, you might specify a range of IDs for scaling up and a single node ID to remove a specific node when scaling down.

In this procedure, we add the scaling annotations to node pools as follows:

- `pool-a` is assigned a range of IDs for scaling up
- `pool-b` is assigned a range of IDs for scaling down

During the scaling operation, IDs are used as follows:

- Scale up picks up the lowest available ID in the range for the new node.
- Scale down removes the node with the highest available ID in the range.

If there are gaps in the sequence of node IDs assigned in the node pool, the next node to be added is assigned an ID that fills the gap.

The annotations don't need to be updated after every scaling operation. Any unused IDs are still valid for the next scaling event.

The Cluster Operator allows you to specify a range of IDs in either ascending or descending order, so you can define them in the order the nodes are scaled. For example, when scaling up, you can specify a range such as [1000-1999], and the new nodes are assigned the next lowest IDs: 1000, 1001, 1002, 1003, and so on. Conversely, when scaling down, you can specify a range like [1999-1000], ensuring that nodes with the next highest IDs are removed: 1003, 1002, 1001, 1000, and so on.

If you don't specify an ID range using the annotations, the Cluster Operator follows its default behavior for handling IDs during scaling operations. Node IDs start at 0 (zero) and run sequentially across the Kafka cluster. The next lowest ID is assigned to a new node. Gaps to node IDs are filled across the cluster. This means that they might not run sequentially within a node pool. The default behavior for scaling up is to add the next lowest available node ID across the cluster; and for scaling down, it is to remove the node in the node pool with the highest available node ID. The default approach is also applied if the assigned range of IDs is misformatted, the scaling up range runs out of IDs, or the scaling down range does not apply to any in-use nodes.

Prerequisites

- [The Cluster Operator must be deployed.](#)
- (Optional) Use the `reserved.broker-max.id` configuration property to extend the allowable range for node IDs within your node pools.

By default, Apache Kafka restricts node IDs to numbers ranging from 0 to 999. To use node ID values greater than 999, add the `reserved.broker-max.id` configuration property to the [Kafka](#) custom resource and specify the required maximum node ID value.

In this example, the maximum node ID is set at 10000. Node IDs can then be assigned up to that

value.

Example configuration for the maximum node ID number

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    config:
      reserved.broker.max.id: 10000
    # ...
```

Procedure

1. Annotate the node pool with the IDs to use when scaling up or scaling down, as shown in the following examples.

IDs for scaling up are assigned to node pool **pool-a**:

Assigning IDs for scaling up

```
kubectl annotate kafkanodepool pool-a strimzi.io/next-node-ids="[0,1,2,10-20,30]"
```

The lowest available ID from this range is used when adding a node to **pool-a**.

IDs for scaling down are assigned to node pool **pool-b**:

Assigning IDs for scaling down

```
kubectl annotate kafkanodepool pool-b strimzi.io/remove-node-ids="[60-50,9,8,7]"
```

The highest available ID from this range is removed when scaling down **pool-b**.

NOTE

If you want to remove a specific node, you can assign a single node ID to the scaling down annotation: `kubectl annotate kafkanodepool pool-b strimzi.io/remove-node-ids="3"`.

2. You can now scale the node pool.

For more information, see the following:

- [Adding nodes to a node pool](#)
- [Removing nodes from a node pool](#)
- [Moving nodes between node pools](#)

On reconciliation, a warning is given if the annotations are misformatted.

3. After you have performed the scaling operation, you can remove the annotation if it's no longer needed.

Removing the annotation for scaling up

```
kubectl annotate kafkanodepool pool-a strimzi.io/next-node-ids-
```

Removing the annotation for scaling down

```
kubectl annotate kafkanodepool pool-b strimzi.io/remove-node-ids-
```

10.4.2. Impact on racks when moving nodes from node pools

If rack awareness is enabled on a Kafka cluster, replicas can be spread across different racks, data centers, or availability zones. When moving nodes from node pools, consider the implications on the cluster topology, particularly regarding rack awareness. Removing specific pods from node pools, especially out of order, may break the cluster topology or cause an imbalance in distribution across racks. An imbalance can impact both the distribution of nodes themselves and the partition replicas within the cluster. An uneven distribution of nodes and partitions across racks can affect the performance and resilience of the Kafka cluster.

Plan the removal of nodes strategically to maintain the required balance and resilience across racks. Use the `strimzi.io/remove-node-ids` annotation to move nodes with specific IDs with caution. Ensure that configuration to spread partition replicas across racks and for clients to consume from the closest replicas is not broken.

TIP Use Cruise Control and the `KafkaRebalance` resource with the `RackAwareGoal` to make sure that replicas remain distributed across different racks.

10.4.3. Adding nodes to a node pool

This procedure describes how to scale up a node pool to add new nodes. Currently, scale up is only possible for broker-only node pools containing nodes that run as dedicated brokers.

In this procedure, we start with three nodes for node pool `pool-a`:

Kafka nodes in the node pool

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0

Node IDs are appended to the name of the node on creation. We add node `my-cluster-pool-a-3`, which has a node ID of `3`.

NOTE During this process, the ID of the node that holds the partition replicas changes.

Consider any dependencies that reference the node ID.

Prerequisites

- The Cluster Operator must be deployed.
- Cruise Control is deployed with Kafka.
- (Optional) For scale up operations, [you can specify the node IDs to use in the operation](#).

If you have assigned a range of node IDs for the operation, the ID of the node being added is determined by the sequence of nodes given. If you have assigned a single node ID, a node is added with the specified ID. Otherwise, the lowest available node ID across the cluster is used.

Procedure

1. Create a new node in the node pool.

For example, node pool `pool-a` has three replicas. We add a node by increasing the number of replicas:

```
kubectl scale kafkanodepool pool-a --replicas=4
```

2. Check the status of the deployment and wait for the pods in the node pool to be created and ready (`1/1`).

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

Output shows four Kafka nodes in the node pool

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-a-3	1/1	Running	0

3. Reassign the partitions after increasing the number of nodes in the node pool.

After scaling up a node pool, use the Cruise Control `add-brokers` mode to move partition replicas from existing brokers to the newly added brokers.

Using Cruise Control to reassign partition replicas

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: add-brokers
  brokers: [3]
```

We are reassigning partitions to node `my-cluster-pool-a-3`. The reassignment can take some time depending on the number of topics and partitions in the cluster.

10.4.4. Removing nodes from a node pool

This procedure describes how to scale down a node pool to remove nodes. Currently, scale down is only possible for broker-only node pools containing nodes that run as dedicated brokers.

In this procedure, we start with four nodes for node pool `pool-a`:

Kafka nodes in the node pool

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-a-3	1/1	Running	0

Node IDs are appended to the name of the node on creation. We remove node `my-cluster-pool-a-3`, which has a node ID of `3`.

NOTE

During this process, the ID of the node that holds the partition replicas changes. Consider any dependencies that reference the node ID.

Prerequisites

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)
- (Optional) For scale down operations, [you can specify the node IDs to use in the operation.](#)

If you have assigned a range of node IDs for the operation, the ID of the node being removed is determined by the sequence of nodes given. If you have assigned a single node ID, the node with the specified ID is removed. Otherwise, the node with the highest available ID in the node pool is removed.

Procedure

1. Reassign the partitions before decreasing the number of nodes in the node pool.

Before scaling down a node pool, use the Cruise Control `remove-brokers` mode to move partition replicas off the brokers that are going to be removed.

Using Cruise Control to reassign partition replicas

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
```

brokers: [3]

We are reassigning partitions from node `my-cluster-pool-a-3`. The reassignment can take some time depending on the number of topics and partitions in the cluster.

- After the reassignment process is complete, and the node being removed has no live partitions, reduce the number of Kafka nodes in the node pool.

For example, node pool `pool-a` has four replicas. We remove a node by decreasing the number of replicas:

```
kubectl scale kafkanodepool pool-a --replicas=3
```

Output shows three Kafka nodes in the node pool

NAME	READY	STATUS	RESTARTS
my-cluster-pool-b-kafka-0	1/1	Running	0
my-cluster-pool-b-kafka-1	1/1	Running	0
my-cluster-pool-b-kafka-2	1/1	Running	0

10.4.5. Moving nodes between node pools

This procedure describes how to move nodes between source and target Kafka node pools without downtime. You create a new node on the target node pool and reassign partitions to move data from the old node on the source node pool. When the replicas on the new node are in-sync, you can delete the old node.

In this procedure, we start with two node pools:

- `pool-a` with three replicas is the target node pool
- `pool-b` with four replicas is the source node pool

We scale up `pool-a`, and reassign partitions and scale down `pool-b`, which results in the following:

- `pool-a` with four replicas
- `pool-b` with three replicas

NOTE

During this process, the ID of the node that holds the partition replicas changes. Consider any dependencies that reference the node ID.

Prerequisites

- The Cluster Operator must be deployed.
- Cruise Control is deployed with Kafka.
- (Optional) For scale up and scale down operations, you can specify the range of node IDs to use.

If you have assigned node IDs for the operation, the ID of the node being added or removed is

determined by the sequence of nodes given. Otherwise, the lowest available node ID across the cluster is used when adding nodes; and the node with the highest available ID in the node pool is removed.

Procedure

1. Create a new node in the target node pool.

For example, node pool `pool-a` has three replicas. We add a node by increasing the number of replicas:

```
kubectl scale kafkaNodePool pool-a --replicas=4
```

2. Check the status of the deployment and wait for the pods in the node pool to be created and ready (1/1).

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

Output shows four Kafka nodes in the source and target node pools

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-4	1/1	Running	0
my-cluster-pool-a-7	1/1	Running	0
my-cluster-pool-b-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0
my-cluster-pool-b-5	1/1	Running	0
my-cluster-pool-b-6	1/1	Running	0

Node IDs are appended to the name of the node on creation. We add node `my-cluster-pool-a-7`, which has a node ID of 7.

3. Reassign the partitions from the old node to the new node.

Before scaling down the source node pool, use the Cruise Control `remove-brokers` mode to move partition replicas off the brokers that are going to be removed.

Using Cruise Control to reassign partition replicas

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
  brokers: [6]
```

We are reassigning partitions from node `my-cluster-pool-b-6`. The reassignment can take some time depending on the number of topics and partitions in the cluster.

- After the reassignment process is complete, reduce the number of Kafka nodes in the source node pool.

For example, node pool `pool-b` has four replicas. We remove a node by decreasing the number of replicas:

```
kubectl scale kafkaNodePool pool-b --replicas=3
```

The node with the highest ID (6) within the pool is removed.

Output shows three Kafka nodes in the source node pool

NAME	READY	STATUS	RESTARTS
my-cluster-pool-b-kafka-2	1/1	Running	0
my-cluster-pool-b-kafka-3	1/1	Running	0
my-cluster-pool-b-kafka-5	1/1	Running	0

10.4.6. Changing node pool roles

Node pools can be used with Kafka clusters that operate in KRaft mode (using Kafka Raft metadata) or use ZooKeeper for metadata management. If you are using KRaft mode, you can specify roles for all nodes in the node pool to operate as brokers, controllers, or both. If you are using ZooKeeper, nodes must be set as brokers only.

In certain circumstances you might want to change the roles assigned to a node pool. For example, you may have a node pool that contains nodes that perform dual broker and controller roles, and then decide to split the roles between two node pools. In this case, you create a new node pool with nodes that act only as brokers, and then reassign partitions from the dual-role nodes to the new brokers. You can then switch the old node pool to a controller-only role.

You can also perform the reverse operation by moving from node pools with controller-only and broker-only roles to a node pool that contains nodes that perform dual broker and controller roles. In this case, you add the `broker` role to the existing controller-only node pool, reassign partitions from the broker-only nodes to the dual-role nodes, and then delete the broker-only node pool.

When removing `broker` roles in the node pool configuration, keep in mind that Kafka does not automatically reassign partitions. Before removing the broker role, ensure that nodes changing to controller-only roles do not have any assigned partitions. If partitions are assigned, the change is prevented. No replicas must be left on the node before removing the broker role. The best way to reassign partitions before changing roles is to apply a Cruise Control optimization proposal in `remove-brokers` mode. For more information, see [Generating optimization proposals](#).

10.4.7. Transitioning to separate broker and controller roles

This procedure describes how to transition to using node pools with separate roles. If your Kafka

cluster is using a node pool with combined controller and broker roles, you can transition to using two node pools with separate roles. To do this, rebalance the cluster to move partition replicas to a node pool with a broker-only role, and then switch the old node pool to a controller-only role.

In this procedure, we start with node pool `pool-a`, which has `controller` and `broker` roles:

Dual-role node pool

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 20Gi
        deleteClaim: false
  # ...
```

The node pool has three nodes:

Kafka nodes in the node pool

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0

Each node performs a combined role of broker and controller. We create a second node pool called `pool-b`, with three nodes that act as brokers only.

NOTE

During this process, the ID of the node that holds the partition replicas changes. Consider any dependencies that reference the node ID.

Prerequisites

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)

Procedure

1. Create a node pool with a `broker` role.

Example node pool configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-b
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
# ...
```

The new node pool also has three nodes. If you already have a broker-only node pool, you can skip this step.

2. Apply the new `KafkaNodePool` resource to create the brokers.
3. Check the status of the deployment and wait for the pods in the node pool to be created and ready (1/1).

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

Output shows pods running in two node pools

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0
my-cluster-pool-b-4	1/1	Running	0
my-cluster-pool-b-5	1/1	Running	0

Node IDs are appended to the name of the node on creation.

4. Use the Cruise Control `remove-brokers` mode to reassigned partition replicas from the dual-role nodes to the newly added brokers.

Using Cruise Control to reassigned partition replicas

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

```

kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
  brokers: [0, 1, 2]

```

The reassignment can take some time depending on the number of topics and partitions in the cluster.

NOTE If nodes changing to controller-only roles have any assigned partitions, the change is prevented. The `status.conditions` of the `Kafka` resource provide details of events preventing the change.

- Remove the `broker` role from the node pool that originally had a combined role.

Dual-role nodes switched to controllers

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 20Gi
        deleteClaim: false
# ...

```

- Apply the configuration change so that the node pool switches to a controller-only role.

10.4.8. Transitioning to dual-role nodes

This procedure describes how to transition from separate node pools with broker-only and controller-only roles to using a dual-role node pool. If your Kafka cluster is using node pools with dedicated controller and broker nodes, you can transition to using a single node pool with both roles. To do this, add the `broker` role to the controller-only node pool, rebalance the cluster to move partition replicas to the dual-role node pool, and then delete the old broker-only node pool.

In this procedure, we start with two node pools `pool-a`, which has only the `controller` role and `pool-b` which has only the `broker` role:

Single role node pools

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
# ...
---
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-b
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
# ...
```

The Kafka cluster has six nodes:

Kafka nodes in the node pools

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0
my-cluster-pool-b-4	1/1	Running	0

```
my-cluster-pool-b-5 1/1 Running 0
```

The **pool-a** nodes perform the role of controller. The **pool-b** nodes perform the role of broker.

NOTE During this process, the ID of the node that holds the partition replicas changes. Consider any dependencies that reference the node ID.

Prerequisites

- The Cluster Operator must be deployed.
- Cruise Control is deployed with Kafka.

Procedure

1. Edit the node pool **pool-a** and add the **broker** role to it.

Example node pool configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - controller
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
  # ...
```

2. Check the status and wait for the pods in the node pool to be restarted and ready (1/1).

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

Output shows pods running in two node pools

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0
my-cluster-pool-b-3	1/1	Running	0

```
my-cluster-pool-b-4 1/1    Running  0  
my-cluster-pool-b-5 1/1    Running  0
```

Node IDs are appended to the name of the node on creation.

3. Use the Cruise Control `remove-brokers` mode to reassign partition replicas from the broker-only nodes to the dual-role nodes.

Using Cruise Control to reassign partition replicas

```
apiVersion: kafka.strimzi.io/v1beta2  
kind: KafkaRebalance  
metadata:  
  # ...  
spec:  
  mode: remove-brokers  
  brokers: [3, 4, 5]
```

The reassignment can take some time depending on the number of topics and partitions in the cluster.

4. Remove the `pool-b` node pool that has the old broker-only nodes.

```
kubectl delete kafkanodepool pool-b -n <my_cluster_operator_namespace>
```

10.4.9. Managing storage using node pools

Storage management in Strimzi is usually straightforward, and requires little change when set up, but there might be situations where you need to modify your storage configurations. Node pools simplify this process, because you can set up separate node pools that specify your new storage requirements.

In this procedure we create and manage storage for a node pool called `pool-a` containing three nodes. We show how to change the storage class (`volumes.class`) that defines the type of persistent storage it uses. You can use the same steps to change the storage size (`volumes.size`).

NOTE

We strongly recommend using block storage. Strimzi is only tested for use with block storage.

Prerequisites

- [The Cluster Operator must be deployed.](#)
- [Cruise Control is deployed with Kafka.](#)
- For storage that uses persistent volume claims for dynamic volume allocation, storage classes are defined and available in the Kubernetes cluster that correspond to the storage solutions you need.

Procedure

1. Create the node pool with its own storage settings.

For example, node pool `pool-a` uses JBOD storage with persistent volumes:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-a
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 500Gi
        class: gp2-ebs
  # ...
```

Nodes in `pool-a` are configured to use Amazon EBS (Elastic Block Store) GP2 volumes.

2. Apply the node pool configuration for `pool-a`.
3. Check the status of the deployment and wait for the pods in `pool-a` to be created and ready (1/1).

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

Output shows three Kafka nodes in the node pool

NAME	READY	STATUS	RESTARTS
my-cluster-pool-a-0	1/1	Running	0
my-cluster-pool-a-1	1/1	Running	0
my-cluster-pool-a-2	1/1	Running	0

4. To migrate to a new storage class, create a new node pool with the required storage configuration:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-b
  labels:
    strimzi.io/cluster: my-cluster
spec:
  roles:
    - broker
```

```

replicas: 3
storage:
  type: jbod
  volumes:
    - id: 0
      type: persistent-claim
      size: 1Ti
      class: gp3-ebs
# ...

```

Nodes in `pool-b` are configured to use Amazon EBS (Elastic Block Store) GP3 volumes.

5. Apply the node pool configuration for `pool-b`.
6. Check the status of the deployment and wait for the pods in `pool-b` to be created and ready.
7. Reassign the partitions from `pool-a` to `pool-b`.

When migrating to a new storage configuration, use the Cruise Control `remove-brokers` mode to move partition replicas off the brokers that are going to be removed.

Using Cruise Control to reassign partition replicas

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  # ...
spec:
  mode: remove-brokers
  brokers: [0, 1, 2]

```

We are reassigning partitions from `pool-a`. The reassignment can take some time depending on the number of topics and partitions in the cluster.

8. After the reassignment process is complete, delete the old node pool:

```
kubectl delete kafkanodepool pool-a
```

10.4.10. Managing storage affinity using node pools

In situations where storage resources, such as local persistent volumes, are constrained to specific worker nodes, or availability zones, configuring storage affinity helps to schedule pods to use the right nodes.

Node pools allow you to configure affinity independently. In this procedure, we create and manage storage affinity for two availability zones: `zone-1` and `zone-2`.

You can configure node pools for separate availability zones, but use the same storage class. We define an `all-zones` persistent storage class representing the storage resources available in each zone.

We also use the `.spec.template.pod` properties to configure the node affinity and schedule Kafka pods on `zone-1` and `zone-2` worker nodes.

The storage class and affinity is specified in node pools representing the nodes in each availability zone:

- `pool-zone-1`
- `pool-zone-2`.

Prerequisites

- [The Cluster Operator must be deployed.](#)
- If you are not familiar with the concepts of affinity, see the [Kubernetes node and pod affinity documentation](#).

Procedure

1. Define the storage class for use with each availability zone:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: all-zones
provisioner: kubernetes.io/my-storage
parameters:
  type: ssd
volumeBindingMode: WaitForFirstConsumer
```

2. Create node pools representing the two availability zones, specifying the `all-zones` storage class and the affinity for each zone:

Node pool configuration for zone-1

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-zone-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 500Gi
        class: all-zones
  template:
    pod:
      affinity:
```

```

nodeAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
      - matchExpressions:
          - key: topology.kubernetes.io/zone
            operator: In
            values:
              - zone-1
# ...

```

Node pool configuration for zone-2

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-zone-2
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 4
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 500Gi
        class: all-zones
  template:
    pod:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: topology.kubernetes.io/zone
                    operator: In
                    values:
                      - zone-2
# ...

```

3. Apply the node pool configuration.
4. Check the status of the deployment and wait for the pods in the node pools to be created and ready (1/1).

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

Output shows 3 Kafka nodes in pool-zone-1 and 4 Kafka nodes in pool-zone-2

NAME	READY	STATUS	RESTARTS
my-cluster-pool-zone-1-kafka-0	1/1	Running	0
my-cluster-pool-zone-1-kafka-1	1/1	Running	0
my-cluster-pool-zone-1-kafka-2	1/1	Running	0
my-cluster-pool-zone-2-kafka-3	1/1	Running	0
my-cluster-pool-zone-2-kafka-4	1/1	Running	0
my-cluster-pool-zone-2-kafka-5	1/1	Running	0
my-cluster-pool-zone-2-kafka-6	1/1	Running	0

10.4.11. Migrating existing Kafka clusters to use Kafka node pools

This procedure describes how to migrate existing Kafka clusters to use Kafka node pools. After you have updated the Kafka cluster, you can use the node pools to manage the configuration of nodes within each pool.

NOTE Currently, replica and storage configuration in the `KafkaNodePool` resource must also be present in the `Kafka` resource. The configuration is ignored when node pools are being used.

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create a new `KafkaNodePool` resource.
 - a. Name the resource `kafka`.
 - b. Point a `strimzi.io/cluster` label to your existing `Kafka` resource.
 - c. Set the replica count and storage configuration to match your current Kafka cluster.
 - d. Set the roles to `broker`.

Example configuration for a node pool used in migrating a Kafka cluster

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: kafka
  labels:
    strimzi.io/cluster: my-cluster
spec:
  replicas: 3
  roles:
    - broker
  storage:
    type: jbod
    volumes:
      - id: 0
```

```
type: persistent-claim  
size: 100Gi  
deleteClaim: false
```

WARNING

To preserve cluster data and the names of its nodes and resources, the node pool name must be `kafka`, and the `strimzi.io/cluster` label matches the Kafka resource name. Otherwise, nodes and resources are created with new names, including the persistent volume storage used by the nodes. Consequently, your previous data may not be available.

2. Apply the `KafkaNodePool` resource:

```
kubectl apply -f <node_pool_configuration_file>
```

By applying this resource, you switch Kafka to using node pools.

There is no change or rolling update and resources are identical to how they were before.

3. Enable support for node pools in the `Kafka` resource using the `strimzi.io/node-pools: enabled` annotation.

Example configuration for a node pool in a cluster using ZooKeeper

```
apiVersion: kafka.strimzi.io/v1beta2  
kind: Kafka  
metadata:  
  name: my-cluster  
  annotations:  
    strimzi.io/node-pools: enabled  
spec:  
  kafka:  
    # ...  
  zookeeper:  
    # ...
```

4. Apply the `Kafka` resource:

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

There is no change or rolling update. The resources remain identical to how they were before.

5. Remove the replicated properties from the `Kafka` custom resource. When the `KafkaNodePool` resource is in use, you can remove the properties that you copied to the `KafkaNodePool` resource, such as the `.spec.kafka.replicas` and `.spec.kafka.storage` properties.

Reversing the migration

To revert to managing Kafka nodes using only `Kafka` custom resources:

1. If you have multiple node pools, consolidate them into a single `KafkaNodePool` named `kafka` with node IDs from 0 to N (where N is the number of replicas).
2. Ensure that the `.spec.kafka` configuration in the `Kafka` resource matches the `KafkaNodePool` configuration, including storage, resources, and replicas.
3. Disable support for node pools in the `Kafka` resource using the `strimzi.io/node-pools: disabled` annotation.
4. Delete the Kafka node pool named `kafka`.

10.5. Configuring the Entity Operator

Use the `entityOperator` property in `Kafka.spec` to configure the Entity Operator. The Entity Operator is responsible for managing Kafka-related entities in a running Kafka cluster. It comprises the following operators:

- Topic Operator to manage Kafka topics
- User Operator to manage Kafka users

By configuring the `Kafka` resource, the Cluster Operator can deploy the Entity Operator, including one or both operators. Once deployed, the operators are automatically configured to handle the topics and users of the Kafka cluster.

Each operator can only monitor a single namespace. For more information, see [Watching Strimzi resources in Kubernetes namespaces](#).

The `entityOperator` property supports several sub-properties:

- `topicOperator`
- `userOperator`
- `template`

The `template` property contains the configuration of the Entity Operator pod, such as labels, annotations, affinity, and tolerations. For more information on configuring templates, see [Customizing Kubernetes resources](#).

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

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

For more information on the properties used to configure the Entity Operator, see the [EntityOperatorSpec schema reference](#).

Example of basic configuration enabling both operators

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

```
name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

If an empty object ({}) is used for the `topicOperator` and `userOperator`, all properties use their default values.

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

10.5.1. Configuring the Topic Operator

Use `topicOperator` properties in `Kafka.spec.entityOperator` to configure the Topic Operator.

The following properties are supported:

`watchedNamespace`

The Kubernetes namespace in which the Topic Operator watches for `KafkaTopic` resources. Default is the namespace where the Kafka cluster is deployed.

`reconciliationIntervalMs`

The interval between periodic reconciliations in milliseconds. Default `120000`.

`image`

The `image` property can be used to configure the container image which is used. To learn more, refer to the information provided on [configuring the `image` property](#).

`resources`

The `resources` property configures the amount of resources allocated to the Topic Operator. You can specify requests and limits for `memory` and `cpu` resources. The requests should be enough to ensure a stable performance of the operator.

`logging`

The `logging` property configures the logging of the Topic Operator. To learn more, refer to the information provided on [Topic Operator logging](#).

Example Topic Operator configuration

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

```
kafka:  
  # ...  
zookeeper:  
  # ...  
entityOperator:  
  # ...  
topicOperator:  
  watchedNamespace: my-topic-namespace  
  reconciliationIntervalMs: 60000  
resources:  
  requests:  
    cpu: "1"  
    memory: 500Mi  
  limits:  
    cpu: "1"  
    memory: 500Mi  
# ...
```

10.5.2. Configuring the User Operator

Use `userOperator` properties in `Kafka.spec.entityOperator` to configure the User Operator. The following properties are supported:

watchedNamespace

The Kubernetes namespace in which the User Operator watches for `KafkaUser` resources. Default is the namespace where the Kafka cluster is deployed.

reconciliationIntervalMs

The interval between periodic reconciliations in milliseconds. Default `120000`.

image

The `image` property can be used to configure the container image which will be used. To learn more, refer to the information provided on [configuring the `image` property](#).

resources

The `resources` property configures the amount of resources allocated to the User Operator. You can specify requests and limits for `memory` and `cpu` resources. The requests should be enough to ensure a stable performance of the operator.

logging

The `logging` property configures the logging of the User Operator. To learn more, refer to the information provided on [User Operator logging](#).

secretPrefix

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

Example User Operator configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
  userOperator:
    watchedNamespace: my-user-namespace
    reconciliationIntervalMs: 60000
    resources:
      requests:
        cpu: "1"
        memory: 500Mi
      limits:
        cpu: "1"
        memory: 500Mi
    # ...
```

10.6. Configuring the Cluster Operator

Use environment variables to configure the Cluster Operator. Specify the environment variables for the container image of the Cluster Operator in its [Deployment](#) configuration file. You can use the following environment variables to configure the Cluster Operator. If you are running Cluster Operator replicas in standby mode, there are additional [environment variables for enabling leader election](#).

Kafka, Kafka Connect, and Kafka MirrorMaker support multiple versions. Use their `STRIMZI_<COMPONENT_NAME>_IMAGES` environment variables to configure the default container images used for each version. The configuration provides a mapping between a version and an image. The required syntax is whitespace or comma-separated `<version> = <image>` pairs, which determine the image to use for a given version. For example, `3.7.1=quay.io/strimzi/kafka:0.42.0-kafka-3.7.1`. These default images are overridden if `image` property values are specified in the configuration of a component. For more information on `image` configuration of components, see the [Strimzi Custom Resource API Reference](#).

NOTE

The [Deployment](#) configuration file provided with the Strimzi release artifacts is `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml`.

STRIMZI_NAMESPACE

A comma-separated list of namespaces that the operator operates in. When not set, set to empty string, or set to `*`, the Cluster Operator operates in all namespaces.

The Cluster Operator deployment might use the downward API to set this automatically to the namespace the Cluster Operator is deployed in.

Example configuration for Cluster Operator namespaces

```
env:  
  - name: STRIMZI_NAMESPACE  
    valueFrom:  
      fieldRef:  
        fieldPath: metadata.namespace
```

STRIMZI_FULL_RECONCILIATION_INTERVAL_MS

Optional, default is 120000 ms. The interval between [periodic reconciliations](#), in milliseconds.

STRIMZI_OPERATION_TIMEOUT_MS

Optional, default 300000 ms. The timeout for internal operations, in milliseconds. Increase this value when using Strimzi on clusters where regular Kubernetes operations take longer than usual (due to factors such as prolonged download times for container images, for example).

STRIMZI_ZOOKEEPER_ADMIN_SESSION_TIMEOUT_MS

Optional, default 10000 ms. The session timeout for the Cluster Operator's ZooKeeper admin client, in milliseconds. Increase the value if ZooKeeper requests from the Cluster Operator are regularly failing due to timeout issues. There is a maximum allowed session time set on the ZooKeeper server side via the `maxSessionTimeout` config. By default, the maximum session timeout value is 20 times the default `tickTime` (whose default is 2000) at 40000 ms. If you require a higher timeout, change the `maxSessionTimeout` ZooKeeper server configuration value.

STRIMZI_OPERATIONS_THREAD_POOL_SIZE

Optional, default 10. The worker thread pool size, which is used for various asynchronous and blocking operations that are run by the Cluster Operator.

STRIMZI_OPERATOR_NAME

Optional, defaults to the pod's hostname. The operator name identifies the Strimzi instance when [emitting Kubernetes events](#).

STRIMZI_OPERATOR_NAMESPACE

The name of the namespace where the Cluster Operator is running. Do not configure this variable manually. Use the downward API.

```
env:  
  - name: STRIMZI_OPERATOR_NAMESPACE  
    valueFrom:  
      fieldRef:  
        fieldPath: metadata.namespace
```

STRIMZI_OPERATOR_NAMESPACE_LABELS

Optional. The labels of the namespace where the Strimzi Cluster Operator is running. Use

namespace labels to configure the namespace selector in [network policies](#). Network policies allow the Strimzi Cluster Operator access only to the operands from the namespace with these labels. When not set, the namespace selector in network policies is configured to allow access to the Cluster Operator from any namespace in the Kubernetes cluster.

```
env:  
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS  
    value: label1=value1,label2=value2
```

[STRIMZI_LABELS_EXCLUSION_PATTERN](#)

Optional, default regex pattern is `^app.kubernetes.io/(?!part-of).*`. The regex exclusion pattern used to filter labels propagation from the main custom resource to its subresources. The labels exclusion filter is not applied to labels in template sections such as `spec.kafka.template.pod.metadata.labels`.

```
env:  
  - name: STRIMZI_LABELS_EXCLUSION_PATTERN  
    value: "^key1.*"
```

[STRIMZI_CUSTOM_<COMPONENT_NAME>_LABELS](#)

Optional. One or more custom labels to apply to all the pods created by the custom resource of the component. The Cluster Operator labels the pods when the custom resource is created or is next reconciled.

Labels can be applied to the following components:

- [KAFKA](#)
- [KAFKA_CONNECT](#)
- [KAFKA_CONNECT_BUILD](#)
- [ZOOKEEPER](#)
- [ENTITY_OPERATOR](#)
- [KAFKA_MIRROR MAKER2](#)
- [KAFKA_MIRROR MAKER](#)
- [CRUISE_CONTROL](#)
- [KAFKA_BRIDGE](#)
- [KAFKA_EXPORTER](#)

[STRIMZI_CUSTOM_RESOURCE_SELECTOR](#)

Optional. The label selector to filter the custom resources handled by the Cluster Operator. The operator will operate only on those custom resources that have the specified labels set. Resources without these labels will not be seen by the operator. The label selector applies to [Kafka](#), [KafkaConnect](#), [KafkaBridge](#), [KafkaMirrorMaker](#), and [KafkaMirrorMaker2](#) resources. [KafkaRebalance](#) and [KafkaConnector](#) resources are operated only when their corresponding Kafka

and Kafka Connect clusters have the matching labels.

```
env:  
  - name: STRIMZI_CUSTOM_RESOURCE_SELECTOR  
    value: label1=value1,label2=value2
```

STRIMZI_KAFKA_IMAGES

Required. The mapping from the Kafka version to the corresponding image containing a Kafka broker for that version. For example `3.6.1=quay.io/strimzi/kafka:0.42.0-kafka-3.6.1`, `3.7.1=quay.io/strimzi/kafka:0.42.0-kafka-3.7.1`.

STRIMZI_KAFKA_CONNECT_IMAGES

Required. The mapping from the Kafka version to the corresponding image of Kafka Connect for that version. For example `3.6.1=quay.io/strimzi/kafka:0.42.0-kafka-3.6.1`, `3.7.1=quay.io/strimzi/kafka:0.42.0-kafka-3.7.1`.

STRIMZI_KAFKA_MIRROR MAKER2 IMAGES

Required. The mapping from the Kafka version to the corresponding image of MirrorMaker 2 for that version. For example `3.6.1=quay.io/strimzi/kafka:0.42.0-kafka-3.6.1`, `3.7.1=quay.io/strimzi/kafka:0.42.0-kafka-3.7.1`.

(Deprecated) STRIMZI_KAFKA_MIRROR MAKER_IMAGES

Required. The mapping from the Kafka version to the corresponding image of MirrorMaker for that version. For example `3.6.1=quay.io/strimzi/kafka:0.42.0-kafka-3.6.1`, `3.7.1=quay.io/strimzi/kafka:0.42.0-kafka-3.7.1`.

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

Optional. The default is `quay.io/strimzi/operator:0.42.0`. The image name to use as the default when deploying the Topic Operator if no image is specified as the `Kafka.spec.entityOperator.topicOperator.image` in the `Kafka` resource.

STRIMZI_DEFAULT_USER_OPERATOR_IMAGE

Optional. The default is `quay.io/strimzi/operator:0.42.0`. The image name to use as the default when deploying the User Operator if no image is specified as the `Kafka.spec.entityOperator.userOperator.image` in the `Kafka` resource.

STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE

Optional. The default is `quay.io/strimzi/kafka:0.42.0-kafka-3.7.1`. The image name to use as the default when deploying the Kafka Exporter if no image is specified as the `Kafka.spec.kafkaExporter.image` in the `Kafka` resource.

STRIMZI_DEFAULT_CRUISE_CONTROL_IMAGE

Optional. The default is `quay.io/strimzi/kafka:0.42.0-kafka-3.7.1`. The image name to use as the default when deploying Cruise Control if no image is specified as the `Kafka.spec.cruiseControl.image` in the `Kafka` resource.

STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE

Optional. The default is `quay.io/strimzi/kafka-bridge:0.29.0`. The image name to use as the default when deploying the Kafka Bridge if no image is specified as the `Kafka.spec.kafkaBridge.image` in the `Kafka` resource.

STRIMZI_DEFAULT_KAFKA_INIT_IMAGE

Optional. The default is `quay.io/strimzi/operator:0.42.0`. The image name to use as the default for the Kafka initializer container if no image is specified in the `brokerRackInitImage` of the `Kafka` resource or the `clientRackInitImage` of the Kafka Connect resource. The init container is started before the Kafka cluster for initial configuration work, such as rack support.

STRIMZI_IMAGE_PULL_POLICY

Optional. The `ImagePullPolicy` that is applied to containers in all pods managed by the Cluster Operator. The valid values are `Always`, `IfNotPresent`, and `Never`. If not specified, the Kubernetes defaults are used. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

STRIMZI_IMAGE_PULL_SECRETS

Optional. A comma-separated list of `Secret` names. The secrets referenced here contain the credentials to the container registries where the container images are pulled from. The secrets are specified in the `imagePullSecrets` property for all pods created by the Cluster Operator. Changing this list results in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

STRIMZI_KUBERNETES_VERSION

Optional. Overrides the Kubernetes version information detected from the API server.

Example configuration for Kubernetes version override

```
env:  
  - name: STRIMZI_KUBERNETES_VERSION  
    value: |  
      major=1  
      minor=16  
      gitVersion=v1.16.2  
      gitCommit=c97fe5036ef3df2967d086711e6c0c405941e14b  
      gitTreeState=clean  
      buildDate=2019-10-15T19:09:08Z  
      goVersion=go1.12.10  
      compiler=gc  
      platform=linux/amd64
```

KUBERNETES_SERVICE_DNS_DOMAIN

Optional. Overrides the default Kubernetes DNS domain name suffix.

By default, services assigned in the Kubernetes cluster have a DNS domain name that uses the default suffix `cluster.local`.

For example, for broker `kafka-0`:

```
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers.<namespace>.svc.cluster.local
```

The DNS domain name is added to the Kafka broker certificates used for hostname verification.

If you are using a different DNS domain name suffix in your cluster, change the `KUBERNETES_SERVICE_DNS_DOMAIN` environment variable from the default to the one you are using in order to establish a connection with the Kafka brokers.

STRIMZI_CONNECT_BUILD_TIMEOUT_MS

Optional, default 300000 ms. The timeout for building new Kafka Connect images with additional connectors, in milliseconds. Consider increasing this value when using Strimzi to build container images containing many connectors or using a slow container registry.

STRIMZI_NETWORK_POLICY_GENERATION

Optional, default `true`. Network policy for resources. Network policies allow connections between Kafka components.

Set this environment variable to `false` to disable network policy generation. You might do this, for example, if you want to use custom network policies. Custom network policies allow more control over maintaining the connections between components.

STRIMZI_DNS_CACHE_TTL

Optional, default `30`. Number of seconds to cache successful name lookups in local DNS resolver. Any negative value means cache forever. Zero means do not cache, which can be useful for avoiding connection errors due to long caching policies being applied.

STRIMZI_POD_SET_RECONCILIATION_ONLY

Optional, default `false`. When set to `true`, the Cluster Operator reconciles only the `StrimziPodSet` resources and any changes to the other custom resources (`Kafka`, `KafkaConnect`, and so on) are ignored. This mode is useful for ensuring that your pods are recreated if needed, but no other changes happen to the clusters.

STRIMZI_FEATURE_GATES

Optional. Enables or disables the features and functionality controlled by [feature gates](#).

STRIMZI_POD_SECURITY_PROVIDER_CLASS

Optional. Configuration for the pluggable `PodSecurityProvider` class, which can be used to provide the security context configuration for Pods and containers.

10.6.1. Restricting access to the Cluster Operator using network policy

Use the `STRIMZI_OPERATOR_NAMESPACE_LABELS` environment variable to establish network policy for the Cluster Operator using namespace labels.

The Cluster Operator can run in the same namespace as the resources it manages, or in a separate namespace. By default, the `STRIMZI_OPERATOR_NAMESPACE` environment variable is configured to use the downward API to find the namespace the Cluster Operator is running in. If the Cluster Operator is running in the same namespace as the resources, only local access is required and allowed by

Strimzi.

If the Cluster Operator is running in a separate namespace to the resources it manages, any namespace in the Kubernetes cluster is allowed access to the Cluster Operator unless network policy is configured. By adding namespace labels, access to the Cluster Operator is restricted to the namespaces specified.

Network policy configured for the Cluster Operator deployment

```
#...
env:
# ...
- name: STRIMZI_OPERATOR_NAMESPACE_LABELS
  value: label1=value1,label2=value2
#...
```

10.6.2. Setting periodic reconciliation of custom resources

Use the `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` variable to set the time interval for periodic reconciliations by the Cluster Operator. Replace its value with the required interval in milliseconds.

Reconciliation period configured for the Cluster Operator deployment

```
#...
env:
# ...
- name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS
  value: "120000"
#...
```

The Cluster Operator reacts to all notifications about applicable cluster resources received from the Kubernetes cluster. If the operator is not running, or if a notification is not received for any reason, resources will get out of sync with the state of the running 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 resources with the current cluster deployments in order to have a consistent state across all of them.

Additional resources

- [Downward API](#)

10.6.3. Pausing reconciliation of custom resources using annotations

Sometimes it is useful to pause the reconciliation of custom resources managed by Strimzi operators, so that you can perform fixes or make updates. If reconciliations are paused, any changes made to custom resources are ignored by the operators until the pause ends.

If you want to pause reconciliation of a custom resource, set the `strimzi.io/pause-reconciliation` annotation to `true` in its configuration. This instructs the appropriate operator to pause reconciliation of the custom resource. For example, you can apply the annotation to the

[KafkaConnect](#) resource so that reconciliation by the Cluster Operator is paused.

You can also create a custom resource with the pause annotation enabled. The custom resource is created, but it is ignored.

Prerequisites

- The Strimzi Operator that manages the custom resource is running.

Procedure

1. Annotate the custom resource in Kubernetes, setting [pause-reconciliation](#) to `true`:

```
kubectl annotate <kind_of_custom_resource> <name_of_custom_resource>
strimzi.io/pause-reconciliation="true"
```

For example, for the [KafkaConnect](#) custom resource:

```
kubectl annotate KafkaConnect my-connect strimzi.io/pause-reconciliation="true"
```

2. Check that the status conditions of the custom resource show a change to [ReconciliationPaused](#):

```
kubectl describe <kind_of_custom_resource> <name_of_custom_resource>
```

The `type` condition changes to [ReconciliationPaused](#) at the `lastTransitionTime`.

Example custom resource with a paused reconciliation condition type

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  annotations:
    strimzi.io/pause-reconciliation: "true"
    strimzi.io/use-connector-resources: "true"
  creationTimestamp: 2021-03-12T10:47:11Z
  #...
spec:
  # ...
status:
  conditions:
  - lastTransitionTime: 2021-03-12T10:47:41.689249Z
    status: "True"
    type: ReconciliationPaused
```

Resuming from pause

- To resume reconciliation, you can set the annotation to `false`, or remove the annotation.

Additional resources

- [Finding the status of a custom resource](#)

10.6.4. Running multiple Cluster Operator replicas with leader election

The default Cluster Operator configuration enables leader election to run multiple parallel replicas of the Cluster Operator. One replica is elected as the active leader and operates the deployed resources. The other replicas run in standby mode. When the leader stops or fails, one of the standby replicas is elected as the new leader and starts operating the deployed resources.

By default, Strimzi runs with a single Cluster Operator replica that is always the leader replica. When a single Cluster Operator replica stops or fails, Kubernetes starts a new replica.

Running the Cluster Operator with multiple replicas is not essential. But it's useful to have replicas on standby in case of large-scale disruptions caused by major failure. For example, suppose multiple worker nodes or an entire availability zone fails. This failure might cause the Cluster Operator pod and many Kafka pods to go down at the same time. If subsequent pod scheduling causes congestion through lack of resources, this can delay operations when running a single Cluster Operator.

Enabling leader election for Cluster Operator replicas

Configure leader election environment variables when running additional Cluster Operator replicas. The following environment variables are supported:

STRIMZI_LEADER_ELECTION_ENABLED

Optional, disabled (`false`) by default. Enables or disables leader election, which allows additional Cluster Operator replicas to run on standby.

NOTE

Leader election is disabled by default. It is only enabled when applying this environment variable on installation.

STRIMZI_LEADER_ELECTIONLEASE_NAME

Required when leader election is enabled. The name of the Kubernetes `Lease` resource that is used for the leader election.

STRIMZI_LEADER_ELECTIONLEASE_NAMESPACE

Required when leader election is enabled. The namespace where the Kubernetes `Lease` resource used for leader election is created. You can use the downward API to configure it to the namespace where the Cluster Operator is deployed.

```
env:
  - name: STRIMZI_LEADER_ELECTIONLEASE_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_LEADER_ELECTION_IDENTITY

Required when leader election is enabled. Configures the identity of a given Cluster Operator

instance used during the leader election. The identity must be unique for each operator instance. You can use the downward API to configure it to the name of the pod where the Cluster Operator is deployed.

```
env:  
- name: STRIMZI_LEADER_ELECTION_IDENTITY  
  valueFrom:  
    fieldRef:  
      fieldPath: metadata.name
```

STRIMZI_LEADER_ELECTIONLEASE_DURATION_MS

Optional, default 15000 ms. Specifies the duration the acquired lease is valid.

STRIMZI_LEADER_ELECTION_RENEW_DEADLINE_MS

Optional, default 10000 ms. Specifies the period the leader should try to maintain leadership.

STRIMZI_LEADER_ELECTION_RETRY_PERIOD_MS

Optional, default 2000 ms. Specifies the frequency of updates to the lease lock by the leader.

Configuring Cluster Operator replicas

To run additional Cluster Operator replicas in standby mode, you will need to increase the number of replicas and enable leader election. To configure leader election, use the leader election environment variables.

To make the required changes, configure the following Cluster Operator installation files located in `install/cluster-operator/`:

- 060-Deployment-strimzi-cluster-operator.yaml
- 022-ClusterRole-strimzi-cluster-operator-role.yaml
- 022-RoleBinding-strimzi-cluster-operator.yaml

Leader election has its own `ClusterRole` and `RoleBinding` RBAC resources that target the namespace where the Cluster Operator is running, rather than the namespace it is watching.

The default deployment configuration creates a `Lease` resource called `strimzi-cluster-operator` in the same namespace as the Cluster Operator. The Cluster Operator uses leases to manage leader election. The RBAC resources provide the permissions to use the `Lease` resource. If you use a different `Lease` name or namespace, update the `ClusterRole` and `RoleBinding` files accordingly.

Prerequisites

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

Procedure

Edit the `Deployment` resource that is used to deploy the Cluster Operator, which is defined in the `060-Deployment-strimzi-cluster-operator.yaml` file.

1. Change the `replicas` property from the default (1) to a value that matches the required number of replicas.

Increasing the number of Cluster Operator replicas

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 3
```

2. Check that the leader election `env` properties are set.

If they are not set, configure them.

To enable leader election, `STRIMZI_LEADER_ELECTION_ENABLED` must be set to `true` (default).

In this example, the name of the lease is changed to `my-strimzi-cluster-operator`.

Configuring leader election environment variables for the Cluster Operator

```
# ...
spec
containers:
- name: strimzi-cluster-operator
# ...
env:
- name: STRIMZI_LEADER_ELECTION_ENABLED
  value: "true"
- name: STRIMZI_LEADER_ELECTIONLEASE_NAME
  value: "my-strimzi-cluster-operator"
- name: STRIMZI_LEADER_ELECTIONLEASE_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
- name: STRIMZI_LEADER_ELECTION_IDENTITY
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
```

For a description of the available environment variables, see [Enabling leader election for Cluster Operator replicas](#).

If you specified a different name or namespace for the `Lease` resource used in leader election, update the RBAC resources.

3. (optional) Edit the `ClusterRole` resource in the `022-ClusterRole-strimzi-cluster-operator-`

`role.yaml` file.

Update `resourceNames` with the name of the `Lease` resource.

Updating the ClusterRole references to the lease

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
rules:
  - apiGroups:
    - coordination.k8s.io
    resourceNames:
      - my-strimzi-cluster-operator
# ...
```

4. (optional) Edit the `RoleBinding` resource in the `022-RoleBinding-strimzi-cluster-operator.yaml` file.

Update `subjects.name` and `subjects.namespace` with the name of the `Lease` resource and the namespace where it was created.

Updating the RoleBinding references to the lease

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: my-strimzi-cluster-operator
    namespace: myproject
# ...
```

5. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n myproject
```

6. Check the status of the deployment:

```
kubectl get deployments -n myproject
```

Output shows the deployment name and readiness

NAME	READY	UP-TO-DATE	AVAILABLE
strimzi-cluster-operator	3/3	3	3

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

10.6.5. Configuring Cluster Operator HTTP proxy settings

If you are running a Kafka cluster behind a HTTP proxy, you can still pass data in and out of the cluster. For example, you can run Kafka Connect with connectors that push and pull data from outside the proxy. Or you can use a proxy to connect with an authorization server.

Configure the Cluster Operator deployment to specify the proxy environment variables. The Cluster Operator accepts standard proxy configuration (`HTTP_PROXY`, `HTTPS_PROXY` and `NO_PROXY`) as environment variables. The proxy settings are applied to all Strimzi containers.

The format for a proxy address is `http://<ip_address>:<port_number>`. To set up a proxy with a name and password, the format is `http://<username>:<password>@<ip-address>:<port_number>`.

Prerequisites

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

Procedure

1. To add proxy environment variables to the Cluster Operator, update its `Deployment` configuration (`install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml`).

Example proxy configuration for the Cluster Operator

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
          # ...
          - name: "HTTP_PROXY"
            value: "http://proxy.com" ①
          - name: "HTTPS_PROXY"
            value: "https://proxy.com" ②
          - name: "NO_PROXY"
            value: "internal.com, other.domain.com" ③
        # ...
```

- ① Address of the proxy server.
- ② Secure address of the proxy server.
- ③ Addresses for servers that are accessed directly as exceptions to the proxy server. The URLs are comma-separated.

Alternatively, edit the [Deployment](#) directly:

```
kubectl edit deployment strimzi-cluster-operator
```

2. If you updated the YAML file instead of editing the [Deployment](#) directly, apply the changes:

```
kubectl create -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
```

Additional resources

- [Host aliases](#)
- [Designating Strimzi administrators](#)

10.6.6. Disabling FIPS mode using Cluster Operator configuration

Strimzi automatically switches to FIPS mode when running on a FIPS-enabled Kubernetes cluster. Disable FIPS mode by setting the `FIPS_MODE` environment variable to `disabled` in the deployment configuration for the Cluster Operator. With FIPS mode disabled, Strimzi automatically disables FIPS in the OpenJDK for all components. With FIPS mode disabled, Strimzi is not FIPS compliant. The Strimzi operators, as well as all operands, run in the same way as if they were running on an Kubernetes cluster without FIPS enabled.

Procedure

1. To disable the FIPS mode in the Cluster Operator, update its [Deployment](#) configuration (`install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml`) and add the `FIPS_MODE` environment variable.

Example FIPS configuration for the Cluster Operator

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
          # ...
          - name: "FIPS_MODE"
```

```
    value: "disabled" ①  
  # ...
```

① Disables the FIPS mode.

Alternatively, edit the [Deployment](#) directly:

```
kubectl edit deployment strimzi-cluster-operator
```

2. If you updated the YAML file instead of editing the [Deployment](#) directly, apply the changes:

```
kubectl apply -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
```

10.7. Configuring Kafka Connect

Update the [spec](#) properties of the [KafkaConnect](#) custom resource to configure your Kafka Connect deployment.

Use Kafka Connect to set up external data connections to your Kafka cluster. Use the properties of the [KafkaConnect](#) resource to configure your Kafka Connect deployment.

For a deeper understanding of the Kafka Connect cluster configuration options, refer to the [Strimzi Custom Resource API Reference](#).

KafkaConnector configuration

[KafkaConnector](#) resources allow you to create and manage connector instances for Kafka Connect in a Kubernetes-native way.

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

To manage connectors, you can use [use KafkaConnector](#) custom resources or the Kafka Connect REST API. [KafkaConnector](#) resources must be deployed to the same namespace as the Kafka Connect cluster they link to. For more information on using these methods to create, reconfigure, or delete connectors, see [Adding connectors](#).

Connector configuration is passed to Kafka Connect as part of an HTTP request and stored within Kafka itself. ConfigMaps and Secrets are standard Kubernetes resources used for storing configurations and confidential data. You can use ConfigMaps and Secrets to configure certain elements of a connector. You can then reference the configuration values in HTTP REST commands, which keeps the configuration separate and more secure, if needed. This method applies especially to confidential data, such as usernames, passwords, or certificates.

Handling high volumes of messages

You can tune the configuration to handle high volumes of messages. For more information, see [Handling high volumes of messages](#).

Example KafkaConnect custom resource configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect ①
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" ②
spec:
  replicas: 3 ③
  authentication: ④
    type: tls
    certificateAndKey:
      certificate: source.crt
      key: source.key
      secretName: my-user-source
  bootstrapServers: my-cluster-kafka-bootstrap:9092 ⑤
  tls: ⑥
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        pattern: "*.crt"
      - secretName: my-cluster-cluster-cert
        pattern: "*.crt"
  config: ⑦
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  build: ⑧
    output: ⑨
      type: docker
      image: my-registry.io/my-org/my-connect-cluster:latest
      pushSecret: my-registry-credentials
  plugins: ⑩
    - name: connector-1
      artifacts:
        - type: tgz
          url: <url_to_download_connector_1_artifact>
          sha512sum: <SHA-512_checksum_of_connector_1_artifact>
    - name: connector-2
```

```

artifacts:
  - type: jar
    url: <url_to_download_connector_2_artifact>
    sha512sum: <SHA-512_checksum_of_connector_2_artifact>
externalConfiguration: ⑪
env:
  - name: AWS_ACCESS_KEY_ID
    valueFrom:
      secretKeyRef:
        name: aws-creds
        key: awsAccessKey
  - name: AWS_SECRET_ACCESS_KEY
    valueFrom:
      secretKeyRef:
        name: aws-creds
        key: awsSecretAccessKey
resources: ⑫
requests:
  cpu: "1"
  memory: 2Gi
limits:
  cpu: "2"
  memory: 2Gi
logging: ⑬
type: inline
loggers:
  log4j.rootLogger: INFO
readinessProbe: ⑭
initialDelaySeconds: 15
timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
metricsConfig: ⑮
type: jmxPrometheusExporter
valueFrom:
  configMapKeyRef:
    name: my-config-map
    key: my-key
jvmOptions: ⑯
  "-Xmx": "1g"
  "-Xms": "1g"
image: my-org/my-image:latest ⑰
rack:
  topologyKey: topology.kubernetes.io/zone ⑱
template: ⑲
pod:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:

```

```

matchExpressions:
  - key: application
    operator: In
    values:
      - postgresql
      - mongodb
  topologyKey: "kubernetes.io/hostname"
connectContainer: ⑯
  env:
    - name: OTEL_SERVICE_NAME
      value: my-otel-service
    - name: OTEL_EXPORTER_OTLP_ENDPOINT
      value: "http://otlp-host:4317"
tracing:
  type: opentelemetry

```

- ① Use [KafkaConnect](#).
- ② Enables KafkaConnectors for the Kafka Connect cluster.
- ③ The number of replica nodes for the workers that run tasks.
- ④ Authentication for the Kafka Connect cluster, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN. By default, Kafka Connect connects to Kafka brokers using a plain text connection.
- ⑤ Bootstrap server for connection to the Kafka cluster.
- ⑥ TLS configuration for encrypted connections to the Kafka cluster, with trusted certificates stored in X.509 format within the specified secrets.
- ⑦ Kafka Connect configuration of workers (not connectors). Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by Strimzi.
- ⑧ Build configuration properties for building a container image with connector plugins automatically.
- ⑨ (Required) Configuration of the container registry where new images are pushed.
- ⑩ (Required) List of connector plugins and their artifacts to add to the new container image. Each plugin must be configured with at least one [artifact](#).
- ⑪ External configuration for connectors using environment variables, as shown here, or volumes. You can also use configuration provider plugins to load configuration values from external sources.
- ⑫ Requests for reservation of supported resources, currently [cpu](#) and [memory](#), and limits to specify the maximum resources that can be consumed.
- ⑬ Specified Kafka Connect loggers and log levels added directly ([inline](#)) or indirectly ([external](#)) through a ConfigMap. A custom Log4j configuration must be placed under the [log4j.properties](#) or [log4j2.properties](#) key in the ConfigMap. For the Kafka Connect [log4j.rootLogger](#) logger, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
- ⑭ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).

- ⑯ Prometheus metrics, which are enabled by referencing a ConfigMap containing configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using a reference to a ConfigMap containing an empty file under `metricsConfig.valueFrom.configMapKeyRef.key`.
- ⑰ JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka Connect.
- ⑱ ADVANCED OPTION: Container image configuration, which is recommended only in special situations.
- ⑲ SPECIALIZED OPTION: Rack awareness configuration for the deployment. This is a specialized option intended for a deployment within the same location, not across regions. Use this option if you want connectors to consume from the closest replica rather than the leader replica. In certain cases, consuming from the closest replica can improve network utilization or reduce costs. The `topologyKey` must match a node label containing the rack ID. The example used in this configuration specifies a zone using the standard `topology.kubernetes.io/zone` label. To consume from the closest replica, enable the `RackAwareReplicaSelector` in the Kafka broker configuration.
- ⑳ Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.
- ㉑ Environment variables are set for distributed tracing.

Distributed tracing is enabled by using OpenTelemetry.

10.7.1. Configuring Kafka Connect for multiple instances

By default, Strimzi configures the group ID and names of the internal topics used by Kafka Connect. When running multiple instances of Kafka Connect, you must change these default settings using the following `config` properties:

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

① The Kafka Connect cluster group ID within Kafka.

② Kafka topic that stores connector offsets.

③ Kafka topic that stores connector and task status configurations.

④ Kafka topic that stores connector and task status updates.

NOTE Values for the three topics must be the same for all instances with the same

`group.id`.

Unless you modify these default settings, each instance connecting to the same Kafka cluster is deployed with the same values. In practice, this means all instances form a cluster and use the same internal topics.

Multiple instances attempting to use the same internal topics will cause unexpected errors, so you must change the values of these properties for each instance.

10.7.2. Configuring Kafka Connect user authorization

When using authorization in Kafka, a Kafka Connect user requires read/write access to the cluster group and internal topics of Kafka Connect. This procedure outlines how access is granted using [simple authorization](#) and ACLs.

Properties for the Kafka Connect cluster group ID and internal topics are configured by Strimzi by default. Alternatively, you can define them explicitly in the [spec](#) of the [KafkaConnect](#) resource. This is useful when [configuring Kafka Connect for multiple instances](#), as the values for the group ID and topics must differ when running multiple Kafka Connect instances.

Simple authorization uses ACL rules managed by the Kafka [AclAuthorizer](#) and [StandardAuthorizer](#) plugins to ensure appropriate access levels. For more information on configuring a [KafkaUser](#) resource to use simple authorization, see the [AclRule schema reference](#).

Prerequisites

- A Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the [authorization](#) property in the [KafkaUser](#) resource to provide access rights to the user.

Access rights are configured for the Kafka Connect topics and cluster group using [literal](#) name values. The following table shows the default names configured for the topics and cluster group ID.

Table 12. Names for the access rights configuration

Property	Name
<code>offset.storage.topic</code>	<code>connect-cluster-offsets</code>
<code>status.storage.topic</code>	<code>connect-cluster-status</code>
<code>config.storage.topic</code>	<code>connect-cluster-configs</code>
<code>group</code>	<code>connect-cluster</code>

In this example configuration, the default names are used to specify access rights. If you are using different names for a Kafka Connect instance, use those names in the ACLs configuration.

Example configuration for simple authorization

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

```
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      # access to offset.storage.topic
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
          operations:
            - Create
            - Describe
            - Read
            - Write
          host: "*"
      # access to status.storage.topic
      - resource:
          type: topic
          name: connect-cluster-status
          patternType: literal
          operations:
            - Create
            - Describe
            - Read
            - Write
          host: "*"
      # access to config.storage.topic
      - resource:
          type: topic
          name: connect-cluster-configs
          patternType: literal
          operations:
            - Create
            - Describe
            - Read
            - Write
          host: "*"
      # cluster group
      - resource:
          type: group
          name: connect-cluster
          patternType: literal
          operations:
            - Read
```

```
host: "*"
```

2. Create or update the resource.

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

10.7.3. Manually stopping or pausing Kafka Connect connectors

If you are using [KafkaConnector](#) resources to configure connectors, use the `state` configuration to either stop or pause a connector. In contrast to the paused state, where the connector and tasks remain instantiated, stopping a connector retains only the configuration, with no active processes. Stopping a connector from running may be more suitable for longer durations than just pausing. While a paused connector is quicker to resume, a stopped connector has the advantages of freeing up memory and resources.

NOTE

The `state` configuration replaces the (deprecated) `pause` configuration in the [KafkaConnectorSpec](#) schema, which allows pauses on connectors. If you were previously using the `pause` configuration to pause connectors, we encourage you to transition to using the `state` configuration only to avoid conflicts.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Find the name of the [KafkaConnector](#) custom resource that controls the connector you want to pause or stop:

```
kubectl get KafkaConnector
```

2. Edit the [KafkaConnector](#) resource to stop or pause the connector.

Example configuration for stopping a Kafka Connect connector

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

```
# ...
```

Change the `state` configuration to `stopped` or `paused`. The default state for the connector when this property is not set is `running`.

3. Apply the changes to the `KafkaConnector` configuration.

You can resume the connector by changing `state` to `running` or removing the configuration.

NOTE

Alternatively, you can [expose the Kafka Connect API](#) and use the `stop` and `pause` endpoints to stop a connector from running. For example, `PUT /connectors/<connector_name>/stop`. You can then use the `resume` endpoint to restart it.

10.7.4. Manually restarting Kafka Connect connectors

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

Prerequisites

- The Cluster Operator is running.

Procedure

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

```
kubectl get KafkaConnector
```

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

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

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

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

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

10.7.5. Manually restarting Kafka Connect connector tasks

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

Prerequisites

- The Cluster Operator is running.

Procedure

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

```
kubectl get KafkaConnector
```

- Find the ID of the task to be restarted from the `KafkaConnector` custom resource:

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

Task IDs are non-negative integers, starting from 0.

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

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

In this example, task `0` is restarted.

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

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

10.8. Configuring Kafka MirrorMaker 2

Update the `spec` properties of the `KafkaMirrorMaker2` custom resource to configure your MirrorMaker 2 deployment. MirrorMaker 2 uses source cluster configuration for data consumption and target cluster configuration for data output.

MirrorMaker 2 is based on the Kafka Connect framework, *connectors* managing the transfer of data between clusters.

You configure MirrorMaker 2 to define the Kafka Connect deployment, including the connection details of the source and target clusters, and then run a set of MirrorMaker 2 connectors to make the connection.

MirrorMaker 2 supports topic configuration synchronization between the source and target clusters. You specify source topics in the MirrorMaker 2 configuration. MirrorMaker 2 monitors the source topics. MirrorMaker 2 detects and propagates changes to the source topics to the remote topics. Changes might include automatically creating missing topics and partitions.

NOTE

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

The configuration must specify:

- Each Kafka cluster
- Connection information for each cluster, including authentication
- The replication flow and direction
 - Cluster to cluster
 - Topic to topic

For a deeper understanding of the Kafka MirrorMaker 2 cluster configuration options, refer to the [Strimzi Custom Resource API Reference](#).

NOTE MirrorMaker 2 resource configuration differs from the previous version of MirrorMaker, which is now deprecated. There is currently no legacy support, so any resources must be manually converted into the new format.

Default configuration

MirrorMaker 2 provides default configuration values for properties such as replication factors. A minimal configuration, with defaults left unchanged, would be something like this example:

Minimal configuration for MirrorMaker 2

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.7.1
  connectCluster: "my-cluster-target"
  clusters:
    - alias: "my-cluster-source"
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092
    - alias: "my-cluster-target"
      bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
    - sourceCluster: "my-cluster-source"
      targetCluster: "my-cluster-target"
      sourceConnector: {}
```

You can configure access control for source and target clusters using mTLS or SASL authentication. This procedure shows a configuration that uses TLS encryption and mTLS authentication for the source and target cluster.

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

specify only the topics and consumer groups you need to avoid causing any unnecessary extra load on the cluster.

Handling high volumes of messages

You can tune the configuration to handle high volumes of messages. For more information, see [Handling high volumes of messages](#).

Example KafkaMirrorMaker2 custom resource configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.7.1 ①
  replicas: 3 ②
  connectCluster: "my-cluster-target" ③
  clusters:
    - alias: "my-cluster-source" ⑤
      authentication: ⑥
        certificateAndKey:
          certificate: source.crt
          key: source.key
          secretName: my-user-source
        type: tls
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092 ⑦
      tls: ⑧
        trustedCertificates:
          - pattern: ".crt"
            secretName: my-cluster-source-cluster-ca-cert
    - alias: "my-cluster-target" ⑩
      authentication: ⑪
        certificateAndKey:
          certificate: target.crt
          key: target.key
          secretName: my-user-target
        type: tls
      bootstrapServers: my-cluster-target-kafka-bootstrap:9092 ⑫
      config: ⑬
        config.storage.replication.factor: 1
        offset.storage.replication.factor: 1
        status.storage.replication.factor: 1
      tls: ⑭
        trustedCertificates:
          - pattern: ".crt"
            secretName: my-cluster-target-cluster-ca-cert
  mirrors: ⑯
    - sourceCluster: "my-cluster-source" ⑮
      targetCluster: "my-cluster-target" ⑯
      sourceConnector: ⑰
      tasksMax: 10 ⑱
```

```

    autoRestart: ⑯
        enabled: true
    config
        replication.factor: 1 ⑰
        offset-syncs.topic.replication.factor: 1
        sync.topic.acls.enabled: "false"
        refresh.topics.interval.seconds: 60
        replication.policy.class:
    "org.apache.kafka.connect.mirror.IdentityReplicationPolicy"
    heartbeatConnector:
        autoRestart:
            enabled: true
        config:
            heartbeats.topic.replication.factor: 1
            replication.policy.class:
    "org.apache.kafka.connect.mirror.IdentityReplicationPolicy"
    checkpointConnector:
        autoRestart:
            enabled: true
        config:
            checkpoints.topic.replication.factor: 1
            refresh.groups.interval.seconds: 600
            sync.group.offsets.enabled: true
            sync.group.offsets.interval.seconds: 60
            emit.checkpoints.interval.seconds: 60
            replication.policy.class:
    "org.apache.kafka.connect.mirror.IdentityReplicationPolicy"
        topicsPattern: "topic1|topic2|topic3"
        groupsPattern: "group1|group2|group3"
    resources:
        requests:
            cpu: "1"
            memory: 2Gi
        limits:
            cpu: "2"
            memory: 2Gi
    logging:
        type: inline
        loggers:
            connect.root.logger.level: INFO
    readinessProbe:
        initialDelaySeconds: 15
        timeoutSeconds: 5
    livenessProbe:
        initialDelaySeconds: 15
        timeoutSeconds: 5
    jvmOptions:
        "-Xmx": "1g"
        "-Xms": "1g"
    image: my-org/my-image:latest
    rack:

```

```

topologyKey: topology.kubernetes.io/zone
template:
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
        topologyKey: "kubernetes.io/hostname"
connectContainer:
  env:
    - name: OTEL_SERVICE_NAME
      value: my-otel-service
    - name: OTEL_EXPORTER_OTLP_ENDPOINT
      value: "http://otlp-host:4317"
tracing:
  type: opentelemetry
externalConfiguration:
  env:
    - name: AWS_ACCESS_KEY_ID
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: awsAccessKey
    - name: AWS_SECRET_ACCESS_KEY
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: awsSecretAccessKey

```

- ① The Kafka Connect and MirrorMaker 2 version, which will always be the same.
- ② The number of replica nodes for the workers that run tasks.
- ③ Kafka cluster alias for Kafka Connect, which must specify the **target** Kafka cluster. The Kafka cluster is used by Kafka Connect for its internal topics.
- ④ Specification for the Kafka clusters being synchronized.
- ⑤ Cluster alias for the source Kafka cluster.
- ⑥ Authentication for the source cluster, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN.
- ⑦ Bootstrap server for connection to the source Kafka cluster.
- ⑧ TLS configuration for encrypted connections to the Kafka cluster, with trusted certificates stored in X.509 format within the specified secrets.

- ⑨ Cluster alias for the target Kafka cluster.
- ⑩ Authentication for the target Kafka cluster is configured in the same way as for the source Kafka cluster.
- ⑪ Bootstrap server for connection to the target Kafka cluster.
- ⑫ Kafka Connect configuration. Standard Apache Kafka configuration may be provided, restricted to those properties not managed directly by Strimzi.
- ⑬ TLS encryption for the target Kafka cluster is configured in the same way as for the source Kafka cluster.
- ⑭ MirrorMaker 2 connectors.
- ⑮ Cluster alias for the source cluster used by the MirrorMaker 2 connectors.
- ⑯ Cluster alias for the target cluster used by the MirrorMaker 2 connectors.
- ⑰ Configuration for the `MirrorSourceConnector` that creates remote topics. The `config` overrides the default configuration options.
- ⑱ The maximum number of tasks that the connector may create. Tasks handle the data replication and run in parallel. If the infrastructure supports the processing overhead, increasing this value can improve throughput. Kafka Connect distributes the tasks between members of the cluster. If there are more tasks than workers, workers are assigned multiple tasks. For sink connectors, aim to have one task for each topic partition consumed. For source connectors, the number of tasks that can run in parallel may also depend on the external system. The connector creates fewer than the maximum number of tasks if it cannot achieve the parallelism.
- ⑲ Enables automatic restarts of failed connectors and tasks. By default, the number of restarts is indefinite, but you can set a maximum on the number of automatic restarts using the `maxRestarts` property.
- ⑳ Replication factor for mirrored topics created at the target cluster.

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

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

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

Adds a policy that overrides the automatic renaming of remote topics. Instead of prepending the name with the name of the source cluster, the topic retains its original name. This optional setting is useful for active/passive backups and data migration. The property must be specified for all connectors. For bidirectional (active/active) replication, use the `DefaultReplicationPolicy` class to automatically rename remote topics and specify the `replication.policy.separator` property for all connectors to add a custom separator.

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

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

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

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

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

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

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

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

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

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

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

Specified Kafka Connect loggers and log levels added directly (`inline`) or indirectly (`external`) through a ConfigMap. A custom Log4j configuration must be placed under the `log4j.properties` or `log4j2.properties` key in the ConfigMap. For the Kafka Connect `log4j.rootLogger` logger, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

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

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

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

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

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

Environment variables are set for distributed tracing.

Distributed tracing is enabled by using OpenTelemetry.

External configuration for a Kubernetes Secret mounted to Kafka MirrorMaker as an environment

variable. You can also use configuration provider plugins to load configuration values from external sources.

10.8.1. Configuring active/active or active/passive modes

You can use MirrorMaker 2 in *active/passive* or *active/active* cluster configurations.

active/active cluster configuration

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

active/passive cluster configuration

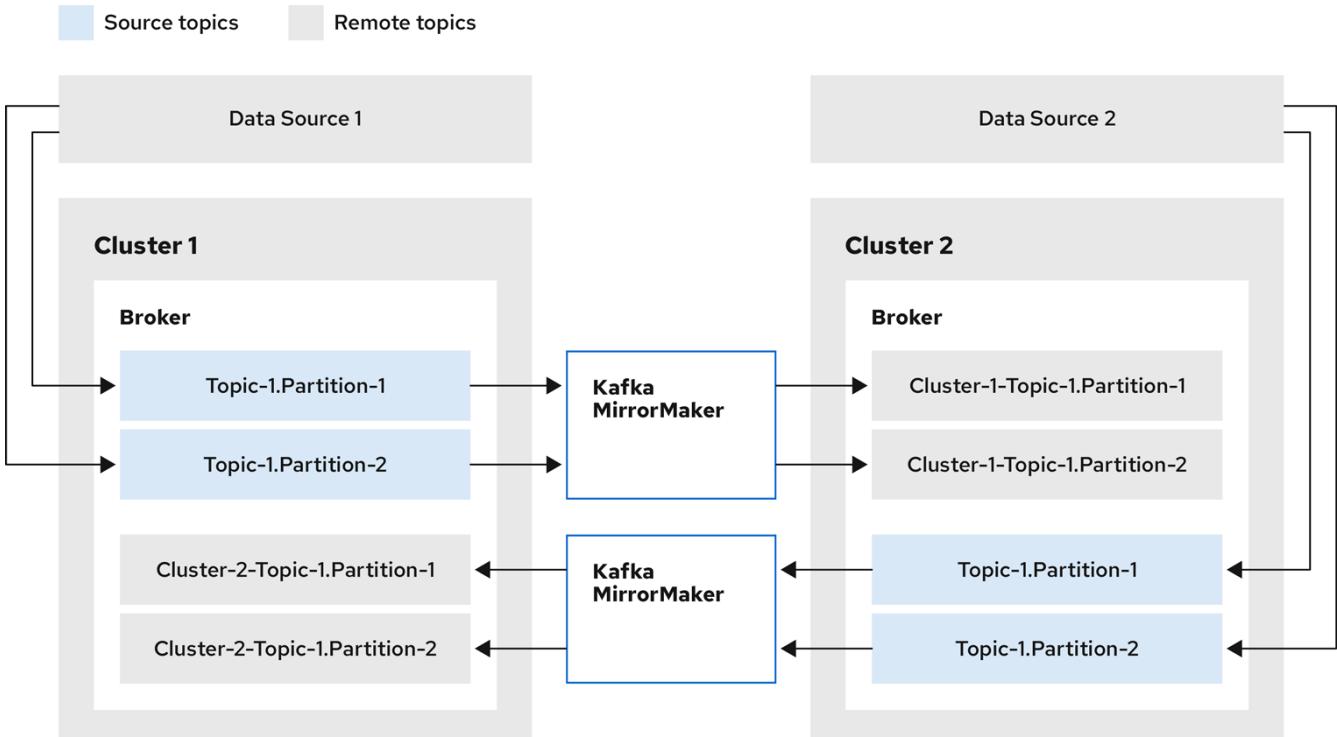
An active/passive configuration has an active cluster replicating data to a passive cluster. The passive cluster remains on standby. You might use the passive cluster for data recovery in the event of system failure.

The expectation is that producers and consumers connect to active clusters only. A MirrorMaker 2 cluster is required at each target destination.

Bidirectional replication (active/active)

The MirrorMaker 2 architecture supports bidirectional replication in an *active/active* cluster configuration.

Each cluster replicates the data of the other cluster using the concept of *source* and *remote* topics. As the same topics are stored in each cluster, remote topics are automatically renamed by MirrorMaker 2 to represent the source cluster. The name of the originating cluster is prepended to the name of the topic.



222_Streams_0322

Figure 3. Topic renaming

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

The concept of replication through *remote* topics is useful when configuring an architecture that requires data aggregation. Consumers can subscribe to source and remote topics within the same cluster, without the need for a separate aggregation cluster.

Unidirectional replication (active/passive)

The MirrorMaker 2 architecture supports unidirectional replication in an *active/passive* cluster configuration.

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

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

10.8.2. Configuring MirrorMaker 2 for multiple instances

By default, Strimzi configures the group ID and names of the internal topics used by the Kafka Connect framework that MirrorMaker 2 runs on. When running multiple instances of MirrorMaker 2, and they share the same `connectCluster` value, you must change these default settings using the following `config` properties:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
```

```

name: my-mirror-maker2
spec:
  connectCluster: "my-cluster-target"
  clusters:
    - alias: "my-cluster-target"
      config:
        group.id: my-connect-cluster ①
        offset.storage.topic: my-connect-cluster-offsets ②
        config.storage.topic: my-connect-cluster-configs ③
        status.storage.topic: my-connect-cluster-status ④
        # ...
      # ...

```

- ① The Kafka Connect cluster group ID within Kafka.
- ② Kafka topic that stores connector offsets.
- ③ Kafka topic that stores connector and task status configurations.
- ④ Kafka topic that stores connector and task status updates.

NOTE

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

The `connectCluster` setting specifies the alias of the target Kafka cluster used by Kafka Connect for its internal topics. As a result, modifications to the `connectCluster`, group ID, and internal topic naming configuration are specific to the target Kafka cluster. You don't need to make changes if two MirrorMaker 2 instances are using the same source Kafka cluster or in an active-active mode where each MirrorMaker 2 instance has a different `connectCluster` setting and target cluster.

However, if multiple MirrorMaker 2 instances share the same `connectCluster`, each instance connecting to the same target Kafka cluster is deployed with the same values. In practice, this means all instances form a cluster and use the same internal topics.

Multiple instances attempting to use the same internal topics will cause unexpected errors, so you must change the values of these properties for each instance.

10.8.3. Configuring MirrorMaker 2 connectors

Use MirrorMaker 2 connector configuration for the internal connectors that orchestrate the synchronization of data between Kafka clusters.

MirrorMaker 2 consists of the following connectors:

MirrorSourceConnector

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

MirrorCheckpointConnector

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

MirrorHeartbeatConnector

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

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

Table 13. MirrorMaker 2 connector configuration properties

Property	sourceConnector	checkpointConnector	heartbeatConnector
admin.timeout.ms Timeout for admin tasks, such as detecting new topics. Default is 60000 (1 minute).	□	□	□
replication.policy.class Policy to define the remote topic naming convention. Default is org.apache.kafka.connect.mirror.DefaultReplicationPolicy .	□	□	□
replication.policy.separator The separator used for topic naming in the target cluster. By default, the separator is set to a dot (.). Separator configuration is only applicable to the DefaultReplicationPolicy replication policy class, which defines remote topic names. The IdentityReplicationPolicy class does not use the property as topics retain their original names.	□	□	□
consumer.poll.timeout.ms Timeout when polling the source cluster. Default is 1000 (1 second).	□	□	
offset-syncs.topic.location The location of the offset-syncs topic, which can be the source (default) or target cluster.	□	□	
topic.filter.class Topic filter to select the topics to replicate. Default is org.apache.kafka.connect.mirror.DefaultTopicFilter .	□	□	

Property	sourceConnector	checkpointConnector	heartbeatConnector
config.property.filter.class Topic filter to select the topic configuration properties to replicate. Default is <code>org.apache.kafka.connect.mirror.DefaultConfigPropertyFilter</code> .	□		
config.properties.exclude Topic configuration properties that should not be replicated. Supports comma-separated property names and regular expressions.	□		
offset.lag.max Maximum allowable (out-of-sync) offset lag before a remote partition is synchronized. Default is <code>100</code> .	□		
offset-syncs.topic.replication.factor Replication factor for the internal <code>offset-syncs</code> topic. Default is <code>3</code> .	□		
refresh.topics.enabled Enables check for new topics and partitions. Default is <code>true</code> .	□		
refresh.topics.interval.seconds Frequency of topic refresh. Default is <code>600</code> (10 minutes). By default, a check for new topics in the source cluster is made every 10 minutes. You can change the frequency by adding <code>refresh.topics.interval.seconds</code> to the source connector configuration.	□		
replication.factor The replication factor for new topics. Default is <code>2</code> .	□		
sync.topic.acls.enabled Enables synchronization of ACLs from the source cluster. Default is <code>true</code> . For more information, see Synchronizing ACL rules for remote topics .	□		

Property	sourceConnector	checkpointConnector	heartbeatConnector
sync.topic.acls.interval.seconds Frequency of ACL synchronization. Default is 600 (10 minutes).	□		
sync.topic.configs.enabled Enables synchronization of topic configuration from the source cluster. Default is true .	□		
sync.topic.configs.interval.seconds Frequency of topic configuration synchronization. Default 600 (10 minutes).	□		
checkpoints.topic.replication.factor Replication factor for the internal checkpoints topic. Default is 3 .		□	
emit.checkpoints.enabled Enables synchronization of consumer offsets to the target cluster. Default is true .		□	
emit.checkpoints.interval.seconds Frequency of consumer offset synchronization. Default is 60 (1 minute).		□	
group.filter.class Group filter to select the consumer groups to replicate. Default is org.apache.kafka.connect.mirror.DefaultGroupFilter .		□	
refresh.groups.enabled Enables check for new consumer groups. Default is true .		□	
refresh.groups.interval.seconds Frequency of consumer group refresh. Default is 600 (10 minutes).		□	
sync.group.offsets.enabled Enables synchronization of consumer group offsets to the target cluster __consumer_offsets topic. Default is false .		□	

Property	sourceConnector	checkpointConnector	heartbeatConnector
sync.group.offsets.interval.seconds Frequency of consumer group offset synchronization. Default is 60 (1 minute).		□	
emit.heartbeats.enabled Enables connectivity checks on the target cluster. Default is true .			□
emit.heartbeats.interval.seconds Frequency of connectivity checks. Default is 1 (1 second).			□
heartbeats.topic.replication.factor Replication factor for the internal heartbeats topic. Default is 3 .			□

Changing the location of the consumer group offsets topic

MirrorMaker 2 tracks offsets for consumer groups using internal topics.

offset-syncs topic

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

checkpoints topic

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

As they are used internally by MirrorMaker 2, you do not interact directly with these topics.

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

The location of the **offset-syncs** topic is the **source** cluster by default. You can use the **offset-syncs.topic.location** connector configuration to change this to the **target** cluster. You need read/write access to the cluster that contains the topic. Using the target cluster as the location of the **offset-syncs** topic allows you to use MirrorMaker 2 even if you have only read access to the source cluster.

Synchronizing consumer group offsets

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

Offset synchronization is particularly useful in an *active/passive* configuration. If the active cluster

goes down, consumer applications can switch to the passive (standby) cluster and pick up from the last transferred offset position.

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

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

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

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

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

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

Deciding when to use the heartbeat connector

The heartbeat connector emits heartbeats to check connectivity between source and target Kafka clusters. An internal `heartbeat` topic is replicated from the source cluster, which means that the heartbeat connector must be connected to the source cluster. The `heartbeat` topic is located on the target cluster, which allows it to do the following:

- Identify all source clusters it is mirroring data from
- Verify the liveness and latency of the mirroring process

This helps to make sure that the process is not stuck or has stopped for any reason. While the heartbeat connector can be a valuable tool for monitoring the mirroring processes between Kafka clusters, it's not always necessary to use it. For example, if your deployment has low network latency or a small number of topics, you might prefer to monitor the mirroring process using log messages or other monitoring tools. If you decide not to use the heartbeat connector, simply omit it from your MirrorMaker 2 configuration.

Aligning the configuration of MirrorMaker 2 connectors

To ensure that MirrorMaker 2 connectors work properly, make sure to align certain configuration settings across connectors. Specifically, ensure that the following properties have the same value across all applicable connectors:

- `replication.policy.class`
- `replication.policy.separator`
- `offset-syncs.topic.location`
- `topic.filter.class`

For example, the value for `replication.policy.class` must be the same for the source, checkpoint, and heartbeat connectors. Mismatched or missing settings cause issues with data replication or offset syncing, so it's essential to keep all relevant connectors configured with the same settings.

10.8.4. Configuring MirrorMaker 2 connector producers and consumers

MirrorMaker 2 connectors use internal producers and consumers. If needed, you can configure these producers and consumers to override the default settings.

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

IMPORTANT

Producer and consumer configuration options depend on the MirrorMaker 2 implementation, and may be subject to change.

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

Table 14. Source connector producers and consumers

Type	Description	Configuration
Producer	Sends topic messages to the target Kafka cluster. Consider tuning the configuration of this producer when it is handling large volumes of data.	<code>mirrors.sourceConnector.config: producer.override.*</code>

Type	Description	Configuration
Producer	Writes to the <code>offset-syncs</code> topic, which maps the source and target offsets for replicated topic partitions.	<code>mirrors.sourceConnector.config: producer.*</code>
Consumer	Retrieves topic messages from the source Kafka cluster.	<code>mirrors.sourceConnector.config: consumer.*</code>

Table 15. Checkpoint connector producers and consumers

Type	Description	Configuration
Producer	Emits consumer offset checkpoints.	<code>mirrors.checkpointConnector.config: producer.override.*</code>
Consumer	Loads the <code>offset-syncs</code> topic.	<code>mirrors.checkpointConnector.config: consumer.*</code>

NOTE

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

Table 16. Heartbeat connector producer

Type	Description	Configuration
Producer	Emits heartbeats.	<code>mirrors.heartbeatConnector.config: producer.override.*</code>

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

Example configuration for connector producers and consumers

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.7.1
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 5
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100

```

```

producer.request.timeout.ms: 30000
consumer.fetch.max.bytes: 52428800
# ...
checkpointConnector:
  config:
    producer.override.request.timeout.ms: 30000
    consumer.max.poll.interval.ms: 300000
    # ...
heartbeatConnector:
  config:
    producer.override.request.timeout.ms: 30000
    # ...

```

10.8.5. Specifying a maximum number of data replication tasks

Connectors create the tasks that are responsible for moving data in and out of Kafka. Each connector comprises one or more tasks that are distributed across a group of worker pods that run the tasks. Increasing the number of tasks can help with performance issues when replicating a large number of partitions or synchronizing the offsets of a large number of consumer groups.

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

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

The heartbeat connector always uses a single task.

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

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

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

Increasing the number of tasks for the source connector

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2

```

```
spec:  
  # ...  
  mirrors:  
    - sourceCluster: "my-cluster-source"  
      targetCluster: "my-cluster-target"  
      sourceConnector:  
        tasksMax: 10  
  # ...
```

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

Increasing the number of tasks for the checkpoint connector

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

By default, MirrorMaker 2 checks for new consumer groups every 10 minutes. You can adjust the `refresh.groups.interval.seconds` configuration to change the frequency. Take care when adjusting lower. More frequent checks can have a negative impact on performance.

Checking connector task operations

If you are using Prometheus and Grafana to monitor your deployment, you can check MirrorMaker 2 performance. The example MirrorMaker 2 Grafana dashboard provided with Strimzi shows the following metrics related to tasks and latency.

- The number of tasks
- Replication latency
- Offset synchronization latency

Additional resources

- [Introducing metrics](#)

10.8.6. Synchronizing ACL rules for remote topics

When using MirrorMaker 2 with Strimzi, it is possible to synchronize ACL rules for remote topics. However, this feature is only available if you are not using the User Operator.

If you are using `type: simple` authorization without the User Operator, the ACL rules that manage access to brokers also apply to remote topics. This means that users who have read access to a source topic can also read its remote equivalent.

NOTE OAuth 2.0 authorization does not support access to remote topics in this way.

10.8.7. Securing a Kafka MirrorMaker 2 deployment

This procedure describes in outline the configuration required to secure a MirrorMaker 2 deployment.

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

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

You can configure authentication and authorization mechanisms. The security options implemented for the source and target Kafka clusters must be compatible with the security options implemented for MirrorMaker 2.

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

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

Before you start

Before starting this procedure, take a look at the [example configuration files](#) provided by Strimzi. They include examples for securing a deployment of MirrorMaker 2 using mTLS or SCRAM-SHA-512 authentication. The examples specify internal listeners for connecting within a Kubernetes cluster.

The examples also provide the configuration for full authorization, including the ACLs that allow user operations on the source and target Kafka clusters.

When configuring user access to source and target Kafka clusters, ACLs must grant access rights to internal MirrorMaker 2 connectors and read/write access to the cluster group and internal topics used by the underlying Kafka Connect framework in the target cluster. If you've renamed the cluster group or internal topics, such as when [configuring MirrorMaker 2 for multiple instances](#), use those names in the ACLs configuration.

Simple authorization uses ACL rules managed by the Kafka `AclAuthorizer` and `StandardAuthorizer` plugins to ensure appropriate access levels. For more information on configuring a `KafkaUser` resource to use simple authorization, see the [AclRule schema reference](#).

Prerequisites

- Strimzi is running
- Separate namespaces for source and target clusters

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

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

Procedure

1. Configure two **Kafka** resources, one to secure the source Kafka cluster and one to secure the target Kafka cluster.

You can add listener configuration for authentication and enable authorization.

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

Example source Kafka cluster configuration with TLS encryption and mTLS authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-source-cluster
spec:
  kafka:
    version: 3.7.1
    replicas: 1
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min_isr: 1
      default.replication.factor: 1
      min.insync.replicas: 1
      inter.broker.protocol.version: "3.7"
    storage:
      type: jbod
      volumes:
        - id: 0
```

```

    type: persistent-claim
    size: 100Gi
    deleteClaim: false
zookeeper:
  replicas: 1
  storage:
    type: persistent-claim
    size: 100Gi
    deleteClaim: false
entityOperator:
  topicOperator: {}
  userOperator: {}

```

Example target Kafka cluster configuration with TLS encryption and mTLS authentication

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-target-cluster
spec:
  kafka:
    version: 3.7.1
    replicas: 1
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min_isr: 1
      default.replication.factor: 1
      min.insync.replicas: 1
      inter.broker.protocol.version: "3.7"
    storage:
      type: jbod
      volumes:
        - id: 0
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
  zookeeper:
    replicas: 1
    storage:
      type: persistent-claim

```

```
size: 100Gi
  deleteClaim: false
entityOperator:
  topicOperator: {}
  userOperator: {}
```

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

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

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

The certificates are created in the secret `<cluster_name>-cluster-ca-cert`.

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

- Configure the same authentication and authorization types as the corresponding source and target Kafka cluster. For example, if you used `tls` authentication and the `simple` authorization type in the **Kafka** configuration for the source Kafka cluster, use the same in the **KafkaUser** configuration.
- Configure the ACLs needed by MirrorMaker 2 to allow operations on the source and target Kafka clusters.

Example source user configuration for mTLS authentication

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

```

type: topic
name: "*"
operations:
- DescribeConfigs
- Read
# MirrorCheckpointConnector
- resource:
  type: cluster
operations:
- Describe
- resource: # Needed for every group for which offsets are synced
  type: group
  name: "*"
operations:
- Describe
- resource: # Not needed if offset-syncs.topic.location=target
  type: topic
  name: mm2-offset-syncs.my-target-cluster.internal
operations:
- Read

```

Example target user configuration for mTLS authentication

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-target-user
  labels:
    strimzi.io/cluster: my-target-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
  acls:
    # cluster group
    - resource:
        type: group
        name: mirrormaker2-cluster
    operations:
      - Read
  # access to config.storage.topic
  - resource:
      type: topic
      name: mirrormaker2-cluster-configs
  operations:
    - Create
    - Describe
    - DescribeConfigs
    - Read

```

```

    - Write
# access to status.storage.topic
- resource:
    type: topic
    name: mirrormaker2-cluster-status
operations:
    - Create
    - Describe
    - DescribeConfigs
    - Read
    - Write
# access to offset.storage.topic
- resource:
    type: topic
    name: mirrormaker2-cluster-offsets
operations:
    - Create
    - Describe
    - DescribeConfigs
    - Read
    - Write
# MirrorSourceConnector
- resource: # Needed for every topic which is mirrored
    type: topic
    name: "*"
operations:
    - Create
    - Alter
    - AlterConfigs
    - Write
# MirrorCheckpointConnector
- resource:
    type: cluster
operations:
    - Describe
- resource:
    type: topic
    name: my-source-cluster.checkpoints.internal
operations:
    - Create
    - Describe
    - Read
    - Write
- resource: # Needed for every group for which the offset is synced
    type: group
    name: "*"
operations:
    - Read
    - Describe
# MirrorHeartbeatConnector
- resource:

```

```
    type: topic
    name: heartbeats
  operations:
    - Create
    - Describe
    - Write
```

NOTE

You can use a certificate issued outside the User Operator by setting `type` to `tls-external`. For more information, see the [KafkaUserSpec schema reference](#).

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

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

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

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

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

Example MirrorMaker 2 configuration with TLS encryption and mTLS authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker-2
spec:
  version: 3.7.1
  replicas: 1
  connectCluster: "my-target-cluster"
  clusters:
    - alias: "my-source-cluster"
      bootstrapServers: my-source-cluster-kafka-bootstrap:9093
      tls: ①
        trustedCertificates:
          - secretName: my-source-cluster-cluster-ca-cert
            pattern: "*.crt"
      authentication: ②
        type: tls
        certificateAndKey:
          secretName: my-source-user
          certificate: user.crt
          key: user.key
    - alias: "my-target-cluster"
```

```

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

```

- ① The TLS certificates for the source Kafka cluster. If they are in a separate namespace, copy the cluster secrets from the namespace of the Kafka cluster.
- ② The user authentication for accessing the source Kafka cluster using the TLS mechanism.
- ③ The TLS certificates for the target Kafka cluster.
- ④ The user authentication for accessing the target Kafka cluster.

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

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

10.8.8. Manually stopping or pausing MirrorMaker 2 connectors

If you are using `KafkaMirrorMaker2` resources to configure internal MirrorMaker connectors, use the `state` configuration to either stop or pause a connector. In contrast to the paused state, where the connector and tasks remain instantiated, stopping a connector retains only the configuration, with no active processes. Stopping a connector from running may be more suitable for longer durations than just pausing. While a paused connector is quicker to resume, a stopped connector has the advantages of freeing up memory and resources.

NOTE

The `state` configuration replaces the (deprecated) `pause` configuration in the `KafkaMirrorMaker2ConnectorSpec` schema, which allows pauses on connectors. If you were previously using the `pause` configuration to pause connectors, we encourage you to transition to using the `state` configuration only to avoid conflicts.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Find the name of the `KafkaMirrorMaker2` custom resource that controls the MirrorMaker 2 connector you want to pause or stop:

```
kubectl get KafkaMirrorMaker2
```

2. Edit the `KafkaMirrorMaker2` resource to stop or pause the connector.

Example configuration for stopping a MirrorMaker 2 connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.7.1
  replicas: 3
  connectCluster: "my-cluster-target"
  clusters:
    # ...
  mirrors:
    - sourceCluster: "my-cluster-source"
      targetCluster: "my-cluster-target"
      sourceConnector:
        tasksMax: 10
        autoRestart:
          enabled: true
          state: stopped
    # ...
```

Change the `state` configuration to `stopped` or `paused`. The default state for the connector when

this property is not set is `running`.

3. Apply the changes to the `KafkaMirrorMaker2` configuration.

You can resume the connector by changing `state` to `running` or removing the configuration.

NOTE

Alternatively, you can [expose the Kafka Connect API](#) and use the `stop` and `pause` endpoints to stop a connector from running. For example, `PUT /connectors/<connector_name>/stop`. You can then use the `resume` endpoint to restart it.

10.8.9. Manually restarting MirrorMaker 2 connectors

Use the `strimzi.io/restart-connector` annotation to manually trigger a restart of a MirrorMaker 2 connector.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Find the name of the `KafkaMirrorMaker2` custom resource that controls the Kafka MirrorMaker 2 connector you want to restart:

```
kubectl get KafkaMirrorMaker2
```

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

```
kubectl describe KafkaMirrorMaker2 <mirrormaker_cluster_name>
```

3. Use the name of the connector to restart the connector by annotating the `KafkaMirrorMaker2` resource in Kubernetes:

```
kubectl annotate KafkaMirrorMaker2 <mirrormaker_cluster_name> "strimzi.io/restart-connector=<mirrormaker_connector_name>"
```

In this example, connector `my-connector` in the `my-mirror-maker-2` cluster is restarted:

```
kubectl annotate KafkaMirrorMaker2 my-mirror-maker-2 "strimzi.io/restart-connector=my-connector"
```

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

The MirrorMaker 2 connector is restarted, as long as the annotation was detected by the reconciliation process. When MirrorMaker 2 accepts the request, the annotation is removed

from the `KafkaMirrorMaker2` custom resource.

10.8.10. Manually restarting MirrorMaker 2 connector tasks

Use the `strimzi.io/restart-connector-task` annotation to manually trigger a restart of a MirrorMaker 2 connector.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Find the name of the `KafkaMirrorMaker2` custom resource that controls the MirrorMaker 2 connector task you want to restart:

```
kubectl get KafkaMirrorMaker2
```

2. Find the name of the connector and the ID of the task to be restarted from the `KafkaMirrorMaker2` custom resource:

```
kubectl describe KafkaMirrorMaker2 <mirrormaker_cluster_name>
```

Task IDs are non-negative integers, starting from 0.

3. Use the name and ID to restart the connector task by annotating the `KafkaMirrorMaker2` resource in Kubernetes:

```
kubectl annotate KafkaMirrorMaker2 <mirrormaker_cluster_name> "strimzi.io/restart-connector-task=<mirrormaker_connector_name>:<task_id>"
```

In this example, task `0` for connector `my-connector` in the `my-mirror-maker-2` cluster is restarted:

```
kubectl annotate KafkaMirrorMaker2 my-mirror-maker-2 "strimzi.io/restart-connector-task=my-connector:0"
```

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

The MirrorMaker 2 connector task is restarted, as long as the annotation was detected by the reconciliation process. When MirrorMaker 2 accepts the request, the annotation is removed from the `KafkaMirrorMaker2` custom resource.

10.9. Configuring Kafka MirrorMaker (deprecated)

Update the `spec` properties of the `KafkaMirrorMaker` custom resource to configure your Kafka MirrorMaker deployment.

You can configure access control for producers and consumers using TLS or SASL authentication. This procedure shows a configuration that uses TLS encryption and mTLS authentication on the consumer and producer side.

For a deeper understanding of the Kafka MirrorMaker cluster configuration options, refer to the [Strimzi Custom Resource API Reference](#).

IMPORTANT

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

Example KafkaMirrorMaker custom resource configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  replicas: 3 ①
  consumer:
    bootstrapServers: my-source-cluster-kafka-bootstrap:9092 ②
    groupId: "my-group" ③
    numStreams: 2 ④
    offsetCommitInterval: 120000 ⑤
    tls: ⑥
      trustedCertificates:
        - secretName: my-source-cluster-ca-cert
          pattern: ".crt"
    authentication: ⑦
      type: tls
      certificateAndKey:
        secretName: my-source-secret
        certificate: public.crt
        key: private.key
    config: ⑧
      max.poll.records: 100
      receive.buffer.bytes: 32768
  producer:
    bootstrapServers: my-target-cluster-kafka-bootstrap:9092
    abortOnSendFailure: false ⑨
    tls:
      trustedCertificates:
        - secretName: my-target-cluster-ca-cert
          pattern: ".crt"
      authentication:
        type: tls
```

```

certificateAndKey:
  secretName: my-target-secret
  certificate: public.crt
  key: private.key
config:
  compression.type: gzip
  batch.size: 8192
include: "my-topic|other-topic" ⑩
resources: ⑪
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
logging: ⑫
  type: inline
  loggers:
    mirrormaker.root.logger: INFO
readinessProbe: ⑬
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
metricsConfig: ⑭
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: my-key
jvmOptions: ⑮
  "-Xmx": "1g"
  "-Xms": "1g"
image: my-org/my-image:latest ⑯
template: ⑰
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
mirrorMakerContainer: ⑱
  env:
    - name: OTEL_SERVICE_NAME

```

```

    value: my-otel-service
  - name: OTEL_EXPORTER_OTLP_ENDPOINT
    value: "http://otlp-host:4317"
  tracing: ⑯
    type: opentelemetry

```

- ① The number of replica nodes.
- ② Bootstrap servers for consumer and producer.
- ③ Group ID for the consumer.
- ④ The number of consumer streams.
- ⑤ The offset auto-commit interval in milliseconds.
- ⑥ TLS configuration for encrypted connections to the Kafka cluster, with trusted certificates stored in X.509 format within the specified secrets.
- ⑦ Authentication for consumer or producer, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN.
- ⑧ Kafka configuration options for consumer and producer.
- ⑨ If the `abortOnSendFailure` property is set to `true`, Kafka MirrorMaker will exit and the container will restart following a send failure for a message.
- ⑩ A list of included topics mirrored from source to target Kafka cluster.
- ⑪ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑫ Specified loggers and log levels added directly (`inline`) or indirectly (`external`) through a ConfigMap. A custom Log4j configuration must be placed under the `log4j.properties` or `log4j2.properties` key in the ConfigMap. MirrorMaker has a single logger called `mirrormaker.root.logger`. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
- ⑬ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
- ⑭ Prometheus metrics, which are enabled by referencing a ConfigMap containing configuration for the Prometheus JMX exporter in this example. You can enable metrics without further configuration using a reference to a ConfigMap containing an empty file under `metricsConfig.valueFrom.configMapKeyRef.key`.
- ⑮ JVM configuration options to optimize performance for the Virtual Machine (VM) running Kafka MirrorMaker.
- ⑯ ADVANCED OPTION: Container image configuration, which is recommended only in special situations.
- ⑰ Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.
- ⑱ Environment variables are set for distributed tracing.
- ⑲ Distributed tracing is enabled by using OpenTelemetry.

WARNING With the `abortOnSendFailure` property set to `false`, the producer attempts to

send the next message in a topic. The original message might be lost, as there is no attempt to resend a failed message.

10.10. Configuring the Kafka Bridge

Update the `spec` properties of the `KafkaBridge` custom resource to configure your Kafka Bridge deployment.

In order to prevent issues arising when client consumer requests are processed by different Kafka Bridge instances, address-based routing must be employed to ensure that requests are routed to the right Kafka Bridge instance. Additionally, each independent Kafka Bridge instance must have a replica. A Kafka Bridge instance has its own state which is not shared with another instances.

For a deeper understanding of the Kafka Bridge cluster configuration options, refer to the [Strimzi Custom Resource API Reference](#).

Example KafkaBridge custom resource configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  replicas: 3 ①
  bootstrapServers: <cluster_name>-cluster-kafka-bootstrap:9092 ②
  tls: ③
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        pattern: "*.crt"
      - secretName: my-cluster-cluster-cert
        certificate: ca2.crt
  authentication: ④
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  http: ⑤
    port: 8080
    cors: ⑥
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  consumer: ⑦
    config:
      auto.offset.reset: earliest
  producer: ⑧
    config:
      delivery.timeout.ms: 300000
  resources: ⑨
    requests:
```

```

cpu: "1"
memory: 2Gi
limits:
  cpu: "2"
  memory: 2Gi
logging: ⑩
  type: inline
  loggers:
    logger.bridge.level: INFO
    # enabling DEBUG just for send operation
    logger.send.name: "http.openapi.operation.send"
    logger.send.level: DEBUG
jvmOptions: ⑪
  "-Xmx": "1g"
  "-Xms": "1g"
readinessProbe: ⑫
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
image: my-org/my-image:latest ⑬
template: ⑭
pod:
  affinity:
    podAntiAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: application
                operator: In
            values:
              - postgresql
              - mongodb
      topologyKey: "kubernetes.io/hostname"
bridgeContainer: ⑮
env:
  - name: OTEL_SERVICE_NAME
    value: my-otel-service
  - name: OTEL_EXPORTER_OTLP_ENDPOINT
    value: "http://otlp-host:4317"
tracing:
  type: opentelemetry ⑯

```

① The number of replica nodes.

② Bootstrap server for connection to the target Kafka cluster. Use the name of the Kafka cluster as the <cluster_name>.

③ TLS configuration for encrypted connections to the Kafka cluster, with trusted certificates stored in X.509 format within the specified secrets.

- ④ Authentication for the Kafka Bridge cluster, specified as mTLS, token-based OAuth, SASL-based SCRAM-SHA-256/SCRAM-SHA-512, or PLAIN. By default, the Kafka Bridge connects to Kafka brokers without authentication.
- ⑤ HTTP access to Kafka brokers.
- ⑥ CORS access specifying selected resources and access methods. Additional HTTP headers in requests describe the origins that are permitted access to the Kafka cluster.
- ⑦ Consumer configuration options.
- ⑧ Producer configuration options.
- ⑨ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑩ Specified Kafka Bridge loggers and log levels added directly (`inline`) or indirectly (`external`) through a ConfigMap. A custom Log4j configuration must be placed under the `log4j.properties` or `log4j2.properties` key in the ConfigMap. For the Kafka Bridge loggers, you can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
- ⑪ JVM configuration options to optimize performance for the Virtual Machine (VM) running the Kafka Bridge.
- ⑫ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
- ⑬ Optional: Container image configuration, which is recommended only in special situations.
- ⑭ Template customization. Here a pod is scheduled with anti-affinity, so the pod is not scheduled on nodes with the same hostname.
- ⑮ Environment variables are set for distributed tracing.
- ⑯ Distributed tracing is enabled by using OpenTelemetry.

Additional resources

- [Using the Kafka Bridge](#)

10.11. Configuring Kafka and ZooKeeper storage

Strimzi provides flexibility in configuring the data storage options of Kafka and ZooKeeper.

The supported storage types are:

- Ephemeral (Recommended for development only)
- Persistent
- JBOD (Kafka only; not available for ZooKeeper)
- Tiered storage (Early access)

To configure storage, you specify `storage` properties in the custom resource of the component. The storage type is set using the `storage.type` property. When using node pools, you can specify storage configuration unique to each node pool used in a Kafka cluster. The same storage properties available to the `Kafka` resource are also available to the `KafkaNodePool` pool resource.

Tiered storage provides more flexibility for data management by leveraging the parallel use of storage types with different characteristics. For example, tiered storage might include the following:

- Higher performance and higher cost block storage
- Lower performance and lower cost object storage

Tiered storage is an early access feature in Kafka. To configure tiered storage, you specify `tieredStorage` properties. Tiered storage is configured only at the cluster level using the `Kafka` custom resource.

The storage-related schema references provide more information on the storage configuration properties:

- `EphemeralStorage` schema reference
- `PersistentClaimStorage` schema reference
- `JbodStorage` schema reference
- `TieredStorageCustom` schema reference

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

10.11.1. Data storage considerations

For Strimzi to work well, an efficient data storage infrastructure is essential. We strongly recommend using block storage. Strimzi is only tested for use with block storage. File storage, such as NFS, is not tested and there is no guarantee it will work.

Choose one of the following options for your block storage:

- A cloud-based block storage solution, such as [Amazon Elastic Block Store \(EBS\)](#)
- Persistent storage using [local persistent volumes](#)
- Storage Area Network (SAN) volumes accessed by a protocol such as *Fibre Channel* or *iSCSI*

NOTE Strimzi does not require Kubernetes raw block volumes.

File systems

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

For more information, refer to [Filesystem Selection](#) in the Kafka documentation.

Disk usage

Use separate disks for Apache Kafka and ZooKeeper.

Solid-state drives (SSDs), though not essential, can improve the performance of Kafka in large clusters where data is sent to and received from multiple topics asynchronously. SSDs are particularly effective with ZooKeeper, which requires fast, low latency data access.

NOTE

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

10.11.2. Ephemeral storage

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

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

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

IMPORTANT

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

To use ephemeral storage, you set the storage type configuration in the `Kafka` or `ZooKeeper` resource to `ephemeral`. If you are using node pools, you can also specify `ephemeral` in the storage configuration of individual node pools.

Example ephemeral storage configuration

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

Mount path of Kafka log directories

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

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

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

10.11.3. Persistent storage

Persistent data storage retains data in the event of system disruption. For pods that use persistent data storage, data is persisted across pod failures and restarts. Because of its permanent nature, persistent storage is recommended for production environments.

The following examples show common types of persistent volumes supported by Kubernetes:

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

To use persistent storage in Strimzi, you specify `persistent-claim` in the storage configuration of the `Kafka` or `ZooKeeper` resources. If you are using node pools, you can also specify `persistent-claim` in the storage configuration of individual node pools.

You configure the resource so that pods use `Persistent Volume Claims` (PVCs) to make storage requests on persistent volumes (PVs). PVs represent storage volumes that are created on demand and are independent of the pods that use them. The PVC requests the amount of storage required when a pod is being created. The underlying storage infrastructure of the PV does not need to be understood. If a PV matches the storage criteria, the PVC is bound to the PV.

You have two options for specifying the storage type:

`storage.type: persistent-claim`

If you choose `persistent-claim` as the storage type, a single persistent storage volume is defined.

`storage.type: jbod`

When you select `jbod` as the storage type, you have the flexibility to define an array of persistent storage volumes using unique IDs.

In a production environment, it is recommended to configure the following:

- For Kafka or node pools, set `storage.type` to `jbod` with one or more persistent volumes.
- For ZooKeeper, set `storage.type` as `persistent-claim` for a single persistent volume.

Persistent storage also has the following configuration options:

`id (optional)`

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

size (required)

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

class (optional)

PVCs can request different types of persistent storage by specifying a [StorageClass](#). Storage classes define storage profiles and dynamically provision PVs based on that profile. If a storage class is not specified, the storage class marked as default in the Kubernetes cluster is used. Persistent storage options might include SAN storage types or [local persistent volumes](#).

selector (optional)

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

deleteClaim (optional)

Boolean value to specify whether the PVC is deleted when the cluster is uninstalled. Default is `false`.

WARNING

Increasing the size of persistent volumes in an existing Strimzi cluster is only supported in Kubernetes versions that support persistent volume resizing. The persistent volume to be resized must use a storage class that supports volume expansion. For other versions of Kubernetes and storage classes that do not support volume expansion, you must decide the necessary storage size before deploying the cluster. Decreasing the size of existing persistent volumes is not possible.

Example persistent storage configuration for Kafka and ZooKeeper

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    storage:
      type: jbod
      volumes:
        - id: 0
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
        - id: 1
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
        - id: 2
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
# ...
```

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

Example persistent storage configuration with specific storage class

```
# ...  
storage:  
  type: persistent-claim  
  size: 500Gi  
  class: my-storage-class  
# ...
```

Use a [selector](#) to specify a labeled persistent volume that provides certain features, such as an SSD.

Example persistent storage configuration with selector

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

Storage class overrides

Instead of using the default storage class, you can specify a different storage class for one or more Kafka or ZooKeeper nodes. This is useful, for example, when storage classes are restricted to different availability zones or data centers. You can use the [overrides](#) field for this purpose.

In this example, the default storage class is named [my-storage-class](#):

Example storage configuration with class overrides

```
apiVersion: kafka.strimzi.io/v1beta2  
kind: Kafka  
metadata:  
  labels:  
    app: my-cluster  
  name: my-cluster  
  namespace: myproject  
spec:  
  # ...  
  kafka:  
    replicas: 3
```

```

storage:
  type: jbod
  volumes:
    - id: 0
      type: persistent-claim
      size: 100Gi
      deleteClaim: false
      class: my-storage-class
      overrides:
        - broker: 0
          class: my-storage-class-zone-1a
        - broker: 1
          class: my-storage-class-zone-1b
        - broker: 2
          class: my-storage-class-zone-1c
      # ...
# ...
zookeeper:
  replicas: 3
  storage:
    deleteClaim: true
    size: 100Gi
    type: persistent-claim
    class: my-storage-class
    overrides:
      - broker: 0
        class: my-storage-class-zone-1a
      - broker: 1
        class: my-storage-class-zone-1b
      - broker: 2
        class: my-storage-class-zone-1c
    # ...

```

As a result of the configured `overrides` property, the volumes use the following storage classes:

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

The `overrides` property is currently used only to override the storage `class`. Overrides for other storage configuration properties is not currently supported.

PVC resources for persistent storage

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

data-cluster-name-kafka-idx

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

data-cluster-name-zookeeper-idx

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

Mount path of Kafka log directories

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

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

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

10.11.4. Resizing persistent volumes

Persistent volumes used by a cluster can be resized without any risk of data loss, as long as the storage infrastructure supports it. Following a configuration update to change the size of the storage, Strimzi instructs the storage infrastructure to make the change. Storage expansion is supported in Strimzi clusters that use persistent-claim volumes.

Storage reduction is only possible when using multiple disks per broker. You can remove a disk after moving all partitions on the disk to other volumes within the same broker (intra-broker) or to other brokers within the same cluster (intra-cluster).

IMPORTANT

You cannot decrease the size of persistent volumes because it is not currently supported in Kubernetes.

Prerequisites

- A Kubernetes cluster with support for volume resizing.
- The Cluster Operator is running.
- A Kafka cluster using persistent volumes created using a storage class that supports volume expansion.

Procedure

1. Edit the `Kafka` resource for your cluster.

Change the `size` property to increase the size of the persistent volume allocated to a Kafka cluster, a ZooKeeper cluster, or both.

- For Kafka clusters, update the `size` property under `spec.kafka.storage`.
- For ZooKeeper clusters, update the `size` property under `spec.zookeeper.storage`.

Kafka configuration to increase the volume size to 2000Gi

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

```
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: persistent-claim
      size: 2000Gi
      class: my-storage-class
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource:

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

Kubernetes increases the capacity of the selected persistent volumes in response to a request from the Cluster Operator. When the resizing is complete, the Cluster Operator restarts all pods that use the resized persistent volumes. This happens automatically.

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

```
kubectl get pv
```

Kafka broker pods with increased storage

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

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

Additional resources

- For more information about resizing persistent volumes in Kubernetes, see [Resizing Persistent Volumes using Kubernetes](#).

10.11.5. JBOD storage

JBOD storage allows you to configure your Kafka cluster to use multiple disks or volumes. This approach provides increased data storage capacity for Kafka nodes, and can lead to performance improvements. A JBOD configuration is defined by one or more volumes, each of which can be either **ephemeral** or **persistent**. The rules and constraints for JBOD volume declarations are the same as those for ephemeral and persistent storage. For example, you cannot decrease the size of a persistent storage volume after it has been provisioned, nor can you change the value of **sizeLimit**

when the type is **ephemeral**.

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

To use JBOD storage, you set the storage type configuration in the **Kafka** resource to **jbod**. If you are using node pools, you can also specify **jbod** in the storage configuration for nodes belonging to a specific node pool.

The **volumes** property allows you to describe the disks that make up your JBOD storage array or configuration.

Example JBOD storage configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    storage:
      type: jbod
      volumes:
        - id: 0
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
        - id: 1
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
# ...
```

The IDs cannot be changed once the JBOD volumes are created. You can add or remove volumes from the JBOD configuration.

PVC resource for JBOD storage

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

data-id-cluster-name-kafka-idx

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

Mount path of Kafka log directories

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

```
/var/lib/kafka/data-id/kafka-logidx
```

Where `id` is the ID of the volume used for storing data for Kafka broker pod `idx`. For example `/var/lib/kafka/data-0/kafka-log0`.

Configuring the storage volume used to store the KRaft metadata log

In KRaft mode, a copy of the Kafka cluster's metadata log is stored on every node, including brokers and controllers. Each node uses one of its data volumes for the KRaft metadata log. By default, the log is stored on the volume with the lowest ID. However, you can specify another volume using the `kraftMetadata` property.

For controller-only nodes, which don't handle data, storage is used only used for the metadata log. The metadata log is always stored only on one volume, so using JBOD storage with multiple volumes does not improve the performance or increase the available disk space.

Meanwhile, broker nodes or nodes combining broker and controller roles share the same volume for storing both the metadata log and partition replica data. This sharing optimizes disk utilization. They can also utilize JBOD storage with multiple volumes so that one of the volumes is shared by the metadata log and partition replica data and any additional volumes are used for partition replica data only.

Changing the volume that stores the metadata log triggers a rolling update of nodes in the cluster. This process involves deleting the old metadata log and creating a new one in the new location. If `kraftMetadata` isn't specified on any volume, adding a new volume with a lower ID also triggers an update and relocation of the metadata log.

NOTE JBOD storage in KRaft mode is considered early-access in Apache Kafka 3.7.x.

Example JBOD storage configuration using volume with ID 1 to store the KRaft metadata

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaNodePool
metadata:
  name: pool-a
  # ...
spec:
  storage:
    type: jbod
    volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 100Gi
        kraftMetadata: shared
        deleteClaim: false
    # ...
```

10.11.6. Adding volumes to JBOD storage

This procedure describes how to add volumes to a Kafka cluster configured to use JBOD storage. It cannot be applied to Kafka clusters configured to use any other storage type.

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

Prerequisites

- A Kubernetes cluster
- A running Cluster Operator
- A Kafka cluster with JBOD storage

Procedure

1. Edit the `spec.kafka.storage.volumes` property in the `Kafka` resource. Add the new volumes to the `volumes` array. For example, add the new volume with id `2`:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: jbod
      volumes:
        - id: 0
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
        - id: 1
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
        - id: 2
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource:

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

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

TIP

Cruise Control is an effective tool for reassigning partitions. To perform an intra-broker disk balance, you set `rebalanceDisk` to `true` under the `KafkaRebalance.spec`.

10.11.7. Removing volumes from JBOD storage

This procedure describes how to remove volumes from a Kafka cluster configured to use JBOD storage. It cannot be applied to Kafka clusters configured to use any other storage type. The JBOD storage always has to contain at least one volume.

IMPORTANT

To avoid data loss, you have to move all partitions before removing the volumes.

Prerequisites

- A Kubernetes cluster
- A running Cluster Operator
- A Kafka cluster with JBOD storage with two or more volumes

Procedure

1. Reassign all partitions from the disks which are you going to remove. Any data in partitions still assigned to the disks which are going to be removed might be lost.

TIP

You can use the `kafka-reassign-partitions.sh` tool to reassign the partitions.

2. Edit the `spec.kafka.storage.volumes` property in the `Kafka` resource. Remove one or more volumes from the `volumes` array. For example, remove the volumes with ids **1** and **2**:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: jbod
      volumes:
        - id: 0
          type: persistent-claim
          size: 100Gi
          deleteClaim: false
        # ...
  zookeeper:
```

```
# ...
```

3. Create or update the resource:

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

10.11.8. Tiered storage (early access)

Tiered storage introduces a flexible approach to managing Kafka data whereby log segments are moved to a separate storage system. For example, you can combine the use of block storage on brokers for frequently accessed data and offload older or less frequently accessed data from the block storage to more cost-effective, scalable remote storage solutions, such as Amazon S3, without compromising data accessibility and durability.

WARNING

Tiered storage is an early access Kafka feature, which is also available in Strimzi. Due to its [current limitations](#), it is not recommended for production environments.

Tiered storage requires an implementation of Kafka's `RemoteStorageManager` interface to handle communication between Kafka and the remote storage system, which is enabled through configuration of the `Kafka` resource. Strimzi uses Kafka's `TopicBasedRemoteLogMetadataManager` for Remote Log Metadata Management (RLMM) when custom tiered storage is enabled. The RLMM manages the metadata related to remote storage.

To use custom tiered storage, do the following:

- Include a tiered storage plugin for Kafka in the Strimzi image by building a custom container image. The plugin must provide the necessary functionality for a Kafka cluster managed by Strimzi to interact with the tiered storage solution.
- Configure Kafka for tiered storage using `tieredStorage` properties in the `Kafka` resource. Specify the class name and path for the custom `RemoteStorageManager` implementation, as well as any additional configuration.
- If required, specify RLMM-specific tiered storage configuration.

Example custom tiered storage configuration for Kafka

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    tieredStorage:
      type: custom ①
      remoteStorageManager: ②
        className: com.example.kafka.tiered.storage.s3.S3RemoteStorageManager
        classPath: /opt/kafka/plugins/tiered-storage-s3/*
```

```

config:
  storage.bucket.name: my-bucket ③
  # ...
config:
  rlmm.config.remote.log.metadata.topic.replication.factor: 1 ④
# ...

```

- ① The `type` must be set to `custom`.
- ② The configuration for the custom `RemoteStorageManager` implementation, including class name and path.
- ③ Configuration to pass to the custom `RemoteStorageManager` implementation, which Strimzi automatically prefixes with `rsm.config..`
- ④ Tiered storage configuration to pass to the RLMM, which requires an `rlmm.config.` prefix. For more information on tiered storage configuration, see the [Apache Kafka documentation](#).

10.12. Configuring CPU and memory resource limits and requests

By default, the Strimzi Cluster Operator does not specify CPU and memory resource requests and limits for its deployed operands. Ensuring an adequate allocation of resources is crucial for maintaining stability and achieving optimal performance in Kafka. The ideal resource allocation depends on your specific requirements and use cases.

It is recommended to configure CPU and memory resources for each container by [setting appropriate requests and limits](#).

10.13. Restrictions on Kubernetes labels

[Kubernetes labels](#) make it easier to organize, manage, and discover Kubernetes resources within your applications. The Cluster Operator is responsible for applying the following Kubernetes labels to the operands it deploys. These labels cannot be overridden through `template` configuration of Strimzi resources:

- `app.kubernetes.io/name`: Identifies the component type within Strimzi, such as `kafka`, `zookeeper`, and `cruise-control`.
- `app.kubernetes.io/instance`: Represents the name of the custom resource to which the operand belongs to. For instance, if a Kafka custom resource is named `my-cluster`, this label will bear that name on the associated pods.
- `app.kubernetes.io/part-of`: Similar to `app.kubernetes.io/instance`, but prefixed with `strimzi-`.
- `app.kubernetes.io/managed-by`: Defines the application responsible for managing the operand, such as `strimzi-cluster-operator` or `strimzi-user-operator`.

Example Kubernetes labels on a Kafka pod when deploying a Kafka custom resource named `my-cluster`

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

```

kind: Pod
metadata:
  name: my-cluster-kafka-0
  labels:
    app.kubernetes.io/instance: my-cluster
    app.kubernetes.io/managed-by: strimzi-cluster-operator
    app.kubernetes.io/name: kafka
    app.kubernetes.io/part-of: strimzi-my-cluster
spec:
  # ...

```

10.14. Configuring pod scheduling

To avoid performance degradation caused by resource conflicts between applications scheduled on the same Kubernetes node, you can schedule Kafka pods separately from critical workloads. This can be achieved by either selecting specific nodes or dedicating a set of nodes exclusively for Kafka.

10.14.1. Specifying affinity, tolerations, and topology spread constraints

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

- `Kafka.spec.kafka.template.pod`
- `Kafka.spec.zookeeper.template.pod`
- `Kafka.spec.entityOperator.template.pod`
- `KafkaConnect.spec.template.pod`
- `KafkaBridge.spec.template.pod`
- `KafkaMirrorMaker.spec.template.pod`
- `KafkaMirrorMaker2.spec.template.pod`

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

- Pod affinity and anti-affinity
- Node affinity

Additional resources

- [Kubernetes node and pod affinity documentation](#)
- [Kubernetes taints and tolerations](#)
- [Kubernetes Topology Spread Constraints](#)

Use pod anti-affinity to avoid critical applications sharing nodes

Use pod anti-affinity to ensure that critical applications are never scheduled on the same disk.

When running a Kafka cluster, it is recommended to use pod anti-affinity to ensure that the Kafka brokers do not share nodes with other workloads, such as databases.

Use node affinity to schedule workloads onto specific nodes

The Kubernetes cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while others 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.

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.

Use node affinity and tolerations for dedicated nodes

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

Cluster administrators can mark selected 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.

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.

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

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

Prerequisites

- A Kubernetes cluster
- A running Cluster Operator

Procedure

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

```

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

```

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

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

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:

```

```

    - labelSelector:
        matchExpressions:
            - key: strimzi.io/cluster
              operator: In
              values:
                  - CLUSTER-NAME
        topologyKey: "kubernetes.io/hostname"

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

    # ...

```

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

3. Create or update the resource.

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

10.14.3. Configuring pod anti-affinity in Kafka components

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

Prerequisites

- A 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/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
            topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...

```

2. Create or update the resource.

This can be done using `kubectl apply`:

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

10.14.4. Configuring node affinity in Kafka components

Prerequisites

- A Kubernetes cluster
- A running Cluster Operator

Procedure

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

This can be done using `kubectl label`:

```
kubectl label node NAME-OF-NODE node-type=fast-network
```

Alternatively, some of the existing labels might be reused.

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

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

```
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: node-type
                      operator: In
                      values:
                        - fast-network
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.

This can be done using `kubectl apply`:

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

10.14.5. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- A 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:

This can be done using `kubectl taint`:

```
kubectl taint node NAME-OF-NODE dedicated=Kafka:NoSchedule
```

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

This can be done using `kubectl label`:

```
kubectl label node NAME-OF-NODE dedicated=Kafka
```

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

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: dedicated
                      operator: In
                      values:
                        - Kafka
      # ...
  zookeeper:
    # ...
```

6. Create or update the resource.

This can be done using `kubectl apply`:

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

10.15. Configuring logging levels

Configure logging levels in the custom resources of Kafka components and Strimzi operators. You can specify the logging levels directly in the `spec.logging` property of the custom resource. Or you can define the logging properties in a ConfigMap that's referenced in the custom resource using the `configMapKeyRef` property.

The advantages of using a ConfigMap are that the logging properties are maintained in one place and are accessible to more than one resource. You can also reuse the ConfigMap for more than one resource. If you are using a ConfigMap to specify loggers for Strimzi Operators, you can also append the logging specification to add filters.

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

- **inline** when specifying logging levels directly
- **external** when referencing a ConfigMap

*Example **inline** logging configuration*

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

*Example **external** logging configuration*

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

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

10.15.1. Logging options for Kafka components and operators

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

Kafka component logging

- [Kafka logging](#)
- [ZooKeeper logging](#)
- [Kafka Connect and MirrorMaker 2 logging](#)
- [MirrorMaker logging](#)
- [Kafka Bridge logging](#)
- [Cruise Control logging](#)

Operator logging

- [Cluster Operator logging](#)
- [Topic Operator logging](#)
- [User Operator logging](#)

10.15.2. Creating a ConfigMap for logging

To use a ConfigMap to define logging properties, you create the ConfigMap and then reference it as part of the logging definition in the `spec` of a resource.

The ConfigMap must contain the appropriate logging configuration.

- `log4j.properties` for Kafka components, ZooKeeper, and the Kafka Bridge
- `log4j2.properties` for the Topic Operator and User Operator

The configuration must be placed under these properties.

In this procedure a ConfigMap defines a root logger for a Kafka resource.

Procedure

1. Create the ConfigMap.

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

ConfigMap example with a root logger definition for Kafka:

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

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

```
kubectl create configmap logging-configmap --from-file=log4j.properties
```

The properties file defines the logging configuration:

```
# Define the logger
kafka.root.logger.level="INFO"
# ...
```

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

```
# ...
logging:
  type: external
  valueFrom:
```

```
configMapKeyRef:  
  name: logging-configmap  
  key: log4j.properties  
# ...
```

3. Create or update the resource.

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

10.15.3. Configuring Cluster Operator logging

Cluster Operator logging is configured through a [ConfigMap](#) named `strimzi-cluster-operator`. A [ConfigMap](#) containing logging configuration is created when installing the Cluster Operator. This [ConfigMap](#) is described in the file `install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml`. You configure Cluster Operator logging by changing the `data.log4j2.properties` values in this [ConfigMap](#).

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

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

Alternatively, edit the [ConfigMap](#) directly:

```
kubectl edit configmap strimzi-cluster-operator
```

With this [ConfigMap](#), you can control various aspects of logging, including the root logger level, log output format, and log levels for different components. The `monitorInterval` setting, determines how often the logging configuration is reloaded. You can also control the logging levels for the Kafka [AdminClient](#), ZooKeeper [ZKTrustManager](#), Netty, and the OkHttp client. Netty is a framework used in Strimzi for network communication, and OkHttp is a library used for making HTTP requests.

If the [ConfigMap](#) is missing when the Cluster Operator is deployed, the default logging values are used.

If the [ConfigMap](#) is accidentally deleted after the Cluster Operator is deployed, the most recently loaded logging configuration is used. Create a new [ConfigMap](#) to load a new logging configuration.

NOTE Do not remove the `monitorInterval` option from the [ConfigMap](#).

10.15.4. Adding logging filters to Strimzi operators

If you are using a [ConfigMap](#) to configure the (log4j2) logging levels for Strimzi operators, you can also define logging filters to limit what's returned in the log.

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

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

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

Basic logging filter configuration

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

- ① The `MarkerFilter` type compares a specified marker for filtering.
- ② The `onMatch` property accepts the log if the marker matches.
- ③ The `onMismatch` property rejects the log if the marker does not match.
- ④ The marker used for filtering is in the format `KIND(NAMESPACE/NAME-OF-RESOURCE)`.

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

Multiple logging filter configuration

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

Adding filters to the Cluster Operator

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

Procedure

1. Update the `050-ConfigMap-strimzi-cluster-operator.yaml` file to add the filter properties to the ConfigMap.

In this example, the filter properties return logs only for the `my-kafka-cluster` Kafka cluster:

```
kind: ConfigMap
```

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

Alternatively, edit the [ConfigMap](#) directly:

```
kubectl edit configmap strimzi-cluster-operator
```

2. If you updated the YAML file instead of editing the [ConfigMap](#) directly, apply the changes by deploying the ConfigMap:

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

Adding filters to the Topic Operator or User Operator

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

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

Procedure

1. Create the ConfigMap.

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

In this example, the filter properties return logs only for the [my-topic](#) topic:

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

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

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

The properties file defines the logging configuration:

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

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

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

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

3. Apply the changes by deploying the Cluster Operator:

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

Additional resources

- [Configuring Kafka](#)
- [Cluster Operator logging](#)
- [Topic Operator logging](#)
- [User Operator logging](#)

10.15.5. Lock acquisition warnings for cluster operations

The Cluster Operator ensures that only one operation runs at a time for each cluster by using locks. If another operation attempts to start while a lock is held, it waits until the current operation completes.

Operations such as cluster creation, rolling updates, scaling down, and scaling up are managed by the Cluster Operator.

If acquiring a lock takes longer than the configured timeout (`STRIMZI_OPERATION_TIMEOUT_MS`), a DEBUG message is logged:

Example DEBUG message for lock acquisition

```
DEBUG AbstractOperator:406 - Reconciliation #55(timer) Kafka(myproject/my-cluster):  
Failed to acquire lock lock::myproject::Kafka::my-cluster within 10000ms.
```

Timed-out operations are retried during the next periodic reconciliation in intervals defined by `STRIMZI_FULL_RECONCILIATION_INTERVAL_MS` (by default 120 seconds).

If an INFO message continues to appear with the same reconciliation number, it might indicate a lock release error:

Example INFO message for reconciliation

```
INFO AbstractOperator:399 - Reconciliation #1(watch) Kafka(myproject/my-cluster):  
Reconciliation is in progress
```

Restarting the Cluster Operator can resolve such issues.

10.16. Using ConfigMaps to add configuration

Add specific configuration to your Strimzi deployment using [ConfigMap](#) resources. ConfigMaps use key-value pairs to store non-confidential data. Configuration data added to ConfigMaps is maintained in one place and can be reused amongst components.

ConfigMaps can only store the following types of configuration data:

- Logging configuration
- Metrics configuration
- External configuration for Kafka Connect connectors

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

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

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

logging configuration.

Example ConfigMap for logging

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

To use a ConfigMap for metrics configuration, you add a reference to the [metricsConfig](#) configuration of the component in the same way.

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

For example, you can use the [configMapKeyRef](#) property to pass configuration data from a ConfigMap as an environment variable.

Example ConfigMap providing environment variable values

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

If you are using ConfigMaps that are managed externally, use configuration providers to load the data in the ConfigMaps.

10.16.1. Naming custom ConfigMaps

Strimzi [creates its own ConfigMaps and other resources](#) when it is deployed to Kubernetes. The ConfigMaps contain data necessary for running components. The ConfigMaps created by Strimzi must not be edited.

Make sure that any custom ConfigMaps you create do not have the same name as these default

ConfigMaps. If they have the same name, they will be overwritten. For example, if your ConfigMap has the same name as the ConfigMap for the Kafka cluster, it will be overwritten when there is an update to the Kafka cluster.

Additional resources

- [List of Kafka cluster resources](#) (including ConfigMaps)
- [Logging configuration](#)
- [metricsConfig](#)
- [ExternalConfiguration schema reference](#)
- [Loading configuration values from external sources](#)

10.17. Loading configuration values from external sources

Use configuration providers to load configuration data from external sources. The providers operate independently of Strimzi. You can use them to load configuration data for all Kafka components, including producers and consumers. You reference the external source in the configuration of the component and provide access rights. The provider loads data without needing to restart the Kafka component or extracting files, even when referencing a new external source. For example, use providers to supply the credentials for the Kafka Connect connector configuration. The configuration must include any access rights to the external source.

10.17.1. Enabling configuration providers

You can enable one or more configuration providers using the `config.providers` properties in the `spec` configuration of a component.

Example configuration to enable a configuration provider

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
    config.providers: env
    config.providers.env.class:
      org.apache.kafka.common.config.provider.EnvVarConfigProvider
      # ...
```

KubernetesSecretConfigProvider

Loads configuration data from Kubernetes secrets. You specify the name of the secret and the

key within the secret where the configuration data is stored. This provider is useful for storing sensitive configuration data like passwords or other user credentials.

KubernetesConfigMapConfigProvider

Loads configuration data from Kubernetes config maps. You specify the name of the config map and the key within the config map where the configuration data is stored. This provider is useful for storing non-sensitive configuration data.

EnvVarConfigProvider

Loads configuration data from environment variables. You specify the name of the environment variable where the configuration data is stored. This provider is useful for configuring applications running in containers, for example, to load certificates or JAAS configuration from environment variables mapped from secrets.

FileConfigProvider

Loads configuration data from a file. You specify the path to the file where the configuration data is stored. This provider is useful for loading configuration data from files that are mounted into containers.

DirectoryConfigProvider

Loads configuration data from files within a directory. You specify the path to the directory where the configuration files are stored. This provider is useful for loading multiple configuration files and for organizing configuration data into separate files.

To use [KubernetesSecretConfigProvider](#) and [KubernetesConfigMapConfigProvider](#), which are part of the Kubernetes Configuration Provider plugin, you must set up access rights to the namespace that contains the configuration file.

You can use the other providers without setting up access rights. You can supply connector configuration for Kafka Connect or MirrorMaker 2 in this way by doing the following:

- Mount config maps or secrets into the Kafka Connect pod as environment variables or volumes
- Enable [EnvVarConfigProvider](#), [FileConfigProvider](#), or [DirectoryConfigProvider](#) in the Kafka Connect or MirrorMaker 2 configuration
- Pass connector configuration using the [externalConfiguration](#) property in the [spec](#) of the [KafkaConnect](#) or [KafkaMirrorMaker2](#) resource

Using providers help prevent the passing of restricted information through the Kafka Connect REST interface. You can use this approach in the following scenarios:

- Mounting environment variables with the values a connector uses to connect and communicate with a data source
- Mounting a properties file with values that are used to configure Kafka Connect connectors
- Mounting files in a directory that contains values for the TLS truststore and keystore used by a connector

NOTE

A restart is required when using a new [Secret](#) or [ConfigMap](#) for a connector, which

can disrupt other connectors.

Additional resources

[ExternalConfiguration schema reference](#)

10.17.2. Loading configuration values from secrets or config maps

Use the [KubernetesSecretConfigProvider](#) to provide configuration properties from a secret or the [KubernetesConfigMapConfigProvider](#) to provide configuration properties from a config map.

In this procedure, a config map provides configuration properties for a connector. The properties are specified as key values of the config map. The config map is mounted into the Kafka Connect pod as a volume.

Prerequisites

- A Kafka cluster is running.
- The Cluster Operator is running.
- You have a config map containing the connector configuration.

Example config map with connector properties

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

Procedure

1. Configure the [KafkaConnect](#) resource.
 - Enable the [KubernetesConfigMapConfigProvider](#)

The specification shown here can support loading values from config maps and secrets.

Example Kafka Connect configuration to use config maps and secrets

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

```

    config.providers.configmaps.class:
      io.strimzi.kafka.KubernetesConfigMapConfigProvider ②
      config.providers.secrets.class: io.strimzi.kafka.KubernetesSecretConfigProvider
    ③
    # ...

```

- ① The alias for the configuration provider is used to define other configuration parameters. The provider parameters use the alias from `config.providers`, taking the form `config.providers.${alias}.class`.
- ② `KubernetesConfigMapConfigProvider` provides values from config maps.
- ③ `KubernetesSecretConfigProvider` provides values from secrets.

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

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

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

Example role to access values from a config map

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: connector-configuration-role
rules:
- apiGroups: []
  resources: ["configmaps"]
  resourceNames: ["my-connector-configuration"]
  verbs: ["get"]
# ...

```

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

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

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

```

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

```

```
apiGroup: rbac.authorization.k8s.io
# ...
```

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

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

5. Reference the config map in the connector configuration.

Example connector configuration referencing the config map

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

The placeholder structure is `configmaps:<path_and_file_name>:<property>`. `KubernetesConfigMapConfigProvider` reads and extracts the `option1` property value from the external config map.

10.17.3. Loading configuration values from environment variables

Use the `EnvVarConfigProvider` to provide configuration properties as environment variables. Environment variables can contain values from config maps or secrets.

In this procedure, environment variables provide configuration properties for a connector to communicate with Amazon AWS. The connector must be able to read the `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. The values of the environment variables are derived from a secret mounted into the Kafka Connect pod.

NOTE

The names of user-defined environment variables cannot start with `KAFKA_` or `STRIMZI_`.

Prerequisites

- A Kafka cluster is running.
- The Cluster Operator is running.
- You have a secret containing the connector configuration.

Example secret with values for environment variables

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-creds
type: Opaque
data:
  awsAccessKey: QUtJQVhYWfhYWFhYWFhYWFg=
  awsSecretAccessKey: Ylhsd1lYTnpkMj15WkE=
```

Procedure

1. Configure the **KafkaConnect** resource.
 - Enable the **EnvVarConfigProvider**
 - Specify the environment variables using the **externalConfiguration** property.

Example Kafka Connect configuration to use external environment variables

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
    config.providers: env ①
    config.providers.env.class:
      org.apache.kafka.common.config.provider.EnvVarConfigProvider ②
      # ...
    externalConfiguration:
      env:
        - name: AWS_ACCESS_KEY_ID ③
          valueFrom:
            secretKeyRef:
              name: aws-creds ④
              key: awsAccessKey ⑤
        - name: AWS_SECRET_ACCESS_KEY
          valueFrom:
            secretKeyRef:
              name: aws-creds
              key: awsSecretAccessKey
      # ...
```

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

`config.providers.${alias}.class.`

- ② `EnvVarConfigProvider` provides values from environment variables.
- ③ The environment variable takes a value from the secret.
- ④ The name of the secret containing the environment variable.
- ⑤ The name of the key stored in the secret.

NOTE The `secretKeyRef` property references keys in a secret. If you are using a config map instead of a secret, use the `configMapKeyRef` property.

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

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

3. Reference the environment variable in the connector configuration.

Example connector configuration referencing the environment variable

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

The placeholder structure is `env:<environment_variable_name>`. `EnvVarConfigProvider` reads and extracts the environment variable values from the mounted secret.

10.17.4. Loading configuration values from a file within a directory

Use the `FileConfigProvider` to provide configuration properties from a file within a directory. Files can be config maps or secrets.

In this procedure, a file provides configuration properties for a connector. A database name and password are specified as properties of a secret. The secret is mounted to the Kafka Connect pod as a volume. Volumes are mounted on the path `/opt/kafka/external-configuration/<volume-name>`.

Prerequisites

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

- You have a secret containing the connector configuration.

Example secret with database properties

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |- ①
    dbUsername: my-username ②
    dbPassword: my-password
```

① The connector configuration in properties file format.

② Database username and password properties used in the configuration.

Procedure

1. Configure the [KafkaConnect](#) resource.

- Enable the [FileConfigProvider](#)
- Specify the file using the [externalConfiguration](#) property.

Example Kafka Connect configuration to use an external property file

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: file ①
    config.providers.file.class:
      org.apache.kafka.common.config.provider.FileConfigProvider ②
      #...
    externalConfiguration:
      volumes:
        - name: connector-config ③
          secret:
            secretName: mysecret ④
```

① The alias for the configuration provider is used to define other configuration parameters.

② [FileConfigProvider](#) provides values from properties files. The parameter uses the alias from [config.providers](#), taking the form [config.providers.\\${alias}.class](#).

③ The name of the volume containing the secret.

④ The name of the secret.

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

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

3. Reference the file properties in the connector configuration as placeholders.

Example connector configuration referencing the file

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    database.hostname: 192.168.99.1
    database.port: "3306"
    database.user: "${file:/opt/kafka/external-configuration/connector-
config/mysecret:dbUsername}"
    database.password: "${file:/opt/kafka/external-configuration/connector-
config/mysecret:dbPassword}"
    database.server.id: "184054"
    #...
```

The placeholder structure is `file:<path_and_file_name>:<property>`. `FileConfigProvider` reads and extracts the database username and password property values from the mounted secret.

10.17.5. Loading configuration values from multiple files within a directory

Use the `DirectoryConfigProvider` to provide configuration properties from multiple files within a directory. Files can be config maps or secrets.

In this procedure, a secret provides the TLS keystore and truststore user credentials for a connector. The credentials are in separate files. The secrets are mounted into the Kafka Connect pod as volumes. Volumes are mounted on the path `/opt/kafka/external-configuration/<volume-name>`.

Prerequisites

- A Kafka cluster is running.
- The Cluster Operator is running.
- You have a secret containing the user credentials.

Example secret with user credentials

```
apiVersion: v1
```

```

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

```

The `my-user` secret provides the keystore credentials (`user.crt` and `user.key`) for the connector.

The `<cluster_name>-cluster-ca-cert` secret generated when deploying the Kafka cluster provides the cluster CA certificate as truststore credentials (`ca.crt`).

Procedure

1. Configure the `KafkaConnect` resource.
 - Enable the `DirectoryConfigProvider`
 - Specify the files using the `externalConfiguration` property.

Example Kafka Connect configuration to use external property files

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: directory ①
    config.providers.directory.class:
      org.apache.kafka.common.config.provider.DirectoryConfigProvider ②
      #...
    externalConfiguration:
      volumes: ③
        - name: cluster-ca ④
          secret:
            secretName: my-cluster-cluster-ca-cert ⑤
        - name: my-user
          secret:
            secretName: my-user ⑥

```

① The alias for the configuration provider is used to define other configuration parameters.

- ② `DirectoryConfigProvider` provides values from files in a directory. The parameter uses the alias from `config.providers`, taking the form `config.providers.${alias}.class`.
 - ③ The names of the volumes containing the secrets.
 - ④ The name of the secret for the cluster CA certificate to supply truststore configuration.
 - ⑤ The name of the secret for the user to supply keystore configuration.
2. Create or update the resource to enable the provider.

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

3. Reference the file properties in the connector configuration as placeholders.

Example connector configuration referencing the files

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    # ...
    database.history.producer.security.protocol: SSL
    database.history.producer.ssl.truststore.type: PEM
    database.history.producer.ssl.truststore.certificates:
      "${directory:/opt/kafka/external-configuration/cluster-ca:ca.crt}"
    database.history.producer.ssl.keystore.type: PEM
    database.history.producer.ssl.keystore.certificate.chain:
      "${directory:/opt/kafka/external-configuration/my-user:user.crt}"
    database.history.producer.ssl.keystore.key: "${directory:/opt/kafka/external-configuration/my-user:user.key}"
    #...
```

The placeholder structure is `directory:<path>:<file_name>`. `DirectoryConfigProvider` reads and extracts the credentials from the mounted secrets.

10.18. Customizing Kubernetes resources

A Strimzi deployment creates Kubernetes resources, such as `Deployment`, `Pod`, and `Service` resources. These resources are managed by Strimzi operators. Only the operator that is responsible for managing a particular Kubernetes resource can change that resource. If you try to manually change an operator-managed Kubernetes resource, the operator will revert your changes back.

Changing an operator-managed Kubernetes resource can be useful if you want to perform certain

tasks, such as the following:

- Adding custom labels or annotations that control how **Pods** are treated by Istio or other services
- Managing how **Loadbalancer**-type Services are created by the cluster

To make the changes to a Kubernetes resource, you can use the **template** property within the **spec** section of various Strimzi custom resources.

Here is a list of the custom resources where you can apply the changes:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator**
- **Kafka.spec.kafkaExporter**
- **Kafka.spec.cruiseControl**
- **KafkaNodePool.spec**
- **KafkaConnect.spec**
- **KafkaMirrorMaker.spec**
- **KafkaMirrorMaker2.spec**
- **KafkaBridge.spec**
- **KafkaUser.spec**

For more information about these properties, see the [Strimzi Custom Resource API Reference](#).

The Strimzi Custom Resource API Reference provides more details about the customizable fields.

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

Example template customization

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

10.18.1. Customizing the image pull policy

Strimzi allows you to customize the image pull policy for containers in all pods deployed by the Cluster Operator. The image pull policy is configured using the environment variable `STRIMZI_IMAGE_PULL_POLICY` in the Cluster Operator deployment. The `STRIMZI_IMAGE_PULL_POLICY` environment variable can be set to three different values:

Always

Container images are pulled from the registry every time the pod is started or restarted.

IfNotPresent

Container images are pulled from the registry only when they were not pulled before.

Never

Container images are never pulled from the registry.

Currently, the image pull policy can only be customized for all Kafka, Kafka Connect, and Kafka MirrorMaker clusters at once. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka MirrorMaker clusters.

Additional resources

- [Disruptions](#).

10.18.2. Applying a termination grace period

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

Specify the time using the `terminationGracePeriodSeconds` property. Add the property to the `template.pod` configuration of the [Kafka](#) custom resource.

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

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

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

Example termination grace period configuration

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

```
spec:  
  kafka:  
    # ...  
  template:  
    pod:  
      terminationGracePeriodSeconds: 120  
      # ...  
    # ...
```

Chapter 11. Using the Topic Operator to manage Kafka topics

The [KafkaTopic](#) resource configures topics, including partition and replication factor settings. When you create, modify, or delete a topic using [KafkaTopic](#), the Topic Operator ensures that these changes are reflected in the Kafka cluster.

For more information on the [KafkaTopic](#) resource, see the [KafkaTopic schema reference](#).

11.1. Topic management

The [KafkaTopic](#) resource is responsible for managing a single topic within a Kafka cluster.

The Topic Operator operates as follows:

- When a [KafkaTopic](#) is created, deleted, or changed, the Topic Operator performs the corresponding operation on the Kafka topic.

If a topic is created, deleted, or modified directly within the Kafka cluster, without the presence of a corresponding [KafkaTopic](#) resource, the Topic Operator does not manage that topic. The Topic Operator will only manage Kafka topics associated with [KafkaTopic](#) resources and does not interfere with topics managed independently within the Kafka cluster. If a [KafkaTopic](#) does exist for a Kafka topic, any configuration changes made outside the resource are reverted.

The Topic Operator can detect cases where multiple [KafkaTopic](#) resources are attempting to manage a Kafka topic using the same `.spec.topicName`. Only the oldest resource is reconciled, while the other resources fail with a resource conflict error.

11.2. Topic naming conventions

A [KafkaTopic](#) resource includes a name for the topic and a label that identifies the name of the Kafka cluster it belongs to.

Label identifying a Kafka cluster for topic handling

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  topicName: topic-name-1
```

The label provides the cluster name of the [Kafka](#) resource. The Topic Operator uses the label as a mechanism for determining which [KafkaTopic](#) resources to manage. If the label does not match the Kafka cluster, the Topic Operator cannot see the [KafkaTopic](#), and the topic is not created.

Kafka and Kubernetes have their own naming validation rules, and a Kafka topic name might not be a valid resource name in Kubernetes. If possible, try and stick to a naming convention that works for both.

Consider the following guidelines:

- Use topic names that reflect the nature of the topic
- Be concise and keep the name under 63 characters
- Use all lower case and hyphens
- Avoid special characters, spaces or symbols

The `KafkaTopic` resource allows you to specify the Kafka topic name using the `metadata.name` field. However, if the desired Kafka topic name is not a valid Kubernetes resource name, you can use the `spec.topicName` property to specify the actual name. The `spec.topicName` field is optional, and when it's absent, the Kafka topic name defaults to the `metadata.name` of the topic. When a topic is created, the topic name cannot be changed later.

Example of supplying a valid Kafka topic name

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic-1 ①
spec:
  topicName: My.Topic.1 ②
  # ...
```

① A valid topic name that works in Kubernetes.

② A Kafka topic name that uses upper case and periods, which are invalid in Kubernetes.

If more than one `KafkaTopic` resource refers to the same Kafka topic, the resource that was created first is considered to be the one managing the topic. The status of the newer resources is updated to indicate a conflict, and their `Ready` status is changed to `False`.

A Kafka client application, such as Kafka Streams, can automatically create topics with invalid Kubernetes resource names. If you want to manage these topics, you must create `KafkaTopic` resources with a different `.metadata.name`, as shown in the previous example.

NOTE

For more information on the requirements for identifiers and names in a cluster, refer to the Kubernetes documentation [Object Names and IDs](#).

11.3. Handling changes to topics

Configuration changes only go in one direction: from the `KafkaTopic` resource to the Kafka topic. Any changes to a Kafka topic managed outside the `KafkaTopic` resource are reverted.

11.3.1. Downgrading to a Strimzi version that uses internal topics to store topic metadata

If you are reverting back to a version of Strimzi earlier than 0.41, which uses internal topics for the storage of topic metadata, you still downgrade your Cluster Operator to the previous version, then downgrade Kafka brokers and client applications to the previous Kafka version as standard.

11.3.2. Downgrading to a Strimzi version that uses ZooKeeper to store topic metadata

If you are reverting back to a version of Strimzi earlier than 0.22, which uses ZooKeeper for the storage of topic metadata, you still downgrade your Cluster Operator to the previous version, then downgrade Kafka brokers and client applications to the previous Kafka version as standard.

However, you must also delete the topics that were created for the topic store using a `kafka-topics` command, specifying the bootstrap address of the Kafka cluster. For example:

```
kubectl run kafka-admin -ti --image=quay.io/stimzi/kafka:0.42.0-kafka-3.7.1 --rm=true  
--restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic  
__strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-  
topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

The command must correspond to the type of listener and authentication used to access the Kafka cluster.

The Topic Operator will reconstruct the ZooKeeper topic metadata from the state of the topics in Kafka.

11.3.3. Automatic creation of topics

Applications can trigger the automatic creation of topics in the Kafka cluster. By default, the Kafka broker configuration `auto.create.topics.enable` is set to `true`, allowing the broker to create topics automatically when an application attempts to produce or consume from a non-existing topic. Applications might also use the Kafka `AdminClient` to automatically create topics. When an application is deployed along with its `KafkaTopic` resources, it is possible that automatic topic creation in the cluster happens before the Topic Operator can react to the `KafkaTopic`.

The topics created for an application deployment are initially created with default topic configuration. If the Topic Operator attempts to reconfigure the topics based on `KafkaTopic` resource specifications included with the application deployment, the operation might fail because the required change to the configuration is not allowed. For example, if the change means lowering the number of topic partitions. For this reason, it is recommended to disable `auto.create.topics.enable` in the Kafka cluster configuration.

11.4. Configuring Kafka topics

Use the properties of the `KafkaTopic` resource to configure Kafka topics. Changes made to topic configuration in the `KafkaTopic` are propagated to Kafka.

You can use `kubectl apply` to create or modify topics, and `kubectl delete` to delete existing topics.

For example:

- `kubectl apply -f <topic_config_file>`
- `kubectl delete KafkaTopic <topic_name>`

To be able to delete topics, `delete.topic.enable` must be set to `true` (default) in the `spec.kafka.config` of the Kafka resource.

This procedure shows how to create a topic with 10 partitions and 2 replicas.

Before you begin

The KafkaTopic resource does not allow the following changes:

- Renaming the topic defined in `spec.topicName`. A mismatch between `spec.topicName` and `status.topicName` will be detected.
- Decreasing the number of partitions using `spec.partitions` (not supported by Kafka).
- Modifying the number of replicas specified in `spec.replicas`.

WARNING

Increasing `spec.partitions` for topics with keys will alter the partitioning of records, which can cause issues, especially when the topic uses semantic partitioning.

Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.
- A running Topic Operator (typically deployed with the Entity Operator).
- For deleting a topic, `delete.topic.enable=true` (default) in the `spec.kafka.config` of the `Kafka` resource.

Procedure

1. Configure the `KafkaTopic` resource.

Example Kafka topic configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
```

TIP

When modifying a topic, you can get the current version of the resource using

```
kubectl get kafkaTopic my-topic-1 -o yaml
```

2. Create the **KafkaTopic** resource in Kubernetes.

```
kubectl apply -f <topic_config_file>
```

3. Wait for the ready status of the topic to change to **True**:

```
kubectl get kafkaTopics -o wide -w -n <namespace>
```

Kafka topic status

NAME	CLUSTER	PARTITIONS	REPLICATION FACTOR	READY
my-topic-1	my-cluster	10	3	True
my-topic-2	my-cluster	10	3	
my-topic-3	my-cluster	10	3	True

Topic creation is successful when the **READY** output shows **True**.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or from the Topic Operator logs.

Status messages provide details on the reason for the current status.

```
oc get kafkaTopics my-topic-2 -o yaml
```

*Details on a topic with a **NotReady** status*

```
# ...
status:
  conditions:
    - lastTransitionTime: "2022-06-13T10:14:43.351550Z"
      message: Number of partitions cannot be decreased
      reason: PartitionDecreaseException
      status: "True"
      type: NotReady
```

In this example, the reason the topic is not ready is because the original number of partitions was reduced in the **KafkaTopic** configuration. Kafka does not support this.

After resetting the topic configuration, the status shows the topic is ready.

```
kubectl get kafkaTopics my-topic-2 -o wide -w -n <namespace>
```

Status update of the topic

NAME	CLUSTER	PARTITIONS	REPLICATION FACTOR	READY
my-topic-2	my-cluster	10	3	True

Fetching the details shows no messages

```
kubectl get kafkatopics my-topic-2 -o yaml
```

Details on a topic with a **READY** status

```
# ...
status:
  conditions:
  - lastTransitionTime: '2022-06-13T10:15:03.761084Z'
    status: 'True'
    type: Ready
```

11.5. Configuring topics for replication and number of partitions

The recommended configuration for topics managed by the Topic Operator is a topic replication factor of 3, and a minimum of 2 in-sync replicas.

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10 ①
  replicas: 3 ②
  config:
    min.insync.replicas: 2 ③
#...
```

① The number of partitions for the topic.

② The number of replica topic partitions. Changing the number of replicas in the topic configuration requires a deployment of Cruise Control. For more information, see [Using Cruise Control to modify topic replication factor](#).

③ The minimum number of replica partitions that a message must be successfully written to, or an exception is raised.

NOTE In-sync replicas are used in conjunction with the `acks` configuration for producer

applications. The `acks` configuration determines the number of follower partitions a message must be replicated to before the message is acknowledged as successfully received. Replicas need to be reassigned when adding or removing brokers (see [Scaling clusters by adding or removing brokers](#)).

Additional resources

- [Downgrading Strimzi](#)
- [Partition reassignment tool overview](#)
- [Using Cruise Control for cluster rebalancing](#)

11.6. Managing KafkaTopic resources without impacting Kafka topics

This procedure describes how to convert Kafka topics that are currently managed through the `KafkaTopic` resource into non-managed topics. This capability can be useful in various scenarios. For instance, you might want to update the `metadata.name` of a `KafkaTopic` resource. You can only do that by deleting the original `KafkaTopic` resource and recreating a new one.

By annotating a `KafkaTopic` resource with `strimzi.io/managed=false`, you indicate that the Topic Operator should no longer manage that particular topic. This allows you to retain the Kafka topic while making changes to the resource's configuration or other administrative tasks.

Prerequisites

- [The Cluster Operator must be deployed.](#)

Procedure

1. Annotate the `KafkaTopic` resource in Kubernetes, setting `strimzi.io/managed` to `false`:

```
kubectl annotate kafka my-topic-1 strimzi.io/managed="false"
```

Specify the `metadata.name` of the topic in your `KafkaTopic` resource, which is `my-topic-1` in this example.

2. Check the status of the `KafkaTopic` resource to make sure the request was successful:

```
oc get kafkatopics my-topic-1 -o yaml
```

Example topic with a Ready status

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  generation: 124
  name: my-topic-1
  finalizer:
```

```

        strimzi.io/topic-operator
    labels:
        strimzi.io/cluster: my-cluster
    spec:
        partitions: 10
        replicas: 2

    # ...
    status:
        observedGeneration: 124 ①
        topicName: my-topic-1
        conditions:
            - type: Ready
              status: True
        lastTransitionTime: 20230301T103000Z

```

① Successful reconciliation of the resource means the topic is no longer managed.

The value of `metadata.generation` (the current version of the deployment) must `match` `status.observedGeneration` (the latest reconciliation of the resource).

3. You can now make changes to the `KafkaTopic` resource without it affecting the Kafka topic it was managing.

For example, to change the `metadata.name`, do as follows:

- a. Delete the original `KafkaTopic` resource:

```
kubectl delete kafkatopic <kafka_topic_name>
```

- b. Recreate the `KafkaTopic` resource with a different `metadata.name`, but use `spec.topicName` to refer to the same topic that was managed by the original

4. If you haven't deleted the original `KafkaTopic` resource, and you wish to resume management of the Kafka topic again, set the `strimzi.io/managed` annotation to `true` or remove the annotation.

11.7. Enabling topic management for existing Kafka topics

This procedure describes how to enable topic management for topics that are not currently managed through the `KafkaTopic` resource. You do this by creating a matching `KafkaTopic` resource.

Prerequisites

- The Cluster Operator must be deployed.

Procedure

1. Create a `KafkaTopic` resource with a `metadata.name` that is the same as the Kafka topic.

Or use `spec.topicName` if the name of the topic in Kafka would not be a legal Kubernetes

resource name.

Example Kafka topic configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
```

In this example, the Kafka topic is named **my-topic-1**.

The Topic Operator checks whether the topic is managed by another **KafkaTopic** resource. If it is, the older resource takes precedence and a resource conflict error is returned in the status of the new resource.

2. Apply the **KafkaTopic** resource:

```
kubectl apply -f <topic_configuration_file>
```

3. Wait for the operator to update the topic in Kafka.

The operator updates the Kafka topic with the **spec** of the **KafkaTopic** that has the same name.

4. Check the status of the **KafkaTopic** resource to make sure the request was successful:

```
oc get kafkatopics my-topic-1 -o yaml
```

Example topic with a Ready status

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  generation: 1
  name: my-topic-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
# ...
status:
  observedGeneration: 1 ①
  topicName: my-topic-1
```

```
conditions:
- type: Ready
  status: True
  lastTransitionTime: 20230301T103000Z
```

① Successful reconciliation of the resource means the topic is now managed.

The value of `metadata.generation` (the current version of the deployment) must `match` `status.observedGeneration` (the latest reconciliation of the resource).

11.8. Deleting managed topics

The Topic Operator supports the deletion of topics managed through the `KafkaTopic` resource with or without Kubernetes finalizers. This is determined by the `STRIMZI_USE_FINALIZERS` Topic Operator environment variable. By default, this is set to `true`, though it can be set to `false` in the Topic Operator `env` configuration if you do not want the Topic Operator to add finalizers.

Finalizers ensure orderly and controlled deletion of `KafkaTopic` resources. A finalizer for the Topic Operator is added to the metadata of the `KafkaTopic` resource:

Finalizer to control topic deletion

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  generation: 1
  name: my-topic-1
  finalizers:
    - strimzi.io/topic-operator
  labels:
    strimzi.io/cluster: my-cluster
```

In this example, the finalizer is added for topic `my-topic-1`. The finalizer prevents the topic from being fully deleted until the finalization process is complete. If you then delete the topic using `kubectl delete kafkatopic my-topic-1`, a timestamp is added to the metadata:

Finalizer timestamp on deletion

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  generation: 1
  name: my-topic-1
  finalizers:
    - strimzi.io/topic-operator
  labels:
    strimzi.io/cluster: my-cluster
  deletionTimestamp: 20230301T000000.000
```

The resource is still present. If the deletion fails, it is shown in the status of the resource.

When the finalization tasks are successfully executed, the finalizer is removed from the metadata, and the resource is fully deleted.

Finalizers also serve to prevent related resources from being deleted. If the Topic Operator is not running, it won't be able to remove its finalizer from the `metadata.finalizers`. And any attempt to directly delete the `KafkaTopic` resources or the namespace will fail or timeout, leaving the namespace in a stuck terminating state. If this happens, you can bypass the finalization process by [removing the finalizers on topics](#).

11.9. Removing finalizers on topics

If the Topic Operator is not running, and you want to bypass the finalization process when deleting managed topics, you must remove the finalizers. You can do this manually by editing the resources directly or by using a command.

To remove finalizers on all topics, use the following command:

Removing finalizers on topics

```
kubectl get kt -o=json | jq '.items[].metadata.finalizers = null' | kubectl apply -f -
```

The command uses the [jq command line JSON parser tool](#) to modify the `KafkaTopic` (`kt`) resources by setting the finalizers to `null`. You can also use the command for a specific topic:

Removing a finalizer on a specific topic

```
kubectl get kt <topic_name> -o=json | jq '.metadata.finalizers = null' | kubectl apply -f -
```

After running the command, you can go ahead and delete the topics. Alternatively, if the topics were already being deleted but were blocked due to outstanding finalizers then their deletion should complete.

WARNING

Be careful when removing finalizers, as any cleanup operations associated with the finalization process are not performed if the Topic Operator is not running. For example, if you remove the finalizer from a `KafkaTopic` resource and subsequently delete the resource, the related Kafka topic won't be deleted.

11.10. Considerations when disabling topic deletion

When the `delete.topic.enable` configuration in Kafka is set to `false`, topics cannot be deleted. This might be required in certain scenarios, but it introduces a consideration when using the Topic Operator.

As topics cannot be deleted, finalizers added to the metadata of a `KafkaTopic` resource to control topic deletion are never removed by the Topic Operator (though they can be [removed manually](#)). Similarly, any Custom Resource Definitions (CRDs) or namespaces associated with topics cannot be

deleted.

Before configuring `delete.topic.enable=false`, assess these implications to ensure it aligns with your specific requirements.

NOTE To avoid using finalizers, you can set the `STRIMZI_USE_FINALIZERS` Topic Operator environment variable to `false`.

11.11. Tuning request batches for topic operations

The Topic Operator uses the request batching capabilities of the Kafka Admin API for operations on topic resources. You can fine-tune the batching mechanism using the following operator configuration properties:

- `STRIMZI_MAX_QUEUE_SIZE` to set the maximum size of the topic event queue. The default value is 1024.
- `STRIMZI_MAX_BATCH_SIZE` to set the maximum number of topic events allowed in a single batch. The default value is 100.
- `MAX_BATCH_LINGER_MS` to specify the maximum time to wait for a batch to accumulate items before processing. The default is 100 milliseconds.

If the maximum size of the request batching queue is exceeded, the Topic Operator shuts down and is restarted. To prevent frequent restarts, consider adjusting the `STRIMZI_MAX_QUEUE_SIZE` property to accommodate the typical load.

Chapter 12. Using the User Operator to manage Kafka users

When you create, modify or delete a user using the `KafkaUser` resource, the User Operator ensures that these changes are reflected in the Kafka cluster.

For more information on the `KafkaUser` resource, see the [KafkaUser schema reference](#).

12.1. Configuring Kafka users

Use the properties of the `KafkaUser` resource to configure Kafka users.

You can use `kubectl apply` to create or modify users, and `kubectl delete` to delete existing users.

For example:

- `kubectl apply -f <user_config_file>`
- `kubectl delete KafkaUser <user_name>`

Users represent Kafka clients. When you configure Kafka users, you enable the user authentication and authorization mechanisms required by clients to access Kafka. The mechanism used must match the equivalent `Kafka` configuration. For more information on using `Kafka` and `KafkaUser` resources to secure access to Kafka brokers, see [Securing access to a Kafka cluster](#).

Prerequisites

- A running Kafka cluster configured with a Kafka broker listener using mTLS authentication and TLS encryption.
- A running User Operator (typically deployed with the Entity Operator).

Procedure

1. Configure the `KafkaUser` resource.

This example specifies mTLS authentication and simple authorization using ACLs.

Example Kafka user configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user-1
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
  acls:
```

```

# Example consumer Acls for topic my-topic using consumer group my-group
- resource:
    type: topic
    name: my-topic
    patternType: literal
  operations:
    - Describe
    - Read
  host: "*"
- resource:
    type: group
    name: my-group
    patternType: literal
  operations:
    - Read
  host: "*"
# Example Producer Acls for topic my-topic
- resource:
    type: topic
    name: my-topic
    patternType: literal
  operations:
    - Create
    - Describe
    - Write
  host: "*"

```

2. Create the **KafkaUser** resource in Kubernetes.

```
kubectl apply -f <user_config_file>
```

3. Wait for the ready status of the user to change to **True**:

```
kubectl get kafkausers -o wide -w -n <namespace>
```

Kafka user status

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
my-user-1	my-cluster	tls	simple	True
my-user-2	my-cluster	tls	simple	
my-user-3	my-cluster	tls	simple	True

User creation is successful when the **READY** output shows **True**.

4. If the **READY** column stays blank, get more details on the status from the resource YAML or User Operator logs.

Messages provide details on the reason for the current status.

```
kubectl get kafkausers my-user-2 -o yaml
```

Details on a user with a NotReady status

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:07:37.238065Z"
    message: Simple authorization ACL rules are configured but not supported in the
              Kafka cluster configuration.
    reason: InvalidResourceException
    status: "True"
    type: NotReady
```

In this example, the reason the user is not ready is because simple authorization is not enabled in the [Kafka](#) configuration.

Kafka configuration for simple authorization

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

After updating the Kafka configuration, the status shows the user is ready.

```
kubectl get kafkausers my-user-2 -o wide -w -n <namespace>
```

Status update of the user

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
my-user-2	my-cluster	tls	simple	True

Fetching the details shows no messages.

```
kubectl get kafkausers my-user-2 -o yaml
```

Details on a user with a READY status

```
# ...
status:
```

```
conditions:  
- lastTransitionTime: "2022-06-10T10:33:40.166846Z"  
  status: "True"  
  type: Ready
```

Chapter 13. Setting up client access to a Kafka cluster

After you have [deployed Strimzi](#), you can set up client access to your Kafka cluster. To verify the deployment, you can deploy example producer and consumer clients. Otherwise, create listeners that provide client access within or outside the Kubernetes cluster.

13.1. Deploying example clients

Send and receive messages from a Kafka cluster installed on Kubernetes.

This procedure describes how to deploy Kafka clients to the Kubernetes cluster, then produce and consume messages to test your installation. The clients are deployed using the Kafka container image.

Prerequisites

- The Kafka cluster is available for the clients.

Procedure

1. Deploy a Kafka producer.

This example deploys a Kafka producer that connects to the Kafka cluster `my-cluster`.

A topic named `my-topic` is created.

Deploying a Kafka producer to Kubernetes

```
kubectl run kafka-producer -ti --image=quay.io/stimzi/kafka:0.42.0-kafka-3.7.1  
--rm=true --restart=Never -- bin/kafka-console-producer.sh --bootstrap-server my-  
cluster-kafka-bootstrap:9092 --topic my-topic
```

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

The consumer should consume messages produced to `my-topic` in the Kafka cluster `my-cluster`.

Deploying a Kafka consumer to Kubernetes

```
kubectl run kafka-consumer -ti --image=quay.io/stimzi/kafka:0.42.0-kafka-3.7.1  
--rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server my-  
cluster-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

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

13.2. Configuring listeners to connect to Kafka

Use listeners to enable client connections to Kafka. Strimzi provides a generic [GenericKafkaListener](#) schema with properties to configure listeners through the [Kafka](#) resource.

When configuring a Kafka cluster, you specify a listener [type](#) based on your requirements, environment, and infrastructure. Services, routes, load balancers, and ingresses for clients to connect to a cluster are created according to the listener type.

Internal and external listener types are supported.

Internal listeners

Use internal listener types to connect clients within a Kubernetes cluster.

- [internal](#) to connect within the same Kubernetes cluster
- [cluster-ip](#) to expose Kafka using per-broker [ClusterIP](#) services

Internal listeners use a headless service and the DNS names assigned to the broker pods. By default, they do not use the Kubernetes service DNS domain (typically [.cluster.local](#)). However, you can customize this configuration using the [useServiceDnsDomain](#) property. Consider using a [cluster-ip](#) type listener if routing through the headless service isn't feasible or if you require a custom access mechanism, such as when integrating with specific Ingress controllers or the Kubernetes Gateway API.

External listeners

Use external listener types to connect clients outside a Kubernetes cluster.

- [nodeport](#) to use ports on Kubernetes nodes
- [loadbalancer](#) to use loadbalancer services
- [ingress](#) to use Kubernetes [Ingress](#) and the [Ingress NGINX Controller for Kubernetes](#) (Kubernetes only)
- [route](#) to use OpenShift [Route](#) and the default HAProxy router (OpenShift only)

External listeners handle access to a Kafka cluster from networks that require different authentication mechanisms. For example, loadbalancers might not be suitable for certain infrastructure, such as bare metal, where node ports provide a better option.

IMPORTANT

Do not use the built-in [ingress](#) controller on OpenShift, use the [route](#) type instead. The Ingress NGINX Controller is only intended for use on Kubernetes. The [route](#) type is only supported on OpenShift.

Each listener is defined as an array in the [Kafka](#) resource.

Example listener configuration

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

```

name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
        configuration:
          useServiceDnsDomain: true
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external1
        port: 9094
        type: route
        tls: true
        configuration:
          brokerCertChainAndKey:
            secretName: my-secret
            certificate: my-certificate.crt
            key: my-key.key
    # ...

```

You can configure as many listeners as required, as long as their names and ports are unique. You can also configure listeners for secure connection using authentication.

NOTE

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

Additional resources

- [GenericKafkaListener schema reference](#)

13.3. Listener naming conventions

From the listener configuration, the resulting listener bootstrap and per-broker service names are structured according to the following naming conventions:

Table 17. Listener naming conventions

Listener type	Bootstrap service name	Per-Broker service name
internal	<cluster_name>-kafka-bootstrap	<i>Not applicable</i>

Listener type	Bootstrap service name	Per-Broker service name
loadbalancer nodeport ingress route cluster-ip	<cluster_name>-kafka-<listener-name>-bootstrap	<cluster_name>-kafka-<listener-name>-<idx>

For example, `my-cluster-kafka-bootstrap`, `my-cluster-kafka-external1-bootstrap`, and `my-cluster-kafka-external1-0`. The names are assigned to the services, routes, load balancers, and ingresses created through the listener configuration.

You can use certain backwards compatible names and port numbers to transition listeners initially configured under the retired `KafkaListeners` schema. The resulting external listener naming convention varies slightly. The specific combinations of listener name and port configuration values in the following table are backwards compatible.

Table 18. Backwards compatible listener name and port combinations

Listener name	Port	Bootstrap service name	Per-Broker service name
plain	9092	<cluster_name>-kafka-bootstrap	<i>Not applicable</i>
tls	9093	<cluster-name>-kafka-bootstrap	<i>Not applicable</i>
external	9094	<cluster_name>-kafka-bootstrap	<cluster_name>-kafka-bootstrap-<idx>

13.4. Accessing Kafka using node ports

Use node ports to access a Kafka cluster from an external client outside the Kubernetes cluster.

To connect to a broker, you specify a hostname and port number for the Kafka bootstrap address, as well as the certificate used for TLS encryption.

The procedure shows basic `nodeport` listener configuration. You can use listener properties to enable TLS encryption (`tls`) and specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the following configuration properties with `nodeport` listeners:

`preferredNodePortAddressType`

Specifies the first address type that's checked as the node address.

`externalTrafficPolicy`

Specifies whether the service routes external traffic to node-local or cluster-wide endpoints.

`nodePort`

Overrides the assigned node port numbers for the bootstrap and broker services.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

Prerequisites

- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external4`.

Procedure

1. Configure a `Kafka` resource with an external listener set to the `nodeport` type.

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
    name: my-cluster
    namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external4
        port: 9094
        type: nodeport
        tls: true
        authentication:
          type: tls
        # ...
      # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

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

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret `my-cluster-cluster-ca-cert`.

`NodePort` type services are created for each Kafka broker, as well as an external bootstrap service.

Node port services created for the bootstrap and brokers

NAME	TYPE	CLUSTER-IP	PORT(S)
my-cluster-kafka-external4-0	NodePort	172.30.55.13	9094:31789/TCP
my-cluster-kafka-external4-1	NodePort	172.30.250.248	9094:30028/TCP
my-cluster-kafka-external4-2	NodePort	172.30.115.81	9094:32650/TCP
my-cluster-kafka-external4-bootstrap	NodePort	172.30.30.23	9094:32650/TCP

The bootstrap address used for client connection is propagated to the `status` of the `Kafka` resource.

Example status for the bootstrap address

```
status:
  clusterId: Y_RJQDGKRXmNF7fEcWldJQ
  conditions:
    - lastTransitionTime: '2023-01-31T14:59:37.113630Z'
      status: 'True'
      type: Ready
  kafkaVersion: 3.7.1
  listeners:
    # ...
    - addresses:
        - host: ip-10-0-224-199.us-west-2.compute.internal
          port: 32650
  bootstrapServers: 'ip-10-0-224-199.us-west-2.compute.internal:32650'
  certificates:
    - |
      -----BEGIN CERTIFICATE-----
      -----END CERTIFICATE-----
    name: external4
  observedGeneration: 2
  operatorLastSuccessfulVersion: 0.42.0
  # ...
```

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

```
kubectl get kafka my-cluster
-o=jsonpath='{.status.listeners[?(@.name=="external4")].bootstrapServers}{"\n"}'
ip-10-0-224-199.us-west-2.compute.internal:32650
```

4. Extract the cluster CA certificate.

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

5. Configure your client to connect to the brokers.

- Specify the bootstrap host and port in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, `ip-10-0-224-199.us-west-2.compute.internal:32650`.
- Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your client.

NOTE If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

13.5. Accessing Kafka using loadbalancers

Use loadbalancers to access a Kafka cluster from an external client outside the Kubernetes cluster.

To connect to a broker, you specify a hostname and port number for the Kafka bootstrap address, as well as the certificate used for TLS encryption.

The procedure shows basic `loadbalancer` listener configuration. You can use listener properties to enable TLS encryption (`tls`) and specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the following configuration properties with `loadbalancer` listeners:

`loadBalancerSourceRanges`

Restricts traffic to a specified list of CIDR (Classless Inter-Domain Routing) ranges.

`externalTrafficPolicy`

Specifies whether the service routes external traffic to node-local or cluster-wide endpoints.

`loadBalancerIP`

Requests a specific IP address when creating a loadbalancer.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

Prerequisites

- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external3`.

Procedure

1. Configure a `Kafka` resource with an external listener set to the `loadbalancer` type.

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  kafka:
```

```

# ...
listeners:
  - name: external3
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    # ...
  # ...
zookeeper:
  # ...

```

2. Create or update the resource.

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

A cluster CA certificate to verify the identity of the kafka brokers is also created in the secret **my-cluster-cluster-ca-cert**.

loadbalancer type services and loadbalancers are created for each Kafka broker, as well as an external bootstrap service.

Loadbalancer services and loadbalancers created for the bootstraps and brokers

NAME	TYPE	CLUSTER-IP	PORT(S)
my-cluster-kafka-external3-0 9094:30011/TCP	LoadBalancer	172.30.204.234	
my-cluster-kafka-external3-1 9094:32544/TCP	LoadBalancer	172.30.164.89	
my-cluster-kafka-external3-2 9094:32504/TCP	LoadBalancer	172.30.73.151	
my-cluster-kafka-external3-bootstrap 9094:30371/TCP	LoadBalancer	172.30.30.228	

NAME	EXTERNAL-IP (loadbalancer)
my-cluster-kafka-external3-0 1132975133.us-west-2.elb.amazonaws.com	a8a519e464b924000b6c0f0a05e19f0d-
my-cluster-kafka-external3-1 611832211.us-west-2.elb.amazonaws.com	ab6adc22b556343afb0db5ea05d07347-
my-cluster-kafka-external3-2 777597560.us-west-2.elb.amazonaws.com	a9173e8ccb1914778aeb17eca98713c0-
my-cluster-kafka-external3-bootstrap west-2.elb.amazonaws.com	a8d4a6fb363bf447fb6e475fc3040176-36312313.us-

The bootstrap address used for client connection is propagated to the **status** of the **Kafka** resource.

Example status for the bootstrap address

```
status:  
  clusterId: Y_RJQDGKRXmNF7fEcWldJQ  
  conditions:  
    - lastTransitionTime: '2023-01-31T14:59:37.113630Z'  
      status: 'True'  
      type: Ready  
  kafkaVersion: 3.7.1  
  listeners:  
    # ...  
    - addresses:  
      - host: >-  
        a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com  
        port: 9094  
  bootstrapServers: >-  
    a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9094  
  certificates:  
    - |  
      -----BEGIN CERTIFICATE-----  
  
      -----END CERTIFICATE-----  
    name: external3  
    observedGeneration: 2  
    operatorLastSuccessfulVersion: 0.42.0  
  # ...
```

The DNS addresses used for client connection are propagated to the **status** of each loadbalancer service.

Example status for the bootstrap loadbalancer

```
status:  
  loadBalancer:  
    ingress:  
      - hostname: >-  
        a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com  
  # ...
```

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

```
kubectl get kafka my-cluster  
-o=jsonpath='{.status.listeners[?(@.name=="external3")].bootstrapServers}{"\n"}'  
a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9094
```

4. Extract the cluster CA certificate.

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

5. Configure your client to connect to the brokers.

- a. Specify the bootstrap host and port in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, a8d4a6fb363bf447fb6e475fc3040176-36312313.us-west-2.elb.amazonaws.com:9094.
- b. Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your client.

NOTE If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

13.6. Accessing Kafka using an Ingress NGINX Controller for Kubernetes

Use an [Ingress NGINX Controller for Kubernetes](#) to access a Kafka cluster from clients outside the Kubernetes cluster.

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

To connect to a broker, you specify a hostname for the ingress bootstrap address, as well as the certificate used for TLS encryption. For access using an ingress, the port used in the Kafka client is typically 443.

The procedure shows basic `ingress` listener configuration. TLS encryption (`tls`) must be enabled. You can also specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the `class` configuration property with `ingress` listeners to specify the ingress controller used.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

TLS passthrough

Make sure that you enable TLS passthrough in your Ingress NGINX Controller for Kubernetes deployment. Kafka uses a binary protocol over TCP, but the Ingress NGINX Controller for Kubernetes is designed to work with a HTTP protocol. To be able to route TCP traffic through ingresses, Strimzi uses TLS passthrough with Server Name Indication (SNI).

SNI helps with identifying and passing connection to Kafka brokers. In passthrough mode, TLS encryption is always used. Because the connection passes to the brokers, the listeners use the TLS certificates signed by the internal cluster CA and not the ingress certificates. To configure listeners to use your own listener certificates, [use the `brokerCertChainAndKey` property](#).

For more information about enabling TLS passthrough, see the [TLS passthrough documentation](#).

Prerequisites

- An Ingress NGINX Controller for Kubernetes is running with TLS passthrough enabled
- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external2`.

Procedure

1. Configure a `Kafka` resource with an external listener set to the `ingress` type.

Specify an ingress hostname for the bootstrap service and each of the Kafka brokers in the Kafka cluster. Add any hostname to the `bootstrap` and `broker-<index>` prefixes that identify the bootstrap and brokers.

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
    name: my-cluster
    namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external2
        port: 9094
        type: ingress
        tls: true ①
        authentication:
          type: tls
    configuration:
      bootstrap:
        host: bootstrap.myingress.com
      brokers:
        - broker: 0
          host: broker-0.myingress.com
        - broker: 1
          host: broker-1.myingress.com
        - broker: 2
          host: broker-2.myingress.com
    class: nginx ②
```

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

- ① For **ingress** type listeners, TLS encryption must be enabled (**true**).
- ② (Optional) Class that specifies the ingress controller to use. You might need to add a class if you have not set up a default and a class name is missing in the ingresses created.

2. Create or update the resource.

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

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret **my-cluster-cluster-ca-cert**.

ClusterIP type services are created for each Kafka broker, as well as an external bootstrap service.

An **ingress** is also created for each service, with a DNS address to expose them using the Ingress NGINX Controller for Kubernetes.

Ingresses created for the bootstrap and brokers

NAME	CLASS	HOSTS	ADDRESS
PORTS			
my-cluster-kafka-external2-0 80,443	nginx	broker-0.myingress.com	192.168.49.2
my-cluster-kafka-external2-1 80,443	nginx	broker-1.myingress.com	192.168.49.2
my-cluster-kafka-external2-2 80,443	nginx	broker-2.myingress.com	192.168.49.2
my-cluster-kafka-external2-bootstrap 80,443	nginx	bootstrap.myingress.com	192.168.49.2

The DNS addresses used for client connection are propagated to the **status** of each ingress.

Status for the bootstrap ingress

```
status:
loadBalancer:
  ingress:
    - ip: 192.168.49.2
# ...
```

3. Use a target broker to check the client-server TLS connection on port 443 using the OpenSSL **s_client**.

```
openssl s_client -connect broker-0.myingress.com:443 -servername broker-
```

```
0.myingress.com -showcerts
```

The server name is the SNI for passing the connection to the broker.

If the connection is successful, the certificates for the broker are returned.

Certificates for the broker

```
Certificate chain
0 s:0 = io.strimzi, CN = my-cluster-kafka
i:0 = io.strimzi, CN = cluster-ca v0
```

4. Extract the cluster CA certificate.

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

5. Configure your client to connect to the brokers.

- Specify the bootstrap host (from the listener [configuration](#)) and port 443 in your Kafka client as the bootstrap address to connect to the Kafka cluster. For example, [bootstrap.myingress.com:443](#).
- Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your client.

NOTE If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

13.7. Accessing Kafka using OpenShift routes

Use OpenShift routes to access a Kafka cluster from clients outside the OpenShift cluster.

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

To connect to a broker, you specify a hostname for the route bootstrap address, as well as the certificate used for TLS encryption. For access using routes, the port is always 443.

WARNING An OpenShift route address comprises the Kafka cluster name, the listener

name, the project name, and the domain of the router. For example, `my-cluster-kafka-external1-bootstrap-my-project.domain.com` (`<cluster_name>-kafka-<listener_name>-bootstrap-<namespace>.domain`). Each DNS label (between periods ".") must not exceed 63 characters, and the total length of the address must not exceed 255 characters.

The procedure shows basic listener configuration. TLS encryption (`tls`) must be enabled. You can also specify a client authentication mechanism (`authentication`). Add additional configuration using `configuration` properties. For example, you can use the `host` configuration property with `route` listeners to specify the hostnames used by the bootstrap and per-broker services.

For more information on listener configuration, see the [GenericKafkaListener schema reference](#).

TLS passthrough

TLS passthrough is enabled for routes created by Strimzi. Kafka uses a binary protocol over TCP, but routes are designed to work with a HTTP protocol. To be able to route TCP traffic through routes, Strimzi uses TLS passthrough with Server Name Indication (SNI).

SNI helps with identifying and passing connection to Kafka brokers. In passthrough mode, TLS encryption is always used. Because the connection passes to the brokers, the listeners use TLS certificates signed by the internal cluster CA and not the ingress certificates. To configure listeners to use your own listener certificates, [use the brokerCertChainAndKey property](#).

Prerequisites

- A running Cluster Operator

In this procedure, the Kafka cluster name is `my-cluster`. The name of the listener is `external1`.

Procedure

1. Configure a `Kafka` resource with an external listener set to the `route` type.

For example:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
    name: my-cluster
    namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external1
        port: 9094
        type: route
        tls: true ①
        authentication:
```

```

    type: tls
    # ...
    # ...
zookeeper:
    # ...

```

① For `route` type listeners, TLS encryption must be enabled (`true`).

2. Create or update the resource.

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

A cluster CA certificate to verify the identity of the kafka brokers is created in the secret `my-cluster-cluster-ca-cert`.

`ClusterIP` type services are created for each Kafka broker, as well as an external bootstrap service.

A `route` is also created for each service, with a DNS address (host/port) to expose them using the default OpenShift HAProxy router.

The routes are preconfigured with TLS passthrough.

Routes created for the bootstraps and brokers

NAME	HOST/PORT
SERVICES	PORT TERMINATION
my-cluster-kafka-external1-0	my-cluster-kafka-external1-0-my-project.router.com 9094 passthrough
my-cluster-kafka-external1-1	my-cluster-kafka-external1-1-my-project.router.com 9094 passthrough
my-cluster-kafka-external1-2	my-cluster-kafka-external1-2-my-project.router.com 9094 passthrough
my-cluster-kafka-external1-bootstrap	my-cluster-kafka-external1-bootstrap-my-project.router.com 9094 passthrough

The DNS addresses used for client connection are propagated to the `status` of each route.

Example status for the bootstrap route

```

status:
  ingress:
    - host: >-
      my-cluster-kafka-external1-bootstrap-my-project.router.com
# ...

```

3. Use a target broker to check the client-server TLS connection on port 443 using the OpenSSL `s_client`.

```
openssl s_client -connect my-cluster-kafka-external1-0-my-project.router.com:443  
-servername my-cluster-kafka-external1-0-my-project.router.com -showcerts
```

The server name is the Server Name Indication (SNI) for passing the connection to the broker.

If the connection is successful, the certificates for the broker are returned.

Certificates for the broker

```
Certificate chain  
0 s:0 = io.strimzi, CN = my-cluster-kafka  
i:0 = io.strimzi, CN = cluster-ca v0
```

4. Retrieve the address of the bootstrap service from the status of the **Kafka** resource.

```
kubectl get kafka my-cluster  
-o=jsonpath='{.status.listeners[?(@.name=="external1")].bootstrapServers}{ "\n" }'  
  
my-cluster-kafka-external1-bootstrap-my-project.router.com:443
```

The address comprises the Kafka cluster name, the listener name, the project name and the domain of the router (**router.com** in this example).

5. Extract the cluster CA certificate.

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

6. Configure your client to connect to the brokers.

- Specify the address for the bootstrap service and port 443 in your Kafka client as the bootstrap address to connect to the Kafka cluster.
- Add the extracted certificate to the truststore of your Kafka client to configure a TLS connection.

If you enabled a client authentication mechanism, you will also need to configure it in your client.

NOTE If you are using your own listener certificates, check whether you need to add the CA certificate to the client's truststore configuration. If it is a public (external) CA, you usually won't need to add it.

13.8. Discovering connection details for clients

Service discovery makes it easier for client applications running in the same Kubernetes cluster as Strimzi to interact with a Kafka cluster.

A service discovery label and annotation are created for the following services:

- Internal Kafka bootstrap service
- Kafka Bridge service

Service discovery label

The service discovery label, `strimzi.io/discovery`, is set to `true` for `Service` resources to make them discoverable for client connections.

Service discovery annotation

The service discovery annotation provides connection details in JSON format for each service for client applications to use to establish connections.

Example internal Kafka bootstrap service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-  
      [ {  
          "port" : 9092,  
          "tls" : false,  
          "protocol" : "kafka",  
          "auth" : "scram-sha-512"  
        }, {  
          "port" : 9093,  
          "tls" : true,  
          "protocol" : "kafka",  
          "auth" : "tls"  
        } ]  
  labels:  
    strimzi.io/cluster: my-cluster  
    strimzi.io/discovery: "true"  
    strimzi.io/kind: Kafka  
    strimzi.io/name: my-cluster-kafka-bootstrap  
  name: my-cluster-kafka-bootstrap  
spec:  
  #...
```

Example Kafka Bridge service

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-  
      [ {  
          "port" : 8080,  
          "tls" : false,  
        } ]
```

```
        "auth" : "none",
        "protocol" : "http"
    } ]
labels:
  strimzi.io/cluster: my-bridge
  strimzi.io/discovery: "true"
  strimzi.io/kind: KafkaBridge
  strimzi.io/name: my-bridge-bridge-service
```

Find services by specifying the discovery label when fetching services from the command line or a corresponding API call.

Returning services using the discovery label

```
kubectl get service -l strimzi.io/discovery=true
```

Connection details are returned when retrieving the service discovery label.

Chapter 14. Securing access to a Kafka cluster

Secure connections by configuring Kafka and Kafka users. Through configuration, you can implement encryption, authentication, and authorization mechanisms.

Kafka configuration

To establish secure access to Kafka, configure the [Kafka](#) resource to set up the following configurations based on your specific requirements:

- Listeners with specified authentication types to define how clients authenticate
 - TLS encryption for communication between Kafka and clients
 - Supported TLS versions and cipher suites for additional security
- Authorization for the entire Kafka cluster
- Network policies for restricting access
- Super users for unconstrained access to brokers

Authentication is configured independently for each listener, while authorization is set up for the whole Kafka cluster.

For more information on access configuration for Kafka, see the [Kafka schema reference](#) and [GenericKafkaListener schema reference](#).

User (client-side) configuration

To enable secure client access to Kafka, configure [KafkaUser](#) resources. These resources represent clients and determine how they authenticate and authorize with the Kafka cluster.

Configure the [KafkaUser](#) resource to set up the following configurations based on your specific requirements:

- Authentication that must match the enabled listener authentication
 - Supported TLS versions and cipher suites that must match the Kafka configuration
- Simple authorization to apply Access Control List (ACL) rules
 - ACLs for fine-grained control over user access to topics and actions
- Quotas to limit client access based on byte rates or CPU utilization

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

For more information on access configuration for users, see the [KafkaUser schema reference](#).

14.1. Configuring client authentication on listeners

Configure client authentication for Kafka brokers when creating listeners. Specify the listener

authentication type using the `Kafka.spec.kafka.listeners.authentication` property in the `Kafka` resource.

For clients inside the Kubernetes cluster, you can create `plain` (without encryption) or `tls internal` listeners. The `internal` listener type use a headless service and the DNS names given to the broker pods. As an alternative to the headless service, you can also create a `cluster-ip` type of internal listener to expose Kafka using per-broker `ClusterIP` services. For clients outside the Kubernetes cluster, you create `external` listeners and specify a connection mechanism, which can be `nodeport`, `loadbalancer`, `ingress` (Kubernetes only), or `route` (OpenShift only).

For more information on the configuration options for connecting an external client, see [Setting up client access to a Kafka cluster](#).

Supported authentication options:

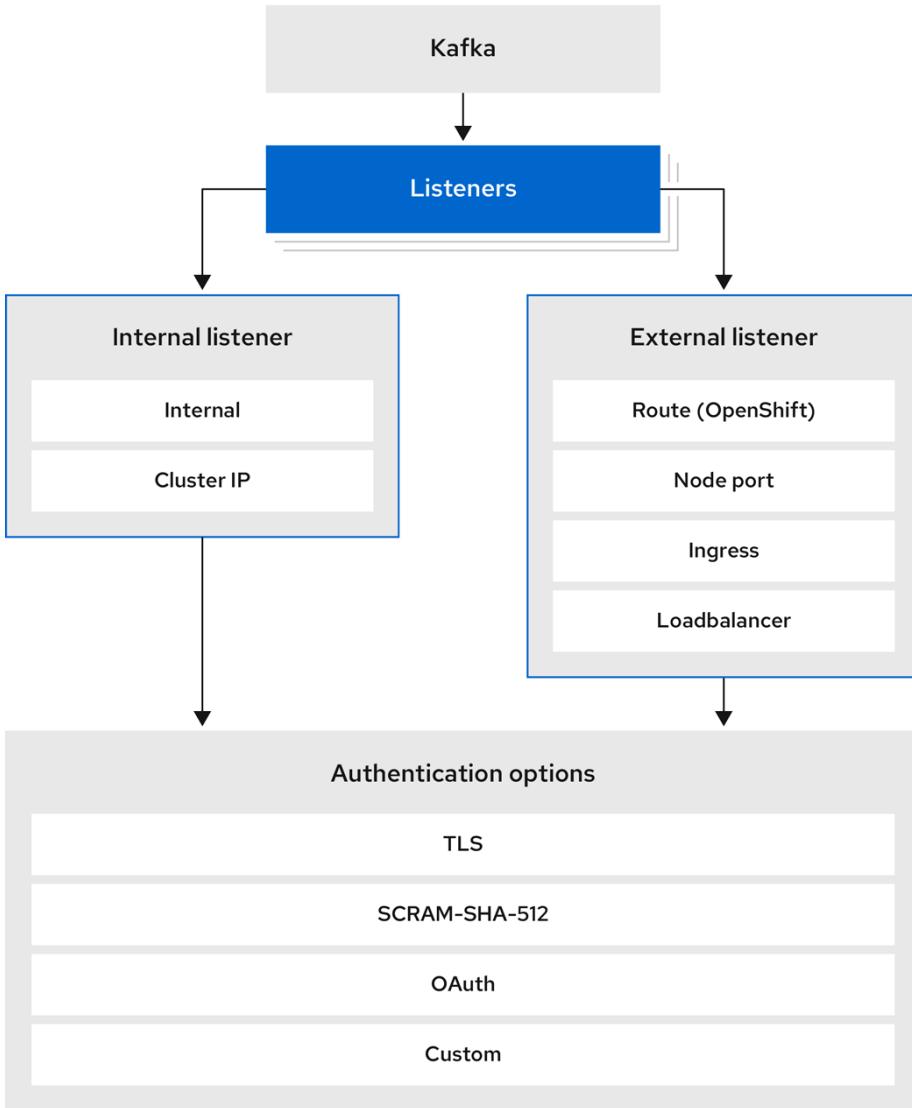
1. mTLS authentication (only on the listeners with TLS enabled encryption)
2. SCRAM-SHA-512 authentication
3. [OAuth 2.0 token-based authentication](#)
4. [Custom authentication](#)
5. [TLS versions and cipher suites](#)

If you're using OAuth 2.0 for client access management, user authentication and authorization credentials are handled through the authorization server.

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

NOTE

Try exploring the standard authentication options before using custom authentication. Custom authentication allows for any type of Kafka-supported authentication. It can provide more flexibility, but also adds complexity.



222_Streams_II22

Figure 4. Kafka listener authentication options

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

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

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

The following example shows:

- A `plain` listener configured for SCRAM-SHA-512 authentication
- A `tls` listener with mTLS authentication
- An `external` listener with mTLS authentication

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

IMPORTANT

When configuring listeners for client access to brokers, you can use port 9092 or higher (9093, 9094, and so on), but with a few exceptions. The

listeners cannot be configured to use the ports reserved for interbroker communication (9090 and 9091), Prometheus metrics (9404), and JMX (Java Management Extensions) monitoring (9999).

Example listener authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: true
        authentication:
          type: scram-sha-512
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        authentication:
          type: tls
    # ...
```

14.1.1. mTLS authentication

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

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. For mutual, or two-way, authentication, both the server and the client present certificates. When you configure mTLS authentication, the broker authenticates the client (client authentication) and the client authenticates the broker (server authentication).

mTLS listener configuration in the [Kafka](#) resource requires the following:

- **tls: true** to specify TLS encryption and server authentication

- `authentication.type`: `tls` to specify the client authentication

When a Kafka cluster is created by the Cluster Operator, it creates a new secret with the name `<cluster_name>-cluster-ca-cert`. The secret contains a CA certificate. The CA certificate is in [PEM](#) and [PKCS #12 format](#). To verify a Kafka cluster, add the CA certificate to the truststore in your client configuration. To verify a client, add a user certificate and key to the keystore in your client configuration. For more information on configuring a client for mTLS, see [Configuring user authentication](#).

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

14.1.2. SCRAM-SHA-512 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 (Simple Authentication and Security Layer) SCRAM-SHA-512 to provide authentication on both unencrypted and encrypted client connections.

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

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

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

When `KafkaUser.spec.authentication.type` is configured with `scram-sha-512` the User Operator will generate a random 32-character password consisting of upper and lowercase ASCII letters and numbers.

14.1.3. Restricting access to listeners with network policies

Control listener access by configuring the `networkPolicyPeers` property in the `Kafka` resource.

By default, Strimzi automatically creates a `NetworkPolicy` resource for every enabled Kafka listener, allowing connections from all namespaces.

To restrict listener access to specific applications or namespaces at the network level, configure the `networkPolicyPeers` property. Each listener can have its own `networkPolicyPeers` configuration. For more information on network policy peers, refer to the [NetworkPolicyPeer API reference](#).

If you want to use custom network policies, you can set the `STRIMZI_NETWORK_POLICY_GENERATION` environment variable to `false` in the Cluster Operator configuration. For more information, see

Configuring the Cluster Operator.

NOTE

Your configuration of Kubernetes must support ingress [NetworkPolicies](#) in order to use network policies.

Prerequisites

- A Kubernetes cluster with support for Ingress NetworkPolicies.
- The Cluster Operator is running.

Procedure

1. Configure the `networkPolicyPeers` property to define the application pods or namespaces allowed to access the Kafka cluster.

This example shows configuration for a `tls` listener to allow connections only from application pods with the label `app` set to `kafka-client`:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    networkPolicyPeers:
      - podSelector:
          matchLabels:
            app: kafka-client
    # ...
  zookeeper:
    # ...
```

2. Apply the changes to the [Kafka](#) resource configuration.

Additional resources

- [networkPolicyPeers configuration](#)
- [NetworkPolicyPeer API reference](#)

14.1.4. Using custom listener certificates for TLS encryption

This procedure shows how to configure custom server certificates for TLS listeners or external listeners which have TLS encryption enabled.

By default, Kafka listeners use certificates signed by Strimzi's internal CA (certificate authority). The

Cluster Operator automatically generates a CA certificate when creating a Kafka cluster. To configure a client for TLS, the CA certificate is included in its truststore configuration to authenticate the Kafka cluster. Alternatively, you have the option to [install and use your own CA certificates](#).

However, if you prefer more granular control by using your own custom certificates at the listener-level, you can configure listeners using `brokerCertChainAndKey` properties. You create a secret with your own private key and server certificate, then specify them in the `brokerCertChainAndKey` configuration.

User-provided certificates allow you to leverage existing security infrastructure. You can use a certificate signed by a public (external) CA or a private CA. Kafka clients need to trust the CA which was used to sign the listener certificate. If signed by a public CA, you usually won't need to add it to a client's truststore configuration.

Custom certificates are not managed by Strimzi, so you need to renew them manually.

NOTE

Listener certificates are used for TLS encryption and server authentication only. They are not used for TLS client authentication. If you want to use your own certificate for TLS client authentication as well, you must [install and use your own clients CA](#).

Prerequisites

- The Cluster Operator is running.
- Each listener requires the following:
 - A compatible server certificate signed by an external CA. (Provide an X.509 certificate in PEM format.)

You can use one listener certificate for multiple listeners.

- Subject Alternative Names (SANs) are specified in the certificate for each listener. For more information, see [Specifying SANs for custom listener certificates](#).

If you are not using a self-signed certificate, you can provide a certificate that includes the whole CA chain in the certificate.

You can only use the `brokerCertChainAndKey` properties if TLS encryption (`tls: true`) is configured for the listener.

NOTE

Strimzi does not support the use of encrypted private keys for TLS. The private key stored in the secret must be unencrypted for this to work.

Procedure

1. Create a `Secret` containing your private key and server certificate:

```
kubectl create secret generic <my_secret> --from-file=<my_listener_key.key> --from-file=<my_listener_certificate.crt>
```

2. Edit the **Kafka** resource for your cluster.

Configure the listener to use your **Secret**, certificate file, and private key file in the `configuration.brokerCertChainAndKey` property.

*Example configuration for a **loadbalancer** external listener with TLS encryption enabled*

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: external3
    port: 9094
    type: loadbalancer
    tls: true
  configuration:
    brokerCertChainAndKey:
      secretName: my-secret
      certificate: my-listener-certificate.crt
      key: my-listener-key.key
# ...
```

Example configuration for a TLS listener

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

3. Apply the changes to the **Kafka** resource configuration.

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

NOTE

A rolling update is also started if you update a Kafka listener certificate in a

Secret that is already used by a listener.

14.1.5. Specifying SANs for custom listener certificates

In order to use TLS hostname verification with custom [Kafka listener certificates](#), you must specify the correct Subject Alternative Names (SANs) for each listener.

The certificate SANs must specify hostnames for the following:

- All of the Kafka brokers in your cluster
- The Kafka cluster bootstrap service

You can use wildcard certificates if they are supported by your CA.

Examples of SANs for internal listeners

Use the following examples to help you specify hostnames of the SANs in your certificates for your internal listeners.

Replace `<cluster-name>` with the name of the Kafka cluster and `<namespace>` with the Kubernetes namespace where the cluster is running.

Wildcards example for a type: internal listener

```
//Kafka brokers
*.<cluster_name>-kafka-brokers
*.<cluster_name>-kafka-brokers.<namespace>.svc

// Bootstrap service
<cluster_name>-kafka-bootstrap
<cluster_name>-kafka-bootstrap.<namespace>.svc
```

Non-wildcards example for a type: internal listener

```
// Kafka brokers
<cluster_name>-kafka-0.<cluster_name>-kafka-brokers
<cluster_name>-kafka-0.<cluster_name>-kafka-brokers.<namespace>.svc
<cluster_name>-kafka-1.<cluster_name>-kafka-brokers
<cluster_name>-kafka-1.<cluster_name>-kafka-brokers.<namespace>.svc
# ...

// Bootstrap service
<cluster_name>-kafka-bootstrap
<cluster_name>-kafka-bootstrap.<namespace>.svc
```

Non-wildcards example for a type: cluster-ip listener

```
// Kafka brokers
<cluster_name>-kafka-<listener-name>-0
```

```

<cluster_name>-kafka-<listener-name>-0.<namespace>.svc
<cluster_name>-kafka-listener-name>-1
<cluster_name>-kafka-<listener-name>-1.<namespace>.svc
# ...

// Bootstrap service
<cluster_name>-kafka-<listener-name>-bootstrap
<cluster_name>-kafka-<listener-name>-bootstrap.<namespace>.svc

```

Examples of SANs for external listeners

For external listeners which have TLS encryption enabled, the hostnames you need to specify in certificates depends on the external listener [type](#).

Table 19. SANs for each type of external listener

External listener type	In the SANs, specify...
ingress	Addresses of all Kafka broker Ingress resources and the address of the bootstrap Ingress . You can use a matching wildcard name.
route	Addresses of all Kafka broker Routes and the address of the bootstrap Route . You can use a matching wildcard name.
loadbalancer	Addresses of all Kafka broker loadbalancers and the bootstrap loadbalancer address. You can use a matching wildcard name.
nodeport	Addresses of all Kubernetes worker nodes that the Kafka broker pods might be scheduled to. You can use a matching wildcard name.

14.2. Configuring authorized access to Kafka

Configure authorized access to a Kafka cluster using the [Kafka.spec.kafka.authorization](#) property in the [Kafka](#) resource. If the [authorization](#) property is missing, no authorization is enabled and clients have no restrictions. When enabled, authorization is applied to all enabled listeners. The authorization method is defined in the [type](#) field.

Supported authorization options:

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

- Custom authorization

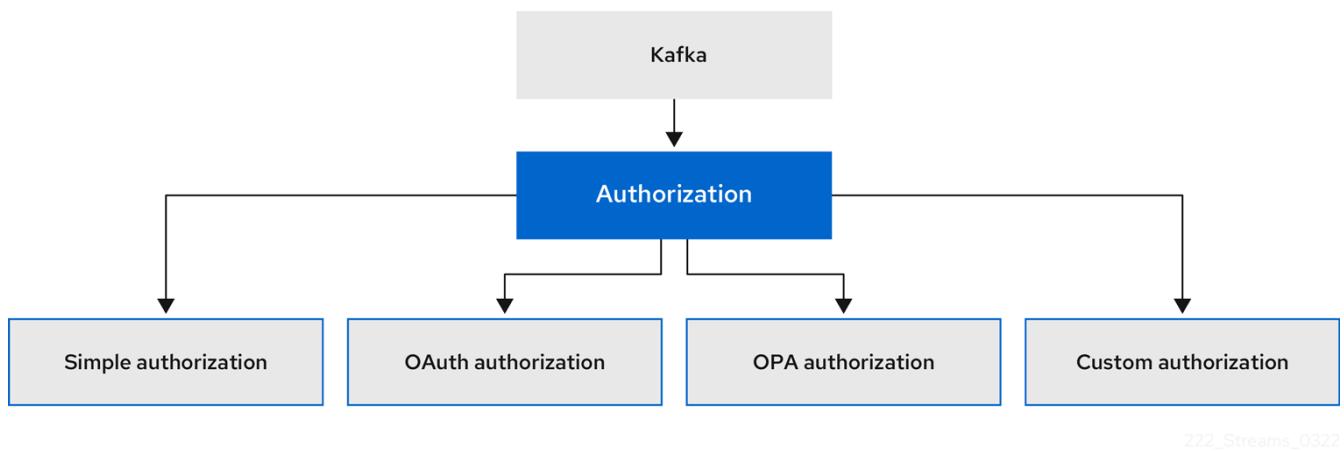


Figure 5. Kafka cluster authorization options

14.2.1. Designating super users

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

To designate super users for a Kafka cluster, add a list of user principals to the `superUsers` property. If a user uses mTLS authentication, the username is the common name from the TLS certificate subject prefixed with `CN=`. If you are not using the User Operator and using your own certificates for mTLS, the username is the full certificate subject. A full certificate subject can have the following fields: `CN=user,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=my_country_code`. Omit any fields that are not present.

An example configuration with super users

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: simple
      superUsers:
        - CN=client_1
        - user_2
        - CN=client_3
        - CN=client_4,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=US
        - CN=client_5,OU=my_ou,O=my_org,C=GB
        - CN=client_6,O=my_org
    # ...
  
```

14.3. Configuring user (client-side) security mechanisms

When configuring security mechanisms in clients, the clients are represented as users. Use the [KafkaUser](#) resource to configure the authentication, authorization, and access rights for Kafka clients.

Authentication permits user access, and authorization constrains user access to permissible actions. You can also create *super users* that have unconstrained access to Kafka brokers.

The authentication and authorization mechanisms must match the [specification for the listener used to access the Kafka brokers](#).

For more information on configuring a [KafkaUser](#) resource to access Kafka brokers securely, see [Example: Setting up secure client access](#).

14.3.1. Associating users with Kafka clusters

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

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

The label enables the User Operator to identify the [KafkaUser](#) resource and create and manage the user.

If the label does not match the Kafka cluster, the User Operator cannot identify the [KafkaUser](#), and the user is not created.

If the status of the [KafkaUser](#) resource remains empty, check your label configuration.

14.3.2. Configuring user authentication

Use the [KafkaUser](#) custom resource to configure authentication credentials for users (clients) that require access to a Kafka cluster. Configure the credentials using the [authentication](#) property in [KafkaUser.spec](#). By specifying a [type](#), you control what credentials are generated.

Supported authentication types:

- [tls](#) for mTLS authentication
- [tls-external](#) for mTLS authentication using external certificates
- [scram-sha-512](#) for SCRAM-SHA-512 authentication

If `tls` or `scram-sha-512` is specified, the User Operator creates authentication credentials when it creates the user. If `tls-external` is specified, the user still uses mTLS, but no authentication credentials are created. Use this option when you're providing your own certificates. When no authentication type is specified, the User Operator does not create the user or its credentials.

You can use `tls-external` to authenticate with mTLS using a certificate issued outside the User Operator. The User Operator does not generate a TLS certificate or a secret. You can still manage ACL rules and quotas through the User Operator in the same way as when you're using the `tls` mechanism. This means that you use the `CN=USER-NAME` format when specifying ACL rules and quotas. `USER-NAME` is the common name given in a TLS certificate.

mTLS authentication

To use mTLS authentication, you set the `type` field in the `KafkaUser` resource to `tls`.

Example user with mTLS authentication enabled

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

The authentication type must match the equivalent configuration for the `Kafka` listener used to access the Kafka cluster.

When the user is created by the User Operator, it creates a new secret with the same name as the `KafkaUser` resource. The secret contains a private and public key for mTLS. The public key is contained in a user certificate, which is signed by a clients CA (certificate authority) when it is created. All keys are in X.509 format.

NOTE If you are using the clients CA generated by the Cluster Operator, the user certificates generated by the User Operator are also renewed when the clients CA is renewed by the Cluster Operator.

The user secret provides keys and certificates in PEM and PKCS #12 formats.

Example secret with user credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
```

```

strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: <public_key> # Public key of the clients CA used to sign this user
  certificate
  user.crt: <user_certificate> # Public key of the user
  user.key: <user_private_key> # Private key of the user
  user.p12: <store> # PKCS #12 store for user certificates and keys
  user.password: <password_for_store> # Protects the PKCS #12 store

```

When you configure a client, you specify the following:

- **Truststore** properties for the public cluster CA certificate to verify the identity of the Kafka cluster
- **Keystore** properties for the user authentication credentials to verify the client

The configuration depends on the file format (PEM or PKCS #12). This example uses PKCS #12 stores, and the passwords required to access the credentials in the stores.

Example client configuration using mTLS in PKCS #12 format

```

bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093 ①
security.protocol=SSL ②
ssl.truststore.location=/tmp/ca.p12 ③
ssl.truststore.password=<truststore_password> ④
ssl.keystore.location=/tmp/user.p12 ⑤
ssl.keystore.password=<keystore_password> ⑥

```

- ① The bootstrap server address to connect to the Kafka cluster.
- ② The security protocol option when using TLS for encryption.
- ③ The truststore location contains the public key certificate (`ca.p12`) for the Kafka cluster. A cluster CA certificate and password is generated by the Cluster Operator in the `<cluster_name>-cluster-ca-cert` secret when the Kafka cluster is created.
- ④ The password (`ca.password`) for accessing the truststore.
- ⑤ The keystore location contains the public key certificate (`user.p12`) for the Kafka user.
- ⑥ The password (`user.password`) for accessing the keystore.

mTLS authentication using a certificate issued outside the User Operator

To use mTLS authentication using a certificate issued outside the User Operator, you set the `type` field in the `KafkaUser` resource to `tls-external`. A secret and credentials are not created for the user.

Example user with mTLS authentication that uses a certificate issued outside the User Operator

```

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

```

```

labels:
  strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls-external
  # ...

```

SCRAM-SHA-512 authentication

To use the SCRAM-SHA-512 authentication mechanism, you set the `type` field in the `KafkaUser` resource to `scram-sha-512`.

Example user with SCRAM-SHA-512 authentication enabled

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  # ...

```

When the user is created by the User Operator, it creates a new secret with the same name as the `KafkaUser` resource. The secret contains the generated password in the `password` key, which is encoded with base64. In order to use the password, it must be decoded.

Example secret with user credentials

```

apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  password: Z2VuZXJhdGVkcGFzc3dvcmQ= ①
  sasl.jaas.config:
    b3JnLmFwYWNoZS5rYWZrYS5jb21tb24uc2VjdXJpdHkuc2NyYW0uU2NyYW1Mb2dpbk1vZHVsZSBzXF1aXJlZC
    B1c2VybmtT0ibXktDXNlcIlgcGFzc3dvcmQ9ImdlbmVyYXR1ZHhc3N3b3JkIjsK ②

```

① The generated password, base64 encoded.

② The JAAS configuration string for SASL SCRAM-SHA-512 authentication, base64 encoded.

Decoding the generated password:

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

Custom password configuration

When a user is created, Strimzi generates a random password. You can use your own password instead of the one generated by Strimzi. To do so, create a secret with the password and reference it in the [KafkaUser](#) resource.

Example user with a password set for SCRAM-SHA-512 authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
    password:
      valueFrom:
        secretKeyRef:
          name: my-secret ①
          key: my-password ②
# ...
```

① The name of the secret containing the predefined password.

② The key for the password stored inside the secret.

14.3.3. Configuring user authorization

Use the [KafkaUser](#) custom resource to configure authorization rules for users (clients) that require access to a Kafka cluster. Configure the rules using the [authorization](#) property in [KafkaUser.spec](#). By specifying a [type](#), you control what rules are used.

To use simple authorization, you set the [type](#) property to [simple](#) in [KafkaUser.spec.authorization](#). The simple authorization uses the Kafka Admin API to manage the ACL rules inside your Kafka cluster. Whether ACL management in the User Operator is enabled or not depends on your authorization configuration in the Kafka cluster.

- For simple authorization, ACL management is always enabled.
- For OPA authorization, ACL management is always disabled. Authorization rules are configured in the OPA server.
- For Keycloak authorization, you can manage the ACL rules directly in Keycloak. You can also delegate authorization to the simple authorizer as a fallback option in the configuration. When delegation to the simple authorizer is enabled, the User Operator will enable management of ACL rules as well.

- For custom authorization using a custom authorization plugin, use the `supportsAdminApi` property in the `.spec.kafka.authorization` configuration of the `Kafka` custom resource to enable or disable the support.

Authorization is cluster-wide. The authorization type must match the equivalent configuration in the `Kafka` custom resource.

If ACL management is not enabled, Strimzi rejects a resource if it contains any ACL rules.

If you're using a standalone deployment of the User Operator, ACL management is enabled by default. You can disable it using the `STRIMZI_ACLS_ADMIN_API_SUPPORTED` environment variable.

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

ACL rules

`simple` authorization uses ACL rules to manage access to Kafka brokers.

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

For more information about the `AclRule` object, see the [AclRule schema reference](#).

Super user access to Kafka brokers

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

For more information on configuring super user access to brokers, see [Kafka authorization](#).

14.3.4. Configuring user quotas

Configure the `spec` for the `KafkaUser` resource to enforce quotas so that a user does not overload the Kafka brokers. Set size-based network usage and time-based CPU utilization thresholds. You can also add a partition mutation quota to control the rate at which requests to change partitions are accepted for user requests.

An example `KafkaUser` with user quotas

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

```
producerByteRate: 1048576 ①
consumerByteRate: 2097152 ②
requestPercentage: 55 ③
controllerMutationRate: 10 ④
```

- ① Byte-per-second quota on the amount of data the user can push to a Kafka broker.
- ② Byte-per-second quota on the amount of data the user can fetch from a Kafka broker.
- ③ CPU utilization limit as a percentage of time for a client group.
- ④ Number of concurrent partition creation and deletion operations (mutations) allowed per second.

For more information on these properties, see the [KafkaUserQuotas schema reference](#).

14.4. Example: Setting up secure client access

This procedure shows how to configure client access to a Kafka cluster from outside Kubernetes or from another Kubernetes cluster. It's split into two parts:

- Securing Kafka brokers
- Securing user access to Kafka

Resource configuration

Client access to the Kafka cluster is secured with the following configuration:

1. An external listener is configured with TLS encryption and mutual TLS (mTLS) authentication in the [Kafka](#) resource, as well as [simple](#) authorization.
2. A [KafkaUser](#) is created for the client, utilizing mTLS authentication, and Access Control Lists (ACLs) are defined for [simple](#) authorization.

At least one listener supporting the desired authentication must be configured for the [KafkaUser](#).

Listeners can be configured for mutual [TLS](#), [SCRAM-SHA-512](#), or [OAuth](#) authentication. While mTLS always uses encryption, it's also recommended when using SCRAM-SHA-512 and OAuth 2.0 authentication.

Authorization options for Kafka include [simple](#), [OAuth](#), [OPA](#), or [custom](#). When enabled, authorization is applied to all enabled listeners.

To ensure compatibility between Kafka and clients, configuration of the following authentication and authorization mechanisms must align:

- For `type: tls` and `type: scram-sha-512` authentication types, `Kafka.spec.kafka.listeners[*].authentication` must match `KafkaUser.spec.authentication`
- For `type: simple` authorization, `Kafka.spec.kafka.authorization` must match `KafkaUser.spec.authorization`

For example, mTLS authentication for a user is only possible if it's also enabled in the Kafka

configuration.

Automation and certificate management

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

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

You add the certificates to your client configuration.

In this procedure, the CA certificates generated by the Cluster Operator are used. Alternatively, you can replace them by [installing your own custom CA certificates](#). You can also configure listeners to [use Kafka listener certificates managed by an external CA](#).

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

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

Prerequisites

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

14.4.1. Securing Kafka brokers

1. Configure the Kafka cluster with a Kafka listener.

- Define the authentication required to access the Kafka broker through the listener.
- Enable authorization on the Kafka broker.

Example listener configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners: ①
    - name: external1 ②
      port: 9094 ③
      type: <listener_type> ④
```

```

    tls: true ⑤
    authentication:
      type: tls ⑥
    configuration: ⑦
      #...
  authorization: ⑧
    type: simple
  superUsers:
    - super-user-name ⑨
# ...

```

- ① Configuration options for enabling external listeners are described in the [Generic Kafka listener schema reference](#).
- ② Name to identify the listener. Must be unique within the Kafka cluster.
- ③ Port number used by the listener inside Kafka. The port number has to be unique within a given Kafka cluster. Allowed port numbers are 9092 and higher with the exception of ports 9404 and 9999, which are already used for Prometheus and JMX. Depending on the listener type, the port number might not be the same as the port number that connects Kafka clients.
- ④ External listener type specified as `route` (OpenShift only), `loadbalancer`, `nodeport` or `ingress` (Kubernetes only). An internal listener is specified as `internal` or `cluster-ip`.
- ⑤ Required. TLS encryption on the listener. For `route` and `ingress` type listeners it must be set to `true`. For mTLS authentication, also use the `authentication` property.
- ⑥ Client authentication mechanism on the listener. For server and client authentication using mTLS, you specify `tls: true` and `authentication.type: tls`.
- ⑦ (Optional) Depending on the requirements of the listener type, you can specify additional [listener configuration](#).
- ⑧ Authorization specified as `simple`, which uses the `AclAuthorizer` and `StandardAuthorizer` Kafka plugins.
- ⑨ (Optional) Super users can access all brokers regardless of any access restrictions defined in ACLs.

WARNING

An OpenShift route address comprises the Kafka cluster name, the listener name, the project name, and the domain of the router. For example, `my-cluster-kafka-external1-bootstrap-my-project.domain.com` (`<cluster_name>-kafka-<listener_name>-bootstrap-<namespace>.domain`). Each DNS label (between periods ".") must not exceed 63 characters, and the total length of the address must not exceed 255 characters.

2. Apply the changes to the Kafka resource configuration.

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

A service is created for each Kafka broker pod.

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

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

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

NOTE

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

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

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

For example:

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

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

14.4.2. Securing user access to Kafka

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

Example user configuration

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

```

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

```

- ① The label must match the label of the Kafka cluster.
- ② Authentication specified as mutual `tls`.
- ③ Simple authorization requires an accompanying list of ACL rules to apply to the user. The rules define the operations allowed on Kafka resources based on the `username` (`my-user`).

2. Apply the changes to the `KafkaUser` resource configuration.

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

Example secret with user credentials

```

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

```

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

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

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

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

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

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

6. Configure your client with the bootstrap address hostname and port for connecting to the Kafka cluster:

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

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

Specify the public cluster CA certificate.

Example truststore configuration

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

SSL is the specified security protocol for mTLS authentication. Specify [SASL_SSL](#) for SCRAM-SHA-512 authentication over TLS. PEM is the file format of the truststore.

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

Specify the public certificate and private key.

Example keystore configuration

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

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

Example keystore configuration

```
props.put(SslConfigs.SSL_KEYSTORE_CERTIFICATE_CHAIN_CONFIG, "-----BEGIN
```

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

14.5. Troubleshooting TLS hostname verification with node ports

Off-cluster access using node ports with TLS encryption enabled does not support TLS hostname verification. Consequently, clients that perform hostname verification will fail to connect.

For example, a Java client will fail with the following exception:

Exception for TLS hostname verification

```
Caused by: java.security.cert.CertificateException: No subject alternative names  
matching IP address 168.72.15.231 found  
...  
...
```

To connect, you must disable hostname verification. In the Java client, set the `ssl.endpoint.identification.algorithm` configuration option to an empty string.

When configuring the client using a properties file, you can do it this way:

```
ssl.endpoint.identification.algorithm=
```

When configuring the client directly in Java, set the configuration option to an empty string:

```
props.put("ssl.endpoint.identification.algorithm", "");
```

Chapter 15. Enabling OAuth 2.0 token-based access

Strimzi supports OAuth 2.0 for securing Kafka clusters by integrating with an OAuth 2.0 authorization server. Kafka brokers and clients both need to be configured to use OAuth 2.0.

OAuth 2.0 enables standardized token-based authentication and authorization between applications, using a central authorization server to issue tokens that grant limited access to resources. You can define specific scopes for fine-grained access control. Scopes correspond to different levels of access to Kafka topics or operations within the cluster.

OAuth 2.0 also supports single sign-on and integration with identity providers.

For more information on using OAuth 2.0, see the [Strimzi OAuth 2.0 for Apache Kafka project](#).

15.1. Configuring an OAuth 2.0 authorization server

Before you can use OAuth 2.0 token-based access, you must configure an authorization server for integration with Strimzi. The steps are dependent on the chosen authorization server. Consult the product documentation for the authorization server for information on how to set up OAuth 2.0 access.

Prepare the authorization server to work with Strimzi by defining *OAuth 2.0 clients* for Kafka and each Kafka client component of your application. In relation to the authorization server, the Kafka cluster and Kafka clients are both regarded as OAuth 2.0 clients.

In general, configure OAuth 2.0 clients in the authorization server with the following client credentials enabled:

- Client ID (for example, `kafka` for the Kafka cluster)
- Client ID and secret as the authentication mechanism

NOTE

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

15.2. Using OAuth 2.0 token-based authentication

Strimzi supports the use of [OAuth 2.0](#) for token-based authentication. An OAuth 2.0 authorization server handles the granting of access and inquiries about access. Kafka clients authenticate to Kafka brokers. Brokers and clients communicate with the authorization server, as necessary, to obtain or validate access tokens.

For a deployment of Strimzi, OAuth 2.0 integration provides the following support:

- Server-side OAuth 2.0 authentication for Kafka brokers

- Client-side OAuth 2.0 authentication for Kafka MirrorMaker, Kafka Connect, and the Kafka Bridge

15.2.1. Configuring OAuth 2.0 authentication on listeners

To secure Kafka brokers with OAuth 2.0 authentication, configure a listener in the [Kafka](#) resource to use OAuth 2.0 authentication and a client authentication mechanism, and add further configuration depending on the authentication mechanism and type of token validation used in the authentication.

Configuring listeners to use `oauth` authentication

Specify a listener in the [Kafka](#) resource with an `oauth` authentication type. You can configure internal and external listeners. We recommend using OAuth 2.0 authentication together with TLS encryption (`tls: true`). Without encryption, the connection is vulnerable to network eavesdropping and unauthorized access through token theft.

Example listener configuration with OAuth 2.0 authentication

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        authentication:
          type: oauth
    #...
```

Enabling SASL authentication mechanisms

Use one or both of the following SASL mechanisms for clients to exchange credentials and establish authenticated sessions with Kafka.

OAUTHBEARER

Using the [OAUTHBEARER](#) authentication mechanism, credentials exchange uses a bearer token provided by an OAuth callback handler. Token provision can be configured to use the following methods:

- Client ID and secret (using the the OAuth 2.0 *client credentials mechanism*)
- Long-lived access token

- Long-lived refresh token obtained manually

OAUTHBEARER is recommended as it provides a higher level of security than **PLAIN**, though it can only be used by Kafka clients that support the **OAUTHBEARER** mechanism at the protocol level. Client credentials are never shared with Kafka.

PLAIN

PLAIN is a simple authentication mechanism used by all Kafka client tools. Consider using **PLAIN** only with Kafka clients that do not support **OAUTHBEARER**. Using the **PLAIN** authentication mechanism, credentials exchange can be configured to use the following methods:

- Client ID and secret (using the the OAuth 2.0 *client credentials mechanism*)

- Long-lived access token

Regardless of the method used, the client must provide `username` and `password` properties to Kafka.

Credentials are handled centrally behind a compliant authorization server, similar to how **OAUTHBEARER** authentication is used. The username extraction process depends on the authorization server configuration.

OAUTHBEARER is automatically enabled in the `oauth` listener configuration for the Kafka broker. To use the **PLAIN** mechanism, you must set the `enablePlain` property to `true`.

In the following example, the **PLAIN** mechanism is enabled, and the **OAUTHBEARER** mechanism is disabled on a listener using the `enableOauthBearer` property.

*Example listener configuration for the **PLAIN** mechanism*

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        authentication:
          type: oauth
          enablePlain: true
          enableOauthBearer: false
    #...
```

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

Configuring fast local JWT token validation

Fast local JWT token validation involves checking a JWT token signature locally to ensure that the token meets the following criteria:

- Contains a `typ` (type) or `token_type` header claim value of `Bearer` to indicate it is an access token
- Is currently valid and not expired
- Has an issuer that matches a `validIssuerURI`

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

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

All communication with the authorization server should be performed using TLS encryption. You can configure a certificate truststore as a Kubernetes `Secret` in your Strimzi project namespace, and use the `tlsTrustedCertificates` property to point to the Kubernetes secret containing the truststore file.

You might want to configure a `userNameClaim` to properly extract a username from the JWT token. If required, you can use a JsonPath expression like `"['user.info'].['user.id']"` to retrieve the username from nested JSON attributes within a token.

If you want to use Kafka ACL authorization, identify the user by their username during authentication. (The `sub` claim in JWT tokens is typically a unique ID, not a username.)

Example configuration for fast local JWT token validation

```
#...
- name: external3
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth ①
    validIssuerUri: https://<auth_server_address>/<issuer-context> ②
    jwksEndpointUri: https://<auth_server_address>/<path_to_jwks_endpoint> ③
    userNameClaim: preferred_username ④
    maxSecondsWithoutReauthentication: 3600 ⑤
    tlsTrustedCertificates: ⑥
      - secretName: oauth-server-cert
        pattern: "* .crt"
    disableTlsHostnameVerification: true ⑦
    jwksExpirySeconds: 360 ⑧
```

```
jwksRefreshSeconds: 300 ⑨  
jwksMinRefreshPauseSeconds: 1 ⑩
```

- ① Listener type set to `oauth`.
- ② URI of the token issuer used for authentication.
- ③ URI of the JWKS certificate endpoint used for local JWT validation.
- ④ The token claim (or key) that contains the actual username used to identify the user. Its value depends on the authorization server. If necessary, a JsonPath expression like `"[user.info].[user.id]"` can be used to retrieve the username from nested JSON attributes within a token.
- ⑤ (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.
- ⑥ (Optional) Certificates stored in X.509 format within the specified secrets for TLS connection to the authorization server.
- ⑦ (Optional) Disable TLS hostname verification. Default is `false`.
- ⑧ The duration the JWKS certificates are considered valid before they expire. Default is `360` seconds. If you specify a longer time, consider the risk of allowing access to revoked certificates.
- ⑨ The period between refreshes of JWKS certificates. The interval must be at least 60 seconds shorter than the expiry interval. Default is `300` seconds.
- ⑩ The minimum pause in seconds between consecutive attempts to refresh JWKS public keys. When an unknown signing key is encountered, the JWKS keys refresh is scheduled outside the regular periodic schedule with at least the specified pause since the last refresh attempt. The refreshing of keys follows the rule of exponential backoff, retrying on unsuccessful refreshes with ever increasing pause, until it reaches `jwksRefreshSeconds`. The default value is 1.

Configuring token validation using an introspection endpoint

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

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

Example token validation configuration using an introspection endpoint

```
- name: external3  
  port: 9094  
  type: loadbalancer  
  tls: true
```

```

authentication:
  type: oauth
  validIssuerUri: https://<auth_server_address>/<issuer-context>
  introspectionEndpointUri:
    https://<auth_server_address>/<path_to_introspection_endpoint> ①
    clientId: kafka-broker ②
    clientSecret: ③
      secretName: my-cluster-oauth
      key: clientSecret
    userNameClaim: preferred_username ④
    maxSecondsWithoutReauthentication: 3600 ⑤
    tlsTrustedCertificates:
      - secretName: oauth-server-cert
        pattern: "*.crt"

```

① URI of the token introspection endpoint.

② Client ID to identify the client.

③ Client Secret and client ID is used for authentication.

④ The token claim (or key) that contains the actual username used to identify the user. Its value depends on the authorization server. If necessary, a JsonPath expression like "[**'user.info'**].["**'user.id'**]" can be used to retrieve the username from nested JSON attributes within a token.

⑤ (Optional) Activates the Kafka re-authentication mechanism that enforces session expiry to the same length of time as the access token. If the specified value is less than the time left for the access token to expire, then the client will have to re-authenticate before the actual token expiry. By default, the session does not expire when the access token expires, and the client does not attempt re-authentication.

Including additional configuration options

Specify additional settings depending on the authentication requirements and the authorization server you are using. Some of these properties apply only to certain authentication mechanisms or when used in combination with other properties.

For example, when using OAuth over **PLAIN**, access tokens are passed as **password** property values with or without an **\$accessToken:** prefix.

- If you configure a token endpoint (**tokenEndpointUri**) in the listener configuration, you need the prefix.
- If you don't configure a token endpoint in the listener configuration, you don't need the prefix. The Kafka broker interprets the password as a raw access token.

If the **password** is set as the access token, the **username** must be set to the same principal name that the Kafka broker obtains from the access token. You can specify username extraction options in your listener using the **userNameClaim**, **fallbackUserNameClaim**, **fallbackUsernamePrefix**, and **userInfoEndpointUri** properties. The username extraction process also depends on your authorization server; in particular, how it maps client IDs to account names.

NOTE

The **PLAIN** mechanism does not support password grant authentication. Use either client credentials (client ID + secret) or an access token for authentication.

Example additional configuration settings

```
# ...
authentication:
  type: oauth
  # ...
  checkIssuer: false ①
  checkAudience: true ②
  fallbackUserNameClaim: client_id ③
  fallbackUserNamePrefix: client-account- ④
  validTokenType: bearer ⑤
  userInfoEndpointUri: https://<auth_server_address>/<path_to_userinfo_endpoint> ⑥
  enableOAuthBearer: false ⑦
  enablePlain: true ⑧
  tokenEndpointUri: https://<auth_server_address>/<path_to_token_endpoint> ⑨
  customClaimCheck: "@.custom == 'custom-value'" ⑩
  clientAudience: audience ⑪
  clientScope: scope ⑫
  connectTimeoutSeconds: 60 ⑬
  readTimeoutSeconds: 60 ⑭
  httpRetries: 2 ⑮
  httpRetryPauseMs: 300 ⑯
  groupsClaim: "$.groups" ⑰
  groupsClaimDelimiter: "," ⑱
  includeAcceptHeader: false ⑲
```

- ① If your authorization server does not provide an `iss` claim, it is not possible to perform an issuer check. In this situation, set `checkIssuer` to `false` and do not specify a `validIssuerUri`. Default is `true`.
- ② If your authorization server provides an `aud` (audience) claim, and you want to enforce an audience check, set `checkAudience` to `true`. Audience checks identify the intended recipients of tokens. As a result, the Kafka broker will reject tokens that do not have its `clientId` in their `aud` claim. Default is `false`.
- ③ An authorization server may not provide a single attribute to identify both regular users and clients. When a client authenticates in its own name, the server might provide a *client ID*. When a user authenticates using a username and password to obtain a refresh token or an access token, the server might provide a *username* attribute in addition to a client ID. Use this fallback option to specify the username claim (attribute) to use if a primary user ID attribute is not available. If necessary, a JsonPath expression like `"['client.info'].['client.id']"` can be used to retrieve the fallback username to retrieve the username from nested JSON attributes within a token.
- ④ In situations where `fallbackUserNameClaim` is applicable, it may also be necessary to prevent name collisions between the values of the username claim, and those of the fallback username claim. Consider a situation where a client called `producer` exists, but also a regular user called `producer` exists. In order to differentiate between the two, you can use this property to add a

prefix to the user ID of the client.

- ⑤ (Only applicable when using `introspectionEndpointUri`) Depending on the authorization server you are using, the introspection endpoint may or may not return the *token type* attribute, or it may contain different values. You can specify a valid token type value that the response from the introspection endpoint has to contain.
- ⑥ (Only applicable when using `introspectionEndpointUri`) The authorization server may be configured or implemented in such a way to not provide any identifiable information in an introspection endpoint response. In order to obtain the user ID, you can configure the URI of the `userinfo` endpoint as a fallback. The `userNameClaim`, `fallbackUserNameClaim`, and `fallbackUserNamePrefix` settings are applied to the response of `userinfo` endpoint.
- ⑦ Set this to `false` to disable the `OAUTHBEARER` mechanism on the listener. At least one of `PLAIN` or `OAUTHBEARER` has to be enabled. Default is `true`.
- ⑧ Set to `true` to enable `PLAIN` authentication on the listener, which is supported for clients on all platforms.
- ⑨ Additional configuration for the `PLAIN` mechanism. If specified, clients can authenticate over `PLAIN` by passing an access token as the `password` using an `$accessToken:` prefix. For production, always use `https://` urls.
- ⑩ Additional custom rules can be imposed on the JWT access token during validation by setting this to a JsonPath filter query. If the access token does not contain the necessary data, it is rejected. When using the `introspectionEndpointUri`, the custom check is applied to the introspection endpoint response JSON.
- ⑪ An `audience` parameter passed to the token endpoint. An *audience* is used when obtaining an access token for inter-broker authentication. It is also used in the name of a client for OAuth 2.0 over `PLAIN` client authentication using a `clientId` and `secret`. This only affects the ability to obtain the token, and the content of the token, depending on the authorization server. It does not affect token validation rules by the listener.
- ⑫ A `scope` parameter passed to the token endpoint. A `scope` is used when obtaining an access token for inter-broker authentication. It is also used in the name of a client for OAuth 2.0 over `PLAIN` client authentication using a `clientId` and `secret`. This only affects the ability to obtain the token, and the content of the token, depending on the authorization server. It does not affect token validation rules by the listener.
- ⑬ The connect timeout in seconds when connecting to the authorization server. The default value is 60.
- ⑭ The read timeout in seconds when connecting to the authorization server. The default value is 60.
- ⑮ The maximum number of times to retry a failed HTTP request to the authorization server. The default value is `0`, meaning that no retries are performed. To use this option effectively, consider reducing the timeout times for the `connectTimeoutSeconds` and `readTimeoutSeconds` options. However, note that retries may prevent the current worker thread from being available to other requests, and if too many requests stall, it could make the Kafka broker unresponsive.
- ⑯ The time to wait before attempting another retry of a failed HTTP request to the authorization server. By default, this time is set to zero, meaning that no pause is applied. This is because many issues that cause failed requests are per-request network glitches or proxy issues that can

be resolved quickly. However, if your authorization server is under stress or experiencing high traffic, you may want to set this option to a value of 100 ms or more to reduce the load on the server and increase the likelihood of successful retries.

- ⑯ A JsonPath query that is used to extract groups information from either the JWT token or the introspection endpoint response. This option is not set by default. By configuring this option, a custom authorizer can make authorization decisions based on user groups.
- ⑰ A delimiter used to parse groups information when it is returned as a single delimited string. The default value is ',' (comma).
- ⑲ Some authorization servers have issues with client sending `Accept: application/json` header. By setting `includeAcceptHeader: false` the header will not be sent. Default is `true`.

15.2.2. Configuring OAuth 2.0 on client applications

To configure OAuth 2.0 on client applications, you must specify the following:

- SASL (Simple Authentication and Security Layer) security protocols
- SASL mechanisms
- A JAAS (Java Authentication and Authorization Service) module
- Authentication properties to access the authorization server

Configuring SASL protocols

Specify SASL protocols in the client configuration:

- `SASL_SSL` for authentication over TLS encrypted connections
- `SASL_PLAINTEXT` for authentication over unencrypted connections

Use `SASL_SSL` for production and `SASL_PLAINTEXT` for local development only.

When using `SASL_SSL`, additional `ssl.truststore` configuration is needed. The truststore configuration is required for secure connection (`https://`) to the OAuth 2.0 authorization server. To verify the OAuth 2.0 authorization server, add the CA certificate for the authorization server to the truststore in your client configuration. You can configure a truststore in PEM or PKCS #12 format.

Configuring SASL authentication mechanisms

Specify SASL mechanisms in the client configuration:

- `OAUTHBEARER` for credentials exchange using a bearer token
- `PLAIN` to pass client credentials (clientId + secret) or an access token

Configuring a JAAS module

Specify a JAAS module that implements the SASL authentication mechanism as a `sasl.jaas.config` property value:

- `org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule` implements the `OAUTHBEARER` mechanism
- `org.apache.kafka.common.security.plain.PlainLoginModule` implements the `PLAIN` mechanism

NOTE

For the **OAUTHBearer** mechanism, Strimzi provides a callback handler for clients that use Kafka Client Java libraries to enable credentials exchange. For clients in other languages, custom code may be required to obtain the access token. For the **PLAIN** mechanism, Strimzi provides server-side callbacks to enable credentials exchange.

To be able to use the **OAUTHBearer** mechanism, you must also add the custom `io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler` class as the callback handler. `JaasClientOauthLoginCallbackHandler` handles OAuth callbacks to the authorization server for access tokens during client login. This enables automatic token renewal, ensuring continuous authentication without user intervention. Additionally, it handles login credentials for clients using the OAuth 2.0 password grant method.

Configuring authentication properties

Configure the client to use credentials or access tokens for OAuth 2.0 authentication.

Using client credentials

Using client credentials involves configuring the client with the necessary credentials (client ID and secret) to obtain a valid access token from an authorization server. This is the simplest mechanism.

Using access tokens

Using access tokens, the client is configured with a valid long-lived access token or refresh token obtained from an authorization server. Using access tokens adds more complexity because there is an additional dependency on authorization server tools. If you are using long-lived access tokens, you may need to configure the client in the authorization server to increase the maximum lifetime of the token.

The only information ever sent to Kafka is the access token. The credentials used to obtain the token are never sent to Kafka. When a client obtains an access token, no further communication with the authorization server is needed.

SASL authentication properties support the following authentication methods:

- OAuth 2.0 client credentials
- OAuth 2.0 password grant (deprecated)
- Access token
- Refresh token

Add the authentication properties as JAAS configuration (`sasl.jaas.config` and `sasl.login.callback.handler.class`).

If the client application is not configured with an access token directly, the client exchanges one of the following sets of credentials for an access token during Kafka session initiation:

- Client ID and secret
- Client ID, refresh token, and (optionally) a secret
- Username and password, with client ID and (optionally) a secret

NOTE You can also specify authentication properties as environment variables, or as Java system properties. For Java system properties, you can set them using `setProperty` and pass them on the command line using the `-D` option.

Example client credentials configuration

```
security.protocol=SASL_SSL ①
sasl.mechanism=OAUTHBEARER ②
ssl.truststore.location=/tmp/truststore.p12 ③
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer OAuthBearerLoginModule
required \
    oauth.token.endpoint.uri=<token_endpoint_url> \ ④
    oauth.client.id=<client_id> \ ⑤
    oauth.client.secret=<client_secret> \ ⑥
    oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \ ⑦
    oauth.ssl.truststore.password="$STOREPASS" \ ⑧
    oauth.ssl.truststore.type="PKCS12" \ ⑨
    oauth.scope=<scope> \ ⑩
    oauth.audience=<audience> ; ⑪
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCa
llbackHandler
```

- ① `SASL_SSL` security protocol for TLS-encrypted connections. Use `SASL_PLAINTEXT` over unencrypted connections for local development only.
- ② The SASL mechanism specified as `OAUTHBEARER` or `PLAIN`.
- ③ The truststore configuration for secure access to the Kafka cluster.
- ④ URI of the authorization server token endpoint.
- ⑤ Client ID, which is the name used when creating the *client* in the authorization server.
- ⑥ Client secret created when creating the *client* in the authorization server.
- ⑦ The location contains the public key certificate (`truststore.p12`) for the authorization server.
- ⑧ The password for accessing the truststore.
- ⑨ The truststore type.
- ⑩ (Optional) The `scope` for requesting the token from the token endpoint. An authorization server may require a client to specify the scope.
- ⑪ (Optional) The `audience` for requesting the token from the token endpoint. An authorization server may require a client to specify the audience.

Example password grants configuration

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
```

```

ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer OAuthBearerLoginModule
required \
    oauth.token.endpoint.uri=<token_endpoint_url> \
    oauth.client.id=<client_id> ①
    oauth.client.secret=<client_secret> ②
    oauth.password.grant.username=<username> ③
    oauth.password.grant.password=<password> ④
    oauth.ssl.truststore.location=/tmp/oauth-truststore.p12 \
    oauth.ssl.truststore.password=$STOREPASS \
    oauth.ssl.truststore.type=PKCS12 \
    oauth.scope=<scope> \
    oauth.audience=<audience> ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCa
llbackHandler

```

① Client ID, which is the name used when creating the *client* in the authorization server.

② (Optional) Client secret created when creating the *client* in the authorization server.

③ Username for password grant authentication. OAuth password grant configuration (username and password) uses the OAuth 2.0 password grant method. To use password grants, create a user account for a client on your authorization server with limited permissions. The account should act like a service account. Use in environments where user accounts are required for authentication, but consider using a refresh token first.

④ Password for password grant authentication.

NOTE SASL **PLAIN** does not support passing a username and password (password grants) using the OAuth 2.0 password grant method.

Example access token configuration

```

security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer OAuthBearerLoginModule
required \
    oauth.token.endpoint.uri=<token_endpoint_url> \
    oauth.access.token=<access_token> ①
    oauth.ssl.truststore.location=/tmp/oauth-truststore.p12 \
    oauth.ssl.truststore.password=$STOREPASS \
    oauth.ssl.truststore.type=PKCS12 ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCa
llbackHandler

```

① Long-lived access token for Kafka clients.

Example refresh token configuration

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer OAuthBearerLoginModule
required \
oauth.token.endpoint.uri=<token_endpoint_url> \
oauth.client.id=<client_id> \
①
oauth.client.secret=<client_secret> \
②
oauth.refresh.token=<refresh_token> \
③
oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
oauth.ssl.truststore.password="$STOREPASS" \
oauth.ssl.truststore.type="PKCS12" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCa
llbackHandler
```

① Client ID, which is the name used when creating the *client* in the authorization server.

② (Optional) Client secret created when creating the *client* in the authorization server.

③ Long-lived refresh token for Kafka clients.

15.2.3. OAuth 2.0 client authentication flows

OAuth 2.0 authentication flows depend on the underlying Kafka client and Kafka broker configuration. The flows must also be supported by the authorization server used.

The Kafka broker listener configuration determines how clients authenticate using an access token. The client can pass a client ID and secret to request an access token.

If a listener is configured to use **PLAIN** authentication, the client can authenticate with a client ID and secret or username and access token. These values are passed as the **username** and **password** properties of the **PLAIN** mechanism.

Listener configuration supports the following token validation options:

- You can use fast local token validation based on JWT signature checking and local token introspection, without contacting an authorization server. The authorization server provides a JWKS endpoint with public certificates that are used to validate signatures on the tokens.
- You can use a call to a token introspection endpoint provided by an authorization server. Each time a new Kafka broker connection is established, the broker passes the access token received from the client to the authorization server. The Kafka broker checks the response to confirm whether or not the token is valid.

NOTE

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

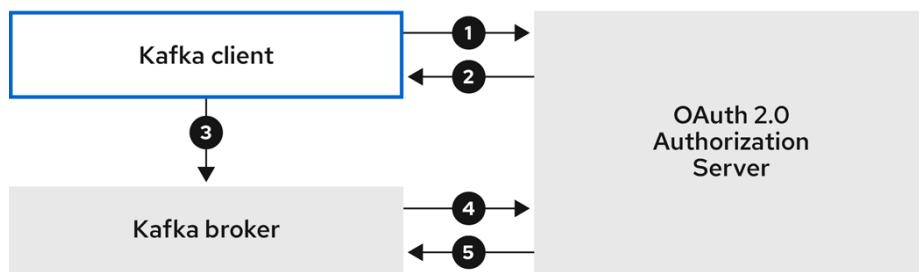
Kafka client credentials can also be configured for the following types of authentication:

- Direct local access using a previously generated long-lived access token
- Contact with the authorization server for a new access token to be issued (using a client ID and a secret, or a refresh token, or a username and a password)

Example client authentication flows using the SASL **OAUTHBEARER** mechanism

You can use the following communication flows for Kafka authentication using the SASL **OAUTHBEARER** mechanism.

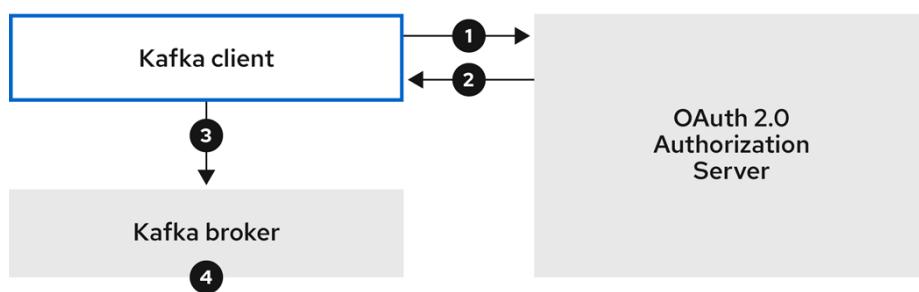
Client using client ID and secret, with broker delegating validation to authorization server



574_AMQ_0424

1. The Kafka client requests an access token from the authorization server using a client ID and secret, and optionally a refresh token. Alternatively, the client may authenticate using a username and a password.
2. The authorization server generates a new access token.
3. The Kafka client authenticates with the Kafka broker using the SASL **OAUTHBEARER** mechanism to pass the access token.
4. The Kafka broker validates the access token by calling a token introspection endpoint on the authorization server using its own client ID and secret.
5. A Kafka client session is established if the token is valid.

Client using client ID and secret, with broker performing fast local token validation

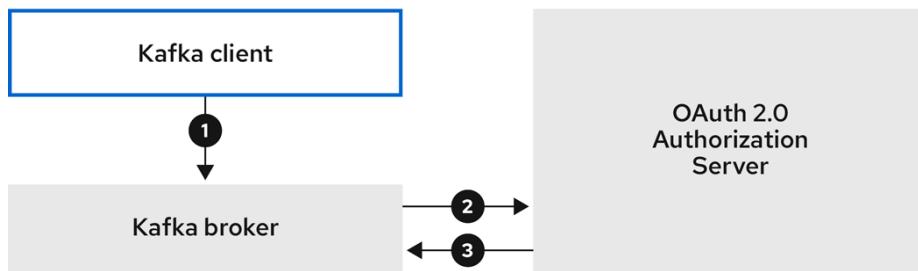


574_AMQ_0424

1. The Kafka client authenticates with the authorization server from the token endpoint, using a client ID and secret, and optionally a refresh token. Alternatively, the client may authenticate using a username and a password.
2. The authorization server generates a new access token.
3. The Kafka client authenticates with the Kafka broker using the SASL **OAUTHBEARER** mechanism to pass the access token.
4. The Kafka broker validates the access token locally using a JWT token signature check, and local

token introspection.

Client using long-lived access token, with broker delegating validation to authorization server



574_AMQ_0424

1. The Kafka client authenticates with the Kafka broker using the SASL **OAUTHBEARER** mechanism to pass the long-lived access token.
2. The Kafka broker validates the access token by calling a token introspection endpoint on the authorization server, using its own client ID and secret.
3. A Kafka client session is established if the token is valid.

Client using long-lived access token, with broker performing fast local validation



574_AMQ_0424

1. The Kafka client authenticates with the Kafka broker using the SASL **OAUTHBEARER** mechanism to pass the long-lived access token.
2. The Kafka broker validates the access token locally using a JWT token signature check and local token introspection.

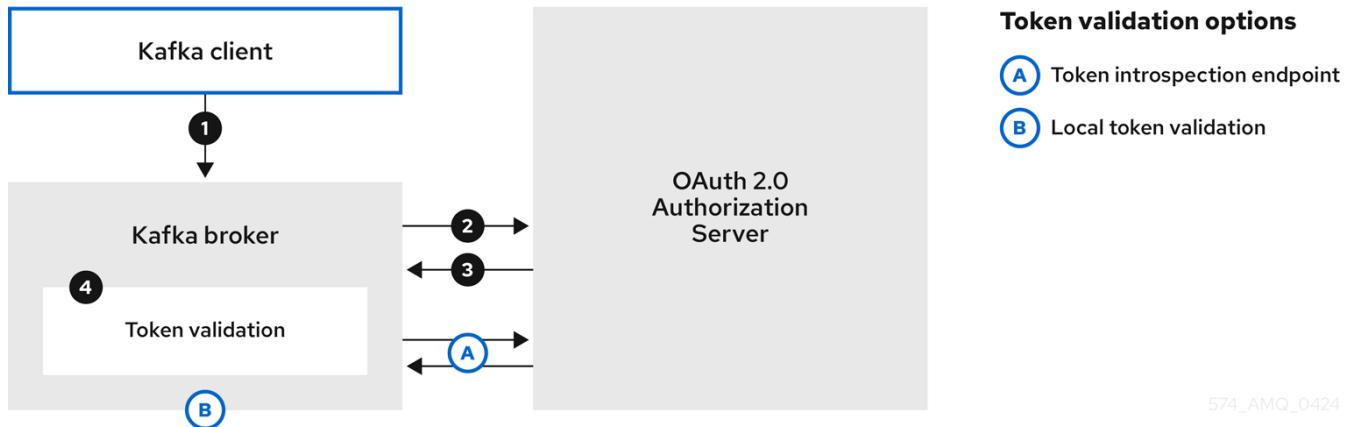
Fast local JWT token signature validation is suitable only for short-lived tokens as there is no check with the authorization server if a token has been revoked. Token expiration is written into the token, but revocation can happen at any time, so cannot be accounted for without contacting the authorization server. Any issued token would be considered valid until it expires.

WARNING

Example client authentication flows using the SASL **PLAIN** mechanism

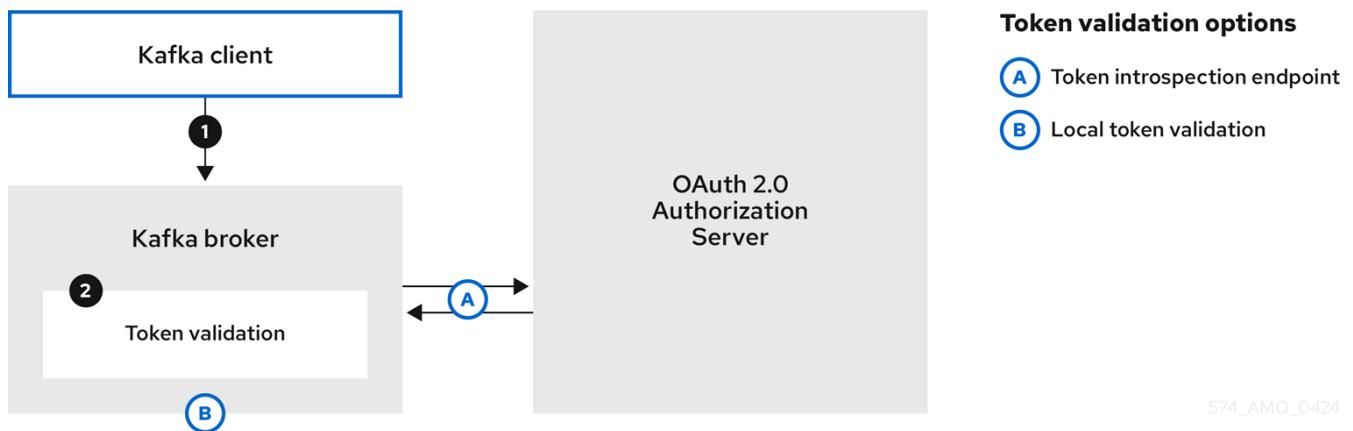
You can use the following communication flows for Kafka authentication using the OAuth **PLAIN** mechanism.

Client using a client ID and secret, with the broker obtaining the access token for the client



1. The Kafka client passes a `clientId` as a username and a `secret` as a password.
2. The Kafka broker uses a token endpoint to pass the `clientId` and `secret` to the authorization server.
3. The authorization server returns a fresh access token or an error if the client credentials are not valid.
4. The Kafka broker validates the token in one of the following ways:
 - a. If a token introspection endpoint is specified, the Kafka broker validates the access token by calling the endpoint on the authorization server. A session is established if the token validation is successful.
 - b. If local token introspection is used, a request is not made to the authorization server. The Kafka broker validates the access token locally using a JWT token signature check.

Client using a long-lived access token without a client ID and secret



1. The Kafka client passes a username and password. The password provides the value of an access token that was obtained manually and configured before running the client.
2. The password is passed with or without an `$accessToken:` string prefix depending on whether or not the Kafka broker listener is configured with a token endpoint for authentication.
 - a. If the token endpoint is configured, the password should be prefixed by `$accessToken:` to let the broker know that the password parameter contains an access token rather than a client secret. The Kafka broker interprets the username as the account username.

- b. If the token endpoint is not configured on the Kafka broker listener (enforcing a [no-client-credentials mode](#)), the password should provide the access token without the prefix. The Kafka broker interprets the username as the account username. In this mode, the client doesn't use a client ID and secret, and the [password](#) parameter is always interpreted as a raw access token.
3. The Kafka broker validates the token in one of the following ways:
- a. If a token introspection endpoint is specified, the Kafka broker validates the access token by calling the endpoint on the authorization server. A session is established if token validation is successful.
 - b. If local token introspection is used, there is no request made to the authorization server. Kafka broker validates the access token locally using a JWT token signature check.

15.2.4. Re-authenticating sessions

Configure [oauth](#) listeners to use Kafka *session re-authentication* for OAuth 2.0 sessions between Kafka clients and Kafka. This mechanism enforces the expiry of an authenticated session between the client and the broker after a defined period of time. When a session expires, the client immediately starts a new session by reusing the existing connection rather than dropping it.

Session re-authentication is disabled by default. To enable it, you set a time value for [maxSecondsWithoutReauthentication](#) in the [oauth](#) listener configuration. The same property is used to configure session re-authentication for [OAUTHBEARER](#) and [PLAIN](#) authentication. For an example configuration, see [Configuring OAuth 2.0 authentication on listeners](#).

Session re-authentication must be supported by the Kafka client libraries used by the client.

Session re-authentication can be used with *fast local JWT* or *introspection endpoint* token validation.

Client re-authentication

When the broker's authenticated session expires, the client must re-authenticate to the existing session by sending a new, valid access token to the broker, without dropping the connection.

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

Session re-authentication also applies to refresh tokens, if used. When the session expires, the client refreshes the access token by using its refresh token. The client then uses the new access token to re-authenticate to the existing session.

Session expiry

When session re-authentication is configured, session expiry works differently for [OAUTHBEARER](#) and [PLAIN](#) authentication.

For [OAUTHBEARER](#) and [PLAIN](#), using the client ID and secret method:

- The broker's authenticated session will expire at the configured

`maxSecondsWithoutReauthentication`.

- The session will expire earlier if the access token expires before the configured time.

For `PLAIN` using the long-lived access token method:

- The broker's authenticated session will expire at the configured `maxSecondsWithoutReauthentication`.
- Re-authentication will fail if the access token expires before the configured time. Although session re-authentication is attempted, `PLAIN` has no mechanism for refreshing tokens.

If `maxSecondsWithoutReauthentication` is *not* configured, `OAUTHBEARER` and `PLAIN` clients can remain connected to brokers indefinitely, without needing to re-authenticate. Authenticated sessions do not end with access token expiry. However, this can be considered when configuring authorization, for example, by using `keycloak` authorization or installing a custom authorizer.

15.2.5. Example: Enabling OAuth 2.0 authentication

This example shows how to configure client access to a Kafka cluster using OAuth 2.0 authentication. The procedures describe the configuration required to set up OAuth 2.0 authentication on Kafka listeners, Kafka Java clients, and Kafka components.

Setting up OAuth 2.0 authentication on listeners

Configure Kafka listeners so that they are enabled to use OAuth 2.0 authentication using an authorization server.

We advise using OAuth 2.0 over an encrypted interface through a listener with `tls: true`. Plain listeners are not recommended.

If the authorization server is using certificates signed by the trusted CA and matching the OAuth 2.0 server hostname, TLS connection works using the default settings. Otherwise, you may need to configure the truststore with proper certificates or disable the certificate hostname validation.

For more information on the configuration of OAuth 2.0 authentication for Kafka broker listeners, see the [KafkaListenerAuthenticationOAuth schema reference](#).

Prerequisites

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

Procedure

1. Specify a listener in the `Kafka` resource with an `oauth` authentication type.

Example listener configuration with OAuth 2.0 authentication

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

```

# ...
listeners:
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication:
      type: oauth
  - name: external3
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: oauth
#

```

2. Configure the OAuth listener depending on the authorization server and validation type:
 - [Configuring fast local JWT token validation](#)
 - [Configuring token validation using an introspection endpoint](#)
 - [Including additional configuration options](#)
3. Apply the changes to the [Kafka](#) configuration.
4. Check the update in the logs or by watching the pod state transitions:

```

kubectl logs -f ${POD_NAME} -c ${CONTAINER_NAME}
kubectl get pod -w

```

The rolling update configures the brokers to use OAuth 2.0 authentication.

What to do next

- [Configure your Kafka clients to use OAuth 2.0](#)

Setting up OAuth 2.0 on Kafka Java clients

Configure Kafka producer and consumer APIs to use OAuth 2.0 for interaction with Kafka brokers. Add a callback plugin to your client [pom.xml](#) file, then configure your client for OAuth 2.0.

How you configure the authentication properties depends on the authentication method you are using to access the OAuth 2.0 authorization server. In this procedure, the properties are specified in a properties file, then loaded into the client configuration.

Prerequisites

- Strimzi and Kafka are running
- An OAuth 2.0 authorization server is deployed and configured for OAuth access to Kafka brokers
- Kafka brokers are configured for OAuth 2.0

Procedure

1. Add the client library with OAuth 2.0 support to the `pom.xml` file for the Kafka client:

```
<dependency>
  <groupId>io.strimzi</groupId>
  <artifactId>kafka-oauth-client</artifactId>
  <version>0.15.0</version>
</dependency>
```

2. Configure the client depending on the OAuth 2.0 authentication method:

- [Example client credentials configuration](#)
- [Example password grants configuration](#)
- [Example access token configuration](#)
- [Example refresh token configuration](#)

For example, specify the properties for the authentication method in a `client.properties` file.

3. Input the client properties for OAUTH 2.0 authentication into the Java client code.

Example showing input of client properties

```
Properties props = new Properties();
try (FileReader reader = new FileReader("client.properties",
StandardCharsets.UTF_8)) {
    props.load(reader);
}
```

4. Verify that the Kafka client can access the Kafka brokers.

Setting up OAuth 2.0 on Kafka components

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

You can configure OAuth 2.0 authentication for the following components:

- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

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

Before you start

For more information on the configuration of OAuth 2.0 authentication for Kafka components, see the [KafkaClientAuthenticationOAuth schema reference](#). The schema reference includes examples of configuration options.

Prerequisites

- Strimzi and Kafka are running
- An OAuth 2.0 authorization server is deployed and configured for OAuth access to Kafka brokers
- Kafka brokers are configured for OAuth 2.0

Procedure

1. Create a client secret and mount it to the component as an environment variable.

For example, here we are creating a client **Secret** for the Kafka Bridge:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Secret
metadata:
  name: my-bridge-oauth
type: Opaque
data:
  clientSecret: MGQ10TRmMzYtZTl1ZS00MDY2LWI50GEtMTM5MzM2Njd1ZjQw ①
```

① The **clientSecret** key must be in base64 format.

2. Create or edit the resource for the Kafka component so that OAuth 2.0 authentication is configured for the authentication property.

For OAuth 2.0 authentication, you can use the following options:

- Client ID and secret
- Client ID and refresh token
- Access token
- Username and password
- TLS

For example, here OAuth 2.0 is assigned to the Kafka Bridge client using a client ID and secret, and TLS:

Example OAuth 2.0 authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: oauth ①
    tokenEndpointUri: https://<auth_server_address>/<path_to_token_endpoint> ②
    clientId: kafka-bridge
```

```

clientSecret:
  secretName: my-bridge-oauth
  key: clientSecret
  tlsTrustedCertificates: ③
    - secretName: oauth-server-cert
      pattern: "*.crt"

```

① Authentication type set to `oauth`.

② URI of the token endpoint for authentication.

③ Certificates stored in X.509 format within the specified secrets for TLS connection to the authorization server.

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

Additional configuration options

```

# ...
spec:
# ...
authentication:
# ...
  disableTlsHostnameVerification: true ①
  checkAccessTokenType: false ②
  accessTokenIsJwt: false ③
  scope: any ④
  audience: kafka ⑤
  connectTimeoutSeconds: 60 ⑥
  readTimeoutSeconds: 60 ⑦
  httpRetries: 2 ⑧
  httpRetryPauseMs: 300 ⑨
  includeAcceptHeader: false ⑩

```

① (Optional) Disable TLS hostname verification. Default is `false`.

② If the authorization server does not return a `typ` (type) claim inside the JWT token, you can apply `checkAccessTokenType: false` to skip the token type check. Default is `true`.

③ If you are using opaque tokens, you can apply `accessTokenIsJwt: false` so that access tokens are not treated as JWT tokens.

④ (Optional) The `scope` for requesting the token from the token endpoint. An authorization server may require a client to specify the scope. In this case it is `any`.

⑤ (Optional) The `audience` for requesting the token from the token endpoint. An authorization server may require a client to specify the audience. In this case it is `kafka`.

⑥ (Optional) The connect timeout in seconds when connecting to the authorization server. The default value is 60.

⑦ (Optional) The read timeout in seconds when connecting to the authorization server. The default value is 60.

- ⑧ (Optional) The maximum number of times to retry a failed HTTP request to the authorization server. The default value is `0`, meaning that no retries are performed. To use this option effectively, consider reducing the timeout times for the `connectTimeoutSeconds` and `readTimeoutSeconds` options. However, note that retries may prevent the current worker thread from being available to other requests, and if too many requests stall, it could make the Kafka broker unresponsive.
- ⑨ (Optional) The time to wait before attempting another retry of a failed HTTP request to the authorization server. By default, this time is set to zero, meaning that no pause is applied. This is because many issues that cause failed requests are per-request network glitches or proxy issues that can be resolved quickly. However, if your authorization server is under stress or experiencing high traffic, you may want to set this option to a value of 100 ms or more to reduce the load on the server and increase the likelihood of successful retries.
- ⑩ (Optional) Some authorization servers have issues with client sending `Accept: application/json` header. By setting `includeAcceptHeader: false` the header will not be sent. Default is `true`.

3. Apply the changes to the resource configuration of the component.
4. Check the update in the logs or by watching the pod state transitions:

```
kubectl logs -f ${POD_NAME} -c ${CONTAINER_NAME}
kubectl get pod -w
```

The rolling updates configure the component for interaction with Kafka brokers using OAuth 2.0 authentication.

15.3. Using OAuth 2.0 token-based authorization

Strimzi supports the use of OAuth 2.0 token-based authorization through [Keycloak Authorization Services](#), which lets you manage security policies and permissions centrally.

Security policies and permissions defined in Keycloak grant access to Kafka resources. Users and clients are matched against policies that permit access to perform specific actions on Kafka brokers.

Kafka allows all users full access to brokers by default, but also provides the `AclAuthorizer` and `StandardAuthorizer` plugins to configure authorization based on Access Control Lists (ACLs). The ACL rules managed by these plugins are used to grant or deny access to resources based on `username`, and these rules are stored within the Kafka cluster itself.

However, OAuth 2.0 token-based authorization with Keycloak offers far greater flexibility on how you wish to implement access control to Kafka brokers. In addition, you can configure your Kafka brokers to use OAuth 2.0 authorization and ACLs.

15.3.1. Example: Enabling OAuth 2.0 authorization

This example procedure shows how to configure Kafka to use OAuth 2.0 authorization using Keycloak Authorization Services. To enable OAuth 2.0 authorization using Keycloak, configure the

Kafka resource to use `keycloak` authorization and specify the properties required to access the authorization server and Keycloak Authorization Services.

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

A Keycloak *authorizer* (`KeycloakAuthorizer`) is provided with Strimzi. The authorizer fetches a list of granted permissions from the authorization server as needed, and enforces authorization locally on Kafka, making rapid authorization decisions for each client request.

Before you begin

Consider the access you require or want to limit for certain users. You can use a combination of Keycloak *groups*, *roles*, *clients*, and *users* to configure access in Keycloak.

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

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

NOTE

`Super users` always have unconstrained access to Kafka regardless of the authorization implemented.

Prerequisites

- Strimzi must be configured to use OAuth 2.0 with Keycloak for `token-based authentication`. You use the same Keycloak server endpoint when you set up authorization.
- OAuth 2.0 authentication must be configured with the `maxSecondsWithoutReauthentication` option to enable re-authentication.

Procedure

1. Access the Keycloak Admin Console or use the Keycloak Admin CLI to enable Authorization Services for the OAuth 2.0 client for Kafka you created when setting up OAuth 2.0 authentication.
2. Use Authorization Services to define resources, authorization scopes, policies, and permissions for the client.
3. Bind the permissions to users and clients by assigning them roles and groups.
4. Configure the `kafka` resource to use `keycloak` authorization, and to be able to access the authorization server and Authorization Services.

Example OAuth 2.0 authorization configuration

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

```

metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    authorization:
      type: keycloak ①
      tokenEndpointUri: <https://<auth-server-
address>/realms/external/protocol/openid-connect/token> ②
      clientId: kafka ③
      delegateToKafkaAcls: false ④
      disableTlsHostnameVerification: false ⑤
      superUsers: ⑥
        - CN=client_1
        - user_2
        - CN=client_3
      tlsTrustedCertificates: ⑦
        - secretName: oauth-server-cert
          pattern: "*.crt"
      grantsRefreshPeriodSeconds: 60 ⑧
      grantsRefreshPoolSize: 5 ⑨
      grantsMaxIdleSeconds: 300 ⑩
      grantsGcPeriodSeconds: 300 ⑪
      grantsAlwaysLatest: false ⑫
      connectTimeoutSeconds: 60 ⑬
      readTimeoutSeconds: 60 ⑭
      httpRetries: 2 ⑮
      enableMetrics: false ⑯
      includeAcceptHeader: false ⑰
    #...

```

① Type `keycloak` enables Keycloak authorization.

② URI of the Keycloak token endpoint. For production, always use `https://` urls. When you configure token-based `oauth` authentication, you specify a `jwksEndpointUri` as the URI for local JWT validation. The hostname for the `tokenEndpointUri` URI must be the same.

③ The client ID of the OAuth 2.0 client definition in Keycloak that has Authorization Services enabled. Typically, `kafka` is used as the ID.

④ (Optional) Delegate authorization to Kafka `AclAuthorizer` and `StandardAuthorizer` if access is denied by Keycloak Authorization Services policies. Default is `false`.

⑤ (Optional) Disable TLS hostname verification. Default is `false`.

⑥ (Optional) Designated super users.

⑦ (Optional) Certificates stored in X.509 format within the specified secrets for TLS connection to the authorization server.

⑧ (Optional) The time between two consecutive grants refresh runs. That is the maximum time for active sessions to detect any permissions changes for the user on Keycloak. The default value is 60.

- ⑨ (Optional) The number of threads to use to refresh (in parallel) the grants for the active sessions. The default value is 5.
- ⑩ (Optional) The time, in seconds, after which an idle grant in the cache can be evicted. The default value is 300.
- ⑪ (Optional) The time, in seconds, between consecutive runs of a job that cleans stale grants from the cache. The default value is 300.
- ⑫ (Optional) Controls whether the latest grants are fetched for a new session. When enabled, grants are retrieved from Keycloak and cached for the user. The default value is `false`.
- ⑬ (Optional) The connect timeout in seconds when connecting to the Keycloak token endpoint. The default value is 60.
- ⑭ (Optional) The read timeout in seconds when connecting to the Keycloak token endpoint. The default value is 60.
- ⑮ (Optional) The maximum number of times to retry (without pausing) a failed HTTP request to the authorization server. The default value is `0`, meaning that no retries are performed. To use this option effectively, consider reducing the timeout times for the `connectTimeoutSeconds` and `readTimeoutSeconds` options. However, note that retries may prevent the current worker thread from being available to other requests, and if too many requests stall, it could make Kafka unresponsive.
- ⑯ (Optional) Enable or disable OAuth metrics. The default value is `false`.
- ⑰ (Optional) Some authorization servers have issues with client sending `Accept: application/json` header. By setting `includeAcceptHeader: false` the header will not be sent. Default is `true`.

5. Apply the changes to the `Kafka` configuration.
6. Check the update in the logs or by watching the pod state transitions:

```
kubectl logs -f ${POD_NAME} -c kafka
kubectl get pod -w
```

The rolling update configures the brokers to use OAuth 2.0 authorization.

7. Verify the configured permissions by accessing Kafka brokers as clients or users with specific roles, ensuring they have the necessary access and do not have unauthorized access.

15.4. Setting up permissions in Keycloak

When using Keycloak as the OAuth 2.0 authorization server, Kafka permissions are granted to user accounts or service accounts using authorization permissions. To grant permissions to access Kafka, create an *OAuth client specification* in Keycloak that maps the authorization models of Keycloak Authorization Services and Kafka.

15.4.1. Kafka and Keycloak authorization models

Kafka and Keycloak Authorization Services use different authorization models.

- Kafka's authorization model uses *resource types* and *operations* to describe ACLs for the *user*
- The Keycloak Authorization Services model has four concepts for defining and granting permissions:
 - *resources*
 - *authorization scopes*
 - *policies*
 - *permissions*

Kafka authorization model

When a Kafka client performs an action on a broker, the broker uses the configured **KeycloakAuthorizer** to check the client's permissions, based on the action and resource type.

Each resource type has a set of available permissions for operations. For example, the **Topic** resource type has **Create** and **Write** permissions among others.

Refer to the [Kafka authorization model](#) in the Kafka documentation for the full list of resources and permissions.

Keycloak Authorization Services model

The Keycloak Authorization Services model defines authorized actions.

Resources

Resources are matched with permitted actions. A resource might be an individual topic, for example, or all topics with names starting with the same prefix. A resource definition is associated with a set of available authorization scopes, which represent a set of all actions available on the resource. Often, only a subset of these actions is actually permitted.

Authorization scopes

An authorization scope is a set of all the available actions on a specific resource definition. When you define a new resource, you add scopes from the set of all scopes.

Policies

A policy is an authorization rule that uses criteria to match against a list of accounts. Policies can match:

- *Service accounts* based on client ID or roles
- *User accounts* based on username, groups, or roles.

Permissions

A permission grants a subset of authorization scopes on a specific resource definition to a set of users.

15.4.2. Mapping authorization models

The Kafka authorization model is used as a basis for defining the Keycloak roles and resources that control access to Kafka.

To grant Kafka permissions to user accounts or service accounts, you first create an *OAuth client specification* in Keycloak for the Kafka cluster. You then specify Keycloak Authorization Services rules on the client. Typically, the client ID of the OAuth client that represents the Kafka cluster is `kafka`. The [example configuration files](#) provided with Strimzi use `kafka` as the OAuth client id.

If you have multiple Kafka clusters, you can use a single OAuth client (`kafka`) for all of them. This gives you a single, unified space in which to define and manage authorization rules. However, you can also use different OAuth client ids (for example, `my-cluster-kafka` or `cluster-dev-kafka`) and define authorization rules for each cluster within each client configuration.

NOTE The `kafka` client definition must have the **Authorization Enabled** option enabled in the Keycloak Admin Console.

All permissions exist within the scope of the `kafka` client. If you have different Kafka clusters configured with different OAuth client IDs, they each need a separate set of permissions even though they're part of the same Keycloak realm.

When the Kafka client uses OAUTHBEARER authentication, the Keycloak authorizer ([KeycloakAuthorizer](#)) uses the access token of the current session to retrieve a list of grants from the Keycloak server. To grant permissions, the authorizer evaluates the grants list (received and cached) from Keycloak Authorization Services based on the access token owner's policies and permissions.

Uploading authorization scopes for Kafka permissions

An initial Keycloak configuration usually involves uploading authorization scopes to create a list of all possible actions that can be performed on each Kafka resource type. This step is performed once only, before defining any permissions. You can add authorization scopes manually instead of uploading them.

Authorization scopes should contain the following Kafka permissions regardless of the resource type:

- `Create`
- `Write`
- `Read`
- `Delete`
- `Describe`
- `Alter`
- `DescribeConfigs`
- `AlterConfigs`

- `ClusterAction`
- `IdempotentWrite`

If you're certain you won't need a permission (for example, `IdempotentWrite`), you can omit it from the list of authorization scopes. However, that permission won't be available to target on Kafka resources.

NOTE The `All` permission is not supported.

Resource patterns for permissions checks

Resource patterns are used for pattern matching against the targeted resources when performing permission checks. The general pattern format is `<resource_type>:<pattern_name>`.

The resource types mirror the Kafka authorization model. The pattern allows for two matching options:

- Exact matching (when the pattern does not end with `*`)
- Prefix matching (when the pattern ends with `*`)

Example patterns for resources

```
Topic:my-topic  
Topic:orders-*  
Group:orders-*  
Cluster:*
```

Additionally, the general pattern format can be prefixed by `kafka-cluster:<cluster_name>` followed by a comma, where `<cluster_name>` refers to the `metadata.name` in the Kafka custom resource.

Example patterns for resources with cluster prefix

```
kafka-cluster:my-cluster,Topic:*
```

```
kafka-cluster:*,Group:b_*
```

When the `kafka-cluster` prefix is missing, it is assumed to be `kafka-cluster:*`.

When defining a resource, you can associate it with a list of possible authorization scopes which are relevant to the resource. Set whatever actions make sense for the targeted resource type.

Though you may add any authorization scope to any resource, only the scopes supported by the resource type are considered for access control.

Policies for applying access permission

Policies are used to target permissions to one or more user accounts or service accounts. Targeting can refer to:

- Specific user or service accounts
- Realm roles or client roles

- User groups

A policy is given a unique name and can be reused to target multiple permissions to multiple resources.

Permissions to grant access

Use fine-grained permissions to pull together the policies, resources, and authorization scopes that grant access to users.

The name of each permission should clearly define which permissions it grants to which users. For example, **Dev Team B can read from topics starting with x**.

15.4.3. Permissions for common Kafka operations

The following examples demonstrate the user permissions required for performing common operations on Kafka.

Create a topic

To create a topic, the **Create** permission is required for the specific topic, or for **Cluster:kafka-cluster**.

```
bin/kafka-topics.sh --create --topic my-topic \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

List topics

If a user has the **Describe** permission on a specified topic, the topic is listed.

```
bin/kafka-topics.sh --list \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

Display topic details

To display a topic's details, **Describe** and **DescribeConfigs** permissions are required on the topic.

```
bin/kafka-topics.sh --describe --topic my-topic \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

Produce messages to a topic

To produce messages to a topic, **Describe** and **Write** permissions are required on the topic.

If the topic hasn't been created yet, and topic auto-creation is enabled, the permissions to create a topic are required.

```
bin/kafka-console-producer.sh --topic my-topic \  
--bootstrap-server my-cluster-kafka-bootstrap:9092  
--producer.config=/tmp/config.properties
```

Consume messages from a topic

To consume messages from a topic, **Describe** and **Read** permissions are required on the topic. Consuming from the topic normally relies on storing the consumer offsets in a consumer group, which requires additional **Describe** and **Read** permissions on the consumer group.

Two **resources** are needed for matching. For example:

```
Topic:my-topic  
Group:my-group-*
```

```
bin/kafka-console-consumer.sh --topic my-topic --group my-group-1 --from-beginning \  
--bootstrap-server my-cluster-kafka-bootstrap:9092 --consumer.config  
/tmp/config.properties
```

Produce messages to a topic using an idempotent producer

As well as the permissions for producing to a topic, an additional **IdempotentWrite** permission is required on the **Cluster:kafka-cluster** resource.

Two **resources** are needed for matching. For example:

```
Topic:my-topic  
Cluster:kafka-cluster
```

```
bin/kafka-console-producer.sh --topic my-topic \  
--bootstrap-server my-cluster-kafka-bootstrap:9092  
--producer.config=/tmp/config.properties --producer-property enable.idempotence=true  
--request-required-acks -1
```

List consumer groups

When listing consumer groups, only the groups on which the user has the **Describe** permissions are returned. Alternatively, if the user has the **Describe** permission on the **Cluster:kafka-cluster**, all the consumer groups are returned.

```
bin/kafka-consumer-groups.sh --list \  
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command  
-config=/tmp/config.properties
```

Display consumer group details

To display a consumer group's details, the **Describe** permission is required on the group and the topics associated with the group.

```
bin/kafka-consumer-groups.sh --describe --group my-group-1 \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

Change topic configuration

To change a topic's configuration, the **Describe** and **Alter** permissions are required on the topic.

```
bin/kafka-topics.sh --alter --topic my-topic --partitions 2 \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

Display Kafka broker configuration

In order to use **kafka-configs.sh** to get a broker's configuration, the **DescribeConfigs** permission is required on the **Cluster:kafka-cluster**.

```
bin/kafka-configs.sh --entity-type brokers --entity-name 0 --describe --all \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

Change Kafka broker configuration

To change a Kafka broker's configuration, **DescribeConfigs** and **AlterConfigs** permissions are required on **Cluster:kafka-cluster**.

```
bin/kafka-configs --entity-type brokers --entity-name 0 --alter --add-config
log.cleaner.threads=2 \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

Delete a topic

To delete a topic, the **Describe** and **Delete** permissions are required on the topic.

```
bin/kafka-topics.sh --delete --topic my-topic \
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command
-config=/tmp/config.properties
```

Select a lead partition

To run leader selection for topic partitions, the **Alter** permission is required on the **Cluster:kafka-cluster**.

```
bin/kafka-leader-election.sh --topic my-topic --partition 0 --election-type PREFERRED
```

```
/  
--bootstrap-server my-cluster-kafka-bootstrap:9092 --admin.config  
/tmp/config.properties
```

Reassign partitions

To generate a partition reassignment file, **Describe** permissions are required on the topics involved.

```
bin/kafka-reassign-partitions.sh --topics-to-move-json-file /tmp/topics-to-move.json  
--broker-list "0,1" --generate \  
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config  
/tmp/config.properties > /tmp/partition-reassignment.json
```

To execute the partition reassignment, **Describe** and **Alter** permissions are required on **Cluster:kafka-cluster**. Also, **Describe** permissions are required on the topics involved.

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-  
reassignment.json --execute \  
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config  
/tmp/config.properties
```

To verify partition reassignment, **Describe**, and **AlterConfigs** permissions are required on **Cluster:kafka-cluster**, and on each of the topics involved.

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-  
reassignment.json --verify \  
--bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config  
/tmp/config.properties
```

15.4.4. Example: Setting up Keycloak Authorization Services

If you are using OAuth 2.0 with Keycloak for token-based authentication, you can also use Keycloak to configure authorization rules to constrain client access to Kafka brokers. This example explains how to use Keycloak Authorization Services with **keycloak** authorization. Set up Keycloak Authorization Services to enforce access restrictions on Kafka clients. Keycloak Authorization Services use authorization scopes, policies and permissions to define and apply access control to resources.

Keycloak Authorization Services REST endpoints provide a list of granted permissions on resources for authenticated users. The list of grants (permissions) is fetched from the Keycloak server as the first action after an authenticated session is established by the Kafka client. The list is refreshed in the background so that changes to the grants are detected. Grants are cached and enforced locally on the Kafka broker for each user session to provide fast authorization decisions.

Strimzi provides [example configuration files](#). These include the following example files for setting up Keycloak:

kafka-ephemeral-oauth-single-keycloak-authz.yaml

An example **Kafka** custom resource configured for OAuth 2.0 token-based authorization using Keycloak. You can use the custom resource to deploy a Kafka cluster that uses **keycloak** authorization and token-based **oauth** authentication.

kafka-authz-realm.json

An example Keycloak realm configured with sample groups, users, roles and clients. You can import the realm into a Keycloak instance to set up fine-grained permissions to access Kafka.

If you want to try the example with Keycloak, use these files to perform the tasks outlined in this section in the order shown.

1. [Setting up permissions in Keycloak](#)
2. [Deploying a Kafka cluster with Keycloak authorization](#)
3. [Preparing TLS connectivity for a CLI Kafka client session](#)
4. [Checking authorized access to Kafka using a CLI Kafka client session](#)

Authentication

When you configure token-based **oauth** authentication, you specify a **jwksEndpointUri** as the URI for local JWT validation. When you configure **keycloak** authorization, you specify a **tokenEndpointUri** as the URI of the Keycloak token endpoint. The hostname for both URIs must be the same.

Targeted permissions with group or role policies

In Keycloak, confidential clients with service accounts enabled can authenticate to the server in their own name using a client ID and a secret. This is convenient for microservices that typically act in their own name, and not as agents of a particular user (like a web site). Service accounts can have roles assigned like regular users. They cannot, however, have groups assigned. As a consequence, if you want to target permissions to microservices using service accounts, you cannot use group policies, and should instead use role policies. Conversely, if you want to limit certain permissions only to regular user accounts where authentication with a username and password is required, you can achieve that as a side effect of using the group policies rather than the role policies. This is what is used in this example for permissions that start with **ClusterManager**. Performing cluster management is usually done interactively using CLI tools. It makes sense to require the user to log in before using the resulting access token to authenticate to the Kafka broker. In this case, the access token represents the specific user, rather than the client application.

Setting up permissions in Keycloak

Set up Keycloak, then connect to its Admin Console and add the preconfigured realm. Use the example **kafka-authz-realm.json** file to import the realm. You can check the authorization rules defined for the realm in the Admin Console. The rules grant access to the resources on the Kafka cluster configured to use the example Keycloak realm.

Prerequisites

- A running Kubernetes cluster.
- The Strimzi [examples/security/keycloak-authorization/kafka-authz-realm.json](#) file that contains the preconfigured realm.

Procedure

1. Install the Keycloak server using the Keycloak Operator as described in [Installing the Keycloak Operator](#) in the Keycloak documentation.
2. Wait until the Keycloak instance is running.
3. Get the external hostname to be able to access the Admin Console.

```
NS=sso  
kubectl get ingress keycloak -n $NS
```

In this example, we assume the Keycloak server is running in the `sso` namespace.

4. Get the password for the `admin` user.

```
kubectl get -n $NS pod keycloak-0 -o yaml | less
```

The password is stored as a secret, so get the configuration YAML file for the Keycloak instance to identify the name of the secret (`secretKeyRef.name`).

5. Use the name of the secret to obtain the clear text password.

```
SECRET_NAME=credential-keycloak  
kubectl get -n $NS secret $SECRET_NAME -o yaml | grep PASSWORD | awk '{print $2}' |  
base64 -D
```

In this example, we assume the name of the secret is `credential-keycloak`.

6. Log in to the Admin Console with the username `admin` and the password you obtained.

Use <https://HOSTNAME> to access the Kubernetes Ingress.

You can now upload the example realm to Keycloak using the Admin Console.

7. Click **Add Realm** to import the example realm.
8. Add the `examples/security/keycloak-authorization/kafka-authz-realm.json` file, and then click **Create**.

You now have `kafka-authz` as your current realm in the Admin Console.

The default view displays the **Master** realm.

9. In the Keycloak Admin Console, go to **Clients > kafka > Authorization > Settings** and check that **Decision Strategy** is set to **Affirmative**.

An affirmative policy means that at least one policy must be satisfied for a client to access the Kafka cluster.

10. In the Keycloak Admin Console, go to **Groups, Users, Roles** and **Clients** to view the realm

configuration.

Groups

Groups are used to create user groups and set user permissions. Groups are sets of users with a name assigned. They are used to compartmentalize users into geographical, organizational or departmental units. Groups can be linked to an LDAP identity provider. You can make a user a member of a group through a custom LDAP server admin user interface, for example, to grant permissions on Kafka resources.

Users

Users are used to create users. For this example, `alice` and `bob` are defined. `alice` is a member of the `ClusterManager` group and `bob` is a member of `ClusterManager-my-cluster` group. Users can be stored in an LDAP identity provider.

Roles

Roles mark users or clients as having certain permissions. Roles are a concept analogous to groups. They are usually used to *tag* users with organizational roles and have the requisite permissions. Roles cannot be stored in an LDAP identity provider. If LDAP is a requirement, you can use groups instead, and add Keycloak roles to the groups so that when users are assigned a group they also get a corresponding role.

Clients

Clients can have specific configurations. For this example, `kafka`, `kafka-cli`, `team-a-client`, and `team-b-client` clients are configured.

- The `kafka` client is used by Kafka brokers to perform the necessary OAuth 2.0 communication for access token validation. This client also contains the authorization services resource definitions, policies, and authorization scopes used to perform authorization on the Kafka brokers. The authorization configuration is defined in the `kafka` client from the **Authorization** tab, which becomes visible when **Authorization Enabled** is switched on from the **Settings** tab.
- The `kafka-cli` client is a public client that is used by the Kafka command line tools when authenticating with username and password to obtain an access token or a refresh token.
- The `team-a-client` and `team-b-client` clients are confidential clients representing services with partial access to certain Kafka topics.

11. In the Keycloak Admin Console, go to **Authorization > Permissions** to see the granted permissions that use the resources and policies defined for the realm.

For example, the `kafka` client has the following permissions:

```
Dev Team A can write to topics that start with x_ on any cluster
Dev Team B can read from topics that start with x_ on any cluster
Dev Team B can update consumer group offsets that start with x_ on any cluster
ClusterManager of my-cluster Group has full access to cluster config on my-cluster
ClusterManager of my-cluster Group has full access to consumer groups on my-cluster
ClusterManager of my-cluster Group has full access to topics on my-cluster
```

Dev Team A

The Dev Team A realm role can write to topics that start with `x_` on any cluster. This combines a resource called `Topic:x_*`, `Describe` and `Write` scopes, and the `Dev Team A` policy. The `Dev Team A` policy matches all users that have a realm role called `Dev Team A`.

Dev Team B

The Dev Team B realm role can read from topics that start with `x_` on any cluster. This combines `Topic:x_*`, `Group:x_*` resources, `Describe` and `Read` scopes, and the `Dev Team B` policy. The `Dev Team B` policy matches all users that have a realm role called `Dev Team B`. Matching users and clients have the ability to read from topics, and update the consumed offsets for topics and consumer groups that have names starting with `x_`.

Deploying a Kafka cluster with Keycloak authorization

Deploy a Kafka cluster configured to connect to the Keycloak server. Use the example `kafka-ephemeral-oauth-single-keycloak-authz.yaml` file to deploy the Kafka cluster as a `Kafka` custom resource. The example deploys a single-node Kafka cluster with `keycloak` authorization and `oauth` authentication.

Prerequisites

- The Keycloak authorization server is deployed to your Kubernetes cluster and loaded with the example realm.
- The Cluster Operator is deployed to your Kubernetes cluster.
- The Strimzi `examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml` custom resource.

Procedure

1. Use the hostname of the Keycloak instance you deployed to prepare a truststore certificate for Kafka brokers to communicate with the Keycloak server.

```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.pem
```

The certificate is required as Kubernetes `Ingress` is used to make a secure (HTTPS) connection.

Usually there is not one single certificate, but a certificate chain. You only have to provide the top-most issuer CA, which is listed last in the `/tmp/sso.pem` file. You can extract it manually or using the following commands:

Example command to extract the top CA certificate in a certificate chain

```
split -p "-----BEGIN CERTIFICATE-----" sso.pem sso-
for f in $(ls sso-*); do mv $f $f.pem; done
```

```
cp $(ls sso-* | sort -r | head -n 1) sso-ca.crt
```

NOTE

A trusted CA certificate is normally obtained from a trusted source, and not by using the `openssl` command.

2. Deploy the certificate to Kubernetes as a secret.

```
kubectl create secret generic oauth-server-cert --from-file=/tmp/sso-ca.crt -n $NS
```

3. Set the hostname as an environment variable

```
SSO_HOST=SSO-HOSTNAME
```

4. Create and deploy the example Kafka cluster.

```
cat examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml | sed -E 's#\${SSO_HOST}#"#$SSO_HOST#" | kubectl create -n $NS -f -
```

Preparing TLS connectivity for a CLI Kafka client session

Create a new pod for an interactive CLI session. Set up a truststore with a Keycloak certificate for TLS connectivity. The truststore is to connect to Keycloak and the Kafka broker.

Prerequisites

- The Keycloak authorization server is deployed to your Kubernetes cluster and loaded with the example realm.

In the Keycloak Admin Console, check the roles assigned to the clients are displayed in **Clients > Service Account Roles**.

- The Kafka cluster configured to connect with Keycloak is deployed to your Kubernetes cluster.

Procedure

1. Run a new interactive pod container using the Kafka image to connect to a running Kafka broker.

```
NS=sso
```

```
kubectl run -ti --restart=Never --image=quay.io/stimzi/kafka:0.42.0-kafka-3.7.1
kafka-cli -n $NS -- /bin/sh
```

NOTE

If `kubectl` times out waiting on the image download, subsequent attempts may result in an *AlreadyExists* error.

2. Attach to the pod container.

```
kubectl attach -ti kafka-cli -n $NS
```

3. Use the hostname of the Keycloak instance to prepare a certificate for client connection using TLS.

```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.pem
```

Usually there is not one single certificate, but a certificate chain. You only have to provide the top-most issuer CA, which is listed last in the `/tmp/sso.pem` file. You can extract it manually or using the following command:

Example command to extract the top CA certificate in a certificate chain

```
split -p "-----BEGIN CERTIFICATE-----" sso.pem sso-
for f in $(ls sso-*); do mv $f $f.pem; done
cp $(ls sso-* | sort -r | head -n 1) sso-ca.crt
```

NOTE

A trusted CA certificate is normally obtained from a trusted source, and not by using the `openssl` command.

4. Create a truststore for TLS connection to the Kafka brokers.

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias sso -storepass
$STOREPASS -import -file /tmp/sso-ca.crt -noprompt
```

5. Use the Kafka bootstrap address as the hostname of the Kafka broker and the `tls` listener port (9093) to prepare a certificate for the Kafka broker.

```
KAFKA_HOST_PORT=my-cluster-kafka-bootstrap:9093
STOREPASS=storepass
```

```
echo "Q" | openssl s_client -showcerts -connect $KAFKA_HOST_PORT 2>/dev/null | awk
' /BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/my-cluster-kafka.pem
```

The obtained `.pem` file is usually not one single certificate, but a certificate chain. You only have to provide the top-most issuer CA, which is listed last in the `/tmp/my-cluster-kafka.pem` file. You can extract it manually or using the following command:

Example command to extract the top CA certificate in a certificate chain

```
split -p "-----BEGIN CERTIFICATE-----" /tmp/my-cluster-kafka.pem kafka-
for f in $(ls kafka-*); do mv $f $f.pem; done
cp $(ls kafka-* | sort -r | head -n 1) my-cluster-kafka-ca.crt
```

NOTE

A trusted CA certificate is normally obtained from a trusted source, and not by using the `openssl` command. For this example we assume the client is running in a pod in the same namespace where the Kafka cluster was deployed. If the client is accessing the Kafka cluster from outside the Kubernetes cluster, you would have to first determine the bootstrap address. In that case you can also get the cluster certificate directly from the Kubernetes secret, and there is no need for `openssl`. For more information, see [Setting up client access to a Kafka cluster](#).

6. Add the certificate for the Kafka broker to the truststore.

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias my-cluster-kafka
-storepass $STOREPASS -import -file /tmp/my-cluster-kafka-ca.crt -noprompt
```

Keep the session open to check authorized access.

Checking authorized access to Kafka using a CLI Kafka client session

Check the authorization rules applied through the Keycloak realm using an interactive CLI session. Apply the checks using Kafka's example producer and consumer clients to create topics with user and service accounts that have different levels of access.

Use the `team-a-client` and `team-b-client` clients to check the authorization rules. Use the `alice` admin user to perform additional administrative tasks on Kafka.

The Kafka image used in this example contains Kafka producer and consumer binaries.

Prerequisites

- ZooKeeper and Kafka are running in the Kubernetes cluster to be able to send and receive messages.
- The [interactive CLI Kafka client session](#) is started.

[Apache Kafka download](#).

Setting up client and admin user configuration

1. Prepare a Kafka configuration file with authentication properties for the `team-a-client` client.

```
SSO_HOST=SSO-HOSTNAME

cat > /tmp/team-a-client.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
EOF
```

```

ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer OAuthBearerLoginModule required \
    oauth.client.id="team-a-client" \
    oauth.client.secret="team-a-client-secret" \
    oauth.ssl.truststore.location="/tmp/truststore.p12" \
    oauth.ssl.truststore.password="$STOREPASS" \
    oauth.ssl.truststore.type="PKCS12" \
    oauth.token.endpoint.uri="https://$SSO_HOST/realms/kafka-authz/protocol/openid-connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler
EOF

```

The SASL **OAUTHBEARER** mechanism is used. This mechanism requires a client ID and client secret, which means the client first connects to the Keycloak server to obtain an access token. The client then connects to the Kafka broker and uses the access token to authenticate.

2. Prepare a Kafka configuration file with authentication properties for the **team-b-client** client.

```

cat > /tmp/team-b-client.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer OAuthBearerLoginModule required \
    oauth.client.id="team-b-client" \
    oauth.client.secret="team-b-client-secret" \
    oauth.ssl.truststore.location="/tmp/truststore.p12" \
    oauth.ssl.truststore.password="$STOREPASS" \
    oauth.ssl.truststore.type="PKCS12" \
    oauth.token.endpoint.uri="https://$SSO_HOST/realms/kafka-authz/protocol/openid-connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler
EOF

```

3. Authenticate admin user **alice** by using **curl** and performing a password grant authentication to obtain a refresh token.

```

USERNAME=alice
PASSWORD=alice-password

```

```

GRANT_RESPONSE=$(curl -X POST "https://$SSO_HOST/realms/kafka-authz/protocol/openid-connect/token" -H 'Content-Type: application/x-www-form-

```

```

"urlencoded' -d
"grant_type=password&username=$USERNAME&password=$PASSWORD&client_id=kafka-
cli&scope=offline_access" -s -k)

REFRESH_TOKEN=$(echo $GRANT_RESPONSE | awk -F "refresh_token\":\"\"" '{printf $2}' |
awk -F "\"" '{printf $1}')

```

The refresh token is an offline token that is long-lived and does not expire.

4. Prepare a Kafka configuration file with authentication properties for the admin user `alice`.

```

cat > /tmp/alice.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModul
e required \
    oauth.refresh.token="$REFRESH_TOKEN" \
    oauth.client.id="kafka-cli" \
    oauth.ssl.truststore.location="/tmp/truststore.p12" \
    oauth.ssl.truststore.password="$STOREPASS" \
    oauth.ssl.truststore.type="PKCS12" \
    oauth.token.endpoint.uri="https://$SSO_HOST/realms/kafka-authz/protocol/openid-
connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLogi
nCallbackHandler
EOF

```

The `kafka-cli` public client is used for the `oauth.client.id` in the `sasl.jaas.config`. Since it's a public client it does not require a secret. The client authenticates with the refresh token that was authenticated in the previous step. The refresh token requests an access token behind the scenes, which is then sent to the Kafka broker for authentication.

Producing messages with authorized access

Use the `team-a-client` configuration to check that you can produce messages to topics that start with `a_` or `x_`.

1. Write to topic `my-topic`.

```

bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093
--topic my-topic \
--producer.config=/tmp/team-a-client.properties
First message

```

This request returns a `Not authorized to access topics: [my-topic]` error.

`team-a-client` has a **Dev Team A** role that gives it permission to perform any supported actions on topics that start with `a_`, but can only write to topics that start with `x_`. The topic named `my-topic` matches neither of those rules.

2. Write to topic `a_messages`.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic a_messages \  
--producer.config /tmp/team-a-client.properties  
First message  
Second message
```

Messages are produced to Kafka successfully.

3. Press CTRL+C to exit the CLI application.
4. Check the Kafka container log for a debug log of `Authorization GRANTED` for the request.

```
kubectl logs my-cluster-kafka-0 -f -n $NS
```

Consuming messages with authorized access

Use the `team-a-client` configuration to consume messages from topic `a_messages`.

1. Fetch messages from topic `a_messages`.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic a_messages \  
--from-beginning --consumer.config /tmp/team-a-client.properties
```

The request returns an error because the **Dev Team A** role for `team-a-client` only has access to consumer groups that have names starting with `a_`.

2. Update the `team-a-client` properties to specify the custom consumer group it is permitted to use.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic a_messages \  
--from-beginning --consumer.config /tmp/team-a-client.properties --group  
a_consumer_group_1
```

The consumer receives all the messages from the `a_messages` topic.

Administering Kafka with authorized access

The `team-a-client` is an account without any cluster-level access, but it can be used with some administrative operations.

1. List topics.

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties --list
```

The **a_messages** topic is returned.

2. List consumer groups.

```
bin/kafka-consumer-groups.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties --list
```

The **a_consumer_group_1** consumer group is returned.

Fetch details on the cluster configuration.

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties \
--entity-type brokers --describe --entity-default
```

The request returns an error because the operation requires cluster level permissions that **team-a-client** does not have.

Using clients with different permissions

Use the **team-b-client** configuration to produce messages to topics that start with **b_**.

1. Write to topic **a_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093
--topic a_messages \
--producer.config /tmp/team-b-client.properties
Message 1
```

This request returns a **Not authorized to access topics: [a_messages]** error.

2. Write to topic **b_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093
--topic b_messages \
--producer.config /tmp/team-b-client.properties
Message 1
Message 2
Message 3
```

Messages are produced to Kafka successfully.

3. Write to topic **x_messages**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--producer.config /tmp/team-b-client.properties  
Message 1
```

A **Not authorized to access topics: [x_messages]** error is returned, The **team-b-client** can only read from topic **x_messages**.

4. Write to topic **x_messages** using **team-a-client**.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--producer.config /tmp/team-a-client.properties  
Message 1
```

This request returns a **Not authorized to access topics: [x_messages]** error. The **team-a-client** can write to the **x_messages** topic, but it does not have a permission to create a topic if it does not yet exist. Before **team-a-client** can write to the **x_messages** topic, an admin *power user* must create it with the correct configuration, such as the number of partitions and replicas.

Managing Kafka with an authorized admin user

Use admin user **alice** to manage Kafka. **alice** has full access to manage everything on any Kafka cluster.

1. Create the **x_messages** topic as **alice**.

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command  
-config /tmp/alice.properties \  
--topic x_messages --create --replication-factor 1 --partitions 1
```

The topic is created successfully.

2. List all topics as **alice**.

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command  
-config /tmp/alice.properties --list  
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command  
-config /tmp/team-a-client.properties --list  
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command  
-config /tmp/team-b-client.properties --list
```

Admin user **alice** can list all the topics, whereas **team-a-client** and **team-b-client** can only list the topics they have access to.

The **Dev Team A** and **Dev Team B** roles both have **Describe** permission on topics that start with **x_**, but they cannot see the other team's topics because they do not have **Describe** permissions on

them.

3. Use the **team-a-client** to produce messages to the **x_messages** topic:

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--producer.config /tmp/team-a-client.properties  
Message 1  
Message 2  
Message 3
```

As **alice** created the **x_messages** topic, messages are produced to Kafka successfully.

4. Use the **team-b-client** to produce messages to the **x_messages** topic.

```
bin/kafka-console-producer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--producer.config /tmp/team-b-client.properties  
Message 4  
Message 5
```

This request returns a **Not authorized to access topics: [x_messages]** error.

5. Use the **team-b-client** to consume messages from the **x_messages** topic:

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--from-beginning --consumer.config /tmp/team-b-client.properties --group  
x_consumer_group_b
```

The consumer receives all the messages from the **x_messages** topic.

6. Use the **team-a-client** to consume messages from the **x_messages** topic.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--from-beginning --consumer.config /tmp/team-a-client.properties --group  
x_consumer_group_a
```

This request returns a **Not authorized to access topics: [x_messages]** error.

7. Use the **team-a-client** to consume messages from a consumer group that begins with **a_**.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--from-beginning --consumer.config /tmp/team-a-client.properties --group  
a_x_consumer_group_a
```

```
a_consumer_group_a
```

This request returns a `Not authorized to access topics: [x_messages]` error.

Dev Team A has no `Read` access on topics that start with a `x_`.

8. Use `alice` to produce messages to the `x_messages` topic.

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093  
--topic x_messages \  
--from-beginning --consumer.config /tmp/alice.properties
```

Messages are produced to Kafka successfully.

`alice` can read from or write to any topic.

9. Use `alice` to read the cluster configuration.

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command  
-config /tmp/alice.properties \  
--entity-type brokers --describe --entity-default
```

The cluster configuration for this example is empty.

Chapter 16. Managing TLS certificates

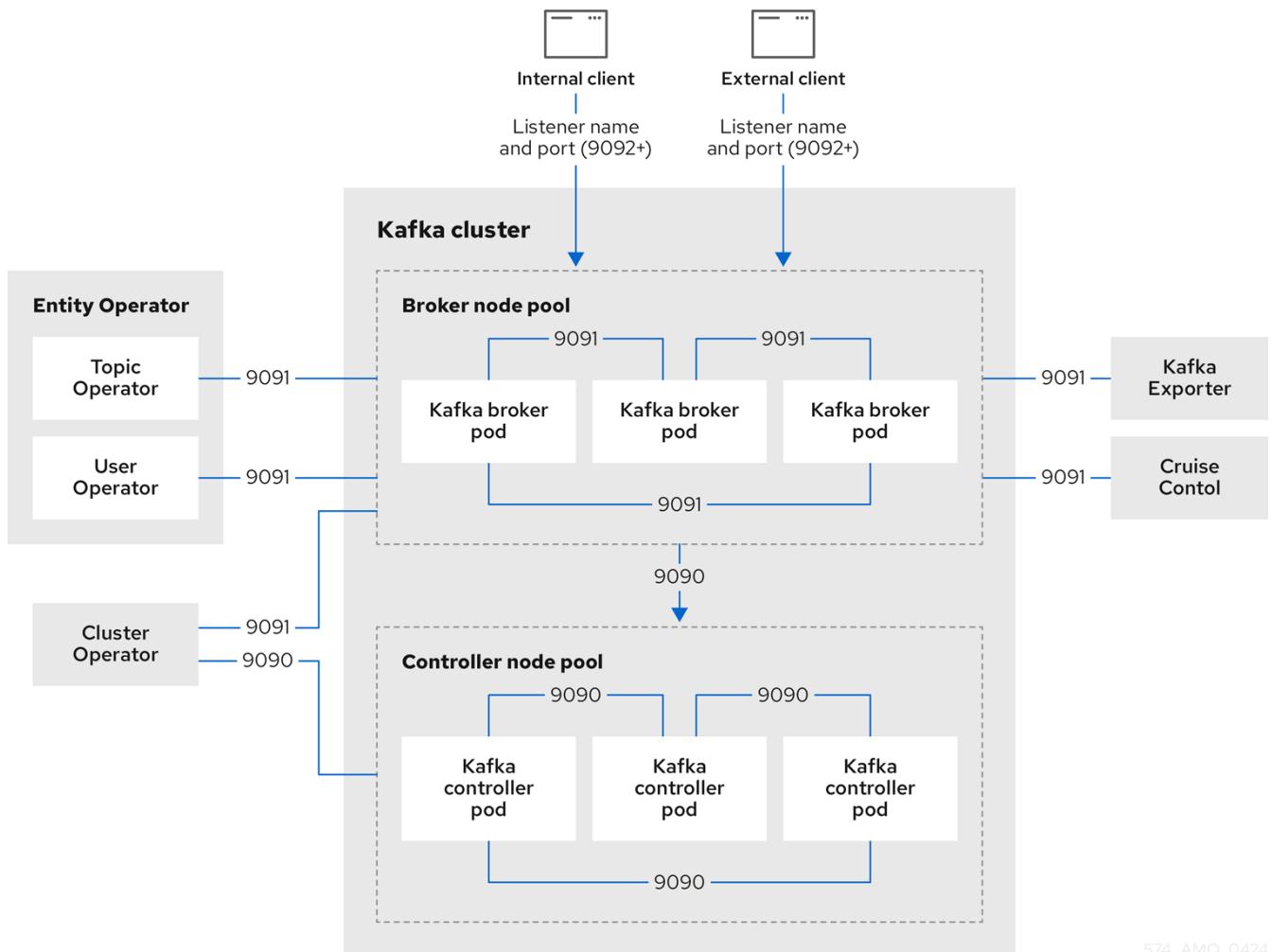
Strimzi supports TLS for encrypted communication between Kafka and Strimzi components.

Strimzi establishes encrypted TLS connections for communication between the following components when using Kafka in KRaft mode:

- Kafka brokers
- Kafka controllers
- Kafka brokers and controllers
- Strimzi operators and Kafka
- Cruise Control and Kafka brokers
- Kafka Exporter and Kafka brokers

Connections between clients and Kafka brokers use listeners that you must configure to use TLS-encrypted communication. You configure these listeners in the [Kafka](#) custom resource and each listener name and port number must be unique within the cluster. Communication between Kafka brokers and Kafka clients is encrypted according to how the `tls` property is configured for the listener. For more information, see [Setting up client access to a Kafka cluster](#).

The following diagram shows the connections for secure communication.



574_AMQ_0424

Figure 6. KRaft-based Kafka communication secured by TLS encryption

The ports shown in the diagram are used as follows:

Control plane listener (9090)

The internal control plane listener on port 9090 facilitates interbroker communication between Kafka controllers and broker-to-controller communication. Additionally, the Cluster Operator communicates with the controllers through the listener. This listener is not accessible to Kafka clients.

Replication listener (9091)

Data replication between brokers, as well as internal connections to the brokers from Strimzi operators, Cruise Control, and the Kafka Exporter, use the replication listener on port 9091. This listener is not accessible to Kafka clients.

Listeners for client connections (9092 or higher)

For TLS-encrypted communication (through configuration of the listener), internal and external clients connect to Kafka brokers. External clients (producers and consumers) connect to the Kafka brokers through the advertised listener port.

IMPORTANT

When configuring listeners for client access to brokers, you can use port 9092 or higher (9093, 9094, and so on), but with a few exceptions. The

listeners cannot be configured to use the ports reserved for interbroker communication (9090 and 9091), Prometheus metrics (9404), and JMX (Java Management Extensions) monitoring (9999).

If you are using ZooKeeper for cluster management, there are TLS connections between ZooKeeper and Kafka brokers and Strimzi operators.

The following diagram shows the connections for secure communication when using ZooKeeper.

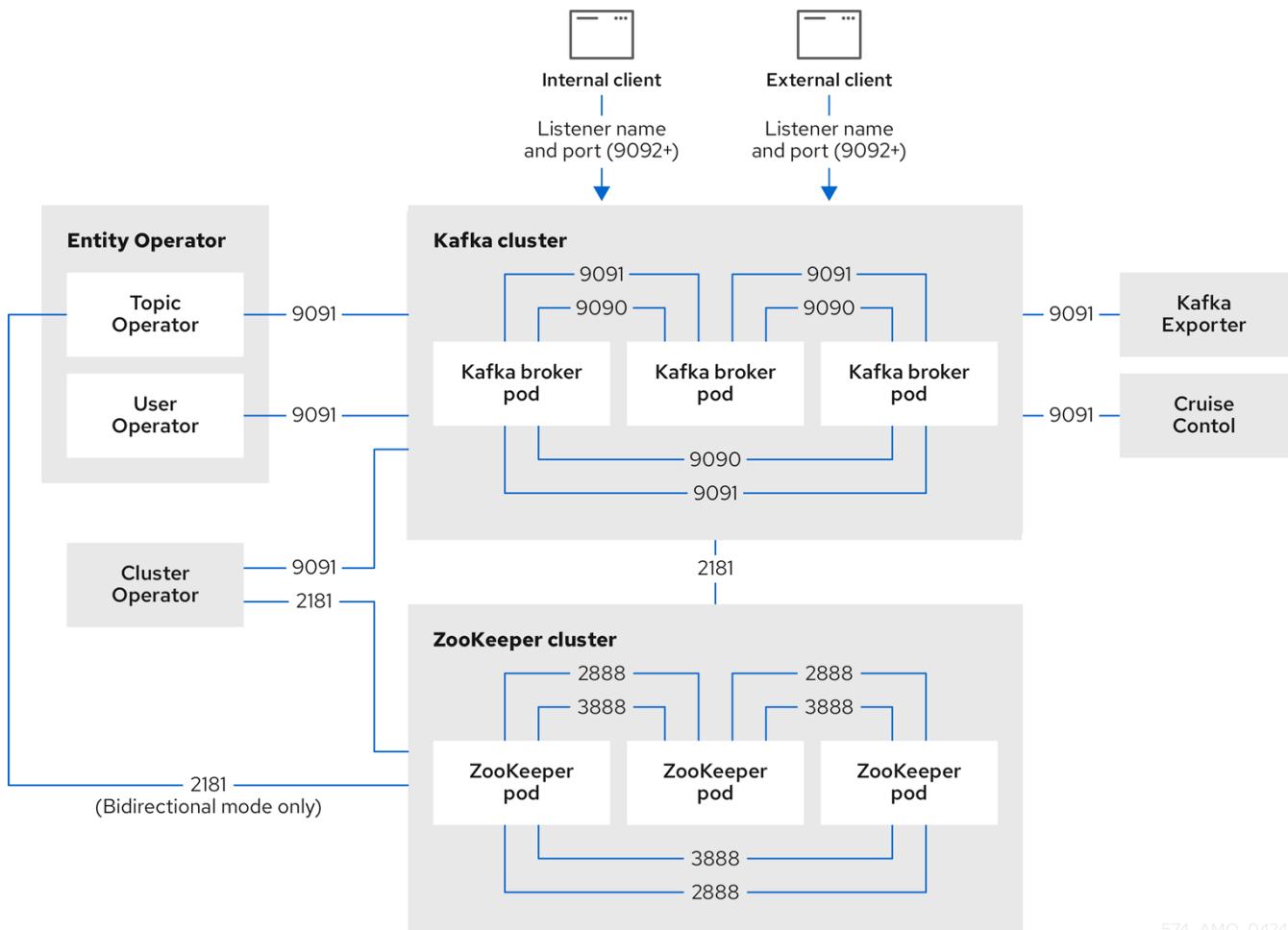


Figure 7. Kafka and ZooKeeper communication secured by TLS encryption

The ZooKeeper ports are used as follows:

ZooKeeper Port (2181)

ZooKeeper port for connection to Kafka brokers. Additionally, the Cluster Operator communicates with ZooKeeper through this port.

ZooKeeper internodal communication port (2888)

ZooKeeper port for internodal communication between ZooKeeper nodes.

ZooKeeper leader election port (3888)

ZooKeeper port for leader election among ZooKeeper nodes in a ZooKeeper cluster.

16.1. Internal cluster CA and clients CA

To support encryption, each Strimzi component needs its own private keys and public key certificates. All component certificates are signed by an internal CA (certificate authority) called the *cluster CA*.

CA (Certificate Authority) certificates are generated by the Cluster Operator to verify the identities of components and clients.

Similarly, each Kafka client application connecting to Strimzi using mTLS needs to use private keys and certificates. A second internal CA, named the *clients CA*, is used to sign certificates for the Kafka clients.

Both the cluster CA and clients CA have a self-signed public key certificate.

Kafka brokers are configured to trust certificates signed by either the cluster CA or clients CA. Components that clients do not need to connect to, such as ZooKeeper, only trust certificates signed by the cluster CA. Unless TLS encryption for external listeners is disabled, client applications must trust certificates signed by the cluster CA. This is also true for client applications that perform mTLS authentication.

By default, Strimzi automatically generates and renews CA certificates issued by the cluster CA or clients CA. You can configure the management of these CA certificates using `Kafka.spec.clusterCa` and `Kafka.spec.clientsCa` properties.

NOTE If you don't want to use the CAs generated by the Cluster Operator, you can [install your own cluster and clients CA certificates](#). Any certificates you provide are not renewed by the Cluster Operator.

16.2. Secrets generated by the operators

The Cluster Operator automatically sets up and renews TLS certificates to enable encryption and authentication within a cluster. It also sets up other TLS certificates if you want to enable encryption or mTLS authentication between Kafka brokers and clients.

Secrets are created when custom resources are deployed, such as `Kafka` and `KafkaUser`. Strimzi uses these secrets to store private and public key certificates for Kafka clusters, clients, and users. The secrets are used for establishing TLS encrypted connections between Kafka brokers, and between brokers and clients. They are also used for mTLS authentication.

Cluster and clients secrets are always pairs: one contains the public key and one contains the private key.

Cluster secret

A cluster secret contains the *cluster CA* to sign Kafka broker certificates. Connecting clients use the certificate to establish a TLS encrypted connection with a Kafka cluster. The certificate verifies broker identity.

Client secret

A client secret contains the *clients CA* for a user to sign its own client certificate. This allows mutual authentication against the Kafka cluster. The broker validates a client's identity through the certificate.

User secret

A user secret contains a private key and certificate. The secret is created and signed by the clients CA when a new user is created. The key and certificate are used to authenticate and authorize the user when accessing the cluster.

NOTE You can provide *Kafka listener certificates* for TLS listeners or external listeners that have TLS encryption enabled. Use Kafka listener certificates to incorporate the security infrastructure you already have in place.

16.2.1. TLS authentication using keys and certificates in PEM or PKCS #12 format

The secrets created by Strimzi provide private keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12 (Public-Key Cryptography Standards) formats. PEM and PKCS #12 are OpenSSL-generated key formats for TLS communications using the SSL protocol.

You can configure mutual TLS (mTLS) authentication that uses the credentials contained in the secrets generated for a Kafka cluster and user.

To set up mTLS, you must first do the following:

- [Configure your Kafka cluster with a listener that uses mTLS](#)
- [Create a KafkaUser that provides client credentials for mTLS](#)

When you deploy a Kafka cluster, a `<cluster_name>-cluster-ca-cert` secret is created with public key to verify the cluster. You use the public key to configure a truststore for the client.

When you create a `KafkaUser`, a `<kafka_user_name>` secret is created with the keys and certificates to verify the user (client). Use these credentials to configure a keystore for the client.

With the Kafka cluster and client set up to use mTLS, you extract credentials from the secrets and add them to your client configuration.

PEM keys and certificates

For PEM, you add the following to your client configuration:

Truststore

- `ca.crt` from the `<cluster_name>-cluster-ca-cert` secret, which is the CA certificate for the cluster.

Keystore

- `user.crt` from the `<kafka_user_name>` secret, which is the public certificate of the user.
- `user.key` from the `<kafka_user_name>` secret, which is the private key of the user.

PKCS #12 keys and certificates

For PKCS #12, you add the following to your client configuration:

Truststore

- `ca.p12` from the `<cluster_name>-cluster-ca-cert` secret, which is the CA certificate for the cluster.
- `ca.password` from the `<cluster_name>-cluster-ca-cert` secret, which is the password to access the public cluster CA certificate.

Keystore

- `user.p12` from the `<kafka_user_name>` secret, which is the public key certificate of the user.
- `user.password` from the `<kafka_user_name>` secret, which is the password to access the public key certificate of the Kafka user.

PKCS #12 is supported by Java, so you can add the values of the certificates directly to your Java client configuration. You can also reference the certificates from a secure storage location. With PEM files, you must add the certificates directly to the client configuration in single-line format. Choose a format that's suitable for establishing TLS connections between your Kafka cluster and client. Use PKCS #12 if you are unfamiliar with PEM.

NOTE

All keys are 2048 bits in size and, by default, are valid for 365 days from the initial generation. You can [change the validity period](#).

16.2.2. Secrets generated by the Cluster Operator

The Cluster Operator generates the following certificates, which are saved as secrets in the Kubernetes cluster. Strimzi uses these secrets by default.

The cluster CA and clients CA have separate secrets for the private key and public key.

`<cluster_name>-cluster-ca`

Contains the private key of the cluster CA. Strimzi and Kafka components use the private key to sign server certificates.

`<cluster_name>-cluster-ca-cert`

Contains the public key of the cluster CA. Kafka clients use the public key to verify the identity of the Kafka brokers they are connecting to with TLS server authentication.

`<cluster_name>-clients-ca`

Contains the private key of the clients CA. Kafka clients use the private key to sign new user certificates for mTLS authentication when connecting to Kafka brokers.

`<cluster_name>-clients-ca-cert`

Contains the public key of the clients CA. Kafka brokers use the public key to verify the identity of clients accessing the Kafka brokers when mTLS authentication is used.

Secrets for communication between Strimzi components contain a private key and a public key certificate signed by the cluster CA.

`<cluster_name>-kafka-brokers`

Contains the private and public keys for Kafka brokers.

`<cluster_name>-zookeeper-nodes`

Contains the private and public keys for ZooKeeper nodes.

`<cluster_name>-cluster-operator-certs`

Contains the private and public keys for encrypting communication between the Cluster Operator and Kafka or ZooKeeper.

`<cluster_name>-entity-topic-operator-certs`

Contains the private and public keys for encrypting communication between the Topic Operator and Kafka or ZooKeeper.

`<cluster_name>-entity-user-operator-certs`

Contains the private and public keys for encrypting communication between the User Operator and Kafka or ZooKeeper.

`<cluster_name>-cruise-control-certs`

Contains the private and public keys for encrypting communication between Cruise Control and Kafka or ZooKeeper.

`<cluster_name>-kafka-exporter-certs`

Contains the private and public keys for encrypting communication between Kafka Exporter and Kafka or ZooKeeper.

NOTE You can [provide your own server certificates and private keys](#) to connect to Kafka brokers using *Kafka listener certificates* rather than certificates signed by the cluster CA.

16.2.3. Cluster CA secrets

Cluster CA secrets are managed by the Cluster Operator in a Kafka cluster.

Only the `<cluster_name>-cluster-ca-cert` secret is required by clients. All other cluster secrets are accessed by Strimzi components. You can enforce this using Kubernetes role-based access controls, if necessary.

NOTE The CA certificates in `<cluster_name>-cluster-ca-cert` must be trusted by Kafka client applications so that they validate the Kafka broker certificates when connecting to Kafka brokers over TLS.

Table 20. Fields in the `<cluster_name>-cluster-ca` secret

Field	Description
<code>ca.key</code>	The current private key for the cluster CA.

Table 21. Fields in the `<cluster_name>-cluster-ca-cert` secret

Field	Description
ca.p12	PKCS #12 store for storing certificates and keys.
ca.password	Password for protecting the PKCS #12 store.
ca.crt	The current certificate for the cluster CA.

Table 22. Fields in the <cluster_name>-kafka-brokers secret

Field	Description
<cluster_name>-kafka-<num>.p12	PKCS #12 store for storing certificates and keys.
<cluster_name>-kafka-<num>.password	Password for protecting the PKCS #12 store.
<cluster_name>-kafka-<num>.crt	Certificate for a Kafka broker pod <num>. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
<cluster_name>-kafka-<num>.key	Private key for a Kafka broker pod <num>.

Table 23. Fields in the <cluster_name>-zookeeper-nodes secret

Field	Description
<cluster_name>-zookeeper-<num>.p12	PKCS #12 store for storing certificates and keys.
<cluster_name>-zookeeper-<num>.password	Password for protecting the PKCS #12 store.
<cluster_name>-zookeeper-<num>.crt	Certificate for ZooKeeper node <num>. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
<cluster_name>-zookeeper-<num>.key	Private key for ZooKeeper pod <num>.

Table 24. Fields in the <cluster_name>-cluster-operator-certs secret

Field	Description
cluster-operator.p12	PKCS #12 store for storing certificates and keys.
cluster-operator.password	Password for protecting the PKCS #12 store.
cluster-operator.crt	Certificate for mTLS communication between the Cluster Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
cluster-operator.key	Private key for mTLS communication between the Cluster Operator and Kafka or ZooKeeper.

Table 25. Fields in the <cluster_name>-entity-topic-operator-certs secret

Field	Description
entity-operator.p12	PKCS #12 store for storing certificates and keys.
entity-operator.password	Password for protecting the PKCS #12 store.

Field	Description
entity-operator.crt	Certificate for mTLS communication between the Topic Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
entity-operator.key	Private key for mTLS communication between the Topic Operator and Kafka or ZooKeeper.

Table 26. Fields in the <cluster_name>-entity-user-operator-certs secret

Field	Description
entity-operator.p12	PKCS #12 store for storing certificates and keys.
entity-operator.password	Password for protecting the PKCS #12 store.
entity-operator.crt	Certificate for mTLS communication between the User Operator and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
entity-operator.key	Private key for mTLS communication between the User Operator and Kafka or ZooKeeper.

Table 27. Fields in the <cluster_name>-cruise-control-certs secret

Field	Description
cruise-control.p12	PKCS #12 store for storing certificates and keys.
cruise-control.password	Password for protecting the PKCS #12 store.
cruise-control.crt	Certificate for mTLS communication between Cruise Control and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
cruise-control.key	Private key for mTLS communication between the Cruise Control and Kafka or ZooKeeper.

Table 28. Fields in the <cluster_name>-kafka-exporter-certs secret

Field	Description
kafka-exporter.p12	PKCS #12 store for storing certificates and keys.
kafka-exporter.password	Password for protecting the PKCS #12 store.
kafka-exporter.crt	Certificate for mTLS communication between Kafka Exporter and Kafka or ZooKeeper. Signed by a current or former cluster CA private key in <cluster_name>-cluster-ca.
kafka-exporter.key	Private key for mTLS communication between the Kafka Exporter and Kafka or ZooKeeper.

16.2.4. Clients CA secrets

Clients CA secrets are managed by the Cluster Operator in a Kafka cluster.

The certificates in `<cluster_name>-clients-ca-cert` are those which the Kafka brokers trust.

The `<cluster_name>-clients-ca` secret is used to sign the certificates of client applications. This secret must be accessible to the Strimzi components and for administrative access if you are intending to issue application certificates without using the User Operator. You can enforce this using Kubernetes role-based access controls, if necessary.

Table 29. Fields in the `<cluster_name>-clients-ca` secret

Field	Description
<code>ca.key</code>	The current private key for the clients CA.

Table 30. Fields in the `<cluster_name>-clients-ca-cert` secret

Field	Description
<code>ca.p12</code>	PKCS #12 store for storing certificates and keys.
<code>ca.password</code>	Password for protecting the PKCS #12 store.
<code>ca.crt</code>	The current certificate for the clients CA.

16.2.5. User secrets generated by the User Operator

User secrets are managed by the User Operator.

When a user is created using the User Operator, a secret is generated using the name of the user.

Table 31. Fields in the `user_name` secret

Secret name	Field within secret	Description
<code><user_name></code>	<code>user.p12</code>	PKCS #12 store for storing certificates and keys.
	<code>user.password</code>	Password for protecting the PKCS #12 store.
	<code>user.crt</code>	Certificate for the user, signed by the clients CA
	<code>user.key</code>	Private key for the user

16.2.6. Adding labels and annotations to cluster CA secrets

By configuring the `clusterCaCert` template property in the `Kafka` custom resource, you can add custom labels and annotations to the Cluster CA secrets created by the Cluster Operator. Labels and annotations are useful for identifying objects and adding contextual information. You configure template properties in Strimzi custom resources.

Example template customization to add labels and annotations to secrets

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      clusterCaCert:
        metadata:
          labels:
            label1: value1
            label2: value2
          annotations:
            annotation1: value1
            annotation2: value2
    # ...
```

16.2.7. Disabling `ownerReference` in the CA secrets

By default, the cluster and clients CA secrets are created with an `ownerReference` property that is set to the `Kafka` custom resource. This means that, when the `Kafka` custom resource is deleted, the CA secrets are also deleted (garbage collected) by Kubernetes.

If you want to reuse the CA for a new cluster, you can disable the `ownerReference` by setting the `generateSecretOwnerReference` property for the cluster and clients CA secrets to `false` in the `Kafka` configuration. When the `ownerReference` is disabled, CA secrets are not deleted by Kubernetes when the corresponding `Kafka` custom resource is deleted.

Example Kafka configuration with disabled `ownerReference` for cluster and clients CAs

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    generateSecretOwnerReference: false
  clientsCa:
    generateSecretOwnerReference: false
# ...
```

Additional resources

- [CertificateAuthority schema reference](#)

16.3. Certificate renewal and validity periods

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

For CA certificates automatically created by the Cluster Operator, configure the validity period for certificates in the `kafka` resource using the following properties:

- `Kafka.spec.clusterCa.validityDays` for Cluster CA certificates
- `Kafka.spec.clientsCa.validityDays` for Clients CA certificates

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

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

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

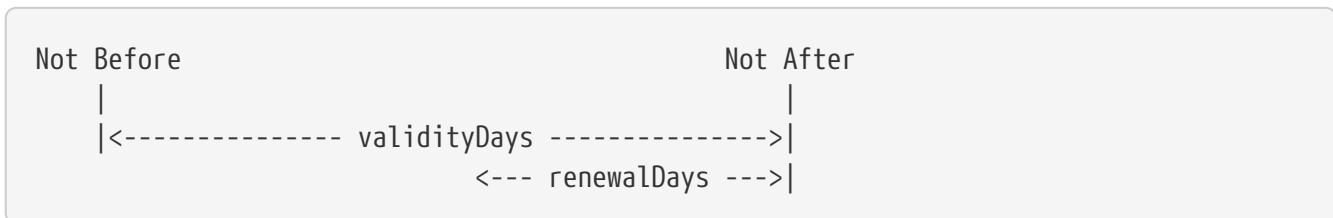
Configure the renewal period of the certificates created by the Cluster Operator in the `kafka` resource using the following properties:

- `Kafka.spec.clusterCa.renewalDays` for Cluster CA certificates
- `Kafka.spec.clientsCa.renewalDays` for Clients CA certificates

The default renewal period for both certificates is 30 days.

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

Validity period against renewal period



To schedule the renewal period at a convenient time, use [maintenance time windows](#).

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

Example Kafka configuration for certificate validity and renewal periods

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

```

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

```

The behavior of the Cluster Operator during the renewal period depends on the settings for the `generateCertificateAuthority` certificate generation properties for the cluster CA and clients CA.

true

If the properties are set to `true`, a CA certificate is generated automatically by the Cluster Operator, and renewed automatically within the renewal period.

false

If the properties are set to `false`, a CA certificate is not generated by the Cluster Operator. Use this option if you are [installing your own certificates](#).

16.3.1. Renewing automatically generated CA Certificates

When it's time to renew CA certificates, the Cluster Operator follows these steps:

1. Generates a new CA certificate, but retains the existing key.

The new certificate replaces the old one with the name `ca.crt` within the corresponding `Secret`.

2. Generates new client certificates (for ZooKeeper nodes, Kafka brokers, and the Entity Operator).

This is not strictly necessary because the signing key has not changed, but it keeps the validity period of the client certificate in sync with the CA certificate.

3. Restarts ZooKeeper nodes to trust the new CA certificate and use the new client certificates.
4. Restarts Kafka brokers to trust the new CA certificate and use the new client certificates.
5. Restarts the Topic Operator and User Operator to trust the new CA certificate and use the new client certificates.

User certificates are signed by the clients CA. The User Operator handles renewing user certificates when the client's CA is renewed.

16.3.2. Renewing client certificates

The Cluster Operator is not aware of the client applications using the Kafka cluster. You must ensure clients continue to work after certificate renewal. The renewal process depends on how the clients are configured.

When connecting to the cluster, and to ensure they operate correctly, client applications must include the following configuration:

- Truststore credentials from the `<cluster_name>-cluster-ca-cert` secret to verify the identity of the Kafka cluster.
- Keystore credentials from the `<user_name>` secret to connect to verify the user when connecting to the Kafka cluster.

The user secret provides credentials in PEM and PKCS #12 format, or it can provide a password when using SCRAM-SHA authentication. The User Operator creates the user credentials when a user is created. For an example of adding certificates to client configuration, see [Securing user access to Kafka](#).

If you are provisioning client certificates and keys manually, you must generate new client certificates and ensure the new certificates are used by clients within the renewal period. Failure to do this by the end of the renewal period could result in client applications being unable to connect to the cluster.

NOTE

For workloads running inside the same Kubernetes cluster and namespace, secrets can be mounted as a volume so the client pods construct their keystores and truststores from the current state of the secrets. For more details on this procedure, see [Configuring internal clients to trust the cluster CA](#).

16.3.3. Scheduling maintenance time windows

Schedule certificate renewal updates by the Cluster Operator to Kafka or ZooKeeper clusters for minimal impact on client applications. Use time windows in conjunction with the [renewal periods of the CA certificates created by the Cluster Operator](#) (`Kafka.spec.clusterCa.renewalDays` and `Kafka.spec.clusterCa.renewalDays`).

Updates are usually triggered by changes to the `Kafka` resource by the user or through user tooling. Rolling restarts for certificate expiration may occur without `Kafka` resource changes. While unscheduled restarts shouldn't affect service availability, they could impact the performance of client applications. Maintenance time windows allow scheduling of these updates for convenient times.

Configure maintenance time windows as follows:

- Configure an array of strings using the `Kafka.spec.maintenanceTimeWindows` property of the `Kafka` resource.
- Each string is a [cron expression](#) interpreted as being in UTC (Coordinated Universal Time)

The following example configures a single maintenance time window that starts at midnight and ends at 01:59am (UTC), on Sundays, Mondays, Tuesdays, Wednesdays, and Thursdays.

Example maintenance time window configuration

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

```
metadata:  
  name: my-cluster  
spec:  
  kafka:  
    #...  
  maintenanceTimeWindows:  
    - "* * 0-1 ? * SUN,MON,TUE,WED,THU *"  
  #...
```

NOTE

The Cluster Operator doesn't adhere strictly to the given time windows for maintenance operations. Maintenance operations are triggered by the first reconciliation that occurs within the specified time window. If the time window is shorter than the interval between reconciliations, there's a risk that the reconciliation may happen outside of the time window. Therefore, maintenance time windows must be at least as long as the interval between reconciliations.

16.3.4. Manually renewing Cluster Operator-managed CA certificates

Cluster and clients CA certificates generated by the Cluster Operator auto-renew at the start of their respective certificate renewal periods. However, you can use the [strimzi.io/force-renew](#) annotation to manually renew one or both of these certificates before the certificate renewal period starts. You might do this for security reasons, or if you have [changed the renewal or validity periods for the certificates](#).

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

NOTE

If you are using your own CA certificates, the [force-renew](#) annotation cannot be used. Instead, follow the procedure for [renewing your own CA certificates](#).

Prerequisites

- [The Cluster Operator must be deployed.](#)
- A Kafka cluster in which CA certificates and private keys are installed.
- The OpenSSL TLS management tool to check the period of validity for CA certificates.

In this procedure, we use a Kafka cluster named [my-cluster](#) within the [my-project](#) namespace.

Procedure

1. Apply the [strimzi.io/force-renew](#) annotation to the secret that contains the CA certificate that you want to renew.

Renewing the Cluster CA secret

```
kubectl annotate secret my-cluster-cluster-ca-cert -n my-project strimzi.io/force-renew="true"
```

Renewing the Clients CA secret

```
kubectl annotate secret my-cluster-clients-ca-cert -n my-project strimzi.io/force-renew="true"
```

2. At the next reconciliation, the Cluster Operator generates new certificates.

If maintenance time windows are configured, the Cluster Operator generates the new CA certificate at the first reconciliation within the next maintenance time window.

3. Check the period of validity for the new CA certificates.

Checking the period of validity for the new cluster CA certificate

```
kubectl get secret my-cluster-cluster-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

Checking the period of validity for the new clients CA certificate

```
kubectl get secret my-cluster-clients-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

The command returns a `notBefore` and `notAfter` date, which is the valid start and end date for the CA certificate.

4. Update client configurations to trust the new cluster CA certificate.

See:

- [Configuring internal clients to trust the cluster CA](#)
- [Configuring external clients to trust the cluster CA](#)

16.3.5. Manually recovering from expired Cluster Operator-managed CA certificates

The Cluster Operator automatically renews the cluster and clients CA certificates when their renewal periods begin. Nevertheless, unexpected operational problems or disruptions may prevent the renewal process, such as prolonged downtime of the Cluster Operator or unavailability of the Kafka cluster. If CA certificates expire, Kafka cluster components cannot communicate with each other and the Cluster Operator cannot renew the CA certificates without manual intervention.

To promptly perform a recovery, follow the steps outlined in this procedure in the order given. You can recover from expired cluster and clients CA certificates. The process involves deleting the secrets containing the expired certificates so that new ones are generated by the Cluster Operator. For more information on the secrets managed in Strimzi, see [Secrets generated by the Cluster Operator](#).

NOTE

If you are using your own CA certificates and they expire, the process is similar, but

you need to [renew the CA certificates](#) rather than use certificates generated by the Cluster Operator.

Prerequisites

- [The Cluster Operator must be deployed.](#)
- A Kafka cluster in which CA certificates and private keys are installed.
- The OpenSSL TLS management tool to check the period of validity for CA certificates.

In this procedure, we use a Kafka cluster named `my-cluster` within the `my-project` namespace.

Procedure

1. Delete the secret containing the expired CA certificate.

Deleting the Cluster CA secret

```
kubectl delete secret my-cluster-cluster-ca-cert -n my-project
```

Deleting the Clients CA secret

```
kubectl delete secret my-cluster-clients-ca-cert -n my-project
```

2. Wait for the Cluster Operator to generate new certificates.

- A new CA cluster certificate to verify the identity of the Kafka brokers is created in a secret of the same name (`my-cluster-cluster-ca-cert`).
- A new CA clients certificate to verify the identity of Kafka users is created in a secret of the same name (`my-cluster-clients-ca-cert`).

3. Check the period of validity for the new CA certificates.

Checking the period of validity for the new cluster CA certificate

```
kubectl get secret my-cluster-cluster-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

Checking the period of validity for the new clients CA certificate

```
kubectl get secret my-cluster-clients-ca-cert -n my-project  
-o=jsonpath='{.data.ca\.crt}' | base64 -d | openssl x509 -noout -dates
```

The command returns a `notBefore` and `notAfter` date, which is the valid start and end date for the CA certificate.

4. Delete the component pods and secrets that use the CA certificates.

- a. Delete the ZooKeeper secret.
- b. Wait for the Cluster Operator to detect the missing ZooKeeper secret and recreate it.

- c. Delete all ZooKeeper pods.
- d. Delete the Kafka secret.
- e. Wait for the Cluster Operator to detect the missing Kafka secret and recreate it.
- f. Delete all Kafka pods.

If you are only recovering the clients CA certificate, you only need to delete the Kafka secret and pods.

You can use the following `kubectl` command to find resources and also verify that they have been removed.

```
kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

Replace `<resource_type>` with the type of the resource, such as `Pod` or `Secret`.

5. Wait for the Cluster Operator to detect the missing Kafka and ZooKeeper pods and recreate them with the updated CA certificates.

On reconciliation, the Cluster Operator automatically updates other components to trust the new CA certificates.

6. Verify that there are no issues related to certificate validation in the Cluster Operator log.
7. Update client configurations to trust the new cluster CA certificate.

See:

- [Configuring internal clients to trust the cluster CA](#)
- [Configuring external clients to trust the cluster CA](#)

16.3.6. Replacing private keys used by Cluster Operator-managed CA certificates

You can replace the private keys used by the cluster CA and clients CA certificates generated by the Cluster Operator. When a private key is replaced, the Cluster Operator generates a new CA certificate for the new private key.

NOTE

If you are using your own CA certificates, the `force-replace` annotation cannot be used. Instead, follow the procedure for [renewing your own CA certificates](#).

Prerequisites

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

Procedure

- Apply the `strimzi.io/force-replace` annotation to the `Secret` that contains the private key that you want to renew.

Table 32. Commands for replacing private keys

Private key for	Secret	Annotate command
Cluster CA	<cluster_name>-cluster-ca	<code>kubectl annotate secret <cluster_name>-cluster-ca strimzi.io/force-replace="true"</code>
Clients CA	<cluster_name>-clients-ca	<code>kubectl annotate secret <cluster_name>-clients-ca strimzi.io/force-replace="true"</code>

At the next reconciliation the Cluster Operator will:

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

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

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

Additional resources

- [Secrets generated by the operators](#)
- [Scheduling maintenance time windows](#)

16.4. Configuring internal clients to trust the cluster CA

This procedure describes how to configure a Kafka client that resides inside the Kubernetes cluster — connecting to a TLS listener — to trust the cluster CA certificate.

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

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

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

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

Prerequisites

- The Cluster Operator must be running.
- There needs to be a **Kafka** resource within the Kubernetes cluster.

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

Using PKCS #12 format (.p12)

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

For example:

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

Here we're mounting the following:

- The PKCS #12 file into an exact path, which can be configured
 - The password into an environment variable, where it can be used for Java configuration
2. Configure the Kafka client with the following properties:
 - A security protocol option:
 - **security.protocol: SSL** when using TLS for encryption (with or without mTLS authentication).
 - **security.protocol: SASL_SSL** when using SCRAM-SHA authentication over TLS.
 - **ssl.truststore.location** with the truststore location where the certificates were imported.
 - **ssl.truststore.password** with the password for accessing the truststore.
 - **ssl.truststore.type=PKCS12** to identify the truststore type.

Using PEM format (.crt)

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

For example:

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
    - name: client-name
      image: client-name
      volumeMounts:
        - name: secret-volume
          mountPath: /data/crt
  volumes:
    - name: secret-volume
      secret:
        secretName: my-cluster-cluster-ca-cert
```

2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

16.5. Configuring external clients to trust the cluster CA

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

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

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

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

IMPORTANT

The `<cluster_name>-cluster-ca-cert` secret contains more than one CA certificate during the CA certificate renewal period. Clients must add *all* of them to their truststores.

Prerequisites

- The Cluster Operator must be running.
- There needs to be a [Kafka](#) resource within the Kubernetes cluster.

- You need a Kafka client application outside the Kubernetes cluster that will connect using TLS, and needs to trust the cluster CA certificate.

Using PKCS #12 format (.p12)

1. Extract the cluster CA certificate and password from the `<cluster_name>-cluster-ca-cert` Secret of the Kafka cluster.

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

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

Replace `<cluster_name>` with the name of the Kafka cluster.

2. Configure the Kafka client with the following properties:

- A security protocol option:
 - `security.protocol: SSL` when using TLS.
 - `security.protocol: SASL_SSL` when using SCRAM-SHA authentication over TLS.
- `ssl.truststore.location` with the truststore location where the certificates were imported.
- `ssl.truststore.password` with the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.
- `ssl.truststore.type=PKCS12` to identify the truststore type.

Using PEM format (.crt)

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

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

2. Use the extracted certificate to configure a TLS connection in clients that use certificates in X.509 format.

16.6. Using your own CA certificates and private keys

Install and use your own CA certificates and private keys instead of using the defaults generated by the Cluster Operator. You can replace the cluster and clients CA certificates and private keys.

You can switch to using your own CA certificates and private keys in the following ways:

- Install your own CA certificates and private keys before deploying your Kafka cluster
- Replace the default CA certificates and private keys with your own after deploying a Kafka

cluster

The steps to replace the default CA certificates and private keys after deploying a Kafka cluster are the same as those used to renew your own CA certificates and private keys.

If you use your own certificates, they won't be renewed automatically. You need to renew the CA certificates and private keys before they expire.

Renewal options:

- Renew the CA certificates only
- Renew CA certificates and private keys (or replace the defaults)

16.6.1. Installing your own CA certificates and private keys

Install your own CA certificates and private keys instead of using the cluster and clients CA certificates and private keys generated by the Cluster Operator.

By default, Strimzi uses the following [cluster CA and clients CA secrets](#), which are renewed automatically.

- Cluster CA secrets
 - `<cluster_name>-cluster-ca`
 - `<cluster_name>-cluster-ca-cert`
- Clients CA secrets
 - `<cluster_name>-clients-ca`
 - `<cluster_name>-clients-ca-cert`

To install your own certificates, use the same names.

Prerequisites

- The Cluster Operator is running.
- A Kafka cluster is not yet deployed.

If you have already deployed a Kafka cluster, you can [replace the default CA certificates with your own](#).

- Your own X.509 certificates and keys in PEM format for the cluster CA or clients CA.
 - If you want to use a cluster or clients CA which is not a Root CA, you have to include the whole chain in the certificate file. The chain should be in the following order:
 1. The cluster or clients CA
 2. One or more intermediate CAs
 3. The root CA
 - All CAs in the chain should be configured using the X509v3 Basic Constraints extension. Basic Constraints limit the path length of a certificate chain.

- The OpenSSL TLS management tool for converting certificates.

Before you begin

The Cluster Operator generates keys and certificates in PEM (Privacy Enhanced Mail) and PKCS #12 (Public-Key Cryptography Standards) formats. You can add your own certificates in either format.

Some applications cannot use PEM certificates and support only PKCS #12 certificates. If you don't have a cluster certificate in PKCS #12 format, use the OpenSSL TLS management tool to generate one from your `ca.crt` file.

Example certificate generation command

```
openssl pkcs12 -export -in ca.crt -nokeys -out ca.p12 -password pass:<P12_password>  
-caname ca.crt
```

Replace <P12_password> with your own password.

Procedure

1. Create a new secret that contains the CA certificate.

Client secret creation with a certificate in PEM format only

```
kubectl create secret generic <cluster_name>-clients-ca-cert --from  
-file=ca.crt=ca.crt
```

Cluster secret creation with certificates in PEM and PKCS #12 format

```
kubectl create secret generic <cluster_name>-cluster-ca-cert \  
--from-file=ca.crt=ca.crt \  
--from-file=ca.p12=ca.p12 \  
--from-literal=ca.password=P12-PASSWORD
```

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

2. Create a new secret that contains the private key.

```
kubectl create secret generic <ca_key_secret> --from-file=ca.key=ca.key
```

3. Label the secrets.

```
kubectl label secret <ca_certificate_secret> strimzi.io/kind=Kafka  
strimzi.io/cluster=<cluster_name>"
```

```
kubectl label secret <ca_key_secret> strimzi.io/kind=Kafka  
strimzi.io/cluster=<cluster_name>"
```

- Label `strimzi.io/kind=Kafka` identifies the Kafka custom resource.
 - Label `strimzi.io/cluster=<cluster_name>` identifies the Kafka cluster.
4. Annotate the secrets

```
kubectl annotate secret <ca_certificate_secret> strimzi.io/ca-cert-generation=<ca_certificate_generation>
```

```
kubectl annotate secret <ca_key_secret> strimzi.io/ca-key-generation=<ca_key_generation>
```

- Annotation `strimzi.io/ca-cert-generation=<ca_certificate_generation>` defines the generation of a new CA certificate.
- Annotation `strimzi.io/ca-key-generation=<ca_key_generation>` defines the generation of a new CA key.

Start from 0 (zero) as the incremental value (`strimzi.io/ca-cert-generation=0`) for your own CA certificate. Set a higher incremental value when you renew the certificates.

5. Create the `Kafka` resource for your cluster, configuring either the `Kafka.spec.clusterCa` or the `Kafka.spec.clientsCa` object to *not* use generated CAs.

Example fragment Kafka resource configuring the cluster CA to use certificates you supply for yourself

```
kind: Kafka
version: kafka.strimzi.io/v1beta2
spec:
  # ...
  clusterCa:
    generateCertificateAuthority: false
```

Additional resources

- [Renewing your own CA certificates](#)
- [Renewing or replacing CA certificates and private keys with your own](#)
- [Using custom listener certificates for TLS encryption](#)

16.6.2. Renewing your own CA certificates

If you are using your own CA certificates, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire.

Perform the steps in this procedure when you are renewing CA certificates and continuing with the same private key. If you are renewing your own CA certificates *and* private keys, see [Renewing or replacing CA certificates and private keys with your own](#).

The procedure describes the renewal of CA certificates in PEM format.

Prerequisites

- The Cluster Operator is running.
- You have new cluster or clients X.509 certificates in PEM format.

Procedure

1. Update the **Secret** for the CA certificate.

Edit the existing secret to add the new CA certificate and update the certificate generation annotation value.

```
kubectl edit secret <ca_certificate_secret_name>
```

`<ca_certificate_secret_name>` is the name of the **Secret**, which is `<kafka_cluster_name>-cluster-ca-cert` for the cluster CA certificate and `<kafka_cluster_name>-clients-ca-cert` for the clients CA certificate.

The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named `my-cluster`.

Example secret configuration for a cluster CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ①
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
    name: my-cluster-cluster-ca-cert
    #...
  type: Opaque
```

① Current base64-encoded CA certificate

② Current CA certificate generation annotation value

2. Encode your new CA certificate into base64.

```
cat <path_to_new_certificate> | base64
```

3. Update the CA certificate.

Copy the base64-encoded CA certificate from the previous step as the value for the `ca.crt`

property under `data`.

4. Increase the value of the CA certificate generation annotation.

Update the `strimzi.io/ca-cert-generation` annotation with a higher incremental value. For example, change `strimzi.io/ca-cert-generation=0` to `strimzi.io/ca-cert-generation=1`. If the `Secret` is missing the annotation, the value is treated as `0`, so add the annotation with a value of `1`.

When Strimzi generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The `strimzi.io/ca-cert-generation` has to be incremented on each CA certificate renewal.

5. Save the secret with the new CA certificate and certificate generation annotation value.

Example secret configuration updated with a new CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFfFDGBOUDYFAZ0F... ①
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

① New base64-encoded CA certificate

② New CA certificate generation annotation value

On the next reconciliation, the Cluster Operator performs a rolling update of ZooKeeper, Kafka, and other components to trust the new CA certificate.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

16.6.3. Renewing or replacing CA certificates and private keys with your own

If you are using your own CA certificates and private keys, you need to renew them manually. The Cluster Operator will not renew them automatically. Renew the CA certificates in the renewal period before they expire. You can also use the same procedure to replace the CA certificates and private keys generated by the Strimzi operators with your own.

Perform the steps in this procedure when you are renewing or replacing CA certificates and private keys. If you are only renewing your own CA certificates, see [Renewing your own CA certificates](#).

The procedure describes the renewal of CA certificates and private keys in PEM format.

Before going through the following steps, make sure that the CN (Common Name) of the new CA certificate is different from the current one. For example, when the Cluster Operator renews certificates automatically it adds a $v<version_number>$ suffix to identify a version. Do the same with your own CA certificate by adding a different suffix on each renewal. By using a different key to generate a new CA certificate, you retain the current CA certificate stored in the [Secret](#).

Prerequisites

- The Cluster Operator is running.
- You have new cluster or clients X.509 certificates and keys in PEM format.

Procedure

1. Pause the reconciliation of the [Kafka](#) custom resource.

- a. Annotate the custom resource in Kubernetes, setting the [pause-reconciliation](#) annotation to [true](#):

```
kubectl annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="true"
```

For example, for a [Kafka](#) custom resource named [my-cluster](#):

```
kubectl annotate Kafka my-cluster strimzi.io/pause-reconciliation="true"
```

- b. Check that the status conditions of the custom resource show a change to [ReconciliationPaused](#):

```
kubectl describe Kafka <name_of_custom_resource>
```

The [type](#) condition changes to [ReconciliationPaused](#) at the [lastTransitionTime](#).

2. Check the settings for the [generateCertificateAuthority](#) properties in your [Kafka](#) custom resource.

If a property is set to [false](#), a CA certificate is not generated by the Cluster Operator. You require this setting if you are using your own certificates.

3. If needed, edit the existing [Kafka](#) custom resource and set the [generateCertificateAuthority](#) properties to [false](#).

```
kubectl edit Kafka <name_of_custom_resource>
```

The following example shows a `Kafka` custom resource with both cluster and clients CA certificates generation delegated to the user.

Example Kafka configuration using your own CA certificates

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    generateCertificateAuthority: false ①
  clientsCa:
    generateCertificateAuthority: false ②
# ...
```

① Use your own cluster CA

② Use your own clients CA

4. Update the `Secret` for the CA certificate.

- Edit the existing secret to add the new CA certificate and update the certificate generation annotation value.

```
kubectl edit secret <ca_certificate_secret_name>
```

`<ca_certificate_secret_name>` is the name of the `Secret`, which is `<kafka_cluster_name>-cluster-ca-cert` for the cluster CA certificate and `<kafka_cluster_name>-clients-ca-cert` for the clients CA certificate.

The following example shows a secret for a cluster CA certificate that's associated with a Kafka cluster named `my-cluster`.

Example secret configuration for a cluster CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ①
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

- ① Current base64-encoded CA certificate
- ② Current CA certificate generation annotation value

b. Rename the current CA certificate to retain it.

Rename the current `ca.crt` property under `data` as `ca-<date>.crt`, where `<date>` is the certificate expiry date in the format `YEAR-MONTH-DAYTHOUR-MINUTE-SECONDZ`. For example `ca-2023-01-26T17-32-00Z.crt`. Leave the value for the property as it is to retain the current CA certificate.

c. Encode your new CA certificate into base64.

```
cat <path_to_new_certificate> | base64
```

d. Update the CA certificate.

Create a new `ca.crt` property under `data` and copy the base64-encoded CA certificate from the previous step as the value for `ca.crt` property.

e. Increase the value of the CA certificate generation annotation.

Update the `strimzi.io/ca-cert-generation` annotation with a higher incremental value. For example, change `strimzi.io/ca-cert-generation=0` to `strimzi.io/ca-cert-generation=1`. If the `Secret` is missing the annotation, the value is treated as `0`, so add the annotation with a value of `1`.

When Strimzi generates certificates, the certificate generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates, set the annotations with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates. The `strimzi.io/ca-cert-generation` has to be incremented on each CA certificate renewal.

f. Save the secret with the new CA certificate and certificate generation annotation value.

Example secret configuration updated with a new CA certificate

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFfFDGBOUDYFAZ0F... ①
  ca-2023-01-26T17-32-00Z.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ②
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" ③
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
```

```
type: Opaque
```

- ① New base64-encoded CA certificate
- ② Old base64-encoded CA certificate
- ③ New CA certificate generation annotation value

5. Update the **Secret** for the CA key used to sign your new CA certificate.

- a. Edit the existing secret to add the new CA key and update the key generation annotation value.

```
kubectl edit secret <ca_key_name>
```

<ca_key_name> is the name of CA key, which is `<kafka_cluster_name>-cluster-ca` for the cluster CA key and `<kafka_cluster_name>-clients-ca` for the clients CA key.

The following example shows a secret for a cluster CA key that's associated with a Kafka cluster named `my-cluster`.

Example secret configuration for a cluster CA key

```
apiVersion: v1
kind: Secret
data:
  ca.key: SA1cKF1GFDzOIIPOIUQBHDNFGDFS... ①
metadata:
  annotations:
    strimzi.io/ca-key-generation: "0" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque
```

- ① Current base64-encoded CA key
- ② Current CA key generation annotation value

- b. Encode the CA key into base64.

```
cat <path_to_new_key> | base64
```

- c. Update the CA key.

Copy the base64-encoded CA key from the previous step as the value for the `ca.key` property under `data`.

- d. Increase the value of the CA key generation annotation.

Update the `strimzi.io/ca-key-generation` annotation with a higher incremental value. For example, change `strimzi.io/ca-key-generation=0` to `strimzi.io/ca-key-generation=1`. If the `Secret` is missing the annotation, it is treated as `0`, so add the annotation with a value of `1`.

When Strimzi generates certificates, the key generation annotation is automatically incremented by the Cluster Operator. For your own CA certificates together with a new CA key, set the annotation with a higher incremental value. The annotation needs a higher value than the one from the current secret so that the Cluster Operator can roll the pods and update the certificates and keys. The `strimzi.io/ca-key-generation` has to be incremented on each CA certificate renewal.

- e. Save the secret with the new CA key and key generation annotation value.

Example secret configuration updated with a new CA key

```
apiVersion: v1
kind: Secret
data:
  ca.key: AB0cKF1GFDz0IiPOIUQWERZJQ0F... ①
metadata:
  annotations:
    strimzi.io/ca-key-generation: "1" ②
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque
```

① New base64-encoded CA key

② New CA key generation annotation value

6. Resume from the pause.

To resume the `Kafka` custom resource reconciliation, set the `pause-reconciliation` annotation to `false`.

```
kubectl annotate --overwrite Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="false"
```

You can also do the same by removing the `pause-reconciliation` annotation.

```
kubectl annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation-
```

On the next reconciliation, the Cluster Operator performs a rolling update of ZooKeeper, Kafka, and other components to trust the new CA certificate. When the rolling update is complete, the Cluster Operator will start a new one to generate new server certificates signed by the new CA key.

If maintenance time windows are configured, the Cluster Operator will roll the pods at the first reconciliation within the next maintenance time window.

7. Wait until the rolling updates to move to the new CA certificate are complete.
8. Remove any outdated certificates from the secret configuration to ensure that the cluster no longer trusts them.

```
kubectl edit secret <ca_certificate_secret_name>
```

Example secret configuration with the old certificate removed

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F...
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1"
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

9. Start a manual rolling update of your cluster to pick up the changes made to the secret configuration.

See [Managing rolling updates](#).

Chapter 17. Applying security context to Strimzi pods and containers

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

17.1. How to configure security context

Use security provider plugins or template configuration to apply security context to Strimzi pods and containers.

Apply security context at the pod or container level:

Pod-level security context

Pod-level security context is applied to all containers in a specific pod.

Container-level security context

Container-level security context is applied to a specific container.

With Strimzi, security context is applied through one or both of the following methods:

Template configuration

Use `template` configuration of Strimzi custom resources to specify security context at the pod or container level.

Pod security provider plugins

Use pod security provider plugins to automatically set security context across all pods and containers using preconfigured settings.

Pod security providers offer a simpler alternative to specifying security context through `template` configuration. You can use both approaches. The `template` approach has a higher priority. Security context configured through `template` properties overrides the configuration set by pod security providers. So you might use pod security providers to automatically configure the security context for most containers. And also use `template` configuration to set container-specific security context where needed.

The `template` approach provides flexibility, but it also means you have to configure security context in numerous places to capture the security you want for all pods and containers. For example, you'll need to apply the configuration to each pod in a Kafka cluster, as well as the pods for deployments of other Kafka components.

To avoid repeating the same configuration, you can use the following pod security provider plugins so that the security configuration is in one place.

Baseline Provider

The Baseline Provider is based on the Kubernetes *baseline* security profile. The baseline profile

prevents privilege escalations and defines other standard access controls and limitations.

Restricted Provider

The Restricted Provider is based on the Kubernetes *restricted* security profile. The restricted profile is more restrictive than the baseline profile, and is used where security needs to be tighter.

For more information on the Kubernetes security profiles, see [Pod security standards](#).

17.1.1. Template configuration for security context

In the following example, security context is configured for Kafka brokers in the `template` configuration of the `Kafka` resource. Security context is specified at the pod and container level.

Example template configuration for security context

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafka:
    template:
      pod: ①
        securityContext:
          runAsUser: 1000001
          fsGroup: 0
      kafkaContainer: ②
        securityContext:
          runAsUser: 2000
    # ...
```

① Pod security context

② Container security context of the Kafka broker container

17.1.2. Baseline Provider for pod security

The Baseline Provider is the default pod security provider. It configures the pods managed by Strimzi with a baseline security profile. The baseline profile is compatible with previous versions of Strimzi.

The Baseline Provider is enabled by default if you don't specify a provider. Though you can enable it explicitly by setting the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable to `baseline` when configuring the Cluster Operator.

Configuration for the Baseline Provider

```
# ...
env:
```

```
# ...
- name: STRIMZI_POD_SECURITY_PROVIDER_CLASS
  value: baseline
# ...
```

Instead of specifying `baseline` as the value, you can specify the `io.strimzi.plugin.securityprofiles.impl.BaselinePodSecurityProvider` fully-qualified domain name.

17.1.3. Restricted Provider for pod security

The Restricted Provider provides a higher level of security than the Baseline Provider. It configures the pods managed by Strimzi with a restricted security profile.

You enable the Restricted Provider by setting the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable to `restricted` when configuring the Cluster Operator.

Configuration for the Restricted Provider

```
# ...
env:
# ...
- name: STRIMZI_POD_SECURITY_PROVIDER_CLASS
  value: restricted
# ...
```

Instead of specifying `restricted` as the value, you can specify the `io.strimzi.plugin.securityprofiles.impl.RestrictedPodSecurityProvider` fully-qualified domain name.

If you change to the Restricted Provider from the default Baseline Provider, the following restrictions are implemented in addition to the constraints defined in the baseline security profile:

- Limits allowed volume types
- Disallows privilege escalation
- Requires applications to run under a non-root user
- Requires `seccomp` (secure computing mode) profiles to be set as `RuntimeDefault` or `Localhost`
- Limits container capabilities to use only the `NET_BIND_SERVICE` capability

With the Restricted Provider enabled, containers created by the Cluster Operator are set with the following security context.

Cluster Operator with restricted security context configuration

```
# ...
securityContext:
  allowPrivilegeEscalation: false
  capabilities:
```

```
drop:  
  - ALL  
runAsNonRoot: true  
seccompProfile:  
  type: RuntimeDefault  
# ...
```

Container capabilities and `seccomp` are Linux kernel features that support container security.

NOTE

- Capabilities add fine-grained privileges for processes running on a container. The `NET_BIND_SERVICE` capability allows non-root user applications to bind to ports below 1024.
- `seccomp` profiles limit the processes running in a container to only a subset of system calls. The `RuntimeDefault` profile provides a default set of system calls. A `LocalHost` profile uses a profile defined in a file on the node.

Additional resources

- [Security context](#) on Kubernetes
- [Pod security standards](#) on Kubernetes (including profile descriptions)

17.2. Enabling the Restricted Provider for the Cluster Operator

Security pod providers configure the security context constraints of the pods and containers created by the Cluster Operator. The Baseline Provider is the default pod security provider used by Strimzi. You can switch to the Restricted Provider by changing the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable in the Cluster Operator configuration.

To make the required changes, configure the `060-Deployment-strimzi-cluster-operator.yaml` Cluster Operator installation file located in `install/cluster-operator/`.

By enabling a new pod security provider, any pods or containers created by the Cluster Operator are subject to the limitations it imposes. Pods and containers that are already running are restarted for the changes to take affect.

Prerequisites

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

Procedure

Edit the `Deployment` resource that is used to deploy the Cluster Operator, which is defined in the `060-Deployment-strimzi-cluster-operator.yaml` file.

1. Add or amend the `STRIMZI_POD_SECURITY_PROVIDER_CLASS` environment variable with a value of `restricted`.

```
# ...
env:
# ...
- name: STRIMZI_POD_SECURITY_PROVIDER_CLASS
  value: restricted
# ...
```

Or you can specify the fully-qualified domain name.

2. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n myproject
```

3. (Optional) Use `template` configuration to set security context for specific components at the pod or container level.

Adding security context through `template` configuration

```
template:
pod:
  securityContext:
    runAsUser: 1000001
    fsGroup: 0
kafkaContainer:
  securityContext:
    runAsUser: 2000
# ...
```

If you apply specific security context for a component using `template` configuration, it takes priority over the general configuration provided by the pod security provider.

17.3. Implementing a custom pod security provider

If Strimzi's Baseline Provider and Restricted Provider don't quite match your needs, you can develop a custom pod security provider to deliver all-encompassing pod and container security context constraints.

Implement a custom pod security provider to apply your own security context profile. You can decide what applications and privileges to include in the profile.

Your custom pod security provider can implement the `PodSecurityProvider.java` interface that gets the security context for pods and containers; or it can extend the Baseline Provider or Restricted Provider classes.

The pod security provider plugins use the Java Service Provider Interface, so your custom pod security provider also requires a provider configuration file for service discovery.

To implement your own provider, the general steps include the following:

1. Build the JAR file for the provider.
2. Add the JAR file to the Cluster Operator image.
3. Specify the custom pod security provider when setting the Cluster Operator environment variable `STRIMZI_POD_SECURITY_PROVIDER_CLASS`.

Additional resources

- [Pod security provider interface](#)
- [Baseline Provider and Restricted Provider classes](#)
- [Provider configuration file](#)
- [Java Service Provider Interface](#)

17.4. Handling of security context by Kubernetes platform

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

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

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

For more information, see the [OpenShift documentation](#).

Chapter 18. Scaling clusters by adding or removing brokers

Scaling Kafka clusters by adding brokers can increase the performance and reliability of the cluster. Adding more brokers increases available resources, allowing the cluster to handle larger workloads and process more messages. It can also improve fault tolerance by providing more replicas and backups. Conversely, removing underutilized brokers can reduce resource consumption and improve efficiency. Scaling must be done carefully to avoid disruption or data loss. By redistributing partitions across all brokers in the cluster, the resource utilization of each broker is reduced, which can increase the overall throughput of the cluster.

NOTE

To increase the throughput of a Kafka topic, you can increase the number of partitions for that topic. This allows the load of the topic to be shared between different brokers in the cluster. However, if every broker is constrained by a specific resource (such as I/O), adding more partitions will not increase the throughput. In this case, you need to add more brokers to the cluster.

Adjusting the `Kafka.spec.kafka.replicas` configuration affects the number of brokers in the cluster that act as replicas. The actual replication factor for topics is determined by settings for the `default.replication.factor` and `min.insync.replicas`, and the number of available brokers. For example, a replication factor of 3 means that each partition of a topic is replicated across three brokers, ensuring fault tolerance in the event of a broker failure.

Example replica configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    # ...
  config:
    # ...
    default.replication.factor: 3
    min.insync.replicas: 2
# ...
```

When adding brokers through the `Kafka` resource configuration, node IDs start at 0 (zero) and the Cluster Operator assigns the next lowest ID to a new node. The broker removal process starts from the broker pod with the highest ID in the cluster.

If you are managing nodes in the cluster using node pools, adjust the `KafkaNodePool.spec.replicas` configuration to change the number of nodes in the node pool. Additionally, when scaling existing clusters with node pools, you can [assign node IDs for the scaling operations](#).

When you add or remove brokers, Kafka does not automatically reassigned partitions. The best way to do this is using Cruise Control. You can use Cruise Control's `add-brokers` and `remove-brokers` modes when scaling a cluster up or down.

- Use the `add-brokers` mode after scaling up a Kafka cluster to move partition replicas from existing brokers to the newly added brokers.
- Use the `remove-brokers` mode before scaling down a Kafka cluster to move partition replicas off the brokers that are going to be removed.

18.1. Skipping checks on scale-down operations

By default, Strimzi performs a check to ensure that there are no partition replicas on brokers before initiating a scale-down operation on a Kafka cluster. The check applies to nodes in node pools that perform the role of broker only or a dual role of broker and controller.

If replicas are found, the scale-down is not done in order to prevent potential data loss. To scale-down the cluster, no replicas must be left on the broker before trying to scale it down again.

However, there may be scenarios where you want to bypass this mechanism. Disabling the check might be necessary on busy clusters, for example, because new topics keep generating replicas for the broker. This situation can indefinitely block the scale-down, even when brokers are nearly empty. Overriding the blocking mechanism in this way has an impact: the presence of topics on the broker being scaled down will likely cause a reconciliation failure for the Kafka cluster.

You can bypass the blocking mechanism by annotating the `Kafka` resource for the Kafka cluster. Annotate the resource by setting the `strimzi.io/skip-broker-scaledown-check` annotation to `true`:

Adding the annotation to skip checks on scale-down operations

```
kubectl annotate Kafka my-kafka-cluster strimzi.io/skip-broker-scaledown-check="true"
```

This annotation instructs Strimzi to skip the scale-down check. Replace `my-kafka-cluster` with the name of your specific `Kafka` resource.

To restore the check for scale-down operations, remove the annotation:

Removing the annotation to skip checks on scale-down operations

```
kubectl annotate Kafka my-kafka-cluster strimzi.io/skip-broker-scaledown-check-
```

Chapter 19. Using Cruise Control for cluster rebalancing

Cruise Control is an open source system that supports the following Kafka operations:

- Monitoring cluster workload
- Rebalancing a cluster based on predefined constraints

The operations help with running a more balanced Kafka cluster that uses broker pods more efficiently.

A typical cluster can become unevenly loaded over time. Partitions that handle large amounts of message traffic might not be evenly distributed across the available brokers. To rebalance the cluster, administrators must monitor the load on brokers and manually reassign busy partitions to brokers with spare capacity.

Cruise Control automates the cluster rebalancing process. It constructs a *workload model* of resource utilization for the cluster—based on CPU, disk, and network load—and generates optimization proposals (that you can approve or reject) for more balanced partition assignments. A set of configurable optimization goals is used to calculate these proposals.

You can generate optimization proposals in specific modes. The default `full` mode rebalances partitions across all brokers. You can also use the `add-brokers` and `remove-brokers` modes to accommodate changes when scaling a cluster up or down.

When you approve an optimization proposal, Cruise Control applies it to your Kafka cluster. You configure and generate optimization proposals using a `KafkaRebalance` resource. You can configure the resource using an annotation so that optimization proposals are approved automatically or manually.

NOTE Strimzi provides [example configuration files for Cruise Control](#).

19.1. Cruise Control components and features

Cruise Control consists of four main components—the Load Monitor, the Analyzer, the Anomaly Detector, and the Executor—and a REST API for client interactions. Strimzi utilizes the REST API to support the following Cruise Control features:

- Generating optimization proposals from optimization goals.
- Rebalancing a Kafka cluster based on an optimization proposal.

Optimization goals

An optimization goal describes a specific objective to achieve from a rebalance. For example, a goal might be to distribute topic replicas across brokers more evenly. You can change what goals to include through configuration. A goal is defined as a hard goal or soft goal. You can add hard goals through Cruise Control deployment configuration. You also have main, default, and user-

provided goals that fit into each of these categories.

- **Hard goals** are preset and must be satisfied for an optimization proposal to be successful.
- **Soft goals** do not need to be satisfied for an optimization proposal to be successful. They can be set aside if it means that all hard goals are met.
- **Main goals** are inherited from Cruise Control. Some are preset as hard goals. Main goals are used in optimization proposals by default.
- **Default goals** are the same as the main goals by default. You can specify your own set of default goals.
- **User-provided goals** are a subset of default goals that are configured for generating a specific optimization proposal.

Optimization proposals

Optimization proposals comprise the goals you want to achieve from a rebalance. You generate an optimization proposal to create a summary of proposed changes and the results that are possible with the rebalance. The goals are assessed in a specific order of priority. You can then choose to approve or reject the proposal. You can reject the proposal to run it again with an adjusted set of goals.

You can generate an optimization proposal in one of three modes.

- **full** is the default mode and runs a full rebalance.
- **add-brokers** is the mode you use after adding brokers when scaling up a Kafka cluster.
- **remove-brokers** is the mode you use before removing brokers when scaling down a Kafka cluster.

Other Cruise Control features are not currently supported, including self healing, notifications, and write-your-own goals.

Additional resources

- [Cruise Control documentation](#)

19.2. Optimization goals overview

Optimization goals are constraints on workload redistribution and resource utilization across a Kafka cluster. To rebalance a Kafka cluster, Cruise Control uses optimization goals to generate [optimization proposals](#), which you can approve or reject.

19.2.1. Goals order of priority

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

1. Rack-awareness
2. Minimum number of leader replicas per broker for a set of topics
3. Replica capacity

4. Capacity goals
 - Disk capacity
 - Network inbound capacity
 - Network outbound capacity
 - CPU capacity
5. Replica distribution
6. Potential network output
7. Resource distribution goals
 - Disk utilization distribution
 - Network inbound utilization distribution
 - Network outbound utilization distribution
 - CPU utilization distribution
8. Leader bytes-in rate distribution
9. Topic replica distribution
10. Leader replica distribution
11. Preferred leader election
12. Intra-broker disk capacity
13. Intra-broker disk usage distribution

For more information on each optimization goal, see [Goals](#) in the Cruise Control Wiki.

NOTE "Write your own" goals and Kafka assigner goals are not yet supported.

19.2.2. Goals configuration in Strimzi custom resources

You configure optimization goals in [Kafka](#) and [KafkaRebalance](#) custom resources. Cruise Control has configurations for hard optimization goals that must be satisfied, as well as main, default, and user-provided optimization goals.

You can specify optimization goals in the following configuration:

- **Main goals** — [Kafka.spec.cruiseControl.config.goals](#)
- **Hard goals** — [Kafka.spec.cruiseControl.config.hard.goals](#)
- **Default goals** — [Kafka.spec.cruiseControl.config.default.goals](#)
- **User-provided goals** — [KafkaRebalance.spec.goals](#)

NOTE Resource distribution goals are subject to [capacity limits](#) on broker resources.

19.2.3. Hard and soft optimization goals

Hard goals are goals that *must* be satisfied in optimization proposals. Goals that are not defined as *hard goals* in the Cruise Control code are known as *soft goals*. You can think of soft goals as *best effort* goals: they do *not* need to be satisfied in optimization proposals, but are included in optimization calculations. An optimization proposal that violates one or more soft goals, but satisfies all hard goals, is valid.

Cruise Control will calculate optimization proposals that satisfy all the hard goals and as many soft goals as possible (in their priority order). An optimization proposal that does *not* satisfy all the hard goals is rejected by Cruise Control and not sent to the user for approval.

For example, you might have a soft goal to distribute a topic's replicas evenly across

NOTE the cluster (the topic replica distribution goal). Cruise Control will ignore this goal if doing so enables all the configured hard goals to be met.

In Cruise Control, the following [main optimization goals](#) are hard goals:

```
RackAwareGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal;  
NetworkOutboundCapacityGoal; CpuCapacityGoal
```

In your Cruise Control deployment configuration, you can specify which hard goals to enforce using the `hard.goals` property in [Kafka.spec.cruiseControl.config](#).

- To enforce execution of all hard goals, simply omit the `hard.goals` property.
- To change which hard goals Cruise Control enforces, specify the required goals in the `hard.goals` property using their fully-qualified domain names.
- To prevent execution of a specific hard goal, ensure that the goal is not included in both the `default.goals` and `hard.goals` list configurations.

NOTE It is not possible to configure which goals are considered soft or hard goals. This distinction is determined by the Cruise Control code.

Example Kafka configuration for hard optimization goals

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

```

brokerCapacity:
  inboundNetwork: 10000KB/s
  outboundNetwork: 10000KB/s
config:
  # Note that 'default.goals' (superset) must also include all 'hard.goals'
(subset)
  default.goals: >
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
  hard.goals: >
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
  # ...

```

Increasing the number of configured hard goals will reduce the likelihood of Cruise Control generating valid optimization proposals.

If `skipHardGoalCheck: true` is specified in the `KafkaRebalance` custom resource, Cruise Control does *not* check that the list of user-provided optimization goals (in `KafkaRebalance.spec.goals`) contains *all* the configured hard goals (`hard.goals`). Therefore, if some, but not all, of the user-provided optimization goals are in the `hard.goals` list, Cruise Control will still treat them as hard goals even if `skipHardGoalCheck: true` is specified.

19.2.4. Main optimization goals

The *main optimization goals* are available to all users. Goals that are not listed in the main optimization goals are not available for use in Cruise Control operations.

Unless you change the Cruise Control [deployment configuration](#), Strimzi will inherit the following main optimization goals from Cruise Control, in descending priority order:

```

RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal;
ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal;
NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal;
CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal;
LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal

```

Some of these goals are preset as [hard goals](#).

To reduce complexity, we recommend that you use the inherited main optimization goals, unless you need to *completely* exclude one or more goals from use in `KafkaRebalance` resources. The priority order of the main optimization goals can be modified, if desired, in the configuration for [default optimization goals](#).

You configure main optimization goals, if necessary, in the Cruise Control deployment configuration: `Kafka.spec.cruiseControl.config.goals`

- To accept the inherited main optimization goals, do not specify the `goals` property in

Kafka.spec.cruiseControl.config.

- If you need to modify the inherited main optimization goals, specify a list of goals, in descending priority order, in the `goals` configuration option.

NOTE To avoid errors when generating optimization proposals, make sure that any changes you make to the `goals` or `default.goals` in `Kafka.spec.cruiseControl.config` include all of the hard goals specified for the `hard.goals` property. To clarify, the hard goals must also be specified (as a subset) for the main optimization goals and default goals.

19.2.5. Default optimization goals

Cruise Control uses the *default optimization goals* to generate the *cached optimization proposal*. For more information about the cached optimization proposal, see [Optimization proposals overview](#).

You can override the default optimization goals by setting [user-provided optimization goals](#) in a `KafkaRebalance` custom resource.

Unless you specify `default.goals` in the Cruise Control [deployment configuration](#), the main optimization goals are used as the default optimization goals. In this case, the cached optimization proposal is generated using the main optimization goals.

- To use the main optimization goals as the default goals, do not specify the `default.goals` property in `Kafka.spec.cruiseControl.config`.
- To modify the default optimization goals, edit the `default.goals` property in `Kafka.spec.cruiseControl.config`. You must use a subset of the main optimization goals.

Example Kafka configuration for default optimization goals

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      # Note that 'default.goals' (superset) must also include all 'hard.goals'
      (subset)
      default.goals: >
```

```
com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,  
com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,  
com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal  
hard.goals: >  
    com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal  
# ...
```

If no default optimization goals are specified, the cached proposal is generated using the main optimization goals.

19.2.6. User-provided optimization goals

User-provided optimization goals narrow down the configured default goals for a particular optimization proposal. You can set them, as required, in `spec.goals` in a `KafkaRebalance` custom resource:

KafkaRebalance.spec.goals

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

User-provided optimization goals must:

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

To ignore the configured hard goals when generating an optimization proposal, add the `skipHardGoalCheck: true` property to the `KafkaRebalance` custom resource. See [Generating optimization proposals](#).

Additional resources

- [Configuring and deploying Cruise Control with Kafka](#)
- [Configurations](#) in the Cruise Control Wiki.

19.3. Optimization proposals overview

Configure a `KafkaRebalance` resource to generate optimization proposals and apply the suggested changes. An *optimization proposal* is a summary of proposed changes that would produce a more balanced Kafka cluster, with partition workloads distributed more evenly among the brokers.

Each optimization proposal is based on the set of *optimization goals* that was used to generate it, subject to any configured [capacity limits on broker resources](#).

All optimization proposals are *estimates* of the impact of a proposed rebalance. You can approve or reject a proposal. You cannot approve a cluster rebalance without first generating the optimization

proposal.

You can run optimization proposals in one of the following rebalancing modes:

- `full`
- `add-brokers`
- `remove-brokers`

19.3.1. Rebalancing modes

You specify a rebalancing mode using the `spec.mode` property of the `KafkaRebalance` custom resource.

`full`

The `full` mode runs a full rebalance by moving replicas across all the brokers in the cluster. This is the default mode if the `spec.mode` property is not defined in the `KafkaRebalance` custom resource.

`add-brokers`

The `add-brokers` mode is used after scaling up a Kafka cluster by adding one or more brokers. Normally, after scaling up a Kafka cluster, new brokers are used to host only the partitions of newly created topics. If no new topics are created, the newly added brokers are not used and the existing brokers remain under the same load. By using the `add-brokers` mode immediately after adding brokers to the cluster, the rebalancing operation moves replicas from existing brokers to the newly added brokers. You specify the new brokers as a list using the `spec.brokers` property of the `KafkaRebalance` custom resource.

`remove-brokers`

The `remove-brokers` mode is used before scaling down a Kafka cluster by removing one or more brokers. If you scale down a Kafka cluster, brokers are shut down even if they host replicas. This can lead to under-replicated partitions and possibly result in some partitions being under their minimum ISR (in-sync replicas). To avoid this potential problem, the `remove-brokers` mode moves replicas off the brokers that are going to be removed. When these brokers are not hosting replicas anymore, you can safely run the scaling down operation. You specify the brokers you're removing as a list in the `spec.brokers` property in the `KafkaRebalance` custom resource.

In general, use the `full` rebalance mode to rebalance a Kafka cluster by spreading the load across brokers. Use the `add-brokers` and `remove-brokers` modes only if you want to scale your cluster up or down and rebalance the replicas accordingly.

The procedure to run a rebalance is actually the same across the three different modes. The only difference is with specifying a mode through the `spec.mode` property and, if needed, listing brokers that have been added or will be removed through the `spec.brokers` property.

19.3.2. The results of an optimization proposal

When an optimization proposal is generated, a summary and broker load is returned.

Summary

The summary is contained in the `KafkaRebalance` resource. The summary provides an overview of the proposed cluster rebalance and indicates the scale of the changes involved. A summary of a successfully generated optimization proposal is contained in the `Status.OptimizationResult` property of the `KafkaRebalance` resource. The information provided is a summary of the full optimization proposal.

Broker load

The broker load is stored in a ConfigMap that contains data as a JSON string. The broker load shows before and after values for the proposed rebalance, so you can see the impact on each of the brokers in the cluster.

19.3.3. Manually approving or rejecting an optimization proposal

An optimization proposal summary shows the proposed scope of changes.

You can use the name of the `KafkaRebalance` resource to return a summary from the command line.

Returning an optimization proposal summary

```
kubectl describe kafka-rebalance <kafka_rebalance_resource_name> -n <namespace>
```

You can also use the [jq command line JSON parser tool](#).

Returning an optimization proposal summary using jq

```
kubectl get kafka-rebalance -o json | jq <jq_query>.
```

Use the summary to decide whether to approve or reject an optimization proposal.

Approving an optimization proposal

You approve the optimization proposal by setting the `strimzi.io/rebalance` annotation of the `KafkaRebalance` resource to `approve`. Cruise Control applies the proposal to the Kafka cluster and starts a cluster rebalance operation.

Rejecting an optimization proposal

If you choose not to approve an optimization proposal, you can [change the optimization goals](#) or [update any of the rebalance performance tuning options](#), and then generate another proposal. You can generate a new optimization proposal for a `KafkaRebalance` resource by setting the `strimzi.io/rebalance` annotation to `refresh`.

Use optimization proposals to assess the movements required for a rebalance. For example, a summary describes inter-broker and intra-broker movements. Inter-broker rebalancing moves data between separate brokers. Intra-broker rebalancing moves data between disks on the same broker when you are using a JBOD storage configuration. Such information can be useful even if you don't go ahead and approve the proposal.

You might reject an optimization proposal, or delay its approval, because of the additional load on a

Kafka cluster when rebalancing.

In the following example, the proposal suggests the rebalancing of data between separate brokers. The rebalance involves the movement of 55 partition replicas, totaling 12MB of data, across the brokers. Though the inter-broker movement of partition replicas has a high impact on performance, the total amount of data is not large. If the total data was much larger, you could reject the proposal, or time when to approve the rebalance to limit the impact on the performance of the Kafka cluster.

Rebalance performance tuning options can help reduce the impact of data movement. If you can extend the rebalance period, you can divide the rebalance into smaller batches. Fewer data movements at a single time reduces the load on the cluster.

Example optimization proposal summary

```
Name:      my-rebalance
Namespace: myproject
Labels:    strimzi.io/cluster=my-cluster
Annotations: API Version: kafka.strimzi.io/v1alpha1
Kind:      KafkaRebalance
Metadata:
# ...
Status:
Conditions:
  Last Transition Time: 2022-04-05T14:36:11.900Z
  Status:             ProposalReady
  Type:               State
Observed Generation: 1
Optimization Result:
  Data To Move MB: 0
  Excluded Brokers For Leadership:
  Excluded Brokers For Replica Move:
  Excluded Topics:
    Intra Broker Data To Move MB: 12
    Monitored Partitions Percentage: 100
    Num Intra Broker Replica Movements: 0
    Num Leader Movements: 24
    Num Replica Movements: 55
    On Demand Balancedness Score After: 82.91290759174306
    On Demand Balancedness Score Before: 78.01176356230222
    Recent Windows: 5
Session Id:          a4f833bd-2055-4213-bfdd-ad21f95bf184
```

The proposal will also move 24 partition leaders to different brokers. This requires a change to the ZooKeeper configuration, which has a low impact on performance.

The balancedness scores are measurements of the overall balance of the Kafka cluster before and after the optimization proposal is approved. A balancedness score is based on optimization goals. If all goals are satisfied, the score is 100. The score is reduced for each goal that will not be met. Compare the balancedness scores to see whether the Kafka cluster is less balanced than it could be

following a rebalance.

19.3.4. Automatically approving an optimization proposal

To save time, you can automate the process of approving optimization proposals. With automation, when you generate an optimization proposal it goes straight into a cluster rebalance.

To enable the optimization proposal auto-approval mechanism, create the [KafkaRebalance](#) resource with the `strimzi.io/rebalance-auto-approval` annotation set to `true`. If the annotation is not set or set to `false`, the optimization proposal requires manual approval.

Example rebalance request with auto-approval mechanism enabled

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: "true"
spec:
  mode: # any mode
  # ...
```

You can still check the status when automatically approving an optimization proposal. The status of the [KafkaRebalance](#) resource moves to [Ready](#) when the rebalance is complete.

19.3.5. Optimization proposal summary properties

The following table explains the properties contained in the optimization proposal's summary section.

Table 33. Properties contained in an optimization proposal summary

JSON property	Description
<code>numIntraBrokerReplicaMovements</code>	The total number of partition replicas that will be transferred between the disks of the cluster's brokers. Performance impact during rebalance operation: Relatively high, but lower than <code>numReplicaMovements</code> .
<code>excludedBrokersForLeadership</code>	Not yet supported. An empty list is returned.
<code>numReplicaMovements</code>	The number of partition replicas that will be moved between separate brokers. Performance impact during rebalance operation: Relatively high.

JSON property	Description
<code>onDemandBalancednessScoreBefore</code> , <code>onDemandBalancednessScoreAfter</code>	<p>A measurement of the overall <i>balancedness</i> of a Kafka Cluster, before and after the optimization proposal was generated.</p> <p>The score is calculated by subtracting the sum of the <code>BalancednessScore</code> of each violated soft goal from 100. Cruise Control assigns a <code>BalancednessScore</code> to every optimization goal based on several factors, including priority—the goal’s position in the list of <code>default.goals</code> or user-provided goals.</p> <p>The <code>Before</code> score is based on the current configuration of the Kafka cluster. The <code>After</code> score is based on the generated optimization proposal.</p>
<code>intraBrokerDataToMoveMB</code>	<p>The sum of the size of each partition replica that will be moved between disks on the same broker (see also <code>numIntraBrokerReplicaMovements</code>).</p> <p>Performance impact during rebalance operation: Variable. The larger the number, the longer the cluster rebalance will take to complete. Moving a large amount of data between disks on the same broker has less impact than between separate brokers (see <code>dataToMoveMB</code>).</p>
<code>recentWindows</code>	<p>The number of metrics windows upon which the optimization proposal is based.</p>
<code>dataToMoveMB</code>	<p>The sum of the size of each partition replica that will be moved to a separate broker (see also <code>numReplicaMovements</code>).</p> <p>Performance impact during rebalance operation: Variable. The larger the number, the longer the cluster rebalance will take to complete.</p>
<code>monitoredPartitionsPercentage</code>	<p>The percentage of partitions in the Kafka cluster covered by the optimization proposal. Affected by the number of <code>excludedTopics</code>.</p>
<code>excludedTopics</code>	<p>If you specified a regular expression in the <code>spec.excludedTopicsRegex</code> property in the <code>KafkaRebalance</code> resource, all topic names matching that expression are listed here. These topics are excluded from the calculation of partition replica/leader movements in the optimization proposal.</p>

JSON property	Description
<code>numLeaderMovements</code>	The number of partitions whose leaders will be switched to different replicas. This involves a change to ZooKeeper configuration. Performance impact during rebalance operation: Relatively low.
<code>excludedBrokersForReplicaMove</code>	Not yet supported. An empty list is returned.

19.3.6. Broker load properties

The broker load is stored in a ConfigMap (with the same name as the KafkaRebalance custom resource) as a JSON formatted string. This JSON string consists of a JSON object with keys for each broker IDs linking to a number of metrics for each broker. Each metric consist of three values. The first is the metric value before the optimization proposal is applied, the second is the expected value of the metric after the proposal is applied, and the third is the difference between the first two values (after minus before).

NOTE

The ConfigMap appears when the KafkaRebalance resource is in the `ProposalReady` state and remains after the rebalance is complete.

You can use the name of the ConfigMap to view its data from the command line.

Returning ConfigMap data

```
kubectl describe configmaps <my_rebalance_configmap_name> -n <namespace>
```

You can also use the `jq` command line JSON parser tool to extract the JSON string from the ConfigMap.

Extracting the JSON string from the ConfigMap using jq

```
kubectl get configmaps <my_rebalance_configmap_name> -o json | jq  
'["data"]["brokerLoad.json"]|fromjson|.'
```

The following table explains the properties contained in the optimization proposal's broker load ConfigMap:

JSON property	Description
<code>leaders</code>	The number of replicas on this broker that are partition leaders.
<code>replicas</code>	The number of replicas on this broker.
<code>cpuPercentage</code>	The CPU utilization as a percentage of the defined capacity.
<code>diskUsedPercentage</code>	The disk utilization as a percentage of the defined capacity.

JSON property	Description
<code>diskUsedMB</code>	The absolute disk usage in MB.
<code>networkOutRate</code>	The total network output rate for the broker.
<code>leaderNetworkInRate</code>	The network input rate for all partition leader replicas on this broker.
<code>followerNetworkInRate</code>	The network input rate for all follower replicas on this broker.
<code>potentialMaxNetworkOutRate</code>	The hypothetical maximum network output rate that would be realized if this broker became the leader of all the replicas it currently hosts.

19.3.7. Cached optimization proposal

Cruise Control maintains a *cached optimization proposal* based on the configured default optimization goals. Generated from the workload model, the cached optimization proposal is updated every 15 minutes to reflect the current state of the Kafka cluster. If you generate an optimization proposal using the default optimization goals, Cruise Control returns the most recent cached proposal.

To change the cached optimization proposal refresh interval, edit the `proposal.expiration.ms` setting in the Cruise Control deployment configuration. Consider a shorter interval for fast changing clusters, although this increases the load on the Cruise Control server.

Additional resources

- [Optimization goals overview](#)
- [Generating optimization proposals](#)
- [Approving an optimization proposal](#)

19.4. Rebalance performance tuning overview

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

19.4.1. Partition reassignment commands

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

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

- Partition movement: Involves transferring the partition replica and its data to a new location. Partition movements can take one of two forms:
 - Inter-broker movement: The partition replica is moved to a log directory on a different broker.

- Intra-broker movement: The partition replica is moved to a different log directory on the same broker.
- Leadership movement: This involves switching the leader of the partition’s replicas.

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

19.4.2. Replica movement strategies

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

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

- [PrioritizeSmallReplicaMovementStrategy](#): Order reassessments in order of ascending size.
- [PrioritizeLargeReplicaMovementStrategy](#): Order reassessments in order of descending size.
- [PostponeUrpReplicaMovementStrategy](#): Prioritize reassessments for replicas of partitions which have no out-of-sync replicas.
- [PrioritizeMinIsrWithOfflineReplicasStrategy](#): Prioritize reassessments with (At/Under)MinISR partitions with offline replicas. This strategy will only work if `cruiseControl.config.concurrency.adjuster.min.isr.check.enabled` is set to `true` in the [Kafka](#) custom resource’s spec.

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

19.4.3. Intra-broker disk balancing

Moving a large amount of data between disks on the same broker has less impact than between separate brokers. If you are running a Kafka deployment that uses JBOD storage with multiple disks on the same broker, Cruise Control can balance partitions between the disks.

NOTE If you are using JBOD storage with a single disk, intra-broker disk balancing will result in a proposal with 0 partition movements since there are no disks to balance between.

To perform an intra-broker disk balance, set `rebalanceDisk` to `true` under the [KafkaRebalance.spec](#). When setting `rebalanceDisk` to `true`, do not set a `goals` field in the [KafkaRebalance.spec](#), as Cruise Control will automatically set the intra-broker goals and ignore the inter-broker goals. Cruise Control does not perform inter-broker and intra-broker balancing at the same time.

19.4.4. Rebalance tuning options

Cruise Control provides several configuration options for tuning the rebalance parameters discussed above. You can set these tuning options when [configuring and deploying Cruise Control with Kafka](#) or [optimization proposal](#) levels:

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

The relevant configurations are summarized in the following table.

Table 34. Rebalance performance tuning configuration

Cruise Control properties	KafkaRebalance properties	Default	Description
<code>num.concurrent.partition.movements.per.broker</code>	<code>concurrentPartitionMovementsPerBroker</code>	5	The maximum number of inter-broker partition movements in each partition reassignment batch
<code>num.concurrent.intra.broker.partition.movements</code>	<code>concurrentIntraBrokerPartitionMovements</code>	2	The maximum number of intra-broker partition movements in each partition reassignment batch
<code>num.concurrent.leader.movements</code>	<code>concurrentLeaderMovements</code>	1000	The maximum number of partition leadership changes in each partition reassignment batch
<code>default.replication.throttle</code>	<code>replicationThrottle</code>	Null (no limit)	The bandwidth (in bytes per second) to assign to partition reassignment

Cruise Control properties	KafkaRebalance properties	Default	Description
<code>default.replica.movement.strategies</code>	<code>replicaMovementStrategies</code>	<code>BaseReplicaMovementStrategy</code>	The list of strategies (in priority order) used to determine the order in which partition reassignment commands are executed for generated proposals. For the server setting, use a comma separated string with the fully qualified names of the strategy class (add <code>com.linkedin.kafka.cruisecontrol.executor.strategy</code> . to the start of each class name). For the KafkaRebalance resource setting use a YAML array of strategy class names.
-	<code>rebalanceDisk</code>	false	Enables intra-broker disk balancing, which balances disk space utilization between disks on the same broker. Only applies to Kafka deployments that use JBOD storage with multiple disks.

Changing the default settings affects the length of time that the rebalance takes to complete, as well

as the load placed on the Kafka cluster during the rebalance. Using lower values reduces the load but increases the amount of time taken, and vice versa.

Additional resources

- [CruiseControlSpec schema reference](#)
- [KafkaRebalanceSpec schema reference](#)

19.5. Configuring and deploying Cruise Control with Kafka

Configure a [Kafka](#) resource to deploy Cruise Control alongside a Kafka cluster. You can use the `cruiseControl` properties of the [Kafka](#) resource to configure the deployment. Deploy one instance of Cruise Control per Kafka cluster.

Use `goals` configuration in the Cruise Control `config` to specify optimization goals for generating optimization proposals. You can use `brokerCapacity` to change the default capacity limits for goals related to resource distribution. If brokers are running on nodes with heterogeneous network resources, you can use `overrides` to set network capacity limits for each broker.

If an empty object `({})` is used for the `cruiseControl` configuration, all properties use their default values.

For more information on the configuration options for Cruise Control, see the [Strimzi Custom Resource API Reference](#).

Prerequisites

- A Kubernetes cluster
- A running Cluster Operator

Procedure

1. Edit the `cruiseControl` property for the [Kafka](#) resource.

The properties you can configure are shown in this example configuration:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    brokerCapacity: ①
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    overrides: ②
    - brokers: [0]
      inboundNetwork: 20000KiB/s
      outboundNetwork: 20000KiB/s
```

```

- brokers: [1, 2]
  inboundNetwork: 30000KiB/s
  outboundNetwork: 30000KiB/s
  # ...
config: ③
  # Note that 'default.goals' (superset) must also include all 'hard.goals'
  (subset)
    default.goals: > ④
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
      # ...
    hard.goals: >
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal
      # ...
    cpu.balance.threshold: 1.1
    metadata.max.age.ms: 30000
    send.buffer.bytes: 131072
    webserver.http.cors.enabled: true ⑤
    webserver.http.cors.origin: "*"
    webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
    # ...
resources: ⑥
  requests:
    cpu: 1
    memory: 512Mi
  limits:
    cpu: 2
    memory: 2Gi
logging: ⑦
  type: inline
  loggers:
    rootLogger.level: INFO
template: ⑧
  pod:
    metadata:
      labels:
        label1: value1
    securityContext:
      runAsUser: 1000001
      fsGroup: 0
    terminationGracePeriodSeconds: 120
readinessProbe: ⑨
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
metricsConfig: ⑩
  type: jmxPrometheusExporter
  valueFrom:

```

```

configMapKeyRef:
  name: cruise-control-metrics
  key: metrics-config.yml
# ...

```

- ① Capacity limits for broker resources.
- ② Overrides set network capacity limits for specific brokers when running on nodes with heterogeneous network resources.
- ③ Cruise Control configuration. Standard Cruise Control configuration may be provided, restricted to those properties not managed directly by Strimzi.
- ④ Optimization goals configuration, which can include configuration for default optimization goals (`default.goals`), main optimization goals (`goals`), and hard goals (`hard.goals`).
- ⑤ CORS enabled and configured for read-only access to the Cruise Control API.
- ⑥ Requests for reservation of supported resources, currently `cpu` and `memory`, and limits to specify the maximum resources that can be consumed.
- ⑦ Cruise Control loggers and log levels added directly (`inline`) or indirectly (`external`) through a ConfigMap. A custom Log4j configuration must be placed under the `log4j.properties` key in the ConfigMap. Cruise Control has a single logger named `rootLogger.level`. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.
- ⑧ Template customization. Here a pod is scheduled with additional security attributes.
- ⑨ Healthchecks to know when to restart a container (liveness) and when a container can accept traffic (readiness).
- ⑩ Prometheus metrics enabled. In this example, metrics are configured for the Prometheus JMX Exporter (the default metrics exporter).

2. Create or update the resource:

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

3. Check the status of the deployment:

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

Output shows the deployment name and readiness

NAME	READY	UP-TO-DATE	AVAILABLE
my-cluster-cruise-control	1/1	1	1

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

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

Auto-created topics

The following table shows the three topics that are automatically created when Cruise Control is deployed. These topics are required for Cruise Control to work properly and must not be deleted or changed. You can change the name of the topic using the specified configuration option.

Table 35. Auto-created topics

Auto-created topic configuration	Default topic name	Created by	Function
<code>metric.reporter.topic</code>	<code>strimzi.cruisecontrol.metrics</code>	Strimzi Metrics Reporter	Stores the raw metrics from the Metrics Reporter in each Kafka broker.
<code>partition.metric.sample.store.topic</code>	<code>strimzi.cruisecontrol.partitionmetricsamples</code>	Cruise Control	Stores the derived metrics for each partition. These are created by the Metric Sample Aggregator .
<code>broker.metric.sample.store.topic</code>	<code>strimzi.cruisecontrol.modeltrainingsamples</code>	Cruise Control	Stores the metrics samples used to create the Cluster Workload Model .

To prevent the removal of records that are needed by Cruise Control, log compaction is disabled in the auto-created topics.

NOTE If the names of the auto-created topics are changed in a Kafka cluster that already has Cruise Control enabled, the old topics will not be deleted and should be manually removed.

What to do next

After configuring and deploying Cruise Control, you can [generate optimization proposals](#).

Additional resources

- [Optimization goals overview](#)

19.6. Generating optimization proposals

When you create or update a [KafkaRebalance](#) resource, Cruise Control generates an [optimization proposal](#) for the Kafka cluster based on the configured [optimization goals](#). Analyze the information in the optimization proposal and decide whether to approve it. You can use the results of the optimization proposal to rebalance your Kafka cluster.

You can run the optimization proposal in one of the following modes:

- `full` (default)
- `add-brokers`
- `remove-brokers`

The mode you use depends on whether you are rebalancing across all the brokers already running

in the Kafka cluster; or you want to rebalance after scaling up or before scaling down your Kafka cluster. For more information, see [Rebalancing modes with broker scaling](#).

Prerequisites

- You have [deployed Cruise Control](#) to your Strimzi cluster.
- You have configured optimization goals and, optionally, capacity limits on broker resources.

For more information on configuring Cruise Control, see [Configuring and deploying Cruise Control with Kafka](#).

Procedure

1. Create a `KafkaRebalance` resource and specify the appropriate mode.

full mode (default)

To use the *default optimization goals* defined in the `Kafka` resource, leave the `spec` property empty. Cruise Control rebalances a Kafka cluster in `full` mode by default.

Example configuration with full rebalancing by default

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec: {}
```

You can also run a full rebalance by specifying the `full` mode through the `spec.mode` property.

Example configuration specifying full mode

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: full
```

add-brokers mode

If you want to rebalance a Kafka cluster after scaling up, specify the `add-brokers` mode.

In this mode, existing replicas are moved to the newly added brokers. You need to specify the brokers as a list.

Example configuration specifying add-brokers mode

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

```
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: add-brokers
  brokers: [3, 4] ①
```

① List of newly added brokers added by the scale up operation. This property is mandatory.

remove-brokers mode

If you want to rebalance a Kafka cluster before scaling down, specify the `remove-brokers` mode.

In this mode, replicas are moved off the brokers that are going to be removed. You need to specify the brokers that are being removed as a list.

Example configuration specifying `remove-brokers` mode

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: remove-brokers
  brokers: [3, 4] ①
```

① List of brokers to be removed by the scale down operation. This property is mandatory.

NOTE

The following steps and the steps to approve or stop a rebalance are the same regardless of the rebalance mode you are using.

2. To configure *user-provided optimization goals* instead of using the default goals, add the `goals` property and enter one or more goals.

In the following example, rack awareness and replica capacity are configured as user-provided optimization goals:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
```

- ReplicaCapacityGoal

3. To ignore the configured hard goals, add the `skipHardGoalCheck: true` property:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

4. (Optional) To approve the optimization proposal automatically, set the `strimzi.io/rebalance-auto-approval` annotation to `true`:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: "true"
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

5. Create or update the resource:

```
kubectl apply -f <kafka_rebalance_configuration_file>
```

The Cluster Operator requests the optimization proposal from Cruise Control. This might take a few minutes depending on the size of the Kafka cluster.

6. If you used the automatic approval mechanism, wait for the status of the optimization proposal to change to `Ready`. If you haven't enabled the automatic approval mechanism, wait for the status of the optimization proposal to change to `ProposalReady`:

```
kubectl get kafkarebalance -o wide -w -n <namespace>
```

PendingProposal

A `PendingProposal` status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

ProposalReady

A `ProposalReady` status means the optimization proposal is ready for review and approval.

When the status changes to `ProposalReady`, the optimization proposal is ready to approve.

7. Review the optimization proposal.

The optimization proposal is contained in the `Status.Optimization Result` property of the `KafkaRebalance` resource.

```
kubectl describe kafka_rebalance <kafka_rebalance_resource_name>
```

Example optimization proposal

```
Status:  
Conditions:  
  Last Transition Time: 2020-05-19T13:50:12.533Z  
  Status: ProposalReady  
  Type: State  
  Observed Generation: 1  
Optimization Result:  
  Data To Move MB: 0  
  Excluded Brokers For Leadership:  
  Excluded Brokers For Replica Move:  
  Excluded Topics:  
    Intra Broker Data To Move MB: 0  
    Monitored Partitions Percentage: 100  
    Num Intra Broker Replica Movements: 0  
    Num Leader Movements: 0  
    Num Replica Movements: 26  
    On Demand Balancedness Score After: 81.8666802863978  
    On Demand Balancedness Score Before: 78.01176356230222  
    Recent Windows: 1  
Session Id: 05539377-ca7b-45ef-b359-e13564f1458c
```

The properties in the `Optimization Result` section describe the pending cluster rebalance operation. For descriptions of each property, see [Contents of optimization proposals](#).

Insufficient CPU capacity

If a Kafka cluster is overloaded in terms of CPU utilization, you might see an insufficient CPU capacity error in the `KafkaRebalance` status. It's worth noting that this utilization value is unaffected by the `excludedTopics` configuration. Although optimization proposals will not reassign replicas of excluded topics, their load is still considered in the utilization calculation.

Example CPU utilization error

```
com.linkedin.kafka.cruisecontrol.exception.OptimizationFailureException:  
[CpuCapacityGoal] Insufficient capacity for cpu (Utilization 615.21, Allowed Capacity  
420.00, Threshold: 0.70). Add at least 3 brokers with the same cpu capacity (100.00)  
as broker-0. Add at least 3 brokers with the same cpu capacity (100.00) as broker-0.
```

NOTE

The error shows CPU capacity as a percentage rather than the number of CPU cores. For this reason, it does not directly map to the number of CPUs configured in the Kafka custom resource. It is like having a single *virtual* CPU per broker, which has the cycles of the CPUs configured in `Kafka.spec.kafka.resources.limits.cpu`. This has no effect on the rebalance behavior, since the ratio between CPU utilization and capacity remains the same.

What to do next

Approving an optimization proposal

Additional resources

- [Optimization proposals overview](#)

19.7. Approving an optimization proposal

You can approve an [optimization proposal](#) generated by Cruise Control, if its status is `ProposalReady`. Cruise Control will then apply the optimization proposal to the Kafka cluster, reassigning partitions to brokers and changing partition leadership.

This is not a dry run. Before you approve an optimization proposal, you must:

CAUTION

- Refresh the proposal in case it has become out of date.
- Carefully review the [contents of the proposal](#).

Prerequisites

- You have [generated an optimization proposal](#) from Cruise Control.
- The `KafkaRebalance` custom resource status is `ProposalReady`.

Procedure

Perform these steps for the optimization proposal that you want to approve.

1. Unless the optimization proposal is newly generated, check that it is based on current information about the state of the Kafka cluster. To do so, refresh the optimization proposal to make sure it uses the latest cluster metrics:
 - a. Annotate the `KafkaRebalance` resource in Kubernetes with `strimzi.io/rebalance=refresh`:

```
kubectl annotate kafka-rebalance <kafka_rebalance_resource_name>  
strimzi.io/rebalance="refresh"
```

- Wait for the status of the optimization proposal to change to **ProposalReady**:

```
kubectl get kafkaebalance -o wide -w -n <namespace>
```

PendingProposal

A **PendingProposal** status means the rebalance operator is polling the Cruise Control API to check if the optimization proposal is ready.

ProposalReady

A **ProposalReady** status means the optimization proposal is ready for review and approval.

When the status changes to **ProposalReady**, the optimization proposal is ready to approve.

- Approve the optimization proposal that you want Cruise Control to apply.

Annotate the **KafkaRebalance** resource in Kubernetes with `strimzi.io/rebalance=approve`:

```
kubectl annotate kafkaebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance="approve"
```

- The Cluster Operator detects the annotated resource and instructs Cruise Control to rebalance the Kafka cluster.

- Wait for the status of the optimization proposal to change to **Ready**:

```
kubectl get kafkaebalance -o wide -w -n <namespace>
```

Rebalancing

A **Rebalancing** status means the rebalancing is in progress.

Ready

A **Ready** status means the rebalance is complete.

NotReady

A **NotReady** status means an error occurred—see [Fixing problems with a KafkaRebalance resource](#).

When the status changes to **Ready**, the rebalance is complete.

To use the same **KafkaRebalance** custom resource to generate another optimization proposal, apply the `refresh` annotation to the custom resource. This moves the custom resource to the **PendingProposal** or **ProposalReady** state. You can then review the optimization proposal and approve it, if desired.

Additional resources

- [Optimization proposals overview](#)

- [Stopping a cluster rebalance](#)

19.8. Stopping a cluster rebalance

Once started, a cluster rebalance operation might take some time to complete and affect the overall performance of the Kafka cluster.

If you want to stop a cluster rebalance operation that is in progress, apply the `stop` annotation to the `KafkaRebalance` custom resource. This instructs Cruise Control to finish the current batch of partition reassessments and then stop the rebalance. When the rebalance has stopped, completed partition reassessments have already been applied; therefore, the state of the Kafka cluster is different when compared to prior to the start of the rebalance operation. If further rebalancing is required, you should generate a new optimization proposal.

NOTE

The performance of the Kafka cluster in the intermediate (stopped) state might be worse than in the initial state.

Prerequisites

- You have [approved the optimization proposal](#) by annotating the `KafkaRebalance` custom resource with `approve`.
- The status of the `KafkaRebalance` custom resource is `Rebalancing`.

Procedure

1. Annotate the `KafkaRebalance` resource in Kubernetes:

```
kubectl annotate kafka-rebalance rebalance-cr-name strimzi.io/rebalance="stop"
```

2. Check the status of the `KafkaRebalance` resource:

```
kubectl describe kafka-rebalance rebalance-cr-name
```

3. Wait until the status changes to `Stopped`.

Additional resources

- [Optimization proposals overview](#)

19.9. Fixing problems with a `KafkaRebalance` resource

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

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

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

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

Prerequisites

- You have [approved an optimization proposal](#).
- The status of the `KafkaRebalance` custom resource for the rebalance operation is `NotReady`.

Procedure

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

```
kubectl describe kafka-rebalance-rebalance-cr-name
```

2. Attempt to resolve the issue in the `KafkaRebalance` resource.

3. Annotate the `KafkaRebalance` resource in Kubernetes:

```
kubectl annotate kafka-rebalance-rebalance-cr-name strimzi.io/rebalance="refresh"
```

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

```
kubectl describe kafka-rebalance-rebalance-cr-name
```

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

Additional resources

- [Optimization proposals overview](#)

Chapter 20. Using Cruise Control to modify topic replication factor

Change the replication factor of topics by updating the [KafkaTopic](#) resource managed by the Topic Operator. You can adjust the replication factor for specific purposes, such as:

- Setting a lower replication factor for non-critical topics or because of resource shortages
- Setting a higher replication factor to improve data durability and fault tolerance

The Topic Operator uses Cruise Control to make the necessary changes, so Cruise Control must be deployed with Strimzi.

The Topic Operator watches and periodically reconciles all managed and unpause Kafka resources to detect changes to `.spec.replicas` configuration by comparing the replication factor of the topic in Kafka. One or more replication factor updates are then sent to Cruise Control for processing in a single request.

Progress is reflected in the status of the [KafkaTopic](#) resource.

Prerequisites

- [The Cluster Operator must be deployed](#).
- [The Topic Operator must be deployed](#) to manage topics through the [KafkaTopic](#) custom resource.
- [Cruise Control is deployed with Kafka](#).

Procedure

1. Edit the [KafkaTopic](#) resource to change the `replicas` value.

In this procedure, we change the `replicas` value for `my-topic` from 1 to 3.

Kafka topic replication factor configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  # ...
```

2. Apply the change to the [KafkaTopic](#) configuration and wait for the Topic Operator to update the topic.
3. Check the status of the [KafkaTopic](#) resource to make sure the request was successful:

```
oc get kafkatopics my-topic -o yaml
```

Status for the replication factor change

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  # ...
# ...
status:
  conditions:
  - lastTransitionTime: "2024-01-18T16:13:50.490918232Z"
    status: "True"
    type: Ready
  observedGeneration: 2
  replicasChange:
    sessionId: 1aa418ca-53ed-4b93-b0a4-58413c4fc0cb ①
    state: ongoing ②
    targetReplicas: 3 ③
  topicName: my-topic
```

- ① The session ID for the Cruise Control operation, which is shown when process moves out of a pending state.
- ② The state of the update. Moves from `pending` to `ongoing`, and then the entire `replicasChange` status is removed when the change is complete.
- ③ The requested change to the number of replicas.

An error message is shown in the status if the request fails before completion. The request is periodically retried if it enters a failed state.

Changing topic replication factor using the standalone Topic Operator

If you are using the standalone Topic Operator and aim to change the topic replication factor through configuration, you still need to use the Topic Operator in unidirectional mode alongside a Cruise Control deployment. You also need to include the following environment variables in the standalone Topic Operator deployment so that it can integrate with Cruise Control.

Example standalone Topic Operator deployment configuration

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-topic-operator
```

```

labels:
  app: strimzi
spec:
# ...
template:
# ...
spec:
# ...
containers:
- name: strimzi-topic-operator
# ...
env:
# ...
- name: STRIMZI_CRUISE_CONTROL_ENABLED ①
  value: true
- name: STRIMZI_CRUISE_CONTROL_RACK_ENABLED ②
  value: false
- name: STRIMZI_CRUISE_CONTROL_HOSTNAME ③
  value: cruise-control-api.namespace.svc
- name: STRIMZI_CRUISE_CONTROL_PORT ④
  value: 9090
- name: STRIMZI_CRUISE_CONTROL_SSL_ENABLED ⑤
  value: true
- name: STRIMZI_CRUISE_CONTROL_AUTH_ENABLED ⑥
  value: true

```

① Integrates Cruise Control with the Topic Operator.

② Flag to indicate whether rack awareness is enabled on the Kafka cluster. If so, replicas can be spread across different racks, data centers, or availability zones.

③ Cruise Control hostname.

④ Cruise control port.

⑤ Enables TLS authentication and encryption for accessing the Kafka cluster.

⑥ Enables basic authorization for accessing the Cruise Control API.

If you enable TLS authentication and authorization, mount the required certificates as follows:

- Public certificates of the Cluster CA (certificate authority) in [/etc/tls-sidecar/cluster-ca-certs/ca.crt](#)
- Basic authorization credentials (user name and password) in [/etc/eto-cc-api/topic-operator.apiAdminName](#) and [/etc/eto-cc-api/topic-operator.apiAdminPassword](#)

Chapter 21. Using the partition reassignment tool

You can use the `kafka-reassign-partitions.sh` tool for the following:

- Adding or removing brokers
- Reassigning partitions across brokers
- Changing the replication factor of topics

However, while `kafka-reassign-partitions.sh` supports these operations, it is generally easier with Cruise Control. Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

To use the `kafka-reassign-partitions.sh` tool, run it as a separate interactive pod rather than within the broker container. Running the Kafka `bin/` scripts within the broker container may cause a JVM to start with the same settings as the Kafka broker, which can potentially cause disruptions. By running the `kafka-reassign-partitions.sh` tool in a separate pod, you can avoid this issue. Running a pod with the `-ti` option creates an interactive pod with a terminal for running shell commands inside the pod.

Running an interactive pod with a terminal

```
kubectl run helper-pod -ti --image=quay.io/stimzi/kafka:0.42.0-kafka-3.7.1 --rm=true  
--restart=Never -- bash
```

21.1. Partition reassignment tool overview

The partition reassignment tool provides the following capabilities for managing Kafka partitions and brokers:

Redistributing partition replicas

Scale your cluster up and down by adding or removing brokers, and move Kafka partitions from heavily loaded brokers to under-utilized brokers. To do this, you must create a partition reassignment plan that identifies which topics and partitions to move and where to move them. Cruise Control is recommended for this type of operation as it [automates the cluster rebalancing process](#).

Scaling topic replication factor up and down

Increase or decrease the replication factor of your Kafka topics. To do this, you must create a partition reassignment plan that identifies the existing replication assignment across partitions and an updated assignment with the replication factor changes.

Changing the preferred leader

Change the preferred leader of a Kafka partition. This can be useful if the current preferred leader is unavailable or if you want to redistribute load across the brokers in the cluster. To do this, you must create a partition reassignment plan that specifies the new preferred leader for

each partition by changing the order of replicas.

Changing the log directories to use a specific JBOD volume

Change the log directories of your Kafka brokers to use a specific JBOD volume. This can be useful if you want to move your Kafka data to a different disk or storage device. To do this, you must create a partition reassignment plan that specifies the new log directory for each topic.

21.1.1. Generating a partition reassignment plan

The partition reassignment tool ([kafka-reassign-partitions.sh](#)) works by generating a partition assignment plan that specifies which partitions should be moved from their current broker to a new broker.

If you are satisfied with the plan, you can execute it. The tool then does the following:

- Migrates the partition data to the new broker
- Updates the metadata on the Kafka brokers to reflect the new partition assignments
- Triggers a rolling restart of the Kafka brokers to ensure that the new assignments take effect

The partition reassignment tool has three different modes:

--generate

Takes a set of topics and brokers and generates a *reassignment JSON file* which will result in the partitions of those topics being assigned to those brokers. Because this operates on whole topics, it cannot be used when you only want to reassign some partitions of some topics.

--execute

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

--verify

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

It is only possible to have one reassignment running in a cluster at any given time, and it is not possible to cancel a running reassignment. If you must cancel a reassignment, wait for it to complete and then perform another reassignment to revert the effects of the first reassignment. The [kafka-reassign-partitions.sh](#) will print the reassignment JSON for this reversion as part of its output. Very large reassessments should be broken down into a number of smaller reassessments in case there is a need to stop in-progress reassignment.

21.1.2. Specifying topics in a partition reassignment JSON file

The [kafka-reassign-partitions.sh](#) tool uses a reassignment JSON file that specifies the topics to

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

A basic reassignment JSON file has the structure presented in the following example, which describes three partitions belonging to two Kafka topics. Each partition is reassigned to a new set of replicas, which are identified by their broker IDs. The `version`, `topic`, `partition`, and `replicas` properties are all required.

Example partition reassignment JSON file structure

```
{  
  "version": 1, ①  
  "partitions": [ ②  
    {  
      "topic": "example-topic-1", ③  
      "partition": 0, ④  
      "replicas": [1, 2, 3] ⑤  
    },  
    {  
      "topic": "example-topic-1",  
      "partition": 1,  
      "replicas": [2, 3, 4]  
    },  
    {  
      "topic": "example-topic-2",  
      "partition": 0,  
      "replicas": [3, 4, 5]  
    }  
  ]  
}
```

- ① The version of the reassignment JSON file format. Currently, only version 1 is supported, so this should always be 1.
- ② An array that specifies the partitions to be reassigned.
- ③ The name of the Kafka topic that the partition belongs to.
- ④ The ID of the partition being reassigned.
- ⑤ An ordered array of the IDs of the brokers that should be assigned as replicas for this partition. The first broker in the list is the leader replica.

NOTE Partitions not included in the JSON are not changed.

If you specify only topics using a `topics` array, the partition reassignment tool reassigns all the partitions belonging to the specified topics.

Example reassignment JSON file structure for reassigning all partitions for a topic

```
{  
  "version": 1,
```

```
"topics": [
    { "topic": "my-topic"}
]
}
```

21.1.3. Reassigning partitions between JBOD volumes

When using JBOD storage in your Kafka cluster, you can reassign the partitions between specific volumes and their log directories (each volume has a single log directory).

To reassign a partition to a specific volume, add `log_dirs` values for each partition in the reassignment JSON file. Each `log_dirs` array contains the same number of entries as the `replicas` array, since each replica should be assigned to a specific log directory. The `log_dirs` array contains either an absolute path to a log directory or the special value `any`. The `any` value indicates that Kafka can choose any available log directory for that replica, which can be useful when reassigning partitions between JBOD volumes.

Example reassignment JSON file structure with log directories

```
{
    "version": 1,
    "partitions": [
        {
            "topic": "example-topic-1",
            "partition": 0,
            "replicas": [1, 2, 3],
            "log_dirs": ["/var/lib/kafka/data-0/kafka-log1", "any", "/var/lib/kafka/data-1/kafka-log2"]
        },
        {
            "topic": "example-topic-1",
            "partition": 1,
            "replicas": [2, 3, 4],
            "log_dirs": ["any", "/var/lib/kafka/data-2/kafka-log3", "/var/lib/kafka/data-3/kafka-log4"]
        },
        {
            "topic": "example-topic-2",
            "partition": 0,
            "replicas": [3, 4, 5],
            "log_dirs": ["/var/lib/kafka/data-4/kafka-log5", "any", "/var/lib/kafka/data-5/kafka-log6"]
        }
    ]
}
```

21.1.4. Throttling partition reassignment

Partition reassignment can be a slow process because it involves transferring large amounts of data

between brokers. To avoid a detrimental impact on clients, you can throttle the reassignment process. Use the `--throttle` parameter with the `kafka-reassign-partitions.sh` tool to throttle a reassignment. You specify a maximum threshold in bytes per second for the movement of partitions between brokers. For example, `--throttle 5000000` sets a maximum threshold for moving partitions of 50 MBps.

Throttling might cause the reassignment to take longer to complete.

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

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

21.2. Generating a reassignment JSON file to reassign partitions

Generate a reassignment JSON file with the `kafka-reassign-partitions.sh` tool to reassign partitions after scaling a Kafka cluster. Adding or removing brokers does not automatically redistribute the existing partitions. To balance the partition distribution and take full advantage of the new brokers, you can reassign the partitions using the `kafka-reassign-partitions.sh` tool.

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

The following procedure describes a secure reassignment process that uses mTLS. You'll need a Kafka cluster that uses TLS encryption and mTLS authentication.

You'll need the following to establish a connection:

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

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

Prerequisites

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

Kafka configuration with TLS encryption and mTLS authentication

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

① Enables TLS encryption for the internal listener.

② Listener authentication mechanism specified as mutual `tls`.

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

Example topic configuration for `my-topic`

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

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

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

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

```

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

```

① User authentication mechanism defined as mutual `tls`.

② Simple authorization and accompanying list of ACL rules.

Procedure

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

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

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

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

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

```
kubectl run --restart=Never --image=quay.io/stimzi/kafka:0.42.0-kafka-3.7.1  
<interactive_pod_name> -- /bin/sh -c "sleep 3600"
```

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

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

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

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

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

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

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

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

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

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

6. Create a **config.properties** file to specify the truststore and keystore used to authenticate connection to the Kafka cluster.

Use the certificates and passwords you extracted in the previous steps.

```
bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093 ①  
security.protocol=SSL ②  
ssl.truststore.location=/tmp/ca.p12 ③  
ssl.truststore.password=<truststore_password> ④  
ssl.keystore.location=/tmp/user.p12 ⑤  
ssl.keystore.password=<keystore_password> ⑥
```

① The bootstrap server address to connect to the Kafka cluster. Use your own Kafka cluster name to replace *<kafka_cluster_name>*.

② The security protocol option when using TLS for encryption.

- ③ The truststore location contains the public key certificate (`ca.p12`) for the Kafka cluster.
- ④ The password (`ca.password`) for accessing the truststore.
- ⑤ The keystore location contains the public key certificate (`user.p12`) for the Kafka user.
- ⑥ The password (`user.password`) for accessing the keystore.

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

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

8. Prepare a JSON file named `topics.json` that specifies the topics to move.

Specify topic names as a comma-separated list.

Example JSON file to reassign all the partitions of my-topic

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

You can also use this file to [change the replication factor of a topic](#).

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

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

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

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

Replace `<namespace>` with the Kubernetes namespace where the pod is running.

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

Example command to move the partitions of my-topic to specified brokers

```
bin/kafka-reassign-partitions.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 \
\ --command-config /tmp/config.properties \
--topics-to-move-json-file /tmp/topics.json \
--broker-list 0,1,2,3,4 \
--generate
```

Additional resources

- [Configuring Kafka](#)
- [Configuring Kafka topics](#)
- [Configuring Kafka users](#)

21.3. Using the partition reassignment tool to reassign partitions after adding brokers

Use a reassignment file generated by the `kafka-reassign-partitions.sh` tool to reassign partitions after increasing the number of brokers in a Kafka cluster. The reassignment file should describe how partitions are reassigned to brokers in the enlarged Kafka cluster. You apply the reassignment specified in the file to the brokers and then verify the new partition assignments.

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

The `kafka-reassign-partitions.sh` tool can be used to reassign partitions within a Kafka cluster, regardless of whether you are managing all nodes through the cluster or using the node pools to manage groups of nodes within the cluster.

NOTE

Though you can use the `kafka-reassign-partitions.sh` tool for this operation, Cruise Control is recommended for automated partition reassessments and cluster rebalancing. Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

Prerequisites

- You have a running Kafka cluster based on a `Kafka` resource configured with internal TLS encryption and mTLS authentication.
- You have [generated a reassignment JSON file named `reassignment.json`](#).
- You are running an interactive pod container that is connected to the running Kafka broker.
- You are connected as a `KafkaUser` configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

Procedure

1. Add as many new brokers as you need by increasing the `Kafka.spec.kafka.replicas` configuration option.
2. Verify that the new broker pods have started.
3. If you haven't done so, [run an interactive pod container to generate a reassignment JSON file named `reassignment.json`](#).
4. Copy the `reassignment.json` file to the interactive pod container.

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

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

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

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

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

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

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

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

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

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

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

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

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

7. Verify that the reassignment has completed using the `kafka-reassign-partitions.sh` command

line tool from any of the broker pods. This is the same command as the previous step, but with the `--verify` option instead of the `--execute` option.

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

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

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

21.4. Using the partition reassignment tool to reassign partitions before removing brokers

Use a reassignment file generated by the `kafka-reassign-partitions.sh` tool to reassign partitions before decreasing the number of brokers in a Kafka cluster. The reassignment file must describe how partitions are reassigned to the remaining brokers in the Kafka cluster. You apply the reassignment specified in the file to the brokers and then verify the new partition assignments. Brokers in the highest numbered pods are removed first.

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

The `kafka-reassign-partitions.sh` tool can be used to reassign partitions within a Kafka cluster, regardless of whether you are managing all nodes through the cluster or using the node pools to manage groups of nodes within the cluster.

NOTE

Though you can use the `kafka-reassign-partitions.sh` tool for this operation, Cruise Control is recommended for automated partition reassessments and cluster rebalancing. Cruise Control can move topics from one broker to another without any downtime, and it is the most efficient way to reassign partitions.

Prerequisites

- You have a running Kafka cluster based on a `Kafka` resource configured with internal TLS encryption and mTLS authentication.
- You have generated a reassignment JSON file named `reassignment.json`.
- You are running an interactive pod container that is connected to the running Kafka broker.
- You are connected as a `KafkaUser` configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

Procedure

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

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

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

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

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

Replace `<namespace>` with the Kubernetes namespace where the pod is running.

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

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

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

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

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

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

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

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

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

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

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

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

6. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.
7. When all the partition reassessments have finished, the brokers being removed should not have responsibility for any of the partitions in the cluster. You can verify this by checking that the broker's data log directory does not contain any live partition logs. If the log directory on the broker contains a directory that does not match the extended regular expression `\.[a-zA-Z0-9]-delete$`, the broker still has live partitions and should not be stopped.

You can check this by executing the command:

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

where n is the number of the pods being deleted.

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

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

21.5. Using the partition reassignment tool to modify topic replication factor

Use a reassignment file generated by the `kafka-reassign-partitions.sh` tool to change the replication factor of topics.

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

NOTE

Though you can use the `kafka-reassign-partitions.sh` tool for this operation, the simplest approach is to enable Cruise Control and make the change through the `KafkaTopic` configuration. For more information, see [Using Cruise Control to modify topic replication factor](#).

Prerequisites

- You have a running Kafka cluster based on a `Kafka` resource configured with internal TLS encryption and mTLS authentication.
- You are running an interactive pod container that is connected to the running Kafka broker.
- You have generated a reassignment JSON file named `reassignment.json`.
- You are connected as a `KafkaUser` configured with ACL rules that specify permission to manage the Kafka cluster and its topics.

See [Generating reassignment JSON files](#).

In this procedure, a topic called `my-topic` has 4 replicas and we want to reduce it to 3. A JSON file named `topics.json` specifies the topic, and was used to generate the `reassignment.json` file.

Example JSON file specifies `my-topic`

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

Procedure

1. If you haven't done so, [run an interactive pod container to generate a reassignment JSON file named `reassignment.json`](#).

Example reassignment JSON file showing the current and proposed replica assignment

```
Current partition replica assignment  
{ "version":1,"partitions": [{"topic": "my-  
topic", "partition":0, "replicas": [3,4,2,0], "log_dirs": ["any", "any", "any", "any"]}, {"t  
opic": "my-  
topic", "partition":1, "replicas": [0,2,3,1], "log_dirs": ["any", "any", "any", "any"]}, {"t
```

```
topic": "my-topic", "partition": 2, "replicas": [1, 3, 0, 4], "log_dirs": ["any", "any", "any", "any"]}]}}
```

Proposed partition reassignment configuration

```
{"version": 1, "partitions": [{"topic": "my-topic", "partition": 0, "replicas": [0, 1, 2, 3], "log_dirs": ["any", "any", "any", "any"]}, {"topic": "my-topic", "partition": 1, "replicas": [1, 2, 3, 4], "log_dirs": ["any", "any", "any", "any"]}, {"topic": "my-topic", "partition": 2, "replicas": [2, 3, 4, 0], "log_dirs": ["any", "any", "any", "any"]}]}]
```

Save a copy of this file locally in case you need to revert the changes later on.

2. Edit the `reassignment.json` to remove a replica from each partition.

For example use the [jq command line JSON parser tool](#) to remove the last replica in the list for each partition of the topic:

Removing the last topic replica for each partition

```
jq '.partitions[].replicas |= del(.-[-1])' reassignment.json > reassignment.json
```

Example reassignment file showing the updated replicas

```
{"version": 1, "partitions": [{"topic": "my-topic", "partition": 0, "replicas": [0, 1, 2], "log_dirs": ["any", "any", "any", "any"]}, {"topic": "my-topic", "partition": 1, "replicas": [1, 2, 3], "log_dirs": ["any", "any", "any", "any"]}, {"topic": "my-topic", "partition": 2, "replicas": [2, 3, 4], "log_dirs": ["any", "any", "any", "any"]}]}]
```

3. Copy the `reassignment.json` file to the interactive pod container.

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

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

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

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

Replace `<namespace>` with the Kubernetes namespace where the pod is running.

5. Make the topic replica change using the `kafka-reassign-partitions.sh` script from the interactive pod container.

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

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

NOTE Removing replicas from a broker does not require any inter-broker data movement, so there is no need to throttle replication. If you are adding replicas, then you may want to change the throttle rate.

6. Verify that the change to the topic replicas has completed using the `kafka-reassign-partitions.sh` command line tool from any of the broker pods. This is the same command as the previous step, but with the `--verify` option instead of the `--execute` option.

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

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

7. Run the `bin/kafka-topics.sh` command with the `--describe` option to see the results of the change to the topics.

```
bin/kafka-topics.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--describe
```

Results of reducing the number of replicas for a topic

```
my-topic Partition: 0 Leader: 0 Replicas: 0,1,2 Isr: 0,1,2
my-topic Partition: 1 Leader: 2 Replicas: 1,2,3 Isr: 1,2,3
my-topic Partition: 2 Leader: 3 Replicas: 2,3,4 Isr: 2,3,4
```

8. Finally, edit the `KafkaTopic` custom resource to change `.spec.replicas` to 3, and then wait the reconciliation.

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

```
spec:  
  partitions: 3  
  replicas: 3
```

Chapter 22. Introducing metrics

Collecting metrics is critical for understanding the health and performance of your Kafka deployment. By monitoring metrics, you can actively identify issues before they become critical and make informed decisions about resource allocation and capacity planning. Without metrics, you may be left with limited visibility into the behavior of your Kafka deployment, which can make troubleshooting more difficult and time-consuming. Setting up metrics can save you time and resources in the long run, and help ensure the reliability of your Kafka deployment.

Metrics are available for each component in Strimzi, providing valuable insights into their individual performance. While other components require configuration to expose metrics, Strimzi operators automatically expose Prometheus metrics by default. These metrics include:

- Reconciliation count
- Custom Resource count being processed
- Reconciliation duration
- JVM metrics

You can also collect metrics specific to `oauth` authentication and `opa` or `keycloak` authorization by enabling the `enableMetrics` property in the listener or authorization configuration of the `Kafka` resource. Similarly, you can enable metrics for `oauth` authentication in custom resources such as `KafkaBridge`, `KafkaConnect`, `KafkaMirrorMaker`, and `KafkaMirrorMaker2`.

You can use Prometheus and Grafana to monitor Strimzi. Prometheus consumes metrics from the running pods in your cluster when configured with Prometheus rules. Grafana visualizes these metrics on dashboards, providing an intuitive interface for monitoring.

To facilitate metrics integration, Strimzi provides example Prometheus rules and Grafana dashboards for Strimzi components. You can customize the example Grafana dashboards to suit your specific deployment requirements. You can use rules to define conditions that trigger alerts based on specific metrics.

Depending on your monitoring requirements, you can do the following:

- [Set up and deploy Prometheus to expose metrics](#)
- [Deploy Kafka Exporter to provide additional metrics](#)
- [Use Grafana to present the Prometheus metrics](#)

Additionally, you can configure your deployment to track messages end-to-end by [setting up distributed tracing](#).

NOTE Strimzi provides example installation files for Prometheus and Grafana, which can serve as a starting point for monitoring your Strimzi deployment. For further support, try engaging with the Prometheus and Grafana developer communities.

Supporting documentation for metrics and monitoring tools

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

- [Prometheus](#)
- [Prometheus configuration](#)
- [Kafka Exporter](#)
- [Grafana Labs](#)
- [Apache Kafka Monitoring](#) describes JMX metrics exposed by Apache Kafka
- [ZooKeeper JMX](#) describes JMX metrics exposed by Apache ZooKeeper

22.1. Monitoring consumer lag with Kafka Exporter

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

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

A Grafana dashboard for Kafka Exporter is one of a number of [example Grafana dashboards](#) provided by Strimzi.

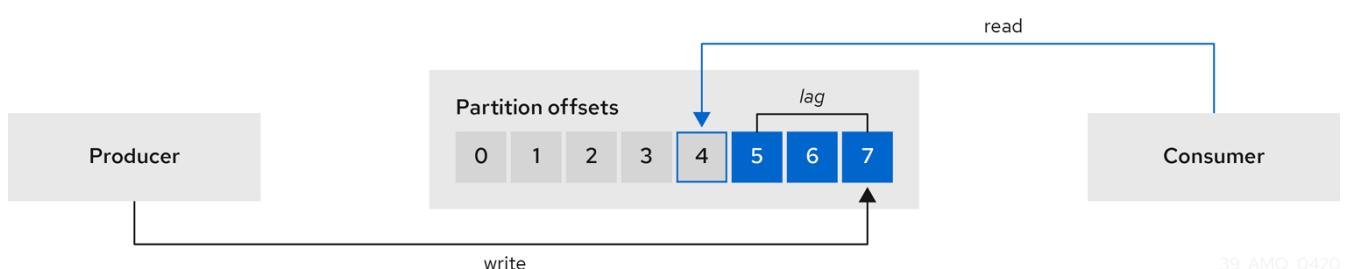
IMPORTANT

Kafka Exporter provides only additional metrics related to consumer lag and consumer offsets. For regular Kafka metrics, you have to configure the Prometheus metrics in [Kafka brokers](#).

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

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

Consumer lag between the producer and consumer offset



39_AMQ_0420

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

Suppose a topic streams 100 messages a second. A lag of 1000 messages between the producer offset

(the topic partition head) and the last offset the consumer has read means a 10-second delay.

The importance of monitoring consumer lag

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

Consumer lag, for example, might be a result of consuming too much old data that has not been purged, or through unplanned shutdowns.

Reducing consumer lag

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

Typical actions to reduce lag include:

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

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

22.2. Monitoring Cruise Control operations

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

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

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

You expose Cruise Control metrics by [enabling the Prometheus JMX Exporter in the Cruise Control configuration](#).

NOTE

For a full list of available Cruise Control metrics, which are known as *sensors*, see the [Cruise Control documentation](#)

22.2.1. Monitoring balancedness scores

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

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

Refreshing the `KafkaRebalance` resource fetches an optimization proposal. The latest cached optimization proposal is fetched if one of the following conditions applies:

NOTE

- KafkaRebalance `goals` match the goals configured in the `default.goals` section of the `Kafka` resource
- KafkaRebalance `goals` are not specified

Otherwise, Cruise Control generates a new optimization proposal based on KafkaRebalance `goals`. If new proposals are generated with each refresh, this can impact performance monitoring.

22.2.2. Setting up alerts for anomaly detection

Cruise control's *anomaly detector* provides metrics data for conditions that block the generation of optimization goals, such as broker failures. If you want more visibility, you can use the metrics provided by the anomaly detector to set up alerts and send out notifications. You can set up Cruise Control's *anomaly notifier* to route alerts based on these metrics through a specified notification channel. Alternatively, you can set up Prometheus to scrape the metrics data provided by the anomaly detector and generate alerts. Prometheus Alertmanager can then route the alerts generated by Prometheus.

The [Cruise Control documentation](#) provides information on `AnomalyDetector` metrics and the anomaly notifier.

22.3. Example metrics files

You can find example Grafana dashboards and other metrics configuration files in the [example configuration files](#) provided by Strimzi.

Example metrics files provided with Strimzi

```
metrics
└── grafana-dashboards ①
    ├── strimzi-cruise-control.json
    ├── strimzi-kafka-bridge.json
    ├── strimzi-kafka-connect.json
    ├── strimzi-kafka-exporter.json
    └── strimzi-kafka-mirror-maker-2.json
```

```

    ├── strimzi-kafka.json
    ├── strimzi-operators.json
    └── strimzi-zookeeper.json
    ├── grafana-install
    │   └── grafana.yaml ②
    ├── prometheus-additional-properties
    │   └── prometheus-additional.yaml ③
    ├── prometheus-alertmanager-config
    │   └── alert-manager-config.yaml ④
    ├── prometheus-install
    │   ├── alert-manager.yaml ⑤
    │   ├── prometheus-rules.yaml ⑥
    │   ├── prometheus.yaml ⑦
    │   └── strimzi-pod-monitor.yaml ⑧
    ├── kafka-bridge-metrics.yaml ⑨
    ├── kafka-connect-metrics.yaml ⑩
    ├── kafka-cruise-control-metrics.yaml ⑪
    ├── kafka-metrics.yaml ⑫
    └── kafka-mirror-maker-2-metrics.yaml ⑬

```

- ① Example Grafana dashboards for the different Strimzi components.
- ② Installation file for the Grafana image.
- ③ Additional configuration to scrape metrics for CPU, memory and disk volume usage, which comes directly from the Kubernetes cAdvisor agent and kubelet on the nodes.
- ④ Hook definitions for sending notifications through Alertmanager.
- ⑤ Resources for deploying and configuring Alertmanager.
- ⑥ Alerting rules examples for use with Prometheus Alertmanager (deployed with Prometheus).
- ⑦ Installation resource file for the Prometheus image.
- ⑧ PodMonitor definitions translated by the Prometheus Operator into jobs for the Prometheus server to be able to scrape metrics data directly from pods.
- ⑨ Kafka Bridge resource with metrics enabled.
- ⑩ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka Connect.
- ⑪ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Cruise Control.
- ⑫ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka and ZooKeeper.
- ⑬ Metrics configuration that defines Prometheus JMX Exporter relabeling rules for Kafka MirrorMaker 2.

22.3.1. Example Prometheus metrics configuration

Strimzi uses the [Prometheus JMX Exporter](#) to expose metrics through an HTTP endpoint, which can be scraped by the Prometheus server.

Grafana dashboards are dependent on Prometheus JMX Exporter relabeling rules, which are defined for Strimzi components in the custom resource configuration.

A label is a name-value pair. Relabeling is the process of writing a label dynamically. For example, the value of a label may be derived from the name of a Kafka server and client ID.

Strimzi provides example custom resource configuration YAML files with relabeling rules. When deploying Prometheus metrics configuration, you can deploy the example custom resource or copy the metrics configuration to your own custom resource definition.

Table 36. Example custom resources with metrics configuration

Component	Custom resource	Example YAML file
Kafka and ZooKeeper	<code>Kafka</code>	<code>kafka-metrics.yaml</code>
Kafka Connect	<code>KafkaConnect</code>	<code>kafka-connect-metrics.yaml</code>
Kafka MirrorMaker 2	<code>KafkaMirrorMaker2</code>	<code>kafka-mirror-maker-2-metrics.yaml</code>
Kafka Bridge	<code>KafkaBridge</code>	<code>kafka-bridge-metrics.yaml</code>
Cruise Control	<code>Kafka</code>	<code>kafka-cruise-control-metrics.yaml</code>

22.3.2. Example Prometheus rules for alert notifications

Example Prometheus rules for alert notifications are provided with the [example metrics configuration files](#) provided by Strimzi. The rules are specified in the example `prometheus-rules.yaml` file for use in a [Prometheus deployment](#).

The `prometheus-rules.yaml` file contains example rules for the following components:

- Kafka
- ZooKeeper
- Entity Operator
- Kafka Connect
- Kafka Bridge
- MirrorMaker
- Kafka Exporter

A description of each of the example rules is provided in the file.

Alerting rules provide notifications about specific conditions observed in metrics. Rules are declared on the Prometheus server, but Prometheus Alertmanager is responsible for alert notifications.

Prometheus alerting rules describe conditions using [PromQL](#) expressions that are continuously evaluated.

When an alert expression becomes true, the condition is met and the Prometheus server sends alert data to the Alertmanager. Alertmanager then sends out a notification using the communication method configured for its deployment.

General points about the alerting rule definitions:

- A `for` property is used with the rules to determine the period of time a condition must persist before an alert is triggered.
- A tick is a basic ZooKeeper time unit, which is measured in milliseconds and configured using the `tickTime` parameter of `Kafka.spec.zookeeper.config`. For example, if ZooKeeper `tickTime=3000`, 3 ticks (3 x 3000) equals 9000 milliseconds.
- The availability of the `ZookeeperRunningOutOfSpace` metric and alert is dependent on the Kubernetes configuration and storage implementation used. Storage implementations for certain platforms may not be able to supply the information on available space required for the metric to provide an alert.

Alertmanager can be configured to use email, chat messages or other notification methods. Adapt the default configuration of the example rules according to your specific needs.

22.3.3. Example Grafana dashboards

If you deploy Prometheus to provide metrics, you can use the example Grafana dashboards provided with Strimzi to monitor Strimzi components.

Example dashboards are provided in the `examples/metrics/grafana-dashboards` directory as JSON files.

All dashboards provide JVM metrics, as well as metrics specific to the component. For example, the Grafana dashboard for Strimzi operators provides information on the number of reconciliations or custom resources they are processing.

The example dashboards don't show all the metrics supported by Kafka. The dashboards are populated with a representative set of metrics for monitoring.

Table 37. Example Grafana dashboard files

Component	Example JSON file
Strimzi operators	<code>strimzi-operators.json</code>
Kafka	<code>strimzi-kafka.json</code>
ZooKeeper	<code>strimzi-zookeeper.json</code>
Kafka Connect	<code>strimzi-kafka-connect.json</code>
Kafka MirrorMaker 2	<code>strimzi-kafka-mirror-maker-2.json</code>
Kafka Bridge	<code>strimzi-kafka-bridge.json</code>
Cruise Control	<code>strimzi-cruise-control.json</code>
Kafka Exporter	<code>strimzi-kafka-exporter.json</code>

NOTE

When metrics are not available to the Kafka Exporter, because there is no traffic in the cluster yet, the Kafka Exporter Grafana dashboard will show `N/A` for numeric fields and `No data to show` for graphs.

22.4. Using Prometheus with Strimzi

You can use Prometheus to provide monitoring data for the example Grafana dashboards provided with Strimzi.

To expose metrics in Prometheus format, you add configuration to a custom resource. You must also make sure that the metrics are scraped by your monitoring stack. Prometheus and Prometheus Alertmanager are used in the examples provided by Strimzi, but you can use also other compatible tools.

Using Prometheus with Strimzi, requires the following:

1. [Enabling Prometheus metrics through configuration](#)
2. [Setting up Prometheus](#)
3. [Deploying Prometheus Alertmanager](#)

Strimzi provides an [example Grafana dashboards](#) to display visualizations of metrics. The exposed metrics provide the monitoring data when you [enable the Grafana dashboard](#).

22.4.1. Enabling Prometheus metrics through configuration

To enable and expose metrics in Strimzi for Prometheus, use metrics configuration properties.

The following components require `metricsConfig` configuration to expose metrics:

- Kafka
- KafkaConnect
- MirrorMaker
- Cruise Control
- ZooKeeper

This configuration enables the [Prometheus JMX Exporter](#) to expose metrics through an HTTP endpoint. The port for the JMX exporter HTTP endpoint is 9404. Prometheus scrapes this endpoint to collect Kafka metrics.

Set the `enableMetrics` property to `true` in order to expose metrics for these components:

- Kafka Bridge
- OAuth 2.0 authentication and authorization framework
- Open Policy Agent (OPA) for authorization

To deploy Prometheus metrics configuration in Strimzi, you can use your own configuration or the [example custom resource configuration files](#) provided with Strimzi:

- `kafka-metrics.yaml`
- `kafka-connect-metrics.yaml`

- `kafka-mirror-maker-2-metrics.yaml`
- `kafka-bridge-metrics.yaml`
- `kafka-cruise-control-metrics.yaml`
- `oauth-metrics.yaml`

These files contain the necessary relabeling rules and configuration to enable Prometheus metrics. They are a good starting point for trying Prometheus with Strimzi.

This procedure shows how to deploy example Prometheus metrics configuration in the [Kafka](#) resource. The process is the same when deploying the example files for other resources.

If you wish to include [Kafka Exporter](#) metrics, add `kafkaExporter` configuration to your [Kafka](#) resource.

IMPORTANT Kafka Exporter only provides additional metrics related to consumer lag and consumer offsets. For regular Kafka metrics, configure Prometheus metrics in the [Kafka](#) resource.

Procedure

1. Deploy the example custom resource with the Prometheus configuration.

For example, for each [Kafka](#) resource you can apply the `kafka-metrics.yaml` file.

Deploying the example configuration

```
kubectl apply -f kafka-metrics.yaml
```

Alternatively, copy the example configuration in `kafka-metrics.yaml` to your own [Kafka](#) resource.

Copying the example configuration

```
kubectl edit kafka <kafka_configuration_file>
```

Copy the `metricsConfig` property and the `ConfigMap` it references to your [Kafka](#) resource.

Example metrics configuration for Kafka

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metricsConfig: ①
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
```

```

name: kafka-metrics
key: kafka-metrics-config.yml
---
kind: ConfigMap ②
apiVersion: v1
metadata:
  name: kafka-metrics
  labels:
    app: strimzi
data:
  kafka-metrics-config.yml: |
    # metrics configuration...

```

① Copy the `metricsConfig` property that references the `ConfigMap` containing metrics configuration.

② Copy the whole `ConfigMap` specifying the metrics configuration.

2. To deploy Kafka Exporter, add `kafkaExporter` configuration.

`kafkaExporter` configuration is specified only in the `Kafka` resource.

Example configuration for deploying Kafka Exporter

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    image: my-registry.io/my-org/my-exporter-cluster:latest ①
    groupRegex: ".*" ②
    topicRegex: ".*" ③
    groupExcludeRegex: "^excluded-.*" ④
    topicExcludeRegex: "^excluded-.*" ⑤
    showAllOffsets: false ⑥
    resources: ⑦
      requests:
        cpu: 200m
        memory: 64Mi
      limits:
        cpu: 500m
        memory: 128Mi
    logging: debug ⑧
    enableSaramaLogging: true ⑨
    template: ⑩
      pod:
        metadata:
          labels:
            label1: value1
        imagePullSecrets:

```

```

    - name: my-docker-credentials
      securityContext:
        runAsUser: 1000001
        fsGroup: 0
      terminationGracePeriodSeconds: 120
      readinessProbe: ⑪
        initialDelaySeconds: 15
        timeoutSeconds: 5
      livenessProbe: ⑫
        initialDelaySeconds: 15
        timeoutSeconds: 5
    # ...

```

- ① ADVANCED OPTION: Container image configuration, which is recommended only in special situations.
- ② A regular expression to specify the consumer groups to include in the metrics.
- ③ A regular expression to specify the topics to include in the metrics.
- ④ A regular expression to specify the consumer groups to exclude in the metrics.
- ⑤ A regular expression to specify the topics to exclude in the metrics.
- ⑥ By default, metrics are collected for all consumers regardless of their connection status.
Setting `showAllOffsets` to `false` stops collecting metrics on disconnected consumers.
- ⑦ CPU and memory resources to reserve.
- ⑧ Logging configuration, to log messages with a given severity (debug, info, warn, error, fatal) or above.
- ⑨ Boolean to enable Sarama logging, a Go client library used by Kafka Exporter.
- ⑩ Customization of deployment templates and pods.
- ⑪ Healthcheck readiness probes.
- ⑫ Healthcheck liveness probes.

NOTE For Kafka Exporter to be able to work properly, consumer groups need to be in use.

Enabling metrics for Kafka Bridge

To expose metrics for Kafka Bridge, set the `enableMetrics` property to `true` in the `KafkaBridge` resource.

Example metrics configuration for Kafka Bridge

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:

```

```
# ...
enableMetrics: true
# ...
```

Enabling metrics for OAuth 2.0 and OPA

To expose metrics for OAuth 2.0 or OPA, set the `enableMetrics` property to `true` in the appropriate custom resource.

OAuth 2.0 metrics

Enable metrics for Kafka cluster authorization and Kafka listener authentication in the [Kafka](#) resource. You can also enable metrics for OAuth 2.0 authentication in the custom resource of other [supported components](#).

OPA metrics

Enable metrics for Kafka cluster authorization in the [Kafka](#) resource similar to OAuth 2.0.

In the following example, metrics are enabled for OAuth 2.0 listener authentication and OAuth 2.0 ([keycloak](#)) cluster authorization.

Example cluster configuration with metrics enabled for OAuth 2.0

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        authentication:
          type: oauth
          enableMetrics: true
        configuration:
          #...
    authorization:
      type: keycloak
      enableMetrics: true
  # ...
```

To use OAuth 2.0 metrics with Prometheus, copy the [ConfigMap](#) configuration from the `oauth-metrics.yaml` file to the same [Kafka](#) resource configuration file where you enabled metrics for OAuth 2.0 and then apply the configuration.

22.4.2. Setting up Prometheus

Prometheus provides an open source set of components for systems monitoring and alert notification.

We describe here how you can use the CoreOS Prometheus Operator to run and manage a Prometheus server that is suitable for use in production environments, but with the correct configuration you can run any Prometheus server.

NOTE

The Prometheus server configuration uses service discovery to discover the pods in the cluster from which it gets metrics. For this feature to work correctly, the service account used for running the Prometheus service pod must have access to the API server so it can retrieve the pod list.

For more information, see [Discovering services](#).

Prometheus configuration

Strimzi provides [example configuration files for the Prometheus server](#).

A Prometheus YAML file is provided for deployment:

- `prometheus.yaml`

Additional Prometheus-related configuration is also provided in the following files:

- `prometheus-additional.yaml`
- `prometheus-rules.yaml`
- `strimzi-pod-monitor.yaml`

For Prometheus to obtain monitoring data:

- [Deploy the Prometheus Operator](#)

Then use the configuration files to:

- [Deploy Prometheus](#)

Alerting rules

The `prometheus-rules.yaml` file provides [example alerting rule examples for use with Alertmanager](#).

Prometheus resources

When you apply the Prometheus configuration, the following resources are created in your Kubernetes cluster and managed by the Prometheus Operator:

- A `ClusterRole` that grants permissions to Prometheus to read the health endpoints exposed by the Kafka and ZooKeeper pods, cAdvisor and the kubelet for container metrics.
- A `ServiceAccount` for the Prometheus pods to run under.

- A `ClusterRoleBinding` which binds the `ClusterRole` to the `ServiceAccount`.
- A `Deployment` to manage the Prometheus Operator pod.
- A `PodMonitor` to manage the configuration of the Prometheus pod.
- A `Prometheus` to manage the configuration of the Prometheus pod.
- A `PrometheusRule` to manage alerting rules for the Prometheus pod.
- A `Secret` to manage additional Prometheus settings.
- A `Service` to allow applications running in the cluster to connect to Prometheus (for example, Grafana using Prometheus as datasource).

Deploying the CoreOS Prometheus Operator

To deploy the Prometheus Operator to your Kafka cluster, apply the YAML bundle resources file from the [Prometheus CoreOS repository](#).

Procedure

1. Download the `bundle.yaml` resources file from the repository:

```
curl -s https://raw.githubusercontent.com/coreos/prometheus-operator/master/bundle.yaml > prometheus-operator-deployment.yaml
```

- Use the latest `master` release as shown, or choose a release that is compatible with your version of Kubernetes (see the [Kubernetes compatibility matrix](#)). The `master` release of the Prometheus Operator works with Kubernetes 1.18+.
- Update the namespace by replacing the example `my-namespace` with your own.

On Linux, use:

```
sed -E -i '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s(namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' prometheus-operator-deployment.yaml
```

On MacOS, use:

```
sed -i '' -e '/[[:space:]]*namespace: [a-zA-Z0-9-]*$/s(namespace:[[:space:]]*[a-zA-Z0-9-]*$/namespace: my-namespace/' prometheus-operator-deployment.yaml
```

NOTE

If using OpenShift, specify a release of the [OpenShift fork](#) of the Prometheus Operator repository.

2. (Optional) If it is not required, you can manually remove the `spec.template.spec.securityContext` property from the `prometheus-operator-deployment.yaml` file.
3. Deploy the Prometheus Operator:

```
kubectl create -f prometheus-operator-deployment.yaml
```

Deploying Prometheus

Use Prometheus to obtain monitoring data in your Kafka cluster.

You can use your own Prometheus deployment or deploy Prometheus using the [example metrics configuration files](#) provided by Strimzi. The example files include a configuration file for a Prometheus deployment and files for Prometheus-related resources:

- [examples/metrics/prometheus-install/prometheus.yaml](#)
- [examples/metrics/prometheus-install/prometheus-rules.yaml](#)
- [examples/metrics/prometheus-install/stimzi-pod-monitor.yaml](#)
- [examples/metrics/prometheus-additional-properties/prometheus-additional.yaml](#)

The deployment process creates a [ClusterRoleBinding](#) and discovers an Alertmanager instance in the namespace specified for the deployment.

NOTE

By default, the Prometheus Operator only supports jobs that include an [endpoints](#) role for service discovery. Targets are discovered and scraped for each endpoint port address. For endpoint discovery, the port address may be derived from service ([role: service](#)) or pod ([role: pod](#)) discovery.

Prerequisites

- Check the [example alerting rules provided](#)

Procedure

1. Modify the Prometheus installation file ([prometheus.yaml](#)) according to the namespace Prometheus is going to be installed into:

On Linux, use:

```
sed -i 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
```

2. Edit the [PodMonitor](#) resource in [stimzi-pod-monitor.yaml](#) to define Prometheus jobs that will scrape the metrics data from pods.

Update the [namespaceSelector.matchNames](#) property with the namespace where the pods to scrape the metrics from are running.

[PodMonitor](#) is used to scrape data directly from pods for Apache Kafka, ZooKeeper, Operators,

the Kafka Bridge and Cruise Control.

3. Edit the `prometheus.yaml` installation file to include additional configuration for scraping metrics directly from nodes.

The Grafana dashboards provided show metrics for CPU, memory and disk volume usage, which come directly from the Kubernetes cAdvisor agent and kubelet on the nodes.

The Prometheus Operator does not have a monitoring resource like `PodMonitor` for scraping the nodes, so the `prometheus-additional.yaml` file contains the additional configuration needed.

- a. Create a `Secret` resource from the configuration file (`prometheus-additional.yaml` in the `examples/metrics/prometheus-additional-properties` directory):

```
kubectl apply -f prometheus-additional.yaml
```

- b. Edit the `additionalScrapeConfigs` property in the `prometheus.yaml` file to include the name of the `Secret` in the `prometheus-additional.yaml` file.

4. Deploy the Prometheus resources:

```
kubectl apply -f strimzi-pod-monitor.yaml  
kubectl apply -f prometheus-rules.yaml  
kubectl apply -f prometheus.yaml
```

22.4.3. Deploying Alertmanager

Use Alertmanager to route alerts to a notification service. [Prometheus Alertmanager](#) is a component for handling alerts and routing them to a notification service. Alertmanager supports an essential aspect of monitoring, which is to be notified of conditions that indicate potential issues based on alerting rules.

You can use the [example metrics configuration files](#) provided by Strimzi to deploy Alertmanager to send notifications to a Slack channel. A configuration file defines the resources for deploying Alertmanager:

- `examples/metrics/prometheus-install/alert-manager.yaml`

An additional configuration file provides the hook definitions for sending notifications from your Kafka cluster:

- `examples/metrics/prometheus-alertmanager-config/alert-manager-config.yaml`

The following resources are defined on deployment:

- An `Alertmanager` to manage the Alertmanager pod.
- A `Secret` to manage the configuration of the Alertmanager.
- A `Service` to provide an easy to reference hostname for other services to connect to

Alertmanager (such as Prometheus).

Prerequisites

- Metrics are configured for the Kafka cluster resource
- Prometheus is deployed

Procedure

1. Update the `alert-manager-config.yaml` file in the `examples/metrics/prometheus-alertmanager-config` directory to replace the following:
 - `slack_api_url` property with the actual value of the Slack API URL related to the application for the Slack workspace
 - `channel` property with the actual Slack channel on which to send notifications
2. Create a `Secret` resource from the Alertmanager configuration file:

```
kubectl apply -f alert-manager-config.yaml
```

3. Deploy Alertmanager:

```
kubectl apply -f alert-manager.yaml
```

22.5. Enabling the example Grafana dashboards

Use Grafana to provide visualizations of Prometheus metrics on customizable dashboards.

You can use your own Grafana deployment or deploy Grafana using the [example metrics configuration files](#) provided by Strimzi. The example files include a configuration file for a Grafana deployment

- `examples/metrics/grafana-install/grafana.yaml`

Strimzi also provides [example dashboard configuration files for Grafana](#) in JSON format.

- `examples/metrics/grafana-dashboards`

This procedure uses the example Grafana configuration file and example dashboards.

The example dashboards are a good starting point for monitoring key metrics, but they don't show all the metrics supported by Kafka. You can modify the example dashboards or add other metrics, depending on your infrastructure.

NOTE No alert notification rules are defined.

When accessing a dashboard, you can use the `port-forward` command to forward traffic from the Grafana pod to the host. The name of the Grafana pod is different for each user.

Prerequisites

- Metrics are configured for the Kafka cluster resource
- Prometheus and Prometheus Alertmanager are deployed

Procedure

- Deploy Grafana.

```
kubectl apply -f grafana.yaml
```

- Get the details of the Grafana service.

```
kubectl get service grafana
```

For example:

NAME	TYPE	CLUSTER-IP	PORT(S)
grafana	ClusterIP	172.30.123.40	3000/TCP

Note the port number for port forwarding.

- Use `port-forward` to redirect the Grafana user interface to `localhost:3000`:

```
kubectl port-forward svc/grafana 3000:3000
```

- In a web browser, access the Grafana login screen using the URL <http://localhost:3000>.

The Grafana Log In page appears.

- Enter your user name and password, and then click **Log In**.

The default Grafana user name and password are both `admin`. After logging in for the first time, you can change the password.

- In **Configuration > Data Sources**, add Prometheus as a *data source*.

- Specify a name
- Add *Prometheus* as the type
- Specify a Prometheus server URL

The Prometheus operator service (`prometheus-operated`) is accessible internally within the Kubernetes cluster on port 9090: <http://prometheus-operated:9090>.

Save and test the connection when you have added the details.

- Click the + icon and then click **Import**.

- In `examples/metrics/grafana-dashboards`, copy the JSON of the dashboard to import.

9. Paste the JSON into the text box, and then click **Load**.
10. Repeat steps 7-9 for the other example Grafana dashboards.

The imported Grafana dashboards are available to view from the **Dashboards** home page.

Chapter 23. Introducing distributed tracing

Distributed tracing tracks the progress of transactions between applications in a distributed system. In a microservices architecture, tracing tracks the progress of transactions between services. Trace data is useful for monitoring application performance and investigating issues with target systems and end-user applications.

In Strimzi, tracing facilitates the end-to-end tracking of messages: from source systems to Kafka, and then from Kafka to target systems and applications. Distributed tracing complements the monitoring of metrics in Grafana dashboards, as well as the component loggers.

Support for tracing is built in to the following Kafka components:

- MirrorMaker to trace messages from a source cluster to a target cluster
- Kafka Connect to trace messages consumed and produced by Kafka Connect
- Kafka Bridge to trace messages between Kafka and HTTP client applications

Tracing is not supported for Kafka brokers.

You enable and configure tracing for these components through their custom resources. You add tracing configuration using `spec.template` properties.

You enable tracing by specifying a tracing type using the `spec.tracing.type` property:

opentelemetry

Specify `type: opentelemetry` to use OpenTelemetry. By Default, OpenTelemetry uses the OTLP (OpenTelemetry Protocol) exporter and endpoint to get trace data. You can specify other tracing systems supported by OpenTelemetry, including Jaeger tracing. To do this, you change the OpenTelemetry exporter and endpoint in the tracing configuration.

CAUTION

Strimzi no longer supports OpenTracing. If you were previously using OpenTracing with the `type: jaeger` option, we encourage you to transition to using OpenTelemetry instead.

23.1. Tracing options

Use OpenTelemetry with the Jaeger tracing system.

OpenTelemetry provides an API specification that is independent from the tracing or monitoring system.

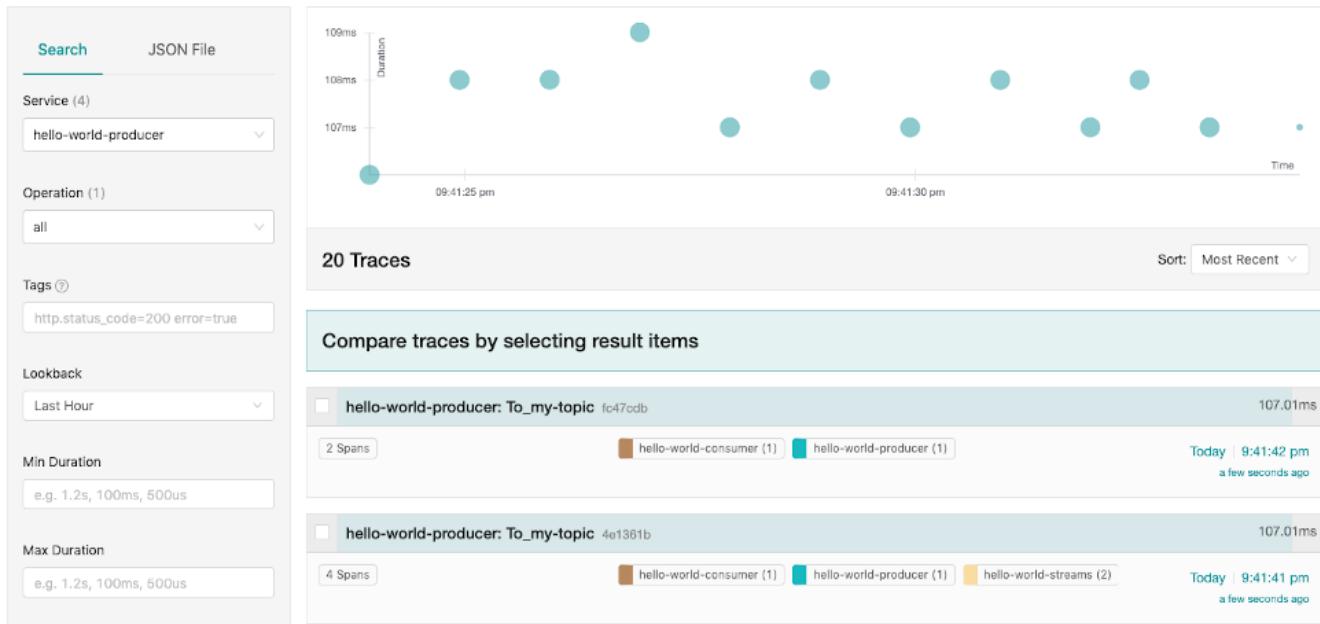
You use the APIs to instrument application code for tracing.

- Instrumented applications generate *traces* for individual requests across the distributed system.
- Traces are composed of *spans* that define specific units of work over time.

Jaeger is a tracing system for microservices-based distributed systems.

- The Jaeger user interface allows you to query, filter, and analyze trace data.

The Jaeger user interface showing a simple query



Additional resources

- [Jaeger documentation](#)
- [OpenTelemetry documentation](#)

23.2. Environment variables for tracing

Use environment variables when you are enabling tracing for Kafka components or initializing a tracer for Kafka clients.

Tracing environment variables are subject to change. For the latest information, see the [OpenTelemetry documentation](#).

The following tables describe the key environment variables for setting up a tracer.

Table 38. OpenTelemetry environment variables

Property	Required	Description
OTEL_SERVICE_NAME	Yes	The name of the Jaeger tracing service for OpenTelemetry.
OTEL_EXPORTER_JAEGER_ENDPOINT	Yes	The exporter used for tracing.
OTEL_TRACES_EXPORTER	Yes	The exporter used for tracing. Set to <code>otlp</code> by default. If using Jaeger tracing, you need to set this environment variable as <code>jaeger</code> . If you are using another tracing implementation, specify the exporter used .

23.3. Setting up distributed tracing

Enable distributed tracing in Kafka components by specifying a tracing type in the custom resource. Instrument tracers in Kafka clients for end-to-end tracking of messages.

To set up distributed tracing, follow these procedures in order:

- [Enable tracing for MirrorMaker, Kafka Connect, and the Kafka Bridge](#)
- Set up tracing for clients:
 - [Initialize a Jaeger tracer for Kafka clients](#)
- Instrument clients with tracers:
 - [Instrument producers and consumers for tracing](#)
 - [Instrument Kafka Streams applications for tracing](#)

23.3.1. Prerequisites

Before setting up distributed tracing, make sure Jaeger backend components are deployed to your Kubernetes cluster. We recommend using the Jaeger operator for deploying Jaeger on your Kubernetes cluster.

For deployment instructions, see the [Jaeger documentation](#).

NOTE

Setting up tracing for applications and systems beyond Strimzi is outside the scope of this content.

23.3.2. Enabling tracing in MirrorMaker, Kafka Connect, and Kafka Bridge resources

Distributed tracing is supported for MirrorMaker, MirrorMaker 2, Kafka Connect, and the Kafka Bridge. Configure the custom resource of the component to specify and enable a tracer service.

Enabling tracing in a resource triggers the following events:

- Interceptor classes are updated in the integrated consumers and producers of the component.
- For MirrorMaker, MirrorMaker 2, and Kafka Connect, the tracing agent initializes a tracer based on the tracing configuration defined in the resource.
- For the Kafka Bridge, a tracer based on the tracing configuration defined in the resource is initialized by the Kafka Bridge itself.

You can enable tracing that uses OpenTelemetry.

Tracing in MirrorMaker and MirrorMaker 2

For MirrorMaker and MirrorMaker 2, messages are traced from the source cluster to the target cluster. The trace data records messages entering and leaving the MirrorMaker or MirrorMaker 2 component.

Tracing in Kafka Connect

For Kafka Connect, only messages produced and consumed by Kafka Connect are traced. To trace messages sent between Kafka Connect and external systems, you must configure tracing in the connectors for those systems.

Tracing in the Kafka Bridge

For the Kafka Bridge, messages produced and consumed by the Kafka Bridge are traced. Incoming HTTP requests from client applications to send and receive messages through the Kafka Bridge are also traced. To have end-to-end tracing, you must configure tracing in your HTTP clients.

Procedure

Perform these steps for each [KafkaMirrorMaker](#), [KafkaMirrorMaker2](#), [KafkaConnect](#), and [KafkaBridge](#) resource.

1. In the `spec.template` property, configure the tracer service.

- Use the [tracing environment variables](#) as template configuration properties.
- For OpenTelemetry, set the `spec.tracing.type` property to `opentelemetry`.

Example tracing configuration for Kafka Connect using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
    tracing:
      type: opentelemetry
      #...
```

Example tracing configuration for MirrorMaker using OpenTelemetry

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  #...
  template:
    mirrorMakerContainer:
      env:
```

```

    - name: OTEL_SERVICE_NAME
      value: my-otel-service
    - name: OTEL_EXPORTER_OTLP_ENDPOINT
      value: "http://otlp-host:4317"
  tracing:
    type: opentelemetry
#...

```

Example tracing configuration for MirrorMaker 2 using OpenTelemetry

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
    tracing:
      type: opentelemetry
#...

```

Example tracing configuration for the Kafka Bridge using OpenTelemetry

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-otel-service
        - name: OTEL_EXPORTER_OTLP_ENDPOINT
          value: "http://otlp-host:4317"
    tracing:
      type: opentelemetry
#...

```

2. Create or update the resource:

```
kubectl apply -f <resource_configuration_file>
```

23.3.3. Initializing tracing for Kafka clients

Initialize a tracer for OpenTelemetry, then instrument your client applications for distributed tracing. You can instrument Kafka producer and consumer clients, and Kafka Streams API applications.

Configure and initialize a tracer using a set of [tracing environment variables](#).

Procedure

In each client application add the dependencies for the tracer:

1. Add the Maven dependencies to the [pom.xml](#) file for the client application:

Dependencies for OpenTelemetry

```
<dependency>
    <groupId>io.opentelemetry.semconv</groupId>
    <artifactId>opentelemetry-semconv</artifactId>
    <version>1.21.0-alpha</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-otlp</artifactId>
    <version>1.34.1</version>
    <exclusions>
        <exclusion>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-exporter-sender-okhttp</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-sender-grpc-managed-channel</artifactId>
    <version>1.34.1</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
    <version>1.34.1</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry.instrumentation</groupId>
    <artifactId>opentelemetry-kafka-clients-2.6</artifactId>
    <version>1.32.0-alpha</version>
</dependency>
```

```

<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk</artifactId>
    <version>1.34.1</version>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-sender-jdk</artifactId>
    <version>1.34.1-alpha</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>1.61.0</version>
</dependency>

```

2. Define the configuration of the tracer using the [tracing environment variables](#).
3. Create a tracer, which is initialized with the environment variables:

Creating a tracer for OpenTelemetry

```
OpenTelemetry ot = GlobalOpenTelemetry.get();
```

4. Register the tracer as a global tracer:

```
GlobalTracer.register(tracer);
```

5. Instrument your client:
 - [Instrumenting producers and consumers for tracing](#)
 - [Instrumenting Kafka Streams applications for tracing](#)

23.3.4. Instrumenting producers and consumers for tracing

Instrument application code to enable tracing in Kafka producers and consumers. Use a decorator pattern or interceptors to instrument your Java producer and consumer application code for tracing. You can then record traces when messages are produced or retrieved from a topic.

OpenTelemetry instrumentation project provides classes that support instrumentation of producers and consumers.

Decorator instrumentation

For decorator instrumentation, create a modified producer or consumer instance for tracing.

Interceptor instrumentation

For interceptor instrumentation, add the tracing capability to the consumer or producer configuration.

Prerequisites

- You have initialized tracing for the client.

You enable instrumentation in producer and consumer applications by adding the tracing JARs as dependencies to your project.

Procedure

Perform these steps in the application code of each producer and consumer application. Instrument your client application code using either a decorator pattern or interceptors.

- To use a decorator pattern, create a modified producer or consumer instance to send or receive messages.

You pass the original `KafkaProducer` or `KafkaConsumer` class.

Example decorator instrumentation for OpenTelemetry

```
// Producer instance
Producer<String, String> op = new KafkaProducer<>(
    configs,
    new StringSerializer(),
    new StringSerializer()
);
Producer<String, String> producer = tracing.wrap(op);
KafkaTracing tracing = KafkaTracing.create(GlobalOpenTelemetry.get());
producer.send(...);

//consumer instance
Consumer<String, String> oc = new KafkaConsumer<>(
    configs,
    new StringDeserializer(),
    new StringDeserializer()
);
Consumer<String, String> consumer = tracing.wrap(oc);
consumer.subscribe(Collections.singleton("mytopic"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
    tracer);
```

- To use interceptors, set the interceptor class in the producer or consumer configuration.

You use the `KafkaProducer` and `KafkaConsumer` classes in the usual way. The `TracingProducerInterceptor` and `TracingConsumerInterceptor` interceptor classes take care of the tracing capability.

Example producer configuration using interceptors

```
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingProducerInterceptor.class.getName());
```

```
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);
producer.send(...);
```

Example consumer configuration using interceptors

```
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
    TracingConsumerInterceptor.class.getName());

KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("messages"));
ConsumerRecords<Integer, String> records = consumer.poll(1000);
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext = TracingKafkaUtils.extractSpanContext(record.headers(),
    tracer);
```

23.3.5. Instrumenting Kafka Streams applications for tracing

Instrument application code to enable tracing in Kafka Streams API applications. Use a decorator pattern or interceptors to instrument your Kafka Streams API applications for tracing. You can then record traces when messages are produced or retrieved from a topic.

Decorator instrumentation

For decorator instrumentation, create a modified Kafka Streams instance for tracing. For OpenTelemetry, you need to create a custom [TracingKafkaClientSupplier](#) class to provide tracing instrumentation for Kafka Streams.

Interceptor instrumentation

For interceptor instrumentation, add the tracing capability to the Kafka Streams producer and consumer configuration.

Prerequisites

- You have [initialized tracing for the client](#).

You enable instrumentation in Kafka Streams applications by adding the tracing JARs as dependencies to your project.

- To instrument Kafka Streams with OpenTelemetry, you'll need to write a custom [TracingKafkaClientSupplier](#).
- The custom [TracingKafkaClientSupplier](#) can extend Kafka's [DefaultKafkaClientSupplier](#), overriding the producer and consumer creation methods to wrap the instances with the telemetry-related code.

Example custom TracingKafkaClientSupplier

```
private class TracingKafkaClientSupplier extends DefaultKafkaClientSupplier {
    @Override
    public Producer<byte[], byte[]> getProducer(Map<String, Object> config) {
```

```

        KafkaTelemetry telemetry =
    KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getProducer(config));
    }

    @Override
    public Consumer<byte[], byte[]> getConsumer(Map<String, Object> config) {
        KafkaTelemetry telemetry =
    KafkaTelemetry.create(GlobalOpenTelemetry.get());
        return telemetry.wrap(super.getConsumer(config));
    }

    @Override
    public Consumer<byte[], byte[]> getRestoreConsumer(Map<String, Object> config)
{
    return this.getConsumer(config);
}

    @Override
    public Consumer<byte[], byte[]> getGlobalConsumer(Map<String, Object> config) {
        return this.getConsumer(config);
}
}

```

Procedure

Perform these steps for each Kafka Streams API application.

- To use a decorator pattern, create an instance of the [TracingKafkaClientSupplier](#) supplier interface, then provide the supplier interface to [KafkaStreams](#).

Example decorator instrumentation

```

KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
supplier);
streams.start();

```

- To use interceptors, set the interceptor class in the Kafka Streams producer and consumer configuration.

The [TracingProducerInterceptor](#) and [TracingConsumerInterceptor](#) interceptor classes take care of the tracing capability.

Example producer and consumer configuration using interceptors

```

props.put(StreamsConfig.PRODUCER_PREFIX +
ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
TracingProducerInterceptor.class.getName());
props.put(StreamsConfig.CONSUMER_PREFIX +
ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,

```

```
TracingConsumerInterceptor.class.getName());
```

23.3.6. Introducing a different OpenTelemetry tracing system

Instead of the default OTLP system, you can specify other tracing systems that are supported by OpenTelemetry. You do this by adding the required artifacts to the Kafka image provided with Strimzi. Any required implementation specific environment variables must also be set. You then enable the new tracing implementation using the `OTEL_TRACES_EXPORTER` environment variable.

This procedure shows how to implement Zipkin tracing.

Procedure

1. Add the tracing artifacts to the `/opt/kafka/libs/` directory of the Kafka image.

You can use the Kafka container image on the [Container Registry](#) as a base image for creating a new custom image.

OpenTelemetry artifact for Zipkin

```
io.opentelemetry:opentelemetry-exporter-zipkin
```

2. Set the tracing exporter and endpoint for the new tracing implementation.

Example Zikpin tracer configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: OTEL_SERVICE_NAME
          value: my-zipkin-service
        - name: OTEL_EXPORTER_ZIPKIN_ENDPOINT
          value: http://zipkin-exporter-host-name:9411/api/v2/spans ①
        - name: OTEL_TRACES_EXPORTER
          value: zipkin ②
    tracing:
      type: opentelemetry
  #...
```

① Specifies the Zipkin endpoint to connect to.

② The Zipkin exporter.

23.3.7. Specifying custom span names for OpenTelemetry

A tracing *span* is a logical unit of work in Jaeger, with an operation name, start time, and duration. Spans have built-in names, but you can specify custom span names in your Kafka client instrumentation where used.

Specifying custom span names is optional and only applies when using a decorator pattern [in producer and consumer client instrumentation](#) or [Kafka Streams instrumentation](#).

Custom span names cannot be specified directly with OpenTelemetry. Instead, you retrieve span names by adding code to your client application to extract additional tags and attributes.

Example code to extract attributes

```
//Defines attribute extraction for a producer
private static class ProducerAttribExtractor implements AttributesExtractor <
ProducerRecord < ?, ? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ProducerRecord < ?, ? >
producerRecord) {
        set(attributes, AttributeKey.stringKey("prod_start"), "prod1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ProducerRecord < ?, ? >
producerRecord, @Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("prod_end"), "prod2");
    }
}
//Defines attribute extraction for a consumer
private static class ConsumerAttribExtractor implements AttributesExtractor <
ConsumerRecord < ?, ? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ConsumerRecord < ?, ? >
producerRecord) {
        set(attributes, AttributeKey.stringKey("con_start"), "con1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ConsumerRecord < ?, ? >
producerRecord, @Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("con_end"), "con2");
    }
}
//Extracts the attributes
public static void main(String[] args) throws Exception {
    Map < String, Object > configs = new HashMap < >
(Collections.singletonMap(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"));
    System.setProperty("otel.traces.exporter", "jaeger");
    System.setProperty("otel.service.name", "myapp1");
    KafkaTracing tracing = KafkaTracing.newBuilder(GlobalOpenTelemetry.get())
        .addProducerAttributesExtractors(new ProducerAttribExtractor())
        .addConsumerAttributesExtractors(new ConsumerAttribExtractor())
```

```
.build();
```

Chapter 24. Evicting pods with the Strimzi Drain Cleaner

Kafka and ZooKeeper pods might be evicted during Kubernetes upgrades, maintenance, or pod rescheduling. If your Kafka and ZooKeeper pods were deployed by Strimzi, you can use the Strimzi Drain Cleaner tool to handle the pod evictions. The Strimzi Drain Cleaner handles the eviction instead of Kubernetes.

By deploying the Strimzi Drain Cleaner, you can use the Cluster Operator to move Kafka pods instead of Kubernetes. The Cluster Operator ensures that the number of in sync replicas for topics are at or above the configured `min.insync.replicas` and Kafka can remain operational during the eviction process. The Cluster Operator waits for topics to synchronize, as the Kubernetes worker nodes drain consecutively.

An admission webhook notifies the Strimzi Drain Cleaner of pod eviction requests to the Kubernetes API. The Strimzi Drain Cleaner then adds a rolling update annotation to the pods to be drained. This informs the Cluster Operator to perform a rolling update of an evicted pod.

NOTE

If you are not using the Strimzi Drain Cleaner, you can [add pod annotations to perform rolling updates manually](#).

Webhook configuration

The Strimzi Drain Cleaner deployment files include a `ValidatingWebhookConfiguration` resource file. The resource provides the configuration for registering the webhook with the Kubernetes API.

The configuration defines the `rules` for the Kubernetes API to follow in the event of a pod eviction request. The rules specify that only `CREATE` operations related to `pods/eviction` sub-resources are intercepted. If these rules are met, the API forwards the notification.

The `clientConfig` points to the Strimzi Drain Cleaner service and `/drainer` endpoint that exposes the webhook. The webhook uses a secure TLS connection, which requires authentication. The `caBundle` property specifies the certificate chain to validate HTTPS communication. Certificates are encoded in Base64.

Webhook configuration for pod eviction notifications

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
# ...
webhooks:
- name: strimzi-drain-cleaner.strimzi.io
  rules:
    - apiGroups: []
      apiVersions: ["v1"]
      operations: ["CREATE"]
      resources: ["pods/eviction"]
      scope: "Namespaced"
  clientConfig:
```

```
service:
  namespace: "strimzi-drain-cleaner"
  name: "strimzi-drain-cleaner"
  path: /drainer
  port: 443
  caBundle: Cg==
# ...
```

24.1. Downloading the Strimzi Drain Cleaner deployment files

To deploy and use the Strimzi Drain Cleaner, you need to download the deployment files.

The Strimzi Drain Cleaner deployment files are available from the [GitHub releases page](#).

24.2. Deploying the Strimzi Drain Cleaner using installation files

Deploy the Strimzi Drain Cleaner to the Kubernetes cluster where the Cluster Operator and Kafka cluster are running.

Strimzi Drain Cleaner can run in two different modes. By default, the Drain Cleaner denies (blocks) the Kubernetes eviction request to prevent Kubernetes from evicting the pods and instead uses the Cluster Operator to move the pod. This mode has better compatibility with various cluster autoscaling tools and does not require any specific [PodDisruptionBudget](#) configuration. Alternatively, you can enable the legacy mode where it allows the eviction request while also instructing the Cluster Operator to move the pod. For the legacy mode to work, you have to configure the [PodDisruptionBudget](#) to not allow any pod evictions by setting the `maxUnavailable` option to `0`.

Prerequisites

- You have [downloaded the Strimzi Drain Cleaner deployment files](#).
- You have a highly available Kafka cluster deployment running with Kubernetes worker nodes that you would like to update.
- Topics are replicated for high availability.

Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
```

```

spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...

```

Excluding Kafka or ZooKeeper

If you don't want to include Kafka or ZooKeeper pods in Drain Cleaner operations, or if you prefer to use the Drain Cleaner in legacy mode, change the default environment variables in the Drain Cleaner [Deployment](#) configuration file:

- Set `STRIMZI_DENY_EVICTION` to `false` to use the legacy mode relying on the [PodDisruptionBudget](#) configuration
- Set `STRIMZI_DRAIN_KAFKA` to `false` to exclude Kafka pods
- Set `STRIMZI_DRAIN_ZOOKEEPER` to `false` to exclude ZooKeeper pods

Example configuration to exclude ZooKeeper pods

```

apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-drain-cleaner
      containers:
        - name: strimzi-drain-cleaner
          # ...
          env:
            - name: STRIMZI_DENY_EVICTION
              value: "true"
            - name: STRIMZI_DRAIN_KAFKA
              value: "true"
            - name: STRIMZI_DRAIN_ZOOKEEPER
              value: "false"
          # ...

```

Procedure

1. If you are using the legacy mode activated by setting the `STRIMZI_DENY_EVICTION` environment variable to `false`, you must also configure the [PodDisruptionBudget](#) resource. Set `maxUnavailable` to `0` (zero) in the Kafka and ZooKeeper sections of the [Kafka](#) resource using `template` settings.

Specifying a pod disruption budget

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka

```

```

metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    template:
      podDisruptionBudget:
        maxUnavailable: 0
    # ...
  zookeeper:
    template:
      podDisruptionBudget:
        maxUnavailable: 0
  # ...

```

This setting prevents the automatic eviction of pods in case of planned disruptions, leaving the Strimzi Drain Cleaner and Cluster Operator to roll the pods on different worker nodes.

Add the same configuration for ZooKeeper if you want to use Strimzi Drain Cleaner to drain ZooKeeper nodes.

2. Update the **Kafka** resource:

```
kubectl apply -f <kafka_configuration_file>
```

3. Deploy the Strimzi Drain Cleaner.

- If you are using **cert-manager** with Kubernetes, apply the resources in the [/install/drain-cleaner/certmanager](#) directory.

```
kubectl apply -f ./install/drain-cleaner/certmanager
```

The TLS certificates for the webhook are generated automatically and injected into the webhook configuration.

- If you are not using **cert-manager** with Kubernetes, do the following:
 - [Add TLS certificates to use in the deployment.](#)

Any certificates you add must be renewed before they expire.

- Apply the resources in the [/install/drain-cleaner/kubernetes](#) directory.

```
kubectl apply -f ./install/drain-cleaner/kubernetes
```

- To run the Drain Cleaner on OpenShift, apply the resources in the [/install/drain-cleaner/openshift](#) directory.

```
kubectl apply -f ./install/drain-cleaner/openshift
```

24.3. Deploying the Strimzi Drain Cleaner using Helm

Helm charts are used to package, configure, and deploy Kubernetes resources. Strimzi provides a Helm chart to deploy the Strimzi Drain Cleaner.

The Drain Cleaner is deployed on the Kubernetes cluster with the default chart configuration, which assumes that `cert-manager` issues the TLS certificates required by the Drain Cleaner.

You can install the Drain Cleaner with `cert-manager` support or provide your own TLS certificates.

Prerequisites

- The Helm client must be installed on a local machine.

Default configuration values

Default configuration values are passed into the chart using parameters defined in a `values.yaml` file. If you don't want to use the default configuration, you can override the defaults when you install the chart using the `--set` argument. You specify values in the format `--set key=value[,key=value]`. The `values.yaml` file supplied with the Helm deployment files describes the available configuration parameters, including those shown in the following table.

You can override the default image settings. You can also set `secret.create` as `true` and add your own TLS certificates instead of using `cert-manager` to generate the certificates. For information on using OpenSSL to generate certificates, see [Adding or renewing the TLS certificates used by the Strimzi Drain Cleaner](#).

Any certificates you add must be renewed before they expire. You can use the configuration to control how certificates are watched for updates using environment variables. For more information on how the environment variables work, see [Watching the TLS certificates used by the Strimzi Drain Cleaner](#).

Table 39. Chart configuration options

Parameter	Description	Default
<code>replicaCount</code>	Number of replicas of the Drain Cleaner webhook	<code>1</code>
<code>image.registry</code>	Drain Cleaner image registry	<code>quay.io</code>
<code>image.repository</code>	Drain Cleaner image repository	<code>strimzi</code>
<code>image.name</code>	Drain Cleaner image name	<code>drain-cleaner</code>
<code>image.tag</code>	Drain Cleaner image tag	<code>latest</code>
<code>image.imagePullPolicy</code>	Image pull policy for all pods deployed by the Drain Cleaner	<code>nil</code>
<code>secret.create</code>	Set to <code>true</code> and add certificates when not using <code>cert-manager</code>	<code>false</code>

Parameter	Description	Default
<code>namespace.name</code>	Default namespace for the Drain Cleaner deployment.	<code>strimzi-drain-cleaner</code>
<code>resources</code>	Configures resources for the Drain Cleaner pod	<code>[]</code>
<code>nodeSelector</code>	Add a node selector to the Drain Cleaner pod	<code>{}</code>
<code>tolerations</code>	Add tolerations to the Drain Cleaner pod	<code>[]</code>
<code>topologySpreadConstraints</code>	Add topology spread constraints to the Drain Cleaner pod	<code>{}</code>
<code>affinity</code>	Add affinities to the Drain Cleaner pod	<code>{}</code>

Procedure

1. Deploy the Drain Cleaner:

```
helm install strimzi-drain-cleaner oci://quay.io/strimzi-helm/strimzi-drain-cleaner
```

Alternatively, you can use parameter values to install a specific version of the Drain Cleaner or specify any changes to the default configuration.

Example configuration that installs a specific version of the Drain Cleaner and changes the number of replicas

```
helm install strimzi-drain-cleaner --set replicaCount=2 --version 1.1.0
oci://quay.io/strimzi-helm/strimzi-drain-cleaner
```

2. Verify that the Drain Cleaner has been deployed successfully:

```
helm ls
```

24.4. Using the Strimzi Drain Cleaner

Use the Strimzi Drain Cleaner in combination with the Cluster Operator to move Kafka broker or ZooKeeper pods from nodes that are being drained. When you run the Strimzi Drain Cleaner, it annotates pods with a rolling update pod annotation. The Cluster Operator performs rolling updates based on the annotation.

Prerequisites

- You have [deployed the Strimzi Drain Cleaner](#).

Considerations when using anti-affinity configuration

When using [anti-affinity](#) with your Kafka or ZooKeeper pods, consider adding a spare worker node to your cluster. Including spare nodes ensures that your cluster has the capacity to reschedule pods during node draining or temporary unavailability of other nodes. When a worker node is drained, and anti-affinity rules restrict pod rescheduling on alternative nodes, spare nodes help prevent restarted pods from becoming unschedulable. This mitigates the risk of the draining operation failing.

Procedure

1. Drain a specified Kubernetes node hosting the Kafka broker or ZooKeeper pods.

```
kubectl get nodes  
kubectl drain <name-of-node> --delete-emptydir-data --ignore-daemonsets  
--timeout=600s --force
```

2. Check the eviction events in the Strimzi Drain Cleaner log to verify that the pods have been annotated for restart.

Strimzi Drain Cleaner log show annotations of pods

```
INFO ... Received eviction webhook for Pod my-cluster-zookeeper-2 in namespace my-project  
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project will be annotated for restart  
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project found and annotated for restart  
  
INFO ... Received eviction webhook for Pod my-cluster-kafka-0 in namespace my-project  
INFO ... Pod my-cluster-kafka-0 in namespace my-project will be annotated for restart  
INFO ... Pod my-cluster-kafka-0 in namespace my-project found and annotated for restart
```

3. Check the reconciliation events in the Cluster Operator log to verify the rolling updates.

Cluster Operator log shows rolling updates

```
INFO PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster):  
Rolling Pod my-cluster-zookeeper-2  
INFO PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster):  
Rolling Pod my-cluster-kafka-0  
INFO AbstractOperator:500 - Reconciliation #13(timer) Kafka(my-project/my-cluster): reconciled
```

24.5. Adding or renewing the TLS certificates used by the Strimzi Drain Cleaner

The Drain Cleaner uses a webhook to receive eviction notifications from the Kubernetes API. The webhook uses a secure TLS connection and authenticates using TLS certificates. If you are not deploying the Drain Cleaner using the [cert-manager](#) or on Openshift, you must create and renew the TLS certificates. You must then add them to the files used to deploy the Drain Cleaner. The certificates must also be renewed before they expire. To renew the certificates, you repeat the steps used to generate and add the certificates to the initial deployment of the Drain Cleaner.

Generate and add certificates to the standard installation files or your Helm configuration when deploying the Drain Cleaner on Kubernetes without [cert-manager](#).

NOTE

If you are using [cert-manager](#) to deploy the Drain Cleaner, you don't need to add or renew TLS certificates. The same applies when deploying the Drain Cleaner on OpenShift, as OpenShift injects the certificates. In both cases, TLS certificates are added and renewed automatically.

Prerequisites

- The [OpenSSL](#) TLS management tool for generating certificates.

Use `openssl help` for command-line descriptions of the options used.

Generating and renewing TLS certificates

1. From the command line, create a directory called [tls-certificate](#):

```
mkdir tls-certificate  
cd tls-certificate
```

Now use OpenSSL to create the certificates in the [tls-certificate](#) directory.

2. Generate a CA (Certificate Authority) public certificate and private key:

```
openssl req -nodes -new -x509 -keyout ca.key -out ca.crt -subj "/CN=Strimzi Drain  
Cleaner CA"
```

A [ca.crt](#) and [ca.key](#) file are created.

3. Generate a private key for the Drain Cleaner:

```
openssl genrsa -out tls.key 2048
```

A [tls.key](#) file is created.

4. Generate a CSR (Certificate Signing Request) and sign it by adding the CA public certificate

(`ca.crt`) you generated:

```
openssl req -new -key tls.key -subj "/CN=strimzi-drain-cleaner.strimzi-drain-cleaner.svc" \
| openssl x509 -req -CA ca.crt -CAkey ca.key -CAcreateserial -extfile <(printf \
"subjectAltName=DNS:strimzi-drain-cleaner.strimzi-drain-cleaner.svc") -out tls.crt
```

A `tls.crt` file is created.

NOTE

If you change the name of the Strimzi Drain Cleaner service or install it into a different namespace, you must change the SAN (Subject Alternative Name) of the certificate, following the format `<service_name>. <namespace>.svc`.

5. Encode the CA public certificate into base64.

```
base64 tls-certificate/ca.crt
```

With the certificates generated, add them to the installation files or to your Helm configuration depending on your deployment method.

Adding the TLS certificates to the Drain Cleaner installation files

1. Copy the base64-encoded CA public certificate as the value for the `caBundle` property of the [070-ValidatingWebhookConfiguration.yaml](#) installation file:

```
# ...
clientConfig:
  service:
    namespace: "strimzi-drain-cleaner"
    name: "strimzi-drain-cleaner"
    path: /drainer
    port: 443
    caBundle: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0...
# ...
```

2. Create a namespace called `strimzi-drain-cleaner` in your Kubernetes cluster:

```
kubectl create ns strimzi-drain-cleaner
```

3. Create a secret named `strimzi-drain-cleaner` with the `tls.crt` and `tls.key` files you generated:

```
kubectl create secret tls strimzi-drain-cleaner \
-n strimzi-drain-cleaner \
--cert=tls-certificate/tls.crt \
--key=tls-certificate/tls.key
```

The secret is used in the Drain Cleaner deployment.

Example secret for the Drain Cleaner deployment

```
apiVersion: v1
kind: Secret
metadata:
  # ...
  name: strimzi-drain-cleaner
  namespace: strimzi-drain-cleaner
# ...
data:
  tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS...
  tls.key: LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLR...
```

You can now use the certificates and updated installation files [to deploy the Drain Cleaner using installation files](#).

Adding the TLS certificates to a Helm deployment

1. Edit the `values.yaml` configuration file used in the Helm deployment.
2. Set the `certManager.create` parameter to `false`.
3. Set the `secret.create` parameter to `true`.
4. Copy the certificates as `secret` parameters.

Example secret configuration for the Drain Cleaner deployment

```
# ...
certManager:
  create: false

secret:
  create: true
  tls_crt: "Cg==" ①
  tls_key: "Cg==" ②
  ca_bundle: "Cg==" ③
```

① The public key (`tls.crt`) signed by the CA public certificate.

② The private key (`tls.key`).

③ The base-64 encoded CA public certificate (`ca.crt`).

You can now use the certificates and updated configuration file [to deploy the Drain Cleaner using Helm](#).

24.6. Watching the TLS certificates used by the Strimzi Drain Cleaner

By default, the Drain Cleaner deployment watches the secret containing the TLS certificates its uses for authentication. The Drain Cleaner watches for changes, such as certificate renewals. If it detects a change, it restarts to reload the TLS certificates. The Drain Cleaner installation files enable this behavior by default. But you can disable the watching of certificates by setting the `STRIMZI_CERTIFICATE_WATCH_ENABLED` environment variable to `false` in the `Deployment` configuration (`060-Deployment.yaml`) of the Drain Cleaner installation files.

With `STRIMZI_CERTIFICATE_WATCH_ENABLED` enabled, you can also use the following environment variables for watching TLS certificates.

Table 40. Drain Cleaner environment variables for watching TLS certificates

Environment Variable	Description	Default
<code>STRIMZI_CERTIFICATE_WATCH_ENABLED</code>	Enables or disables the certificate watch	<code>false</code>
<code>STRIMZI_CERTIFICATE_WATCH_NAMESPACE</code>	The namespace where the Drain Cleaner is deployed and where the certificate secret exists	<code>strimzi-drain-cleaner</code>
<code>STRIMZI_CERTIFICATE_WATCH_POD_NAME</code>	The Drain Cleaner pod name	-
<code>STRIMZI_CERTIFICATE_WATCH_SECRET_NAME</code>	The name of the secret containing TLS certificates	<code>strimzi-drain-cleaner</code>
<code>STRIMZI_CERTIFICATE_WATCH_SECRET_KEYS</code>	The list of fields inside the secret that contain the TLS certificates	<code>tls.crt, tls.key</code>

Example environment variable configuration to control watch operations

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-drain-cleaner
  labels:
    app: strimzi-drain-cleaner
    namespace: strimzi-drain-cleaner
spec:
  # ...
  spec:
    serviceAccountName: strimzi-drain-cleaner
    containers:
      - name: strimzi-drain-cleaner
        # ...
        env:
          - name: STRIMZI_DRAIN_KAFKA
            value: "true"
          - name: STRIMZI_DRAIN_ZOOKEEPER
```

```
    value: "true"
- name: STRIMZI_CERTIFICATE_WATCH_ENABLED
  value: "true"
- name: STRIMZI_CERTIFICATE_WATCH_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
- name: STRIMZI_CERTIFICATE_WATCH_POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
# ...
```

TIP

Use the [Downward API](#) mechanism to configure `STRIMZI_CERTIFICATE_WATCH_NAMESPACE` and `STRIMZI_CERTIFICATE_WATCH_POD_NAME`.

Chapter 25. Managing rolling updates

Use annotations to manually trigger a rolling update of Kafka and other operands through the Cluster Operator. Initiate rolling updates of Kafka, Kafka Connect, MirrorMaker 2, and ZooKeeper clusters.

Manually performing a rolling update on a specific pod or set of pods is usually only required in exceptional circumstances. However, rather than deleting the pods directly, if you perform the rolling update through the Cluster Operator you ensure the following:

- The manual deletion of the pod does not conflict with simultaneous Cluster Operator operations, such as deleting other pods in parallel.
- The Cluster Operator logic handles the Kafka configuration specifications, such as the number of in-sync replicas.

25.1. Performing a rolling update using a pod management annotation

This procedure describes how to trigger a rolling update of Kafka, Kafka Connect, MirrorMaker 2, or ZooKeeper clusters. To trigger the update, you add an annotation to the `StrimziPodSet` that manages the pods running on the cluster.

Prerequisites

To perform a manual rolling update, you need a running Cluster Operator. The cluster for the component you are updating, whether it's Kafka, Kafka Connect, MirrorMaker 2, or ZooKeeper, must also be running.

Procedure

1. Find the name of the resource that controls the pods you want to manually update.

For example, if your Kafka cluster is named *my-cluster*, the corresponding names are *my-cluster-kafka* and *my-cluster-zookeeper*. For a Kafka Connect cluster named *my-connect-cluster*, the corresponding name is *my-connect-cluster-connect*. And for a MirrorMaker 2 cluster named *my-mm2-cluster*, the corresponding name is *my-mm2-cluster-mirrormaker2*.

2. Use `kubectl annotate` to annotate the appropriate resource in Kubernetes.

Annotating a `StrimziPodSet`

```
kubectl annotate strimzipodset <cluster_name>-kafka strimzi.io/manual-rolling-update="true"
```

```
kubectl annotate strimzipodset <cluster_name>-zookeeper strimzi.io/manual-rolling-update="true"
```

```
kubectl annotate strimzipodset <cluster_name>-connect strimzi.io/manual-rolling-update="true"
```

```
kubectl annotate strimzipodset <cluster_name>-mirrormaker2 strimzi.io/manual-rolling-update="true"
```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of all pods within the annotated resource is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of all the pods is complete, the annotation is automatically removed from the resource.

25.2. Performing a rolling update using a pod annotation

This procedure describes how to manually trigger a rolling update of existing Kafka, Kafka Connect, MirrorMaker 2, or ZooKeeper clusters using a Kubernetes **Pod** annotation. When multiple pods are annotated, consecutive rolling updates are performed within the same reconciliation run.

Prerequisites

To perform a manual rolling update, you need a running Cluster Operator. The cluster for the component you are updating, whether it's Kafka, Kafka Connect, MirrorMaker 2, or ZooKeeper, must also be running.

You can perform a rolling update on a Kafka cluster regardless of the topic replication factor used. But for Kafka to stay operational during the update, you'll need the following:

- A highly available Kafka cluster deployment running with nodes that you wish to update.
- Topics replicated for high availability.

Topic configuration specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

Procedure

1. Find the name of the **Pod** you want to manually update.

Pod naming conventions are as follows:

- <cluster_name>-kafka-<index_number> for a Kafka cluster
- <cluster_name>-zookeeper-<index_number> for a ZooKeeper cluster
- <cluster_name>-connect-<index_number> for a Kafka Connect cluster
- <cluster_name>-mirrormaker2-<index_number> for a MirrorMaker 2 cluster

The <index_number> assigned to a pod starts at zero and ends at the total number of replicas minus one.

2. Use `kubectl annotate` to annotate the **Pod** resource in Kubernetes:

```
kubectl annotate pod <cluster_name>-kafka-<index_number> strimzi.io/manual-rolling-update="true"

kubectl annotate pod <cluster_name>-zookeeper-<index_number> strimzi.io/manual-rolling-update="true"

kubectl annotate pod <cluster_name>-connect-<index_number> strimzi.io/manual-rolling-update="true"

kubectl annotate pod <cluster_name>-mirrormaker2-<index_number> strimzi.io/manual-rolling-update="true"
```

3. Wait for the next reconciliation to occur (every two minutes by default). A rolling update of the annotated **Pod** is triggered, as long as the annotation was detected by the reconciliation process. When the rolling update of a pod is complete, the annotation is automatically removed from the **Pod**.

NOTE

If the `ContinueReconciliationOnManualRollingUpdateFailure` feature gate is enabled, reconciliation continues even if the manual rolling update of the cluster fails. This allows the Cluster Operator to recover from certain rectifiable situations that can be addressed later in the reconciliation. For example, it can recreate a missing Persistent Volume Claim (PVC) or Persistent Volume (PV) that caused the update to fail.

Chapter 26. Finding information on Kafka restarts

After the Cluster Operator restarts a Kafka pod in a Kubernetes cluster, it emits a Kubernetes event into the pod's namespace explaining why the pod restarted. For help in understanding cluster behavior, you can check restart events from the command line.

TIP You can export and monitor restart events using metrics collection tools like Prometheus. Use the metrics tool with an *event exporter* that can export the output in a suitable format.

26.1. Reasons for a restart event

The Cluster Operator initiates a restart event for a specific reason. You can check the reason by fetching information on the restart event.

Table 41. Restart reasons

Event	Description
CaCertHasOldGeneration	The pod is still using a server certificate signed with an old CA, so needs to be restarted as part of the certificate update.
CaCertRemoved	Expired CA certificates have been removed, and the pod is restarted to run with the current certificates.
CaCertRenewed	CA certificates have been renewed, and the pod is restarted to run with the updated certificates.
ClientCaCertKeyReplaced	The key used to sign clients CA certificates has been replaced, and the pod is being restarted as part of the CA renewal process.
ClusterCaCertKeyReplaced	The key used to sign the cluster's CA certificates has been replaced, and the pod is being restarted as part of the CA renewal process.
ConfigChangeRequiresRestart	Some Kafka configuration properties are changed dynamically, but others require that the broker be restarted.
FileSystemResizeNeeded	The file system size has been increased, and a restart is needed to apply it.
KafkaCertificatesChanged	One or more TLS certificates used by the Kafka broker have been updated, and a restart is needed to use them.
ManualRollingUpdate	A user annotated the pod, or the StrimziPodSet set it belongs to, to trigger a restart.
PodForceRestartOnError	An error occurred that requires a pod restart to rectify.

Event	Description
PodHasOldRevision	A disk was added or removed from the Kafka volumes, and a restart is needed to apply the change. When using StrimziPodSet resources, the same reason is given if the pod needs to be recreated.
PodHasOldRevision	The StrimziPodSet that the pod is a member of has been updated, so the pod needs to be recreated. When using StrimziPodSet resources, the same reason is given if a disk was added or removed from the Kafka volumes.
PodStuck	The pod is still pending, and is not scheduled or cannot be scheduled, so the operator has restarted the pod in a final attempt to get it running.
PodUnresponsive	Strimzi was unable to connect to the pod, which can indicate a broker not starting correctly, so the operator restarted it in an attempt to resolve the issue.

26.2. Restart event filters

When checking restart events from the command line, you can specify a [field-selector](#) to filter on Kubernetes event fields.

The following fields are available when filtering events with [field-selector](#).

`regardingObject.kind`

The object that was restarted, and for restart events, the kind is always `Pod`.

`regarding.namespace`

The namespace that the pod belongs to.

`regardingObject.name`

The pod's name, for example, `strimzi-cluster-kafka-0`.

`regardingObject.uid`

The unique ID of the pod.

`reason`

The reason the pod was restarted, for example, `JbodVolumesChanged`.

`reportingController`

The reporting component is always `strimzi.io/cluster-operator` for Strimzi restart events.

`source`

`source` is an older version of `reportingController`. The reporting component is always `strimzi.io/cluster-operator` for Strimzi restart events.

type

The event type, which is either `Warning` or `Normal`. For Strimzi restart events, the type is `Normal`.

NOTE In older versions of Kubernetes, the fields using the `regarding` prefix might use an `involvedObject` prefix instead. `reportingController` was previously called `reportingComponent`.

26.3. Checking Kafka restarts

Use a `kubectl` command to list restart events initiated by the Cluster Operator. Filter restart events emitted by the Cluster Operator by setting the Cluster Operator as the reporting component using the `reportingController` or `source` event fields.

Prerequisites

- The Cluster Operator is running in the Kubernetes cluster.

Procedure

1. Get all restart events emitted by the Cluster Operator:

```
kubectl -n kafka get events --field-selector  
reportingController=strimzi.io/cluster-operator
```

Example showing events returned

LAST SEEN	TYPE	REASON	OBJECT	MESSAGE
2m	Normal	CaCertRenewed certificate renewed	pod/strimzi-cluster-kafka-0	CA
58m	Normal	PodForceRestartOnError needs to be forcibly restarted due to an error	pod/strimzi-cluster-kafka-1	Pod
5m47s	Normal	ManualRollingUpdate manually annotated to be rolled	pod/strimzi-cluster-kafka-2	Pod was

You can also specify a `reason` or other `field-selector` options to constrain the events returned.

Here, a specific reason is added:

```
kubectl -n kafka get events --field-selector  
reportingController=strimzi.io/cluster-operator,reason=PodForceRestartOnError
```

2. Use an output format, such as YAML, to return more detailed information about one or more events.

```
kubectl -n kafka get events --field-selector  
reportingController=strimzi.io/cluster-operator,reason=PodForceRestartOnError -o  
yaml
```

Example showing detailed events output

```
apiVersion: v1
items:
- action: StrimziInitiatedPodRestart
  apiVersion: v1
  eventTime: "2022-05-13T00:22:34.168086Z"
  firstTimestamp: null
  involvedObject:
    kind: Pod
    name: strimzi-cluster-kafka-1
    namespace: kafka
  kind: Event
  lastTimestamp: null
  message: Pod needs to be forcibly restarted due to an error
  metadata:
    creationTimestamp: "2022-05-13T00:22:34Z"
    generateName: strimzi-event
    name: strimzi-eventwppk6
    namespace: kafka
    resourceVersion: "432961"
    uid: 29fcdb9e-f2cf-4c95-a165-a5efcd48edfc
  reason: PodForceRestartOnError
  reportingController: strimzi.io/cluster-operator
  reportingInstance: strimzi-cluster-operator-6458cfb4c6-6bpdp
  source: {}
  type: Normal
  kind: List
  metadata:
    resourceVersion: ""
    selfLink: "
```

The following fields are deprecated, so they are not populated for these events:

- `firstTimestamp`
- `lastTimestamp`
- `source`

Chapter 27. Upgrading Strimzi

Upgrade your Strimzi installation to version 0.42.0 and benefit from new features, performance improvements, and enhanced security options. During the upgrade, Kafka is also be updated to the latest supported version, introducing additional features and bug fixes to your Strimzi deployment.

Use the same method to upgrade the Cluster Operator as the initial method of deployment. For example, if you used the Strimzi installation files, modify those files to perform the upgrade. After you have upgraded your Cluster Operator to 0.42.0, the next step is to upgrade all Kafka nodes to the latest supported version of Kafka. Kafka upgrades are performed by the Cluster Operator through rolling updates of the Kafka nodes.

If you encounter any issues with the new version, Strimzi can be [downgraded](#) to the previous version.

Released Strimzi versions can be found at [GitHub releases page](#).

Upgrade without downtime

For topics configured with high availability (replication factor of at least 3 and evenly distributed partitions), the upgrade process should not cause any downtime for consumers and producers.

The upgrade triggers rolling updates, where brokers are restarted one by one at different stages of the process. During this time, overall cluster availability is temporarily reduced, which may increase the risk of message loss in the event of a broker failure.

27.1. Required upgrade sequence

To upgrade brokers and clients without downtime, you *must* complete the Strimzi upgrade procedures in the following order:

1. Make sure your Kubernetes cluster version is supported.

Strimzi 0.42.0 requires Kubernetes 1.23 and later.

You can [upgrade Kubernetes with minimal downtime](#).

2. [Upgrade the Cluster Operator](#).
3. Upgrade Kafka depending on the cluster configuration:
 - a. If using Kafka in KRaft mode, update the Kafka version and `spec.kafka.metadataVersion` to [upgrade all Kafka brokers and client applications](#).
 - b. If using ZooKeeper-based Kafka, update the Kafka version and `inter.broker.protocol.version` to [upgrade all Kafka brokers and client applications](#).

NOTE

From Strimzi 0.39, upgrades and downgrades between KRaft-based clusters are supported.

27.2. Strimzi upgrade paths

Two upgrade paths are available for Strimzi.

Incremental upgrade

An incremental upgrade involves upgrading Strimzi from the previous minor version to version 0.42.0.

Multi-version upgrade

A multi-version upgrade involves upgrading an older version of Strimzi to version 0.42.0 within a single upgrade, skipping one or more intermediate versions. For example, upgrading directly from Strimzi 0.30.0 to Strimzi 0.42.0 is possible.

27.2.1. Support for Kafka versions when upgrading

When upgrading Strimzi, it is important to ensure compatibility with the Kafka version being used.

Multi-version upgrades are possible even if the supported Kafka versions differ between the old and new versions. However, if you attempt to upgrade to a new Strimzi version that does not support the current Kafka version, [an error indicating that the Kafka version is not supported is generated](#). In this case, you must upgrade the Kafka version as part of the Strimzi upgrade by changing the `spec.kafka.version` in the `Kafka` custom resource to the supported version for the new Strimzi version.

You can review supported Kafka versions in the [Supported versions](#) table.

NOTE

- The **Operators** column lists all released Strimzi versions (the Strimzi version is often called the "Operator version").
- The **Kafka versions** column lists the supported Kafka versions for each Strimzi version.

27.2.2. Upgrading from a Strimzi version earlier than 0.22

If you are upgrading to the latest version of Strimzi from a version prior to version 0.22, do the following:

1. Upgrade Strimzi to version 0.22 following the [standard sequence](#).
2. Convert Strimzi custom resources to `v1beta2` using the *API conversion tool* provided with Strimzi.
3. Do one of the following:
 - Upgrade Strimzi to a version between 0.23 and 0.26 (where the `ControlPlaneListener` feature gate is disabled by default).
 - Upgrade Strimzi to a version between 0.27 and 0.31 (where the `ControlPlaneListener` feature gate is enabled by default) with the `ControlPlaneListener` feature gate disabled.
4. Enable the `ControlPlaneListener` feature gate.

5. Upgrade to Strimzi 0.42.0 following the [standard sequence](#).

Strimzi custom resources started using the `v1beta2` API version in release 0.22. CRDs and custom resources must be converted **before** upgrading to Strimzi 0.23 or newer. For information on using the API conversion tool, see the [Strimzi 0.24.0 upgrade documentation](#).

NOTE

As an alternative to first upgrading to version 0.22, you can install the custom resources from version 0.22 and then convert the resources.

The `ControlPlaneListener` feature is now permanently enabled in Strimzi. You must upgrade to a version of Strimzi where it is disabled, then enable it using the `STRIMZI_FEATURE_GATES` environment variable in the Cluster Operator configuration.

Disabling the `ControlPlaneListener` feature gate

```
env:  
  - name: STRIMZI_FEATURE_GATES  
    value: -ControlPlaneListener
```

Enabling the `ControlPlaneListener` feature gate

```
env:  
  - name: STRIMZI_FEATURE_GATES  
    value: +ControlPlaneListener
```

27.2.3. Kafka version and image mappings

When upgrading Kafka, consider your settings for the `STRIMZI_KAFKA_IMAGES` environment variable and the `Kafka.spec.kafka.version` property.

- Each `Kafka` resource can be configured with a `Kafka.spec.kafka.version`, which defaults to the latest supported Kafka version (3.7.1) if not specified.
- The Cluster Operator's `STRIMZI_KAFKA_IMAGES` environment variable provides a mapping (`<kafka_version>=<image>`) between a Kafka version and the image to be used when a specific Kafka version is requested in a given `Kafka` resource. For example, `3.7.1=quay.io/stimzi/kafka:0.42.0-kafka-3.7.1`.
 - If `Kafka.spec.kafka.image` is not configured, the default image for the given version is used.
 - If `Kafka.spec.kafka.image` is configured, the default image is overridden.

WARNING

The Cluster Operator cannot validate that an image actually contains a Kafka broker of the expected version. Take care to ensure that the given image corresponds to the given Kafka version.

27.3. Strategies for upgrading clients

Upgrading Kafka clients ensures that they benefit from the features, fixes, and improvements that

are introduced in new versions of Kafka. Upgraded clients maintain compatibility with other upgraded Kafka components. The performance and stability of the clients might also be improved.

Consider the best approach for upgrading Kafka clients and brokers to ensure a smooth transition. The chosen upgrade strategy depends on whether you are upgrading brokers or clients first. Since Kafka 3.0, you can upgrade brokers and client independently and in any order. The decision to upgrade clients or brokers first depends on several factors, such as the number of applications that need to be upgraded and how much downtime is tolerable.

If you upgrade clients before brokers, some new features may not work as they are not yet supported by brokers. However, brokers can handle producers and consumers running with different versions and supporting different log message versions.

27.4. Upgrading Kubernetes with minimal downtime

If you are upgrading Kubernetes, refer to the Kubernetes upgrade documentation to check the upgrade path and the steps to upgrade your nodes correctly. Before upgrading Kubernetes, [check the supported versions for your version of Strimzi](#).

When performing your upgrade, ensure the availability of your Kafka clusters by following these steps:

1. Configure pod disruption budgets
2. Roll pods using one of these methods:
 - a. Use the Strimzi Drain Cleaner (recommended)
 - b. Apply an annotation to your pods to roll them manually

For Kafka to stay operational, topics must also be replicated for high availability. This requires topic configuration that specifies a replication factor of at least 3 and a minimum number of in-sync replicas to 1 less than the replication factor.

Kafka topic replicated for high availability

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

In a highly available environment, the Cluster Operator maintains a minimum number of in-sync

replicas for topics during the upgrade process so that there is no downtime.

27.4.1. Rolling pods using Drain Cleaner

When using the Strimzi Drain Cleaner to evict nodes during Kubernetes upgrade, it annotates pods with a manual rolling update annotation to inform the Cluster Operator to perform a rolling update of the pod that should be evicted and have it moved away from the Kubernetes node that is being upgraded.

For more information, see [Evicting pods with the Strimzi Drain Cleaner](#).

27.4.2. Rolling pods manually (alternative to Drain Cleaner)

As an alternative to using the Drain Cleaner to roll pods, you can trigger a manual rolling update of pods through the Cluster Operator. Using [Pod](#) resources, rolling updates restart the pods of resources with new pods. To replicate the operation of the Drain Cleaner by keeping topics available, you must also set the `maxUnavailable` value to zero for the pod disruption budget. Reducing the pod disruption budget to zero prevents Kubernetes from evicting pods automatically.

Specifying a pod disruption budget

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    template:
      podDisruptionBudget:
        maxUnavailable: 0
# ...
```

You need to watch the pods that need to be drained. You then add a pod annotation to make the update.

Here, the annotation updates a Kafka pod named `my-cluster-pool-a-1`.

Performing a manual rolling update on a Kafka pod

```
kubectl annotate pod my-cluster-pool-a-1 strimzi.io/manual-rolling-update="true"
```

Additional resources

- [Draining pods using the Strimzi Drain Cleaner](#)
- [Performing a rolling update using a pod annotation](#)
- [PodDisruptionBudgetTemplate schema reference](#)
- [Kubernetes documentation](#)

27.5. Upgrading the Cluster Operator

Use the same method to upgrade the Cluster Operator as the initial method of deployment.

27.5.1. Upgrading the Cluster Operator using installation files

This procedure describes how to upgrade a Cluster Operator deployment to use Strimzi 0.42.0.

Follow this procedure if you deployed the Cluster Operator using the installation YAML files.

The availability of Kafka clusters managed by the Cluster Operator is not affected by the upgrade operation.

NOTE

Refer to the documentation supporting a specific version of Strimzi for information on how to upgrade to that version.

Prerequisites

- An existing Cluster Operator deployment is available.
- You have [downloaded the release artifacts for Strimzi 0.42.0](#).

Procedure

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the `/install/cluster-operator` directory). Any changes will be **overwritten** by the new version of the Cluster Operator.
2. Update your custom resources to reflect the supported configuration options available for Strimzi version 0.42.0.
3. Update the Cluster Operator.
 - a. Modify the installation files for the new Cluster Operator version according to the namespace the Cluster Operator is running in.

On Linux, use:

```
sed -i 's/namespace: .*namespace: my-cluster-operator-namespace/'  
install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*namespace: my-cluster-operator-namespace/'  
install/cluster-operator/*RoleBinding*.yaml
```

- b. If you modified one or more environment variables in your existing Cluster Operator [Deployment](#), edit the `install/cluster-operator/060-Deployment-stimzi-cluster-operator.yaml` file to use those environment variables.
4. When you have an updated configuration, deploy it along with the rest of the installation resources:

```
kubectl replace -f install/cluster-operator
```

Wait for the rolling updates to complete.

5. If the new Operator version no longer supports the Kafka version you are upgrading from, the Cluster Operator returns an error message to say the version is not supported. Otherwise, no error message is returned.
 - If the error message is returned, upgrade to a Kafka version that is supported by the new Cluster Operator version:
 - a. Edit the **Kafka** custom resource.
 - b. Change the `spec.kafka.version` property to a supported Kafka version.
 - If the error message is *not* returned, go to the next step. You will upgrade the Kafka version later.
6. Get the image for the Kafka pod to ensure the upgrade was successful:

```
kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The image tag shows the new Strimzi version followed by the Kafka version:

```
quay.io/strimzi/kafka:0.42.0-kafka-3.7.1
```

You can also [check the upgrade has completed successfully from the status of the Kafka resource](#).

The Cluster Operator is upgraded to version 0.42.0, but the version of Kafka running in the cluster it manages is unchanged.

27.5.2. Upgrading the Cluster Operator using the OperatorHub

If you deployed Strimzi from [OperatorHub.io](#), use the Operator Lifecycle Manager (OLM) to change the update channel for the Strimzi operators to a new Strimzi version.

Updating the channel starts one of the following types of upgrade, depending on your chosen upgrade strategy:

- An automatic upgrade is initiated
- A manual upgrade that requires approval before installation begins

NOTE

If you subscribe to the *stable* channel, you can get automatic updates without changing channels. However, enabling automatic updates is not recommended because of the potential for missing any pre-installation upgrade steps. Use automatic upgrades only on version-specific channels.

For more information on using OperatorHub to upgrade Operators, see the [Operator Lifecycle](#)

27.5.3. Upgrading the Cluster Operator using a Helm chart

If you deployed the Cluster Operator using a Helm chart, use `helm upgrade`.

The `helm upgrade` command does not upgrade the [Custom Resource Definitions for Helm](#). Install the new CRDs manually after upgrading the Cluster Operator. You can access the CRDs from the [GitHub releases page](#) or find them in the `crd` subdirectory inside the Helm Chart.

27.5.4. Upgrading the Cluster Operator returns Kafka version error

If you upgrade the Cluster Operator to a version that does not support the current version of Kafka you are using, you get an *unsupported Kafka version* error. This error applies to all installation methods and means that you must upgrade Kafka to a supported Kafka version. Change the `spec.kafka.version` in the `Kafka` resource to the supported version.

You can use `kubectl` to check for error messages like this in the `status` of the `Kafka` resource.

Checking the Kafka status for errors

```
kubectl get kafka <kafka_cluster_name> -n <namespace> -o jsonpath='{.status.conditions}'
```

Replace `<kafka_cluster_name>` with the name of your Kafka cluster and `<namespace>` with the Kubernetes namespace where the pod is running.

27.5.5. Upgrading from a Strimzi version using the Bidirectional Topic Operator

When deploying the Topic Operator to manage topics, the Cluster Operator enables unidirectional topic management. This means that the Topic Operator only manages Kafka topics associated with `KafkaTopic` resources and does not interfere with topics managed independently within the Kafka cluster.

Previously, the Topic Operator worked in bidirectional mode, which meant it could also perform operations on topics within the Kafka cluster. If you are switching from a version of Strimzi that uses the Bidirectional Topic Operator, after upgrading the Cluster Operator, perform some cleanup tasks on the following internal topics that were used by the operator:

- `strimzi-store-topic`
- `strimzi-topic-operator`
- `consumer-offsets`
- `transaction-state`

For the `strimzi-store-topic` and `strimzi-topic-operator` topics, delete the resources that were used to manage them:

Deleting internal topics used by the operator

```
kubectl delete $(kubectl get kt -n <namespace> -o name | grep strimzi-store-topic) -n <namespace> \
&& kubectl delete $(kubectl get kt -n <namespace> -o name | grep strimzi-topic-operator) -n <namespace>
```

For the internal topics for storing consumer offsets and transaction states, [consumer-offsets](#) and [transaction-state](#), you want to retain them in Kafka, but you don't want them to be managed by the Topic Operator.

Discontinue their management before deleting their resources. Annotating the [KafkaTopic](#) resources with [strimzi.io/managed="false"](#) indicates that the Topic Operator should no longer manage those topics:

Discontinuing management of internal topics

```
kubectl annotate $(kubectl get kt -n <namespace> -o name | grep consumer-offsets) strimzi.io/managed="false" -n <namespace> \
&& kubectl annotate $(kubectl get kt -n <namespace> -o name | grep transaction-state) strimzi.io/managed="false" -n <namespace>
```

Check the statuses of the [KafkaTopic](#) resources to make sure the reconciliation was successful and the topics are no longer managed, as shown in the [procedure to stop managing topics](#).

Having discontinued their management, delete the [KafkaTopic](#) resources:

Deleting the resources for managing internal topics

```
kubectl delete $(kubectl get kt -n <namespace> -o name | grep consumer-offsets) -n <namespace> \
&& kubectl delete $(kubectl get kt -n <namespace> -o name | grep transaction-state) -n <namespace>
```

By discontinuing their management, they won't also be deleted in Kafka.

27.6. Upgrading KRaft-based Kafka clusters

Upgrade a KRaft-based Kafka cluster to a newer supported Kafka version and KRaft metadata version.

NOTE

Refer to the Apache Kafka documentation for the latest on support for KRaft-based upgrades.

Prerequisites

- The Cluster Operator is up and running.
- Before you upgrade the Kafka cluster, check that the properties of the [Kafka](#) resource do *not*

contain configuration options that are not supported in the new Kafka version.

Procedure

1. Update the Kafka cluster configuration:

```
kubectl edit kafka <kafka_configuration_file>
```

2. If configured, check that the current `spec.kafka.metadataVersion` is set to a version supported by the version of Kafka you are upgrading to.

For example, the current version is 3.6-IV2 if upgrading from Kafka version 3.6.1 to 3.7.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    metadataVersion: 3.6-IV2
    version: 3.6.1
    # ...
```

If `metadataVersion` is not configured, Strimzi automatically updates it to the current default after the update to the Kafka version in the next step.

NOTE The value of `metadataVersion` must be a string to prevent it from being interpreted as a floating point number.

3. Change the `Kafka.spec.kafka.version` to specify the new Kafka version; leave the `metadataVersion` at the default for the *current* Kafka version.

NOTE Changing the `kafka.version` ensures that all brokers in the cluster are upgraded to start using the new broker binaries. During this process, some brokers are using the old binaries while others have already upgraded to the new ones. Leaving the `metadataVersion` unchanged at the current setting ensures that the Kafka brokers and controllers can continue to communicate with each other throughout the upgrade.

For example, if upgrading from Kafka 3.6.1 to 3.7.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
```

```
replicas: 3
metadataVersion: 3.6-IV2 ①
version: 3.7.1 ②
# ...
```

① Metadata version is unchanged

② Kafka version is changed to the new version.

4. If the image for the Kafka cluster is defined in `Kafka.spec.kafka.image` of the `Kafka` custom resource, update the `image` to point to a container image with the new Kafka version.

See [Kafka version and image mappings](#)

5. Save and exit the editor, then wait for the rolling updates to upgrade the Kafka nodes to complete.

Check the progress of the rolling updates by watching the pod state transitions:

```
kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The rolling updates ensure that each pod is using the broker binaries for the new version of Kafka.

6. If required, set the `version` property for Kafka Connect and MirrorMaker as the new version of Kafka:
 - a. For Kafka Connect, update `KafkaConnect.spec.version`.
 - b. For MirrorMaker, update `KafkaMirrorMaker.spec.version`.
 - c. For MirrorMaker 2, update `KafkaMirrorMaker2.spec.version`.

NOTE

If you are using custom images that are built manually, you must rebuild those images to ensure that they are up-to-date with the latest Strimzi base image. For example, if you [created a container image from the base Kafka Connect image](#), update the Dockerfile to point to the latest base image and build configuration.

7. If configured, update the Kafka resource to use the new `metadataVersion` version. Otherwise, go to step 9.

For example, if upgrading to Kafka 3.7.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
```

```
metadataVersion: 3.7-IV4
version: 3.7.1
# ...
```

WARNING

Exercise caution when changing the `metadataVersion`, as downgrading may not be possible. You cannot downgrade Kafka if the `metadataVersion` for the new Kafka version is higher than the Kafka version you wish to downgrade to. However, understand the potential implications on support and compatibility when maintaining an older version.

8. Wait for the Cluster Operator to update the cluster.

Check the upgrade has completed successfully from the [status of the Kafka resource](#).

Upgrading client applications

Ensure all Kafka client applications are updated to use the new version of the client binaries as part of the upgrade process and verify their compatibility with the Kafka upgrade. If needed, coordinate with the team responsible for managing the client applications.

TIP

To check that a client is using the latest message format, use the `kafka.server:type=BrokerTopicMetrics,name={Produce|Fetch}MessageConversionsPerSec` metric. The metric shows `0` if the latest message format is being used.

27.7. Upgrading Kafka when using ZooKeeper

If you are using a ZooKeeper-based Kafka cluster, an upgrade requires an update to the Kafka version and the inter-broker protocol version.

If you want to switch a Kafka cluster from using ZooKeeper for metadata management to operating in KRaft mode, the steps must be performed separately from the upgrade. For information on migrating to a KRaft-based cluster, see [Migrating to KRaft mode](#).

27.7.1. Updating Kafka versions

Upgrading Kafka when using ZooKeeper for cluster management requires updates to the Kafka version (`Kafka.spec.kafka.version`) and its inter-broker protocol version (`inter.broker.protocol.version`) in the configuration of the `Kafka` resource. Each version of Kafka has a compatible version of the inter-broker protocol. The inter-broker protocol is used for inter-broker communication. The minor version of the protocol typically increases to match the minor version of Kafka, as shown in the preceding table. The inter-broker protocol version is set cluster wide in the `Kafka` resource. To change it, you edit the `inter.broker.protocol.version` property in `Kafka.spec.kafka.config`.

The following table shows the differences between Kafka versions:

Table 42. Kafka version differences

Kafka version	Inter-broker protocol version	Log message format version	ZooKeeper version
3.6.0	3.6	3.6	3.8.2
3.6.1	3.6	3.6	3.8.3
3.6.2	3.6	3.6	3.8.4
3.7.0	3.7	3.7	3.8.3
3.7.1	3.7	3.7	3.8.4

Log message format version

When a producer sends a message to a Kafka broker, the message is encoded using a specific format. The format can change between Kafka releases, so messages specify which version of the message format they were encoded with.

The properties used to set a specific message format version are as follows:

- `message.format.version` property for topics
- `log.message.format.version` property for Kafka brokers

From Kafka 3.0.0, the message format version values are assumed to match the `inter.broker.protocol.version` and don't need to be set. The values reflect the Kafka version used.

When upgrading to Kafka 3.0.0 or higher, you can remove these settings when you update the `inter.broker.protocol.version`. Otherwise, you can set the message format version based on the Kafka version you are upgrading to.

The default value of `message.format.version` for a topic is defined by the `log.message.format.version` that is set on the Kafka broker. You can manually set the `message.format.version` of a topic by modifying its topic configuration.

Rolling updates from Kafka version changes

The Cluster Operator initiates rolling updates to Kafka brokers when the Kafka version is updated. Further rolling updates depend on the configuration for `inter.broker.protocol.version` and `log.message.format.version`.

If <code>Kafka.spec.kafka.config</code> contains...	The Cluster Operator initiates...
Both the <code>inter.broker.protocol.version</code> and the <code>log.message.format.version</code> .	A single rolling update. After the update, the <code>inter.broker.protocol.version</code> must be updated manually, followed by <code>log.message.format.version</code> . Changing each will trigger a further rolling update.
Either the <code>inter.broker.protocol.version</code> or the <code>log.message.format.version</code> .	Two rolling updates.
No configuration for the <code>inter.broker.protocol.version</code> or the <code>log.message.format.version</code> .	Two rolling updates.

IMPORTANT

From Kafka 3.0.0, when the `inter.broker.protocol.version` is set to `3.0` or higher, the `log.message.format.version` option is ignored and doesn't need to be set. The `log.message.format.version` property for brokers and the `message.format.version` property for topics are deprecated and will be removed in a future release of Kafka.

As part of the Kafka upgrade, the Cluster Operator initiates rolling updates for ZooKeeper.

- A single rolling update occurs even if the ZooKeeper version is unchanged.
- Additional rolling updates occur if the new version of Kafka requires a new ZooKeeper version.

27.7.2. Upgrading clients with older message formats

Before Kafka 3.0, you could configure a specific message format for brokers using the `log.message.format.version` property (or the `message.format.version` property at the topic level). This allowed brokers to accommodate older Kafka clients that were using an outdated message format. Though Kafka inherently supports older clients without explicitly setting this property, brokers would then need to convert the messages from the older clients, which came with a significant performance cost.

Apache Kafka Java clients have supported the latest message format version since version 0.11. If all of your clients are using the latest message version, you can remove the `log.message.format.version` or `message.format.version` overrides when upgrading your brokers.

However, if you still have clients that are using an older message format version, we recommend upgrading your clients first. Start with the consumers, then upgrade the producers before removing the `log.message.format.version` or `message.format.version` overrides when upgrading your brokers. This will ensure that all of your clients can support the latest message format version and that the upgrade process goes smoothly.

You can track Kafka client names and versions using this metric:

- `kafka.server:type=socket-server-metrics,clientSoftwareName=<name>,clientSoftwareVersion=<version>,listener=<listener>,networkProcessor=<processor>`

The following Kafka broker metrics help monitor the performance of message down-conversion:

TIP

- `kafka.network:type=RequestMetrics,name=MessageConversionsTimeMs,request={Produce|Fetch}` provides metrics on the time taken to perform message conversion.
- `kafka.server:type=BrokerTopicMetrics,name={Produce|Fetch}MessageConversionsPerSec,topic=([-.\w]+)` provides metrics on the number of messages converted over a period of time.

27.7.3. Upgrading ZooKeeper-based Kafka clusters

Upgrade a ZooKeeper-based Kafka cluster to a newer supported Kafka version and inter-broker

protocol version.

Prerequisites

- The Cluster Operator is up and running.
- Before you upgrade the Kafka cluster, check that the properties of the **Kafka** resource do *not* contain configuration options that are not supported in the new Kafka version.

Procedure

1. Update the Kafka cluster configuration:

```
kubectl edit kafka <kafka_configuration_file>
```

2. If configured, check that the `inter.broker.protocol.version` and `log.message.format.version` properties are set to the *current* version.

For example, the current version is 3.6 if upgrading from Kafka version 3.6.1 to 3.7.1:

```
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.6.1
    config:
      log.message.format.version: "3.6"
      inter.broker.protocol.version: "3.6"
      # ...
```

If `log.message.format.version` and `inter.broker.protocol.version` are not configured, Strimzi automatically updates these versions to the current defaults after the update to the Kafka version in the next step.

NOTE The value of `log.message.format.version` and `inter.broker.protocol.version` must be strings to prevent them from being interpreted as floating point numbers.

3. Change the `Kafka.spec.kafka.version` to specify the new Kafka version; leave the `log.message.format.version` and `inter.broker.protocol.version` at the defaults for the *current* Kafka version.

NOTE Changing the `kafka.version` ensures that all brokers in the cluster are upgraded to start using the new broker binaries. During this process, some brokers are using the old binaries while others have already upgraded to the new ones. Leaving the `inter.broker.protocol.version` unchanged at the current setting ensures that the brokers can continue to communicate with each other throughout the upgrade.

For example, if upgrading from Kafka 3.6.1 to 3.7.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.7.1 ①
    config:
      log.message.format.version: "3.6" ②
      inter.broker.protocol.version: "3.6" ③
      # ...
```

① Kafka version is changed to the new version.

② Message format version is unchanged.

③ Inter-broker protocol version is unchanged.

WARNING

You cannot downgrade Kafka if the `inter.broker.protocol.version` for the new Kafka version changes. The inter-broker protocol version determines the schemas used for persistent metadata stored by the broker, including messages written to `_consumer_offsets`. The downgraded cluster will not understand the messages.

4. If the image for the Kafka cluster is defined in `Kafka.spec.kafka.image` of the `Kafka` custom resource, update the `image` to point to a container image with the new Kafka version.

See [Kafka version and image mappings](#)

5. Save and exit the editor, then wait for rolling updates to complete.

Check the progress of the rolling updates by watching the pod state transitions:

```
kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The rolling updates ensure that each pod is using the broker binaries for the new version of Kafka.

6. If required, set the `version` property for Kafka Connect and MirrorMaker as the new version of Kafka:
 - a. For Kafka Connect, update `KafkaConnect.spec.version`.
 - b. For MirrorMaker, update `KafkaMirrorMaker.spec.version`.
 - c. For MirrorMaker 2, update `KafkaMirrorMaker2.spec.version`.

NOTE

If you are using custom images that are built manually, you must rebuild those images to ensure that they are up-to-date with the latest Strimzi base image. For example, if you [created a container image from the base Kafka](#)

[Connect image](#), update the Dockerfile to point to the latest base image and build configuration.

7. If configured, update the Kafka resource to use the new `inter.broker.protocol.version` version. Otherwise, go to step 9.

For example, if upgrading to Kafka 3.7.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.7.1
    config:
      log.message.format.version: "3.6"
      inter.broker.protocol.version: "3.7"
      # ...
```

8. Wait for the Cluster Operator to update the cluster.
9. If configured, update the Kafka resource to use the new `log.message.format.version` version. Otherwise, go to step 10.

For example, if upgrading to Kafka 3.7.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.7.1
    config:
      log.message.format.version: "3.7"
      inter.broker.protocol.version: "3.7"
      # ...
```

IMPORTANT

From Kafka 3.0.0, when the `inter.broker.protocol.version` is set to `3.0` or higher, the `log.message.format.version` option is ignored and doesn't need to be set.

10. Wait for the Cluster Operator to update the cluster.

Check the upgrade has completed successfully from the [status of the Kafka resource](#).

Upgrading Kafka client applications

Ensure all Kafka client applications are updated to use the new version of the client binaries as part of the upgrade process and verify their compatibility with the Kafka upgrade. If needed, coordinate

with the team responsible for managing the client applications.

TIP To check that a client is using the latest message format, use the `kafka.server:type=BrokerTopicMetrics,name={Produce|Fetch}MessageConversionsPerSec` metric. The metric shows `0` if the latest message format is being used.

27.8. Checking the status of an upgrade

When performing an upgrade (or downgrade), you can check it completed successfully in the status of the `Kafka` custom resource. The status provides information on the Strimzi and Kafka versions being used.

To ensure that you have the correct versions after completing an upgrade, verify the `kafkaVersion` and `operatorLastSuccessfulVersion` values in the Kafka status.

- `operatorLastSuccessfulVersion` is the version of the Strimzi operator that last performed a successful reconciliation.
- `kafkaVersion` is the the version of Kafka being used by the Kafka cluster.
- `kafkaMetadataVersion` is the metadata version used by KRaft-based Kafka clusters

You can use these values to check an upgrade of Strimzi or Kafka has completed.

Checking an upgrade from the Kafka status

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
spec:
  # ...
status:
  # ...
  kafkaVersion: 3.7.1
  operatorLastSuccessfulVersion: 0.42.0
  kafkaMetadataVersion: 3.7
```

27.9. Switching to FIPS mode when upgrading Strimzi

Upgrade Strimzi to run in FIPS mode on FIPS-enabled Kubernetes clusters. Until Strimzi 0.33, running on FIPS-enabled Kubernetes clusters was possible only by disabling FIPS mode using the `FIPS_MODE` environment variable. From release 0.33, Strimzi supports FIPS mode. If you run Strimzi on a FIPS-enabled Kubernetes cluster with the `FIPS_MODE` set to `disabled`, you can enable it by following this procedure.

Prerequisites

- FIPS-enabled Kubernetes cluster
- An existing Cluster Operator deployment with the `FIPS_MODE` environment variable set to `disabled`

Procedure

1. Upgrade the Cluster Operator to version 0.33 or newer but keep the `FIPS_MODE` environment variable set to `disabled`.
2. If you initially deployed a Strimzi version older than 0.30, it might use old encryption and digest algorithms in its PKCS #12 stores, which are not supported with FIPS enabled. To recreate the certificates with updated algorithms, renew the cluster and clients CA certificates.
 - a. To renew the CAs generated by the Cluster Operator, [add the `force-renew` annotation to the CA secrets to trigger a renewal](#).
 - b. To renew your own CAs, [add the new certificate to the CA secret and update the `ca-cert-generation` annotation with a higher incremental value to capture the update](#).
3. If you use SCRAM-SHA-512 authentication, check the password length of your users. If they are less than 32 characters long, generate a new password in one of the following ways:
 - a. Delete the user secret so that the User Operator generates a new one with a new password of sufficient length.
 - b. If you provided your password using the `.spec.authentication.password` properties of the `KafkaUser` custom resource, update the password in the Kubernetes secret referenced in the same password configuration. Don't forget to update your clients to use the new passwords.
4. Ensure that the CA certificates are using the correct algorithms and the SCRAM-SHA-512 passwords are of sufficient length. You can then enable the FIPS mode.
5. Remove the `FIPS_MODE` environment variable from the Cluster Operator deployment. This restarts the Cluster Operator and rolls all the operands to enable the FIPS mode. After the restart is complete, all Kafka clusters now run with FIPS mode enabled.

Chapter 28. Downgrading Strimzi

If you are encountering issues with the version of Strimzi you upgraded to, you can revert your installation to the previous version.

If you used the YAML installation files to install Strimzi, you can use the YAML installation files from the previous release to perform the downgrade procedures. You can downgrade Strimzi by updating the Cluster Operator and the version of Kafka you are using. Kafka version downgrades are performed by the Cluster Operator.

WARNING

The following downgrade instructions are only suitable if you installed Strimzi using the installation files. If you installed Strimzi using another method, like [OperatorHub.io](#), downgrade may not be supported by that method unless specified in their documentation. To ensure a successful downgrade process, it is essential to use a supported approach.

28.1. Downgrading the Cluster Operator to a previous version

If you are encountering issues with Strimzi, you can revert your installation.

This procedure describes how to downgrade a Cluster Operator deployment to a previous version.

Prerequisites

- An existing Cluster Operator deployment is available.
- You have [downloaded the installation files for the previous version](#).

Before you begin

Check the downgrade requirements of the [Strimzi feature gates](#). If a feature gate is permanently enabled, you may need to downgrade to a version that allows you to disable it before downgrading to your target version.

Procedure

1. Take note of any configuration changes made to the existing Cluster Operator resources (in the `/install/cluster-operator` directory). Any changes will be **overwritten** by the previous version of the Cluster Operator.
2. Revert your custom resources to reflect the supported configuration options available for the version of Strimzi you are downgrading to.
3. Update the Cluster Operator.
 - a. Modify the installation files for the previous version according to the namespace the Cluster Operator is running in.

On Linux, use:

```
sed -i 's/namespace: .*namespace: my-cluster-operator-namespace/'
```

```
install/cluster-operator/*RoleBinding*.yaml
```

On MacOS, use:

```
sed -i '' 's/namespace: .*namespace: my-cluster-operator-namespace/'  
install/cluster-operator/*RoleBinding*.yaml
```

- b. If you modified one or more environment variables in your existing Cluster Operator [Deployment](#), edit the [install/cluster-operator/060-Deployment-stimzi-cluster-operator.yaml](#) file to use those environment variables.
4. When you have an updated configuration, deploy it along with the rest of the installation resources:

```
kubectl replace -f install/cluster-operator
```

Wait for the rolling updates to complete.

5. Get the image for the Kafka pod to ensure the downgrade was successful:

```
kubectl get pod my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The image tag shows the new Strimzi version followed by the Kafka version. For example, [`<strimzi_version>-kafka-<kafka_version>`](#).

You can also [check the downgrade has completed successfully from the status of the Kafka resource](#).

28.2. Downgrading KRaft-based Kafka clusters and client applications

Downgrade a KRaft-based Kafka cluster to an earlier version. When downgrading a KRaft-based Kafka cluster to a lower version, like moving from 3.7.1 to 3.6.1, ensure that the metadata version used by the Kafka cluster is a version supported by the Kafka version you want to downgrade to. The metadata version for the Kafka version you are downgrading from must not be higher than the version you are downgrading to.

NOTE

Consult the Apache Kafka documentation for information regarding the support and limitations associated with KRaft-based downgrades.

Prerequisites

- The Cluster Operator is up and running.
- Before you downgrade the Kafka cluster, check the following for the [Kafka](#) resource:
 - The [Kafka](#) custom resource does not contain options that are not supported by the Kafka

version being downgraded to.

- `spec.kafka.metadataVersion` is set to a version that is supported by the Kafka version being downgraded to.

Procedure

1. Update the Kafka cluster configuration.

```
kubectl edit kafka <kafka_configuration_file>
```

2. Change the `metadataVersion` version to a version supported by the Kafka version you are downgrading to; leave the `Kafka.spec.kafka.version` unchanged at the *current* Kafka version.

For example, if downgrading from Kafka 3.7.1 to 3.6.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    metadataVersion: 3.6-IV2 ①
    version: 3.7.1 ②
    # ...
```

① Metadata version is changed to a version supported by the earlier Kafka version.

② Kafka version is unchanged.

NOTE

The value of `metadataVersion` must be a string to prevent it from being interpreted as a floating point number.

3. Save the change, and wait for Cluster Operator to update `.status.kafkaMetadataVersion` for the `Kafka` resource.

4. Change the `Kafka.spec.kafka.version` to the previous version.

For example, if downgrading from Kafka 3.7.1 to 3.6.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3
    metadataVersion: 3.6-IV2 ①
    version: 3.6.1 ②
```

```
# ...
```

- ① Metadata version is supported by the Kafka version.
 - ② Kafka version is changed to the new version.
5. If the image for the Kafka version is different from the image defined in `STRIMZI_KAFKA_IMAGES` for the Cluster Operator, update `Kafka.spec.kafka.image`.

See [Kafka version and image mappings](#).

6. Wait for the Cluster Operator to update the cluster.

You can [check the downgrade has completed successfully from the status of the Kafka resource](#).

7. Downgrade all client applications (consumers) to use the previous version of the client binaries.

The Kafka cluster and clients are now using the previous Kafka version.

28.3. Downgrading Kafka when using ZooKeeper

If you are using Kafka in ZooKeeper mode, the downgrade process involves changing the Kafka version and the related `log.message.format.version` and `inter.broker.protocol.version` properties.

28.3.1. Kafka version compatibility for downgrades

Kafka downgrades are dependent on compatible current and target [Kafka versions](#), and the state at which messages have been logged.

You cannot revert to the previous Kafka version if that version does not support any of the `inter.broker.protocol.version` settings which have *ever been used* in that cluster, or messages have been added to message logs that use a newer `log.message.format.version`.

The `inter.broker.protocol.version` determines the schemas used for persistent metadata stored by the broker, such as the schema for messages written to `_consumer_offsets`. If you downgrade to a version of Kafka that does not understand an `inter.broker.protocol.version` that has ever been previously used in the cluster the broker will encounter data it cannot understand.

If the target downgrade version of Kafka has:

- The *same* `log.message.format.version` as the current version, the Cluster Operator downgrades by performing a single rolling restart of the brokers.
- A *different* `log.message.format.version`, downgrading is only possible if the running cluster has *always had* `log.message.format.version` set to the version used by the downgraded version. This is typically only the case if the upgrade procedure was aborted before the `log.message.format.version` was changed. In this case, the downgrade requires:
 - Two rolling restarts of the brokers if the interbroker protocol of the two versions is different
 - A single rolling restart if they are the same

Downgrading is *not possible* if the new version has ever used a `log.message.format.version` that is not supported by the previous version, including when the default value for `log.message.format.version` is used. For example, this resource can be downgraded to Kafka version 3.6.1 because the `log.message.format.version` has not been changed:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.7.1
    config:
      log.message.format.version: "3.6"
      # ...
```

The downgrade would not be possible if the `log.message.format.version` was set at "3.7" or a value was absent, so that the parameter took the default value for a 3.7.1 broker of 3.7.

IMPORTANT

From Kafka 3.0.0, when the `inter.broker.protocol.version` is set to 3.0 or higher, the `log.message.format.version` option is ignored and doesn't need to be set.

28.3.2. Downgrading ZooKeeper-based Kafka clusters and client applications

Downgrade a ZooKeeper-based Kafka cluster to an earlier version. When downgrading a ZooKeeper-based Kafka cluster to a lower version, like moving from 3.7.1 to 3.6.1, ensure that the inter-broker protocol version used by the Kafka cluster is a version supported by the Kafka version you want to downgrade to. The inter-broker protocol version for the Kafka version you are downgrading from must not be higher than the version you are downgrading to.

NOTE

Consult the Apache Kafka documentation for information regarding the support and limitations associated with ZooKeeper-based downgrades.

Prerequisites

- The Cluster Operator is up and running.
- Before you downgrade the Kafka cluster, check the following for the `Kafka` resource:
 - IMPORTANT: [Compatibility of Kafka versions](#).
 - The `Kafka` custom resource does not contain options that are not supported by the Kafka version being downgraded to.
 - `Kafka.spec.kafka.config` has a `log.message.format.version` and `inter.broker.protocol.version` that is supported by the Kafka version being downgraded to.

From Kafka 3.0.0, when the `inter.broker.protocol.version` is set to 3.0 or higher, the `log.message.format.version` option is ignored and doesn't need to be set.

Procedure

1. Update the Kafka cluster configuration.

```
kubectl edit kafka <kafka_configuration_file>
```

2. Change the `inter.broker.protocol.version` version (and `log.message.format.version`, if applicable) to a version supported by the Kafka version you are downgrading to; leave the `Kafka.spec.kafka.version` unchanged at the *current* Kafka version.

For example, if downgrading from Kafka 3.7.1 to 3.6.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafka:
    version: 3.7.1 ①
    config:
      inter.broker.protocol.version: "3.6" ②
      log.message.format.version: "3.6"
      # ...
```

① Kafka version is unchanged.

② Inter-broker protocol version is changed to a version supported by the earlier Kafka version.

NOTE The value of `log.message.format.version` and `inter.broker.protocol.version` must be strings to prevent them from being interpreted as floating point numbers.

3. Save and exit the editor, then wait for rolling updates to complete.

Check the progress of the rolling updates by watching the pod state transitions:

```
kubectl get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

The rolling updates ensure that each pod is using the specified Kafka inter-broker protocol version.

4. Change the `Kafka.spec.kafka.version` to the previous version.

For example, if downgrading from Kafka 3.7.1 to 3.6.1:

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
```

```
metadata:  
  name: my-cluster  
spec:  
  # ...  
  kafka:  
    version: 3.6.1 ①  
    config:  
      inter.broker.protocol.version: "3.6" ②  
      log.message.format.version: "3.6"  
      # ...
```

① Kafka version is changed to the new version.

② Inter-broker protocol version is supported by the Kafka version.

5. If the image for the Kafka version is different from the image defined in `STRIMZI_KAFKA_IMAGES` for the Cluster Operator, update `Kafka.spec.kafka.image`.

See [Kafka version and image mappings](#).

6. Wait for the Cluster Operator to update the cluster.

You can [check the downgrade has completed successfully](#) from the status of the `Kafka` resource.

7. Downgrade all client applications (consumers) to use the previous version of the client binaries.

The Kafka cluster and clients are now using the previous Kafka version.

8. If you are reverting back to a version of Strimzi earlier than 0.22, which uses ZooKeeper for the storage of topic metadata, delete the internal topic store topics from the Kafka cluster.

```
kubectl run kafka-admin -ti --image=quay.io/strimzi/kafka:0.42.0-kafka-3.7.1  
  --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server  
  localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog  
  --delete 88 ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic  
  __strimzi_store_topic --delete
```

Chapter 29. Uninstalling Strimzi

You can uninstall Strimzi using the CLI or by unsubscribing from OperatorHub.io.

Use the same approach you used to install Strimzi.

When you uninstall Strimzi, you will need to identify resources created specifically for a deployment and referenced from the Strimzi resource.

Such resources include:

- Secrets (Custom CAs and certificates, Kafka Connect secrets, and other Kafka secrets)
- Logging [ConfigMaps](#) (of type `external`)

These are resources referenced by [Kafka](#), [KafkaConnect](#), [KafkaMirrorMaker](#), or [KafkaBridge](#) configuration.

Deleting CRDs and related custom resources

WARNING

When a [CustomResourceDefinition](#) is deleted, custom resources of that type are also deleted. This includes the [Kafka](#), [KafkaConnect](#), [KafkaMirrorMaker](#), and [KafkaBridge](#) resources managed by Strimzi, as well as the [StrimziPodSet](#) resource Strimzi uses to manage the pods of the Kafka components. In addition, any Kubernetes resources created by these custom resources, such as [Deployment](#), [Pod](#), [Service](#), and [ConfigMap](#) resources, are also removed. Always exercise caution when deleting these resources to avoid unintended data loss.

29.1. Uninstalling Strimzi using the CLI

This procedure describes how to use the [kubectl](#) command-line tool to uninstall Strimzi and remove resources related to the deployment.

Prerequisites

- Access to a Kubernetes cluster using an account with `cluster-admin` or `strimzi-admin` permissions.
- You have identified the resources to be deleted.

You can use the following [kubectl](#) CLI command to find resources and also verify that they have been removed when you have uninstalled Strimzi.

Command to find resources related to a Strimzi deployment

```
kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

Replace `<resource_type>` with the type of the resource you are checking, such as `secret` or `configmap`.

Procedure

1. Delete the Cluster Operator [Deployment](#), related [CustomResourceDefinitions](#), and [RBAC](#) resources.

Specify the installation files used to deploy the Cluster Operator.

```
kubectl delete -f install/cluster-operator
```

2. Delete the resources you identified in the prerequisites.

```
kubectl delete <resource_type> <resource_name> -n <namespace>
```

Replace *<resource_type>* with the type of resource you are deleting and *<resource_name>* with the name of the resource.

Example to delete a secret

```
kubectl delete secret my-cluster-clients-ca-cert -n my-project
```

29.2. Uninstalling Strimzi from OperatorHub.io

This procedure describes how to uninstall Strimzi from OperatorHub.io and remove resources related to the deployment.

You perform the steps using the [kubectl](#) command-line tool.

Prerequisites

- Access to a Kubernetes cluster using an account with [cluster-admin](#) or [strimzi-admin](#) permissions.
- You have identified the resources to be deleted.

You can use the following [kubectl](#) CLI command to find resources and also verify that they have been removed when you have uninstalled Strimzi.

Command to find resources related to a Strimzi deployment

```
kubectl get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

Replace *<resource_type>* with the type of the resource you are checking, such as [secret](#) or [configmap](#).

Procedure

1. Delete the Strimzi subscription.

```
kubectl delete subscription strimzi-cluster-operator -n <namespace>
```

2. Delete the cluster service version (CSV).

```
kubectl delete csv strimzi-cluster-operator.<version> -n <namespace>
```

3. Remove related CRDs.

```
kubectl get crd -l app=strimzi -o name | xargs kubectl delete
```

Chapter 30. Cluster recovery from persistent volumes

You can recover a Kafka cluster from persistent volumes (PVs) if they are still present.

30.1. Cluster recovery scenarios

Recovering from PVs is possible in the following scenarios:

- Unintentional deletion of a namespace
- Loss of an entire Kubernetes cluster while PVs remain in the infrastructure

The recovery procedure for both scenarios is to recreate the original `PersistentVolumeClaim` (PVC) resources.

30.1.1. Recovering from namespace deletion

When you delete a namespace, all resources within that namespace—including PVCs, pods, and services—are deleted. If the `reclaimPolicy` for the PV resource specification is set to `Retain`, the PV retains its data and is not deleted. This configuration allows you to recover from namespace deletion.

PV configuration to retain data

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
  # ...
  persistentVolumeReclaimPolicy: Retain
```

Alternatively, PVs can inherit the reclaim policy from an associated storage class. Storage classes are used for dynamic volume allocation.

By configuring the `reclaimPolicy` property for the storage class, PVs created with this class use the specified reclaim policy. The storage class is assigned to the PV using the `storageClassName` property.

Storage class configuration to retain data

```
apiVersion: v1
kind: StorageClass
metadata:
  name: gp2-retain
parameters:
  # ...
# ...
reclaimPolicy: Retain
```

Storage class specified for PV

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
# ...
storageClassName: gp2-retain
```

NOTE

When using **Retain** as the reclaim policy, you must manually delete PVs if you intend to delete the entire cluster.

30.1.2. Recovering from cluster loss

If you lose the entire Kubernetes cluster, all resources—including PVs, PVCs, and namespaces—are lost. However, it's possible to recover if the physical storage backing the PVs remains intact.

To recover, you need to set up a new Kubernetes cluster and manually reconfigure the PVs to use the existing storage.

30.2. Recovering a deleted Kafka cluster

This procedure describes how to recover a deleted cluster from persistent volumes (PVs) by recreating the original **PersistentVolumeClaim** (PVC) resources.

If the Topic Operator and User Operator are deployed, you can recover **KafkaTopic** and **KafkaUser** resources by recreating them. It is important that you recreate the **KafkaTopic** resources with the same configurations, or the Topic Operator will try to update them in Kafka. This procedure shows how to recreate both resources.

WARNING

If the User Operator is enabled and Kafka users are not recreated, users are deleted from the Kafka cluster immediately after recovery.

Before you begin

In this procedure, it is essential that PVs are mounted into the correct PVC to avoid data corruption. A **volumeName** is specified for the PVC and this must match the name of the PV.

For more information, see [Persistent storage](#).

Procedure

1. Check information on the PVs in the cluster:

```
kubectl get pv
```

Information is presented for PVs with data.

Example PV output

NAME	RECLAIMPOLICY	CLAIM
pvc-5e9c5c7f-3317-11ea-a650-06e1eadd9a4c ...	Retain ...	myproject/data-my-cluster-zookeeper-1
pvc-5e9cc72d-3317-11ea-97b0-0aef8816c7ea ...	Retain ...	myproject/data-my-cluster-zookeeper-0
pvc-5ead43d1-3317-11ea-97b0-0aef8816c7ea ...	Retain ...	myproject/data-my-cluster-zookeeper-2
pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c ...	Retain ...	myproject/data-0-my-cluster-kafka-0
pvc-7e21042e-3317-11ea-9786-02deaf9aa87e ...	Retain ...	myproject/data-0-my-cluster-kafka-1
pvc-7e226978-3317-11ea-97b0-0aef8816c7ea ...	Retain ...	myproject/data-0-my-cluster-kafka-2

- **NAME** is the name of each PV.
- **RECLAIMPOLICY** shows that PVs are retained, meaning that the PV is not automatically deleted when the PVC is deleted.
- **CLAIM** shows the link to the original PVCs.

2. Recreate the original namespace:

```
kubectl create namespace my-project
```

Here, we recreate the **my-project** namespace.

3. Recreate the original PVC resource specifications, linking the PVCs to the appropriate PV:

Example PVC resource specification

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-0-my-cluster-kafka-0
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
  storageClassName: gp2-retain
  volumeMode: Filesystem
  volumeName: pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c
```

4. Edit the PV specifications to delete the **claimRef** properties that bound the original PVC.

Example PV specification

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    kubernetes.io/createdby: aws-ebs-dynamic-provisioner
    pv.kubernetes.io/bound-by-controller: "yes"
    pv.kubernetes.io/provisioned-by: kubernetes.io/aws-ebs
  creationTimestamp: "<date>"
  finalizers:
  - kubernetes.io/pv-protection
  labels:
    failure-domain.beta.kubernetes.io/region: eu-west-1
    failure-domain.beta.kubernetes.io/zone: eu-west-1c
  name: pvc-7e226978-3317-11ea-97b0-0aef8816c7ea
  resourceVersion: "39431"
  selfLink: /api/v1/persistentvolumes/pvc-7e226978-3317-11ea-97b0-0aef8816c7ea
  uid: 7efe6b0d-3317-11ea-a650-06e1eadd9a4c
spec:
  accessModes:
  - ReadWriteOnce
  awsElasticBlockStore:
    fsType: xfs
    volumeID: aws://eu-west-1c/vol-09db3141656d1c258
  capacity:
    storage: 100Gi
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: data-0-my-cluster-kafka-2
    namespace: myproject
    resourceVersion: "39113"
    uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: failure-domain.beta.kubernetes.io/zone
          operator: In
          values:
          - eu-west-1c
        - key: failure-domain.beta.kubernetes.io/region
          operator: In
          values:
          - eu-west-1
  persistentVolumeReclaimPolicy: Retain
  storageClassName: gp2-retain
  volumeMode: Filesystem
```

In the example, the following properties are deleted:

```
claimRef:  
  apiVersion: v1  
  kind: PersistentVolumeClaim  
  name: data-0-my-cluster-kafka-2  
  namespace: myproject  
  resourceVersion: "39113"  
  uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
```

5. Deploy the Cluster Operator:

```
kubectl create -f install/cluster-operator -n my-project
```

6. Recreate all **KafkaTopic** resources by applying the **KafkaTopic** resource configuration:

```
kubectl apply -f <topic_configuration_file>
```

7. Recreate all **KafkaUser** resources:

- If user passwords and certificates need to be retained, recreate the user secrets before recreating the **KafkaUser** resources.

If the secrets are not recreated, the User Operator will generate new credentials automatically. Ensure that the recreated secrets have exactly the same name, labels, and fields as the original secrets.

- Apply the **KafkaUser** resource configuration:

```
kubectl apply -f <user_configuration_file>
```

8. Deploy the Kafka cluster using the original configuration for the **Kafka** resource:

```
kubectl apply -f <kafka_resource_configuration>.yaml -n my-project
```

9. Verify the recovery of the **KafkaTopic** resources:

```
kubectl get kafkatopics -o wide -w -n my-project
```

Kafka topic status

NAME	CLUSTER	PARTITIONS	REPLICATION FACTOR	READY
my-topic-1	my-cluster	10	3	True
my-topic-2	my-cluster	10	3	True

my-topic-3	my-cluster	10	3	True
------------	------------	----	---	------

KafkaTopic custom resource creation is successful when the READY output shows True.

10. Verify the recovery of the KafkaUser resources:

```
kubectl get kafkausers -o wide -w -n my-project
```

Kafka user status

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
my-user-1	my-cluster	tls	simple	True
my-user-2	my-cluster	tls	simple	True
my-user-3	my-cluster	tls	simple	True

KafkaUser custom resource creation is successful when the READY output shows True.

Chapter 31. Tuning Kafka configuration

Fine-tuning the performance of your Kafka deployment involves optimizing various configuration properties according to your specific requirements. This section provides an introduction to common configuration options available for Kafka brokers, producers, and consumers.

While a minimum set of configurations is necessary for Kafka to function, Kafka properties allow for extensive adjustments. Through configuration properties, you can enhance latency, throughput, and overall efficiency, ensuring that your Kafka deployment meets the demands of your applications.

For effective tuning, take a methodical approach. Begin by analyzing relevant metrics to identify potential bottlenecks or areas for improvement. Adjust configuration parameters iteratively, monitoring the impact on performance metrics, and then refine your settings accordingly.

For more information about Apache Kafka configuration properties, see the [Apache Kafka documentation](#).

NOTE The guidance provided here offers a starting point for tuning your Kafka deployment. Finding the optimal configuration depends on factors such as workload, infrastructure, and performance objectives.

31.1. Tools that help with tuning

The following tools help with Kafka tuning:

- Cruise Control generates optimization proposals that you can use to assess and implement a cluster rebalance
- Kafka Static Quota plugin sets limits on brokers
- Rack configuration spreads broker partitions across racks and allows consumers to fetch data from the nearest replica

31.2. Managed broker configurations

When you deploy Strimzi on Kubernetes, you can specify broker configuration through the `config` property of the [Kafka](#) custom resource. However, certain broker configuration options are managed directly by Strimzi.

As such, you cannot configure the following options:

- `broker.id` to specify the ID of the Kafka broker
- `log.dirs` directories for log data
- `zookeeper.connect` configuration to connect Kafka with ZooKeeper
- `listeners` to expose the Kafka cluster to clients
- `authorization` mechanisms to allow or decline actions executed by users

- **authentication** mechanisms to prove the identity of users requiring access to Kafka

Broker IDs start from 0 (zero) and correspond to the number of broker replicas. Log directories are mounted to `/var/lib/kafka/data/kafka-log IDX` based on the `spec.kafka.storage` configuration in the `Kafka` custom resource. IDX is the Kafka broker pod index.

For a list of exclusions, see the [KafkaClusterSpec schema reference](#).

31.3. Kafka broker configuration tuning

Use configuration properties to optimize the performance of Kafka brokers. You can use standard Kafka broker configuration options, except for properties managed directly by Strimzi.

31.3.1. Basic broker configuration

A typical broker configuration will include settings for properties related to topics, threads and logs.

Basic broker configuration properties

```
# ...
num.partitions=1
default.replication.factor=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min_isr=2
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
num.network.threads=3
num.io.threads=8
num.recovery.threads.per.data.dir=1
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
group.initial.rebalance.delay.ms=0
zookeeper.connection.timeout.ms=6000
# ...
```

31.3.2. Replicating topics for high availability

Basic topic properties set the default number of partitions and replication factor for topics, which will apply to topics that are created without these properties being explicitly set, including when topics are created automatically.

```
# ...
num.partitions=1
auto.create.topics.enable=false
default.replication.factor=3
```

```
min.insync.replicas=2  
replica.fetch.max.bytes=1048576  
# ...
```

For high availability environments, it is advisable to increase the replication factor to at least 3 for topics and set the minimum number of in-sync replicas required to 1 less than the replication factor.

The `auto.create.topics.enable` property is enabled by default so that topics that do not already exist are created automatically when needed by producers and consumers. If you are using automatic topic creation, you can set the default number of partitions for topics using `num.partitions`. Generally, however, this property is disabled so that more control is provided over topics through explicit topic creation.

For `data durability`, you should also set `min.insync.replicas` in your *topic* configuration and message delivery acknowledgments using `acks=all` in your *producer* configuration.

Use `replica.fetch.max.bytes` to set the maximum size, in bytes, of messages fetched by each follower that replicates the leader partition. Change this value according to the average message size and throughput. When considering the total memory allocation required for read/write buffering, the memory available must also be able to accommodate the maximum replicated message size when multiplied by all followers.

The `delete.topic.enable` property is enabled by default to allow topics to be deleted. In a production environment, you should disable this property to avoid accidental topic deletion, resulting in data loss. You can, however, temporarily enable it and delete topics and then disable it again.

NOTE

When running Strimzi on Kubernetes, the Topic Operator can provide operator-style topic management. You can use the `KafkaTopic` resource to create topics. For topics created using the `KafkaTopic` resource, the replication factor is set using `spec.replicas`. If `delete.topic.enable` is enabled, you can also delete topics using the `KafkaTopic` resource.

```
# ...  
auto.create.topics.enable=false  
delete.topic.enable=true  
# ...
```

31.3.3. Internal topic settings for transactions and commits

If you are `using transactions` to enable atomic writes to partitions from producers, the state of the transactions is stored in the internal `_transaction_state` topic. By default, the brokers are configured with a replication factor of 3 and a minimum of 2 in-sync replicas for this topic, which means that a minimum of three brokers are required in your Kafka cluster.

```
# ...
```

```
transaction.state.log.replication.factor=3  
transaction.state.log.min.isr=2  
# ...
```

Similarly, the internal `__consumer_offsets` topic, which stores consumer state, has default settings for the number of partitions and replication factor.

```
# ...  
offsets.topic.num.partitions=50  
offsets.topic.replication.factor=3  
# ...
```

Do not reduce these settings in production. You can increase the settings in a *production* environment. As an exception, you might want to reduce the settings in a single-broker *test* environment.

31.3.4. Improving request handling throughput by increasing I/O threads

Network threads handle requests to the Kafka cluster, such as produce and fetch requests from client applications. Produce requests are placed in a request queue. Responses are placed in a response queue.

The number of network threads per listener should reflect the replication factor and the levels of activity from client producers and consumers interacting with the Kafka cluster. If you are going to have a lot of requests, you can increase the number of threads, using the amount of time threads are idle to determine when to add more threads.

To reduce congestion and regulate the request traffic, you can limit the number of requests allowed in the request queue. When the request queue is full, all incoming traffic is blocked.

I/O threads pick up requests from the request queue to process them. Adding more threads can improve throughput, but the number of CPU cores and disk bandwidth imposes a practical upper limit. At a minimum, the number of I/O threads should equal the number of storage volumes.

```
# ...  
num.network.threads=3 ①  
queued.max.requests=500 ②  
num.io.threads=8 ③  
num.recovery.threads.per.data.dir=4 ④  
# ...
```

① The number of network threads for the Kafka cluster.

② The number of requests allowed in the request queue.

③ The number of I/O threads for a Kafka broker.

④ The number of threads used for log loading at startup and flushing at shutdown. Try setting to a value of at least the number of cores.

Configuration updates to the thread pools for all brokers might occur dynamically at the cluster level. These updates are restricted to between half the current size and twice the current size.

The following Kafka broker metrics can help with working out the number of threads required:

- `kafka.network:type=SocketServer, name=NetworkProcessorAvgIdlePercent` provides metrics on the average time network threads are idle as a percentage.
- `kafka.server:type=KafkaRequestHandlerPool, name=RequestHandlerAvgIdlePercent` provides metrics on the average time I/O threads are idle as a percentage.

TIP

If there is 0% idle time, all resources are in use, which means that adding more threads might be beneficial. When idle time goes below 30%, performance may start to suffer.

If threads are slow or limited due to the number of disks, you can try increasing the size of the buffers for network requests to improve throughput:

```
# ...
replica.socket.receive.buffer.bytes=65536
# ...
```

And also increase the maximum number of bytes Kafka can receive:

```
# ...
socket.request.max.bytes=104857600
# ...
```

31.3.5. Increasing bandwidth for high latency connections

Kafka batches data to achieve reasonable throughput over high-latency connections from Kafka to clients, such as connections between datacenters. However, if high latency is a problem, you can increase the size of the buffers for sending and receiving messages.

```
# ...
socket.send.buffer.bytes=1048576
socket.receive.buffer.bytes=1048576
# ...
```

You can estimate the optimal size of your buffers using a *bandwidth-delay product* calculation, which multiplies the maximum bandwidth of the link (in bytes/s) with the round-trip delay (in seconds) to give an estimate of how large a buffer is required to sustain maximum throughput.

31.3.6. Managing Kafka logs with delete and compact policies

Kafka relies on logs to store message data. A log consists of a series of segments, where each segment is associated with offset-based and timestamp-based indexes. New messages are written to an *active* segment and are never subsequently modified. When serving fetch requests from consumers, the segments are read. Periodically, the active segment is *rolled* to become read-only, and a new active segment is created to replace it. There is only one active segment per topic-partition per broker. Older segments are retained until they become eligible for deletion.

Configuration at the broker level determines the maximum size in bytes of a log segment and the time in milliseconds before an active segment is rolled:

```
# ...
log.segment.bytes=1073741824
log.roll.ms=604800000
# ...
```

These settings can be overridden at the topic level using `segment.bytes` and `segment.ms`. The choice to lower or raise these values depends on the policy for segment deletion. A larger size means the active segment contains more messages and is rolled less often. Segments also become eligible for deletion less frequently.

In Kafka, log cleanup policies determine how log data is managed. In most cases, you won't need to change the default configuration at the cluster level, which specifies the `delete` cleanup policy and enables the log cleaner used by the `compact` cleanup policy:

```
# ...
log.cleanup.policy=delete
log.cleaner.enable=true
# ...
```

Delete cleanup policy

Delete cleanup policy is the default cluster-wide policy for all topics. The policy is applied to topics that do not have a specific topic-level policy configured. Kafka removes older segments based on time-based or size-based log retention limits.

Compact cleanup policy

Compact cleanup policy is generally configured as a topic-level policy (`cleanup.policy=compact`). Kafka's log cleaner applies compaction on specific topics, retaining only the most recent value for a key in the topic. You can also configure topics to use both policies (`cleanup.policy=compact,delete`).

Setting up retention limits for the delete policy

Delete cleanup policy corresponds to managing logs with data retention. The policy is suitable when data does not need to be retained forever. You can establish time-based or size-based log retention and cleanup policies to keep logs bounded.

When log retention policies are employed, non-active log segments are removed when retention limits are reached. Deletion of old segments helps to prevent exceeding disk capacity.

For time-based log retention, you set a retention period based on hours, minutes, or milliseconds:

```
# ...
log.retention.ms=1680000
# ...
```

The retention period is based on the time messages were appended to the segment. Kafka uses the timestamp of the latest message within a segment to determine if that segment has expired or not. The milliseconds configuration has priority over minutes, which has priority over hours. The minutes and milliseconds configurations are null by default, but the three options provide a substantial level of control over the data you wish to retain. Preference should be given to the milliseconds configuration, as it is the only one of the three properties that is dynamically updateable.

If `log.retention.ms` is set to -1, no time limit is applied to log retention, and all logs are retained. However, this setting is not generally recommended as it can lead to issues with full disks that are difficult to rectify.

For size-based log retention, you specify a minimum log size (in bytes):

```
# ...
log.retention.bytes=1073741824
# ...
```

This means that Kafka will ensure there is always at least the specified amount of log data available.

For example, if you set `log.retention.bytes` to 1000 and `log.segment.bytes` to 300, Kafka will keep 4 segments plus the active segment, ensuring a minimum of 1000 bytes are available. When the active segment becomes full and a new segment is created, the oldest segment is deleted. At this point, the size on disk may exceed the specified 1000 bytes, potentially ranging between 1200 and 1500 bytes (excluding index files).

A potential issue with using a log size is that it does not take into account the time messages were appended to a segment. You can use time-based and size-based log retention for your cleanup policy to get the balance you need. Whichever threshold is reached first triggers the cleanup.

To add a time delay before a segment file is deleted from the system, you can use `log.segment.delete.delay.ms` at the broker level for all topics:

```
# ...
log.segment.delete.delay.ms=60000
# ...
```

Or configure `file.delete.delay.ms` at the topic level.

You set the frequency at which the log is checked for cleanup in milliseconds:

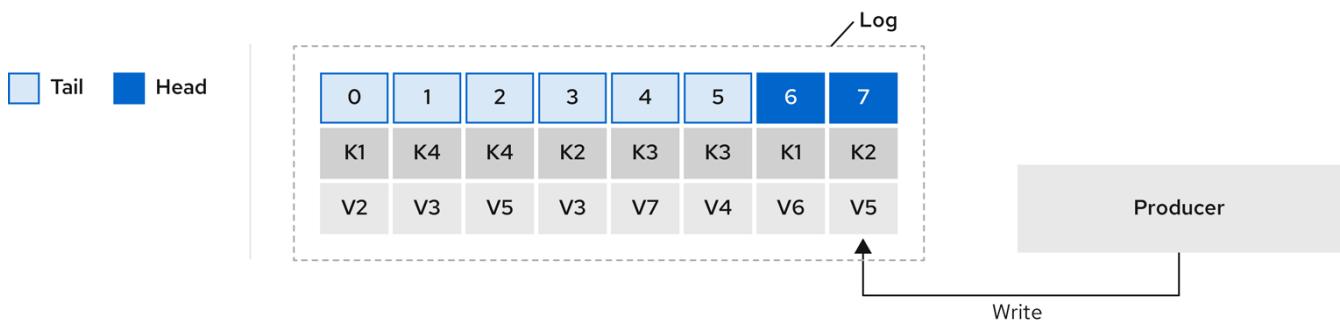
```
# ...
log.retention.check.interval.ms=300000
# ...
```

Adjust the log retention check interval in relation to the log retention settings. Smaller retention sizes might require more frequent checks. The frequency of cleanup should be often enough to manage the disk space but not so often it affects performance on a broker.

Retaining the most recent messages using compact policy

When you enable log compaction for a topic by setting `cleanup.policy=compact`, Kafka uses the log cleaner as a background thread to perform the compaction. The compact policy guarantees that the most recent message for each message key is retained, effectively cleaning up older versions of records. The policy is suitable when message values are changeable, and you want to retain the latest update.

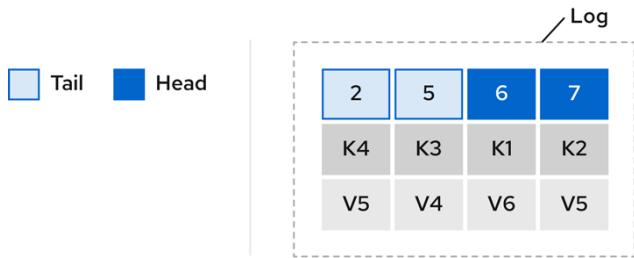
If a cleanup policy is set for log compaction, the *head* of the log operates as a standard Kafka log, with writes for new messages appended in order. In the *tail* of a compacted log, where the log cleaner operates, records are deleted if another record with the same key occurs later in the log. Messages with null values are also deleted. To use compaction, you must have keys to identify related messages because Kafka guarantees that the latest messages for each key will be retained, but it does not guarantee that the whole compacted log will not contain duplicates.



212_Streams_0322

Figure 8. Log showing key value writes with offset positions before compaction

Using keys to identify messages, Kafka compaction keeps the latest message (with the highest offset) that is present in the log tail for a specific message key, eventually discarding earlier messages that have the same key. The message in its latest state is always available, and any out-of-date records of that particular message are eventually removed when the log cleaner runs. You can restore a message back to a previous state. Records retain their original offsets even when surrounding records get deleted. Consequently, the tail can have non-contiguous offsets. When consuming an offset that's no longer available in the tail, the record with the next higher offset is found.



212_Streams_0322

Figure 9. Log after compaction

If appropriate, you can add a delay to the compaction process:

```
# ...
log.cleaner.delete.retention.ms=86400000
# ...
```

The deleted data retention period gives time to notice the data is gone before it is irretrievably deleted.

To delete all messages related to a specific key, a producer can send a *tombstone* message. A tombstone has a null value and acts as a marker to inform consumers that the corresponding message for that key has been deleted. After some time, only the tombstone marker is retained. Assuming new messages continue to come in, the marker is retained for a duration specified by `log.cleaner.delete.retention.ms` to allow consumers enough time to recognize the deletion.

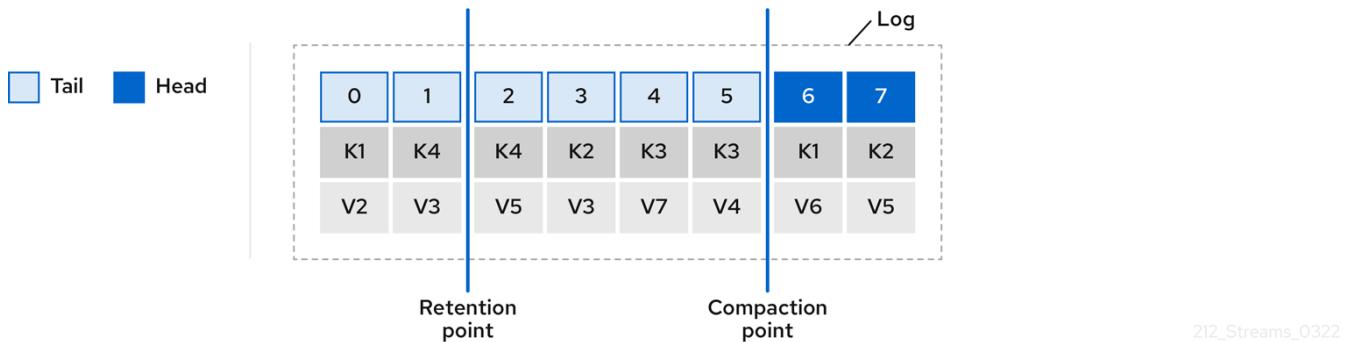
You can also set a time in milliseconds to put the cleaner on standby if there are no logs to clean:

```
# ...
log.cleaner.backoff.ms=15000
# ...
```

Using combined compact and delete policies

If you choose only a compact policy, your log can still become arbitrarily large. In such cases, you can set the cleanup policy for a topic to compact and delete logs. Kafka applies log compaction, removing older versions of records and retaining only the latest version of each key. Kafka also deletes records based on the specified time-based or size-based log retention settings.

For example, in the following diagram only the latest message (with the highest offset) for a specific message key is retained up to the compaction point. If there are any records remaining up to the retention point they are deleted. In this case, the compaction process would remove all duplicates.



2I2_Streams_0322

Figure 10. Log retention point and compaction point

31.3.7. Managing efficient disk utilization for compaction

When employing the compact policy and log cleaner to handle topic logs in Kafka, consider optimizing memory allocation.

You can fine-tune memory allocation using the deduplication property (`dedupe.buffer.size`), which determines the total memory allocated for cleanup tasks across all log cleaner threads. Additionally, you can establish a maximum memory usage limit by defining a percentage through the `buffer.load.factor` property.

```
# ...
log.cleaner.dedupe.buffer.size=134217728
log.cleaner.io.buffer.load.factor=0.9
# ...
```

Each log entry uses exactly 24 bytes, so you can work out how many log entries the buffer can handle in a single run and adjust the setting accordingly.

If possible, consider increasing the number of log cleaner threads if you are looking to reduce the log cleaning time:

```
# ...
log.cleaner.threads=8
# ...
```

If you are experiencing issues with 100% disk bandwidth usage, you can throttle the log cleaner I/O so that the sum of the read/write operations is less than a specified double value based on the capabilities of the disks performing the operations:

```
# ...
log.cleaner.io.max.bytes.per.second=1.7976931348623157E308
# ...
```

31.3.8. Controlling the log flush of message data

Generally, the recommendation is to not set explicit flush thresholds and let the operating system perform background flush using its default settings. Partition replication provides greater data durability than writes to any single disk, as a failed broker can recover from its in-sync replicas.

Log flush properties control the periodic writes of cached message data to disk. The scheduler specifies the frequency of checks on the log cache in milliseconds:

```
# ...
log.flush.scheduler.interval.ms=2000
# ...
```

You can control the frequency of the flush based on the maximum amount of time that a message is kept in-memory and the maximum number of messages in the log before writing to disk:

```
# ...
log.flush.interval.ms=50000
log.flush.interval.messages=100000
# ...
```

The wait between flushes includes the time to make the check and the specified interval before the flush is carried out. Increasing the frequency of flushes can affect throughput.

If you are using application flush management, setting lower flush thresholds might be appropriate if you are using faster disks.

31.3.9. Partition rebalancing for availability

Partitions can be replicated across brokers for fault tolerance. For a given partition, one broker is elected leader and handles all produce requests (writes to the log). Partition followers on other brokers replicate the partition data of the partition leader for data reliability in the event of the leader failing.

Followers do not normally serve clients, though [rack](#) configuration allows a consumer to consume messages from the closest replica when a Kafka cluster spans multiple datacenters. Followers operate only to replicate messages from the partition leader and allow recovery should the leader fail. Recovery requires an in-sync follower. Followers stay in sync by sending fetch requests to the leader, which returns messages to the follower in order. The follower is considered to be in sync if it has caught up with the most recently committed message on the leader. The leader checks this by looking at the last offset requested by the follower. An out-of-sync follower is usually not eligible as a leader should the current leader fail, unless [unclean leader election is allowed](#).

You can adjust the lag time before a follower is considered out of sync:

```
# ...
replica.lag.time.max.ms=30000
```

```
# ...
```

Lag time puts an upper limit on the time to replicate a message to all in-sync replicas and how long a producer has to wait for an acknowledgment. If a follower fails to make a fetch request and catch up with the latest message within the specified lag time, it is removed from in-sync replicas. You can reduce the lag time to detect failed replicas sooner, but by doing so you might increase the number of followers that fall out of sync needlessly. The right lag time value depends on both network latency and broker disk bandwidth.

When a leader partition is no longer available, one of the in-sync replicas is chosen as the new leader. The first broker in a partition's list of replicas is known as the *preferred* leader. By default, Kafka is enabled for automatic partition leader rebalancing based on a periodic check of leader distribution. That is, Kafka checks to see if the preferred leader is the *current* leader. A rebalance ensures that leaders are evenly distributed across brokers and brokers are not overloaded.

You can use Cruise Control for Strimzi to figure out replica assignments to brokers that balance load evenly across the cluster. Its calculation takes into account the differing load experienced by leaders and followers. A failed leader affects the balance of a Kafka cluster because the remaining brokers get the extra work of leading additional partitions.

For the assignment found by Cruise Control to actually be balanced it is necessary that partitions are lead by the preferred leader. Kafka can automatically ensure that the preferred leader is being used (where possible), changing the current leader if necessary. This ensures that the cluster remains in the balanced state found by Cruise Control.

You can control the frequency, in seconds, of the rebalance check and the maximum percentage of imbalance allowed for a broker before a rebalance is triggered.

```
#...
auto.leader.rebalance.enable=true
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
#...
```

The percentage leader imbalance for a broker is the ratio between the current number of partitions for which the broker is the current leader and the number of partitions for which it is the preferred leader. You can set the percentage to zero to ensure that preferred leaders are always elected, assuming they are in sync.

If the checks for rebalances need more control, you can disable automated rebalances. You can then choose when to trigger a rebalance using the [kafka-leader-election.sh](#) command line tool.

NOTE

The Grafana dashboards provided with Strimzi show metrics for under-replicated partitions and partitions that do not have an active leader.

31.3.10. Unclean leader election

Leader election to an in-sync replica is considered clean because it guarantees no loss of data. And this is what happens by default. But what if there is no in-sync replica to take on leadership? Perhaps the ISR (in-sync replica) only contained the leader when the leader's disk died. If a minimum number of in-sync replicas is not set, and there are no followers in sync with the partition leader when its hard drive fails irrevocably, data is already lost. Not only that, but *a new leader cannot be elected* because there are no in-sync followers.

You can configure how Kafka handles leader failure:

```
# ...
unclean.leader.election.enable=false
# ...
```

Unclean leader election is disabled by default, which means that out-of-sync replicas cannot become leaders. With clean leader election, if no other broker was in the ISR when the old leader was lost, Kafka waits until that leader is back online before messages can be written or read. Unclean leader election means out-of-sync replicas can become leaders, but you risk losing messages. The choice you make depends on whether your requirements favor availability or durability.

You can override the default configuration for specific topics at the topic level. If you cannot afford the risk of data loss, then leave the default configuration.

31.3.11. Avoiding unnecessary consumer group rebalances

For consumers joining a new consumer group, you can add a delay so that unnecessary rebalances to the broker are avoided:

```
# ...
group.initial.rebalance.delay.ms=3000
# ...
```

The delay is the amount of time that the coordinator waits for members to join. The longer the delay, the more likely it is that all the members will join in time and avoid a rebalance. But the delay also prevents the group from consuming until the period has ended.

31.4. Kafka producer configuration tuning

Use configuration properties to optimize the performance of Kafka producers. You can use standard Kafka producer configuration options. Adjusting your configuration to maximize throughput might increase latency or vice versa. You will need to experiment and tune your producer configuration to get the balance you need.

When configuring a producer, consider the following aspects carefully, as they significantly impact its performance and behavior:

Compression

By compressing messages before they are sent over the network, you can conserve network bandwidth and reduce disk storage requirements, but with the additional cost of increased CPU utilization due to the compression and decompression processes.

Batching

Adjusting the batch size and time intervals when the producer sends messages can affect throughput and latency.

Partitioning

Partitioning strategies in the Kafka cluster can support producers through parallelism and load balancing, whereby producers can write to multiple partitions concurrently and each partition receives an equal share of messages. Other strategies might include topic replication for fault tolerance.

Securing access

Implement security measures for authentication, encryption, and authorization by setting up user accounts to [manage secure access to Kafka](#).

31.4.1. Basic producer configuration

Connection and serializer properties are required for every producer. Generally, it is good practice to add a client id for tracking, and use compression on the producer to reduce batch sizes in requests.

In a basic producer configuration:

- The order of messages in a partition is not guaranteed.
- The acknowledgment of messages reaching the broker does not guarantee durability.

Basic producer configuration properties

```
# ...
bootstrap.servers=localhost:9092 ①
key.serializer=org.apache.kafka.common.serialization.StringSerializer ②
value.serializer=org.apache.kafka.common.serialization.StringSerializer ③
client.id=my-client ④
compression.type=gzip ⑤
# ...
```

① (Required) Tells the producer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The producer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a server is down, but it's not necessary to provide a list of all the brokers in the cluster.

② (Required) Serializer to transform the key of each message to bytes prior to them being sent to a broker.

③ (Required) Serializer to transform the value of each message to bytes prior to them being sent to a broker.

- ④ (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request.
- ⑤ (Optional) The codec for compressing messages, which are sent and might be stored in compressed format and then decompressed when reaching a consumer. Compression is useful for improving throughput and reducing the load on storage, but might not be suitable for low latency applications where the cost of compression or decompression could be prohibitive.

31.4.2. Data durability

Message delivery acknowledgments minimize the likelihood that messages are lost. By default, acknowledgments are enabled with the `acks` property set to `acks=all`. To control the maximum time the producer waits for acknowledgments from the broker and handle potential delays in sending messages, you can use the `delivery.timeout.ms` property.

Acknowledging message delivery

```
# ...
acks=all ①
delivery.timeout.ms=120000 ②
# ...
```

① `acks=all` forces a leader replica to replicate messages to a certain number of followers before acknowledging that the message request was successfully received.

② The maximum time in milliseconds to wait for a complete send request. You can set the value to `MAX_LONG` to delegate to Kafka an indefinite number of retries. The default is `120000` or 2 minutes.

The `acks=all` setting offers the strongest guarantee of delivery, but it will increase the latency between the producer sending a message and receiving acknowledgment. If you don't require such strong guarantees, a setting of `acks=0` or `acks=1` provides either no delivery guarantees or only acknowledgment that the leader replica has written the record to its log.

With `acks=all`, the leader waits for all in-sync replicas to acknowledge message delivery. A topic's `min.insync.replicas` configuration sets the minimum required number of in-sync replica acknowledgements. The number of acknowledgements include that of the leader and followers.

A typical starting point is to use the following configuration:

- Producer configuration:
 - `acks=all` (default)
- Broker configuration for topic replication:
 - `default.replication.factor=3` (default = 1)
 - `min.insync.replicas=2` (default = 1)

When you create a topic, you can override the default replication factor. You can also override `min.insync.replicas` at the topic level in the topic configuration.

Strimzi uses this configuration in the example configuration files for multi-node deployment of

Kafka.

The following table describes how this configuration operates depending on the availability of followers that replicate the leader replica.

Table 43. Follower availability

Number of followers available and in-sync	Acknowledgements	Producer can send messages?
2	The leader waits for 2 follower acknowledgements	Yes
1	The leader waits for 1 follower acknowledgement	Yes
0	The leader raises an exception	No

A topic replication factor of 3 creates one leader replica and two followers. In this configuration, the producer can continue if a single follower is unavailable. Some delay can occur whilst removing a failed broker from the in-sync replicas or a creating a new leader. If the second follower is also unavailable, message delivery will not be successful. Instead of acknowledging successful message delivery, the leader sends an error (*not enough replicas*) to the producer. The producer raises an equivalent exception. With `retries` configuration, the producer can resend the failed message request.

NOTE If the system fails, there is a risk of unsent data in the buffer being lost.

31.4.3. Ordered delivery

Idempotent producers avoid duplicates as messages are delivered exactly once. IDs and sequence numbers are assigned to messages to ensure the order of delivery, even in the event of failure. If you are using `acks=all` for data consistency, using idempotency makes sense for ordered delivery. Idempotency is enabled for producers by default. With idempotency enabled, you can set the number of concurrent in-flight requests to a maximum of 5 for message ordering to be preserved.

Ordered delivery with idempotency

```
# ...
enable.idempotence=true ①
max.in.flight.requests.per.connection=5 ②
acks=all ③
retries=2147483647 ④
# ...
```

① Set to `true` to enable the idempotent producer.

② With idempotent delivery the number of in-flight requests may be greater than 1 while still providing the message ordering guarantee. The default is 5 in-flight requests.

③ Set `acks` to `all`.

④ Set the number of attempts to resend a failed message request.

If you choose not to use `acks=all` and disable idempotency because of the performance cost, set the number of in-flight (unacknowledged) requests to 1 to preserve ordering. Otherwise, a situation is possible where *Message-A* fails only to succeed after *Message-B* was already written to the broker.

Ordered delivery without idempotency

```
# ...
enable.idempotence=false ①
max.in.flight.requests.per.connection=1 ②
retries=2147483647
# ...
```

① Set to `false` to disable the idempotent producer.

② Set the number of in-flight requests to exactly 1.

31.4.4. Reliability guarantees

Idempotence is useful for exactly once writes to a single partition. Transactions, when used with idempotence, allow exactly once writes across multiple partitions.

Transactions guarantee that messages using the same transactional ID are produced once, and either *all* are successfully written to the respective logs or *none* of them are.

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID ①
transaction.timeout.ms=900000 ②
# ...
```

① Specify a unique transactional ID.

② Set the maximum allowed time for transactions in milliseconds before a timeout error is returned. The default is `900000` or 15 minutes.

The choice of `transactional.id` is important in order that the transactional guarantee is maintained. Each transactional id should be used for a unique set of topic partitions. For example, this can be achieved using an external mapping of topic partition names to transactional ids, or by computing the transactional id from the topic partition names using a function that avoids collisions.

31.4.5. Optimizing producers for throughput and latency

Usually, the requirement of a system is to satisfy a particular throughput target for a proportion of messages within a given latency. For example, targeting 500,000 messages per second with 95% of messages being acknowledged within 2 seconds.

It's likely that the messaging semantics (message ordering and durability) of your producer are

defined by the requirements for your application. For instance, it's possible that you don't have the option of using `acks=0` or `acks=1` without breaking some important property or guarantee provided by your application.

Broker restarts have a significant impact on high percentile statistics. For example, over a long period the 99th percentile latency is dominated by behavior around broker restarts. This is worth considering when designing benchmarks or comparing performance numbers from benchmarking with performance numbers seen in production.

Depending on your objective, Kafka offers a number of configuration parameters and techniques for tuning producer performance for throughput and latency.

Message batching (`linger.ms` and `batch.size`)

Message batching delays sending messages in the hope that more messages destined for the same broker will be sent, allowing them to be batched into a single produce request. Batching is a compromise between higher latency in return for higher throughput. Time-based batching is configured using `linger.ms`, and size-based batching is configured using `batch.size`.

Compression (`compression.type`)

Message compression adds latency in the producer (CPU time spent compressing the messages), but makes requests (and potentially disk writes) smaller, which can increase throughput. Whether compression is worthwhile, and the best compression to use, will depend on the messages being sent. Compression happens on the thread which calls `KafkaProducer.send()`, so if the latency of this method matters for your application you should consider using more threads.

Pipelining (`max.in.flight.requests.per.connection`)

Pipelining means sending more requests before the response to a previous request has been received. In general more pipelining means better throughput, up to a threshold at which other effects, such as worse batching, start to counteract the effect on throughput.

Lowering latency

When your application calls the `KafkaProducer.send()` method, messages undergo a series of operations before being sent:

- Interception: Messages are processed by any configured interceptors.
- Serialization: Messages are serialized into the appropriate format.
- Partition assignment: Each message is assigned to a specific partition.
- Compression: Messages are compressed to conserve network bandwidth.
- Batching: Compressed messages are added to a batch in a partition-specific queue.

During these operations, the `send()` method is momentarily blocked. It also remains blocked if the `buffer.memory` is full or if metadata is unavailable.

Batches will remain in the queue until one of the following occurs:

- The batch is full (according to `batch.size`).
- The delay introduced by `linger.ms` has passed.

- The sender is ready to dispatch batches for other partitions to the same broker and can include this batch.
- The producer is being flushed or closed.

To minimize the impact of `send()` blocking on latency, optimize batching and buffering configurations. Use the `linger.ms` and `batch.size` properties to batch more messages into a single produce request for higher throughput.

```
# ...
linger.ms=100 ①
batch.size=16384 ②
buffer.memory=33554432 ③
# ...
```

- ① The `linger.ms` property adds a delay in milliseconds so that larger batches of messages are accumulated and sent in a request. The default is `0`.
- ② If a maximum `batch.size` in bytes is used, a request is sent when the maximum is reached, or messages have been queued for longer than `linger.ms` (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.
- ③ The buffer size must be at least as big as the batch size, and be able to accommodate buffering, compression, and in-flight requests.

Increasing throughput

You can improve throughput of your message requests by directing messages to a specified partition using a custom partitioner to replace the default.

```
# ...
partitioner.class=my-custom-partitioner ①
# ...
```

- ① Specify the class name of your custom partitioner.

31.5. Kafka consumer configuration tuning

Use configuration properties to optimize the performance of Kafka consumers. When tuning your consumers your primary concern will be ensuring that they cope efficiently with the amount of data ingested. As with the producer tuning, be prepared to make incremental changes until the consumers operate as expected.

When tuning a consumer, consider the following aspects carefully, as they significantly impact its performance and behavior:

Scaling

Consumer groups enable parallel processing of messages by distributing the load across multiple consumers, enhancing scalability and throughput. The number of topic partitions determines

the maximum level of parallelism that you can achieve, as one partition can only be assigned to one consumer in a consumer group.

Message ordering

If absolute ordering within a topic is important, use a single-partition topic. A consumer observes messages in a single partition in the same order that they were committed to the broker, which means that Kafka only provides ordering guarantees for messages in a single partition. It is also possible to maintain message ordering for events specific to individual entities, such as users. If a new entity is created, you can create a new topic dedicated to that entity. You can use a unique ID, like a user ID, as the message key and route all messages with the same key to a single partition within the topic.

Offset reset policy

Setting the appropriate offset policy ensures that the consumer consumes messages from the desired starting point and handles message processing accordingly. The default Kafka reset value is `latest`, which starts at the end of the partition, and consequently means some messages might be missed, depending on the consumer's behavior and the state of the partition. Setting `auto.offset.reset` to `earliest` ensures that when connecting with a new `group.id`, all messages are retrieved from the beginning of the log.

Securing access

Implement security measures for authentication, encryption, and authorization by setting up user accounts to [manage secure access to Kafka](#).

31.5.1. Basic consumer configuration

Connection and deserializer properties are required for every consumer. Generally, it is good practice to add a client id for tracking.

In a consumer configuration, irrespective of any subsequent configuration:

- The consumer fetches from a given offset and consumes the messages in order, unless the offset is changed to skip or re-read messages.
- The broker does not know if the consumer processed the responses, even when committing offsets to Kafka, because the offsets might be sent to a different broker in the cluster.

Basic consumer configuration properties

```
# ...
bootstrap.servers=localhost:9092 ①
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer ②
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer ③
client.id=my-client ④
group.id=my-group-id ⑤
# ...
```

① (Required) Tells the consumer to connect to a Kafka cluster using a *host:port* bootstrap server address for a Kafka broker. The consumer uses the address to discover and connect to all brokers in the cluster. Use a comma-separated list to specify two or three addresses in case a

server is down, but it is not necessary to provide a list of all the brokers in the cluster. If you are using a loadbalancer service to expose the Kafka cluster, you only need the address for the service because the availability is handled by the loadbalancer.

- ② (Required) Deserializer to transform the bytes fetched from the Kafka broker into message keys.
- ③ (Required) Deserializer to transform the bytes fetched from the Kafka broker into message values.
- ④ (Optional) The logical name for the client, which is used in logs and metrics to identify the source of a request. The id can also be used to throttle consumers based on processing time quotas.
- ⑤ (Conditional) A group id is *required* for a consumer to be able to join a consumer group.

31.5.2. Scaling data consumption using consumer groups

Consumer groups share a typically large data stream generated by one or multiple producers from a given topic. Consumers are grouped using a `group.id` property, allowing messages to be spread across the members. One of the consumers in the group is elected leader and decides how the partitions are assigned to the consumers in the group. Each partition can only be assigned to a single consumer.

If you do not already have as many consumers as partitions, you can scale data consumption by adding more consumer instances with the same `group.id`. Adding more consumers to a group than there are partitions will not help throughput, but it does mean that there are consumers on standby should one stop functioning. If you can meet throughput goals with fewer consumers, you save on resources.

Consumers within the same consumer group send offset commits and heartbeats to the same broker. The consumer sends heartbeats to the Kafka broker to indicate its activity within the consumer group. So the greater the number of consumers in the group, the higher the request load on the broker.

```
# ...
group.id=my-group-id ①
# ...
```

① Add a consumer to a consumer group using a group id.

31.5.3. Choosing the right partition assignment strategy

Select an appropriate partition assignment strategy, which determines how Kafka topic partitions are distributed among consumer instances in a group.

Partition strategies are supported by the following classes:

- `org.apache.kafka.clients.consumer.RangeAssignor`
- `org.apache.kafka.clients.consumer.RoundRobinAssignor`
- `org.apache.kafka.clients.consumer.StickyAssignor`

- `org.apache.kafka.clients.consumer.CooperativeStickyAssignor`

Specify a class using the `partition.assignment.strategy` consumer configuration property. The **range** assignment strategy assigns a range of partitions to each consumer, and is useful when you want to process related data together.

Alternatively, opt for a **round robin** assignment strategy for equal partition distribution among consumers, which is ideal for high-throughput scenarios requiring parallel processing.

For more stable partition assignments, consider the **sticky** and **cooperative sticky** strategies. Sticky strategies aim to maintain assigned partitions during rebalances, when possible. If a consumer was previously assigned certain partitions, the sticky strategies prioritize retaining those same partitions with the same consumer after a rebalance, while only revoking and reassigned the partitions that are actually moved to another consumer. Leaving partition assignments in place reduces the overhead on partition movements. The cooperative sticky strategy also supports cooperative rebalances, enabling uninterrupted consumption from partitions that are not reassigned.

If none of the available strategies suit your data, you can create a custom strategy tailored to your specific requirements.

31.5.4. Message ordering guarantees

Kafka brokers receive fetch requests from consumers that ask the broker to send messages from a list of topics, partitions and offset positions.

A consumer observes messages in a single partition in the same order that they were committed to the broker, which means that Kafka **only** provides ordering guarantees for messages in a single partition. Conversely, if a consumer is consuming messages from multiple partitions, the order of messages in different partitions as observed by the consumer does not necessarily reflect the order in which they were sent.

If you want a strict ordering of messages from one topic, use one partition per consumer.

31.5.5. Optimizing consumers for throughput and latency

Control the number of messages returned when your client application calls `KafkaConsumer.poll()`.

Use the `fetch.max.wait.ms` and `fetch.min.bytes` properties to increase the minimum amount of data fetched by the consumer from the Kafka broker. Time-based batching is configured using `fetch.max.wait.ms`, and size-based batching is configured using `fetch.min.bytes`.

If CPU utilization in the consumer or broker is high, it might be because there are too many requests from the consumer. You can adjust `fetch.max.wait.ms` and `fetch.min.bytes` properties higher so that there are fewer requests and messages are delivered in bigger batches. By adjusting higher, throughput is improved with some cost to latency. You can also adjust higher if the amount of data being produced is low.

For example, if you set `fetch.max.wait.ms` to 500ms and `fetch.min.bytes` to 16384 bytes, when Kafka receives a fetch request from the consumer it will respond when the first of either threshold is

reached.

Conversely, you can adjust the `fetch.max.wait.ms` and `fetch.min.bytes` properties lower to improve end-to-end latency.

```
# ...
fetch.max.wait.ms=500 ①
fetch.min.bytes=16384 ②
# ...
```

① The maximum time in milliseconds the broker will wait before completing fetch requests. The default is `500` milliseconds.

② If a minimum batch size in bytes is used, a request is sent when the minimum is reached, or messages have been queued for longer than `fetch.max.wait.ms` (whichever comes sooner). Adding the delay allows batches to accumulate messages up to the batch size.

Lowering latency by increasing the fetch request size

Use the `fetch.max.bytes` and `max.partition.fetch.bytes` properties to increase the maximum amount of data fetched by the consumer from the Kafka broker.

The `fetch.max.bytes` property sets a maximum limit in bytes on the amount of data fetched from the broker at one time.

The `max.partition.fetch.bytes` sets a maximum limit in bytes on how much data is returned for each partition, which must always be larger than the number of bytes set in the broker or topic configuration for `max.message.bytes`.

The maximum amount of memory a client can consume is calculated approximately as:

```
NUMBER-OF-BROKERS * fetch.max.bytes and NUMBER-OF-PARTITIONS *
max.partition.fetch.bytes
```

If memory usage can accommodate it, you can increase the values of these two properties. By allowing more data in each request, latency is improved as there are fewer fetch requests.

```
# ...
fetch.max.bytes=52428800 ①
max.partition.fetch.bytes=1048576 ②
# ...
```

① The maximum amount of data in bytes returned for a fetch request.

② The maximum amount of data in bytes returned for each partition.

31.5.6. Avoiding data loss or duplication when committing offsets

The Kafka *auto-commit mechanism* allows a consumer to commit the offsets of messages automatically. If enabled, the consumer will commit offsets received from polling the broker at

5000ms intervals.

The auto-commit mechanism is convenient, but it introduces a risk of data loss and duplication. If a consumer has fetched and transformed a number of messages, but the system crashes with processed messages in the consumer buffer when performing an auto-commit, that data is lost. If the system crashes after processing the messages, but before performing the auto-commit, the data is duplicated on another consumer instance after rebalancing.

Auto-committing can avoid data loss only when all messages are processed before the next poll to the broker, or the consumer closes.

To minimize the likelihood of data loss or duplication, you can set `enable.auto.commit` to `false` and develop your client application to have more control over committing offsets. Or you can use `auto.commit.interval.ms` to decrease the intervals between commits.

```
# ...
enable.auto.commit=false ①
# ...
```

① Auto commit is set to false to provide more control over committing offsets.

By setting to `enable.auto.commit` to `false`, you can commit offsets after **all** processing has been performed and the message has been consumed. For example, you can set up your application to call the Kafka `commitSync` and `commitAsync` commit APIs.

The `commitSync` API commits the offsets in a message batch returned from polling. You call the API when you are finished processing all the messages in the batch. If you use the `commitSync` API, the application will not poll for new messages until the last offset in the batch is committed. If this negatively affects throughput, you can commit less frequently, or you can use the `commitAsync` API. The `commitAsync` API does not wait for the broker to respond to a commit request, but risks creating more duplicates when rebalancing. A common approach is to combine both commit APIs in an application, with the `commitSync` API used just before shutting the consumer down or rebalancing to make sure the final commit is successful.

Controlling transactional messages

Consider using transactional ids and enabling idempotence (`enable.idempotence=true`) on the producer side to guarantee exactly-once delivery. On the consumer side, you can then use the `isolation.level` property to control how transactional messages are read by the consumer.

The `isolation.level` property has two valid values:

- `read_committed`
- `read_uncommitted` (default)

Use `read_committed` to ensure that only transactional messages that have been committed are read by the consumer. However, this will cause an increase in end-to-end latency, because the consumer will not be able to return a message until the brokers have written the transaction markers that record the result of the transaction (*committed* or *aborted*).

```
# ...
enable.auto.commit=false
isolation.level=read_committed ①
# ...
```

① Set to `read_committed` so that only committed messages are read by the consumer.

31.5.7. Recovering from failure to avoid data loss

In the event of failures within a consumer group, Kafka provides a rebalance protocol designed for effective detection and recovery. To minimize the potential impact of these failures, one key strategy is to adjust the `max.poll.records` property to balance efficient processing with system stability. This property determines the maximum number of records a consumer can fetch in a single poll. Fine-tuning `max.poll.records` helps to maintain a controlled consumption rate, preventing the consumer from overwhelming itself or the Kafka broker.

Additionally, Kafka offers advanced configuration properties like `session.timeout.ms` and `heartbeat.interval.ms`. These settings are typically reserved for more specialized use cases and may not require adjustment in standard scenarios.

The `session.timeout.ms` property specifies the maximum amount of time a consumer can go without sending a heartbeat to the Kafka broker to indicate it is active within the consumer group. If a consumer fails to send a heartbeat within the session timeout, it is considered inactive. A consumer marked as inactive triggers a rebalancing of the partitions for the topic. Setting the `session.timeout.ms` property value too low can result in false-positive outcomes, while setting it too high can lead to delayed recovery from failures.

The `heartbeat.interval.ms` property determines how frequently a consumer sends heartbeats to the Kafka broker. A shorter interval between consecutive heartbeats allows for quicker detection of consumer failures. The heartbeat interval must be lower, usually by a third, than the session timeout. Decreasing the heartbeat interval reduces the chance of accidental rebalancing, but more frequent heartbeats increases the overhead on broker resources.

31.5.8. Managing offset policy

Use the `auto.offset.reset` property to control how a consumer behaves when no offsets have been committed, or a committed offset is no longer valid or deleted.

Suppose you deploy a consumer application for the first time, and it reads messages from an existing topic. Because this is the first time the `group.id` is used, the `__consumer_offsets` topic does not contain any offset information for this application. The new application can start processing all existing messages from the start of the log or only new messages. The default reset value is `latest`, which starts at the end of the partition, and consequently means some messages are missed. To avoid data loss, but increase the amount of processing, set `auto.offset.reset` to `earliest` to start at the beginning of the partition.

Also consider using the `earliest` option to avoid messages being lost when the offsets retention period (`offsets.retention.minutes`) configured for a broker has ended. If a consumer group or standalone consumer is inactive and commits no offsets during the retention period, previously

committed offsets are deleted from `__consumer_offsets`.

```
# ...
heartbeat.interval.ms=3000 ①
session.timeout.ms=45000 ②
auto.offset.reset=earliest ③
# ...
```

- ① Adjust the heartbeat interval lower according to anticipated rebalances.
- ② If no heartbeats are received by the Kafka broker before the timeout duration expires, the consumer is removed from the consumer group and a rebalance is initiated. If the broker configuration has a `group.min.session.timeout.ms` and `group.max.session.timeout.ms`, the session timeout value must be within that range.
- ③ Set to `earliest` to return to the start of a partition and avoid data loss if offsets were not committed.

If the amount of data returned in a single fetch request is large, a timeout might occur before the consumer has processed it. In this case, you can lower `max.partition.fetch.bytes` or increase `session.timeout.ms`.

31.5.9. Minimizing the impact of rebalances

The rebalancing of a partition between active consumers in a group is the time it takes for the following to take place:

- Consumers to commit their offsets
- The new consumer group to be formed
- The group leader to assign partitions to group members
- The consumers in the group to receive their assignments and start fetching

The rebalancing process can increase the downtime of a service, particularly if it happens repeatedly during a rolling restart of a consumer group cluster.

In this situation, you can introduce *static membership* by assigning a unique identifier (`group.instance.id`) to each consumer instance within the group. Static membership uses persistence so that a consumer instance is recognized during a restart after a session timeout. Consequently, the consumer maintains its assignment of topic partitions, reducing unnecessary rebalancing when it rejoins the group after a failure or restart.

Additionally, adjusting the `max.poll.interval.ms` configuration can prevent rebalances caused by prolonged processing tasks, allowing you to specify the maximum interval between polls for new messages. Use the `max.poll.records` property to cap the number of records returned from the consumer buffer during each poll. Reducing the number of records allows the consumer to process fewer messages more efficiently. In cases where lengthy message processing is unavoidable, consider offloading such tasks to a pool of worker threads. This parallel processing approach prevents delays and potential rebalances caused by overwhelming the consumer with a large volume of records.

```
# ...
group.instance.id=UNIQUE-ID ①
max.poll.interval.ms=300000 ②
max.poll.records=500 ③
# ...
```

- ① The unique instance id ensures that a new consumer instance receives the same assignment of topic partitions.
- ② Set the interval to check the consumer is continuing to process messages.
- ③ Sets the number of processed records returned from the consumer.

31.6. Handling high volumes of messages

If your Strimzi deployment needs to handle a high volume of messages, you can use configuration options to optimize for throughput and latency.

Producer and consumer configuration can help control the size and frequency of requests to Kafka brokers. For more information on the configuration options, see the following:

- [Apache Kafka configuration documentation for producers](#)
- [Apache Kafka configuration documentation for consumers](#)

You can also use the same configuration options with the producers and consumers used by the Kafka Connect runtime source connectors (including MirrorMaker 2) and sink connectors.

Source connectors

- Producers from the Kafka Connect runtime send messages to the Kafka cluster.
- For MirrorMaker 2, since the source system is Kafka, consumers retrieve messages from a source Kafka cluster.

Sink connectors

- Consumers from the Kafka Connect runtime retrieve messages from the Kafka cluster.

For consumers, you might increase the amount of data fetched in a single fetch request to reduce latency. You increase the fetch request size using the `fetch.max.bytes` and `max.partition.fetch.bytes` properties. You can also set a maximum limit on the number of messages returned from the consumer buffer using the `max.poll.records` property.

For MirrorMaker 2, configure the `fetch.max.bytes`, `max.partition.fetch.bytes`, and `max.poll.records` values at the source connector level (`consumer.*`), as they relate to the specific consumer that fetches messages from the source.

For producers, you might increase the size of the message batches sent in a single produce request. You increase the batch size using the `batch.size` property. A larger batch size reduces the number of outstanding messages ready to be sent and the size of the backlog in the message queue. Messages being sent to the same partition are batched together. A produce request is sent to the target cluster when the batch size is reached. By increasing the batch size, produce requests are delayed and

more messages are added to the batch and sent to brokers at the same time. This can improve throughput when you have just a few topic partitions that handle large numbers of messages.

Consider the number and size of the records that the producer handles for a suitable producer batch size.

Use `linger.ms` to add a wait time in milliseconds to delay produce requests when producer load decreases. The delay means that more records can be added to batches if they are under the maximum batch size.

Configure the `batch.size` and `linger.ms` values at the source connector level (`producer.override.*`), as they relate to the specific producer that sends messages to the target Kafka cluster.

For Kafka Connect source connectors, the data streaming pipeline to the target Kafka cluster is as follows:

Data streaming pipeline for Kafka Connect source connector

external data source → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

For Kafka Connect sink connectors, the data streaming pipeline to the target external data source is as follows:

Data streaming pipeline for Kafka Connect sink connector

source Kafka topic → (Kafka Connect tasks) sink message queue → consumer buffer → external data source

For MirrorMaker 2, the data mirroring pipeline to the target Kafka cluster is as follows:

Data mirroring pipeline for MirrorMaker 2

source Kafka topic → (Kafka Connect tasks) source message queue → producer buffer → target Kafka topic

The producer sends messages in its buffer to topics in the target Kafka cluster. While this is happening, Kafka Connect tasks continue to poll the data source to add messages to the source message queue.

The size of the producer buffer for the source connector is set using the `producer.override.buffer.memory` property. Tasks wait for a specified timeout period (`offset.flush.timeout.ms`) before the buffer is flushed. This should be enough time for the sent messages to be acknowledged by the brokers and offset data committed. The source task does not wait for the producer to empty the message queue before committing offsets, except during shutdown.

If the producer is unable to keep up with the throughput of messages in the source message queue, buffering is blocked until there is space available in the buffer within a time period bounded by `max.block.ms`. Any unacknowledged messages still in the buffer are sent during this period. New messages are not added to the buffer until these messages are acknowledged and flushed.

You can try the following configuration changes to keep the underlying source message queue of

outstanding messages at a manageable size:

- Increasing the default value in milliseconds of the `offset.flush.timeout.ms`
- Ensuring that there are enough CPU and memory resources
- Increasing the number of tasks that run in parallel by doing the following:
 - Increasing the number of tasks that run in parallel using the `tasksMax` property
 - Increasing the number of worker nodes that run tasks using the `replicas` property

Consider the number of tasks that can run in parallel according to the available CPU and memory resources and number of worker nodes. You might need to keep adjusting the configuration values until they have the desired effect.

31.6.1. Configuring Kafka Connect for high-volume messages

Kafka Connect fetches data from the source external data system and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows configuration for Kafka Connect using the `KafkaConnect` custom resource.

Example Kafka Connect configuration for handling high volumes of messages

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  replicas: 3
  config:
    offset.flush.timeout.ms: 10000
    # ...
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  # ...
```

Producer configuration is added for the source connector, which is managed using the `KafkaConnector` custom resource.

Example source connector configuration for handling high volumes of messages

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
```

```
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
    producer.override.batch.size: 327680
    producer.override.linger.ms: 100
    # ...
```

NOTE [FileStreamSourceConnector](#) and [FileStreamSinkConnector](#) are provided as example connectors. For information on deploying them as [KafkaConnector](#) resources, see [Deploying KafkaConnector resources](#).

Consumer configuration is added for the sink connector.

Example sink connector configuration for handling high volumes of messages

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector
  tasksMax: 2
  config:
    consumer.fetch.max.bytes: 52428800
    consumer.max.partition.fetch.bytes: 1048576
    consumer.max.poll.records: 500
    # ...
```

If you are using the Kafka Connect API instead of the [KafkaConnector](#) custom resource to manage your connectors, you can add the connector configuration as a JSON object.

Example curl request to add source connector configuration for handling high volumes of messages

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
  "config": {
    "connector.class": "org.apache.kafka.connect.file.FileStreamSourceConnector",
    "file": "/opt/kafka/LICENSE",
    "topic": "my-topic",
```

```

    "tasksMax": "4",
    "type": "source"
    "producer.override.batch.size": 327680
    "producer.override.linger.ms": 100
}
}'

```

31.6.2. Configuring MirrorMaker 2 for high-volume messages

MirrorMaker 2 fetches data from the source cluster and hands it to the Kafka Connect runtime producers so that it's replicated to the target cluster.

The following example shows the configuration for MirrorMaker 2 using the [KafkaMirrorMaker2](#) custom resource.

Example MirrorMaker 2 configuration for handling high volumes of messages

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.7.1
  replicas: 1
  connectCluster: "my-cluster-target"
  clusters:
    - alias: "my-cluster-source"
      bootstrapServers: my-cluster-source-kafka-bootstrap:9092
    - alias: "my-cluster-target"
      config:
        offset.flush.timeout.ms: 10000
      bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
    - sourceCluster: "my-cluster-source"
      targetCluster: "my-cluster-target"
      sourceConnector:
        tasksMax: 2
        config:
          producer.override.batch.size: 327680
          producer.override.linger.ms: 100
          consumer.fetch.max.bytes: 52428800
          consumer.max.partition.fetch.bytes: 1048576
          consumer.max.poll.records: 500
        # ...
  resources:
    requests:
      cpu: "1"
      memory: Gi
    limits:
      cpu: "2"

```

31.6.3. Checking the MirrorMaker 2 message flow

If you are using Prometheus and Grafana to monitor your deployment, you can check the MirrorMaker 2 message flow.

The example MirrorMaker 2 Grafana dashboards provided with Strimzi show the following metrics related to the flush pipeline.

- The number of messages in Kafka Connect’s outstanding messages queue
- The available bytes of the producer buffer
- The offset commit timeout in milliseconds

You can use these metrics to gauge whether or not you need to tune your configuration based on the volume of messages.

Additional resources

- [Introducing metrics](#)
- [Adding Kafka Connect connectors](#)

31.7. Handling large message sizes

Kafka’s default batch size for messages is 1MB, which is optimal for maximum throughput in most use cases. Kafka can accommodate larger batches at a reduced throughput, assuming adequate disk capacity.

Large message sizes can be handled in four ways:

1. Brokers, producers, and consumers are configured to accommodate larger message sizes.
2. [Producer-side message compression](#) writes compressed messages to the log.
3. Reference-based messaging sends only a reference to data stored in some other system in the message’s payload.
4. Inline messaging splits messages into chunks that use the same key, which are then combined on output using a stream-processor like Kafka Streams.

Unless you are handling very large messages, the configuration approach is recommended. The reference-based messaging and message compression options cover most other situations. With any of these options, care must be taken to avoid introducing performance issues.

31.7.1. Configuring Kafka components to handle larger messages

Large messages can impact system performance and introduce complexities in message processing. If they cannot be avoided, there are configuration options available. To handle larger messages efficiently and prevent blocks in the message flow, consider adjusting the following configurations:

- Adjusting the maximum record batch size:
 - Set `message.max.bytes` at the broker level to support larger record batch sizes for all topics.
 - Set `max.message.bytes` at the topic level to support larger record batch sizes for individual topics.
- Increasing the maximum size of messages fetched by each partition follower (`replica.fetch.max.bytes`).
- Increasing the batch size (`batch.size`) for producers to increase the size of message batches sent in a single produce request.
- Configuring a higher maximum request size for producers (`max.request.size`) and consumers (`fetch.max.bytes`) to accommodate larger record batches.
- Setting a higher maximum limit (`max.partition.fetch.bytes`) on how much data is returned to consumers for each partition.

Ensure that the maximum size for batch requests is at least as large as `message.max.bytes` to accommodate the largest record batch size.

Example broker configuration

```
message.max.bytes: 10000000
replica.fetch.max.bytes: 10485760
```

Example producer configuration

```
batch.size: 327680
max.request.size: 10000000
```

Example consumer configuration

```
fetch.max.bytes: 10000000
max.partition.fetch.bytes: 10485760
```

It's also possible to configure the producers and consumers used by other Kafka components like Kafka Bridge, Kafka Connect, and MirrorMaker 2 to handle larger messages more effectively.

Kafka Bridge

Configure the Kafka Bridge using specific producer and consumer configuration properties:

- `producer.config` for producers
- `consumer.config` for consumers

Example Kafka Bridge configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
```

```
spec:  
  # ...  
  producer:  
    config:  
      batch.size: 327680  
      max.request.size: 10000000  
  consumer:  
    config:  
      fetch.max.bytes: 10000000  
      max.partition.fetch.bytes: 10485760  
    # ...
```

Kafka Connect

For Kafka Connect, configure the source and sink connectors responsible for sending and retrieving messages using prefixes for producer and consumer configuration properties:

- **producer.override** for the producer used by the source connector to send messages to a Kafka cluster
- **consumer** for the consumer used by the sink connector to retrieve messages from a Kafka cluster

Example Kafka Connect source connector configuration

```
apiVersion: kafka.strimzi.io/v1beta2  
kind: KafkaConnector  
metadata:  
  name: my-source-connector  
  labels:  
    strimzi.io/cluster: my-connect-cluster  
spec:  
  # ...  
  config:  
    producer.override.batch.size: 327680  
    producer.override.max.request.size: 10000000  
  # ...
```

Example Kafka Connect sink connector configuration

```
apiVersion: kafka.strimzi.io/v1beta2  
kind: KafkaConnector  
metadata:  
  name: my-sink-connector  
  labels:  
    strimzi.io/cluster: my-connect-cluster  
spec:  
  # ...  
  config:  
    consumer.fetch.max.bytes: 10000000  
    consumer.max.partition.fetch.bytes: 10485760
```

```
# ...
```

MirrorMaker 2

For MirrorMaker 2, configure the source connector responsible for retrieving messages from the source Kafka cluster using prefixes for producer and consumer configuration properties:

- `producer.override` for the runtime Kafka Connect producer used to replicate data to the target Kafka cluster
- `consumer` for the consumer used by the sink connector to retrieve messages from the source Kafka cluster

Example MirrorMaker 2 source connector configuration

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
    - sourceCluster: "my-cluster-source"
      targetCluster: "my-cluster-target"
      sourceConnector:
        tasksMax: 2
        config:
          producer.override.batch.size: 327680
          producer.override.max.request.size: 10000000
          consumer.fetch.max.bytes: 10000000
          consumer.max.partition.fetch.bytes: 10485760
          # ...
```

31.7.2. Producer-side compression

For producer configuration, you specify a `compression.type`, such as Gzip, which is then applied to batches of data generated by the producer. Using the broker configuration `compression.type=producer` (default), the broker retains whatever compression the producer used. Whenever producer and topic compression do not match, the broker has to compress batches again prior to appending them to the log, which impacts broker performance.

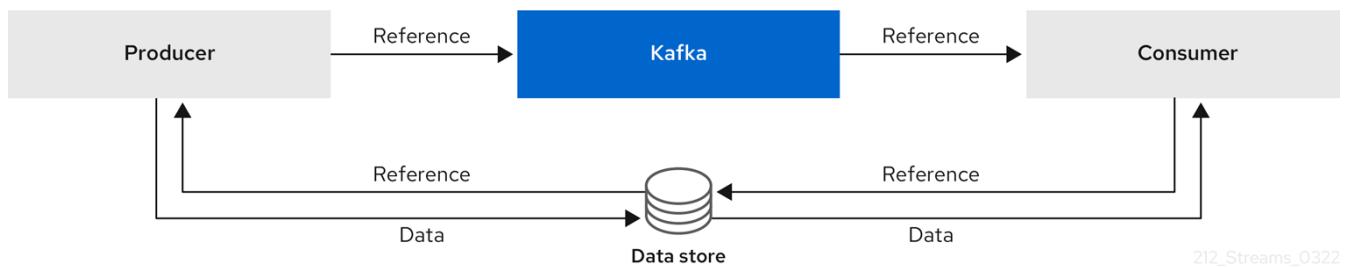
Compression also adds additional processing overhead on the producer and decompression overhead on the consumer, but includes more data in a batch, so is often beneficial to throughput when message data compresses well.

Combine producer-side compression with fine-tuning of the batch size to facilitate optimum throughput. Using metrics helps to gauge the average batch size needed.

31.7.3. Reference-based messaging

Reference-based messaging is useful for data replication when you do not know how big a message will be. The external data store must be fast, durable, and highly available for this configuration to work. Data is written to the data store and a reference to the data is returned. The producer sends a message containing the reference to Kafka. The consumer gets the reference from the message and uses it to fetch the data from the data store.

31.7.4. Reference-based messaging flow



As the message passing requires more trips, end-to-end latency will increase. Another significant drawback of this approach is there is no automatic clean up of the data in the external system when the Kafka message gets cleaned up. A hybrid approach would be to only send large messages to the data store and process standard-sized messages directly.

Inline messaging

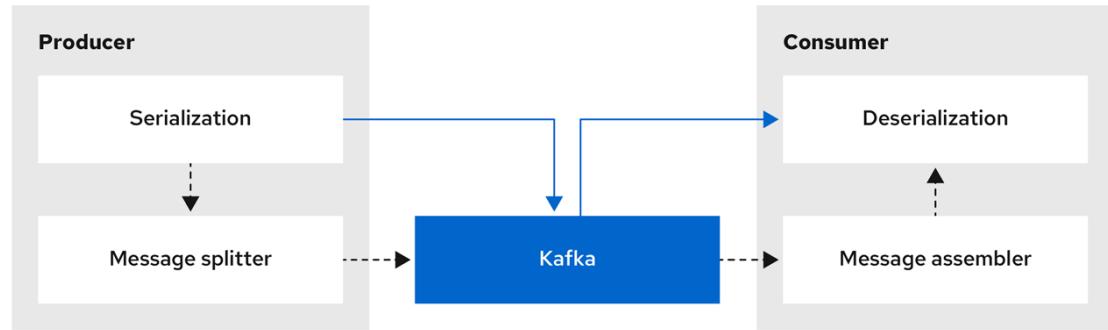
Inline messaging is complex, but it does not have the overhead of depending on external systems like reference-based messaging.

The producing client application has to serialize and then chunk the data if the message is too big. The producer then uses the Kafka `ByteArraySerializer` or similar to serialize each chunk again before sending it. The consumer tracks messages and buffers chunks until it has a complete message. The consuming client application receives the chunks, which are assembled before deserialization. Complete messages are delivered to the rest of the consuming application in order according to the offset of the first or last chunk for each set of chunked messages.

The consumer should commit its offset only after receiving and processing all message chunks to ensure accurate tracking of message delivery and prevent duplicates during rebalancing. Chunks might be spread across segments. Consumer-side handling should cover the possibility that a chunk becomes unavailable if a segment is subsequently deleted.

→ Standard message

→ Large message



2I2_Streams_0322

Figure 11. Inline messaging flow

Inline messaging has a performance overhead on the consumer side because of the buffering required, particularly when handling a series of large messages in parallel. The chunks of large messages can become interleaved, so that it is not always possible to commit when all the chunks of a message have been consumed if the chunks of another large message in the buffer are incomplete. For this reason, the buffering is usually supported by persisting message chunks or by implementing commit logic.