



Red Hat AMQ Streams 2.6

Using AMQ Streams on RHEL

Configure and manage a deployment of AMQ Streams 2.6 on Red Hat Enterprise Linux

Red Hat AMQ Streams 2.6 Using AMQ Streams on RHEL

Configure and manage a deployment of AMQ Streams 2.6 on Red Hat Enterprise Linux

Legal Notice

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

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

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE	7
CHAPTER 1. OVERVIEW OF AMQ STREAMS	8
1.1. USING THE KAFKA BRIDGE TO CONNECT WITH A KAFKA CLUSTER	9
1.2. DOCUMENT CONVENTIONS	9
CHAPTER 2. FIPS SUPPORT	10
2.1. INSTALLING AMQ STREAMS WITH FIPS MODE ENABLED	10
CHAPTER 3. GETTING STARTED	11
3.1. INSTALLATION ENVIRONMENT	11
3.1.1. Supported environment versions	11
3.1.2. Data storage considerations	11
3.1.3. File systems	12
3.1.4. Apache Kafka and ZooKeeper storage	12
3.2. DOWNLOADING AMQ STREAMS	12
3.3. INSTALLING KAFKA	12
3.4. RUNNING A SINGLE-NODE KAFKA CLUSTER	14
3.5. SENDING AND RECEIVING MESSAGES FROM A TOPIC	15
3.6. STOPPING THE AMQ STREAMS SERVICES	16
CHAPTER 4. RUNNING A MULTI-NODE ENVIRONMENT	18
4.1. RUNNING A MULTI-NODE ZOOKEEPER CLUSTER	18
4.2. RUNNING A MULTI-NODE KAFKA CLUSTER	19
4.3. PERFORMING A GRACEFUL ROLLING RESTART OF KAFKA BROKERS	21
CHAPTER 5. RUNNING KAFKA IN KRAFT MODE (DEVELOPMENT PREVIEW)	25
5.1. USING AMQ STREAMS WITH KAFKA IN KRAFT MODE	25
5.2. RUNNING A KAFKA CLUSTER IN KRAFT MODE	26
CHAPTER 6. CONFIGURING AN AMQ STREAMS ON RHEL DEPLOYMENT	28
6.1. USING STANDARD KAFKA CONFIGURATION PROPERTIES	28
6.2. LOADING CONFIGURATION VALUES FROM ENVIRONMENT VARIABLES	29
6.3. CONFIGURING ZOOKEEPER	30
6.3.1. Basic configuration	30
6.3.2. ZooKeeper cluster configuration	30
6.3.3. Authentication	32
6.3.3.1. Authentication with SASL	32
6.3.3.2. Enabling server-to-server authentication using DIGEST-MD5	34
6.3.3.3. Enabling Client-to-server authentication using DIGEST-MD5	35
6.3.4. Authorization	36
6.3.5. TLS	36
6.3.6. Additional configuration options	36
6.4. CONFIGURING KAFKA	37
6.4.1. ZooKeeper	37
6.4.2. Listeners	37
6.4.3. Commit logs	39
6.4.4. Broker ID	39
6.4.5. ZooKeeper authentication	39
6.4.5.1. JAAS Configuration	39
6.4.5.2. Enabling ZooKeeper authentication	40
6.4.6. Authorization	40
6.4.6.1. Simple ACL authorizer	41

6.4.6.1.1. ACL rules	41
6.4.6.1.2. Principals	41
6.4.6.1.3. Authentication of users	42
6.4.6.1.4. Super users	42
6.4.6.1.5. Replica broker authentication	42
6.4.6.1.6. Supported resources	42
6.4.6.1.7. Supported operations	42
6.4.6.1.8. ACL management options	43
6.4.6.2. Enabling authorization	47
6.4.6.3. Adding ACL rules	47
6.4.6.4. Listing ACL rules	48
6.4.6.5. Removing ACL rules	48
6.4.7. ZooKeeper authorization	49
6.4.7.1. ACL Configuration	49
6.4.7.2. Enabling ZooKeeper ACLs for a new Kafka cluster	50
6.4.7.3. Enabling ZooKeeper ACLs in an existing Kafka cluster	50
6.4.8. Encryption and authentication	51
6.4.8.1. Listener configuration	51
6.4.8.2. TLS Encryption	52
6.4.8.3. Enabling TLS encryption	53
6.4.8.4. Authentication	53
6.4.8.4.1. TLS client authentication	53
6.4.8.4.2. SASL authentication	54
6.4.8.5. Enabling TLS client authentication	57
6.4.8.6. Enabling SASL PLAIN authentication	58
6.4.8.7. Enabling SASL SCRAM authentication	58
6.4.8.8. Adding SASL SCRAM users	59
6.4.8.9. Deleting SASL SCRAM users	60
6.4.9. Using OAuth 2.0 token-based authentication	60
6.4.9.1. OAuth 2.0 authentication mechanisms	61
6.4.9.1.1. Configuring OAuth 2.0 with properties or variables	63
6.4.9.2. OAuth 2.0 Kafka broker configuration	63
6.4.9.2.1. OAuth 2.0 client configuration on an authorization server	63
6.4.9.2.2. OAuth 2.0 authentication configuration in the Kafka cluster	64
6.4.9.2.3. Fast local JWT token validation configuration	68
6.4.9.2.4. OAuth 2.0 introspection endpoint configuration	70
6.4.9.3. Session re-authentication for Kafka brokers	70
6.4.9.4. OAuth 2.0 Kafka client configuration	72
6.4.9.5. OAuth 2.0 client authentication flows	72
6.4.9.5.1. Example client authentication flows using the SASL OAUTHBEARER mechanism	73
6.4.9.5.2. Example client authentication flows using the SASL PLAIN mechanism	75
6.4.9.6. Configuring OAuth 2.0 authentication	76
6.4.9.6.1. Configuring Red Hat Single Sign-On as an OAuth 2.0 authorization server	76
6.4.9.6.2. Configuring OAuth 2.0 support for Kafka brokers	78
6.4.9.6.3. Configuring Kafka Java clients to use OAuth 2.0	82
6.4.10. Using OAuth 2.0 token-based authorization	87
6.4.10.1. OAuth 2.0 authorization mechanism	87
6.4.10.1.1. Kafka broker custom authorizer	87
6.4.10.2. Configuring OAuth 2.0 authorization support	87
6.4.11. Using OPA policy-based authorization	91
6.4.11.1. Defining OPA policies	91
6.4.11.2. Connecting to the OPA	91
6.4.11.3. Configuring OPA authorization support	92

CHAPTER 7. CREATING AND MANAGING TOPICS	94
7.1. PARTITIONS AND REPLICAS	94
7.2. MESSAGE RETENTION	94
7.3. TOPIC AUTO-CREATION	95
7.4. TOPIC DELETION	95
7.5. TOPIC CONFIGURATION	95
7.6. INTERNAL TOPICS	96
7.7. CREATING A TOPIC	97
7.8. LISTING AND DESCRIBING TOPICS	98
7.9. MODIFYING A TOPIC CONFIGURATION	98
7.10. DELETING A TOPIC	100
CHAPTER 8. USING AMQ STREAMS WITH KAFKA CONNECT	101
8.1. USING KAFKA CONNECT IN STANDALONE MODE	101
8.1.1. Configuring Kafka Connect in standalone mode	101
8.1.2. Running Kafka Connect in standalone mode	102
8.2. USING KAFKA CONNECT IN DISTRIBUTED MODE	102
8.2.1. Configuring Kafka Connect in distributed mode	102
8.2.2. Running Kafka Connect in distributed mode	103
8.3. MANAGING CONNECTORS	104
8.3.1. Limiting access to the Kafka Connect API	104
8.3.2. Configuring connectors	104
8.3.2.1. Using the Kafka Connect REST API to manage connectors	105
8.3.2.2. Specifying connector configuration properties	106
8.3.3. Creating connectors using the Kafka Connect API	107
8.3.4. Deleting connectors using the Kafka Connect API	107
8.3.5. Adding connector plugins	108
CHAPTER 9. USING AMQ STREAMS WITH MIRRORMAKER 2	110
9.1. CONFIGURING ACTIVE/ACTIVE OR ACTIVE/PASSIVE MODES	110
9.1.1. Bidirectional replication (active/active)	110
9.1.2. Unidirectional replication (active/passive)	111
9.2. CONFIGURING MIRRORMAKER 2 CONNECTORS	111
9.2.1. Changing the location of the consumer group offsets topic	115
9.2.2. Synchronizing consumer group offsets	116
9.2.3. Deciding when to use the heartbeat connector	117
9.2.4. Aligning the configuration of MirrorMaker 2 connectors	117
9.3. CONNECTOR PRODUCER AND CONSUMER CONFIGURATION	117
9.4. SPECIFYING A MAXIMUM NUMBER OF TASKS	118
9.5. ACL RULES SYNCHRONIZATION	119
9.6. RUNNING MIRRORMAKER 2 IN DEDICATED MODE	119
9.7. USING MIRRORMAKER 2 IN LEGACY MODE	122
CHAPTER 10. CONFIGURING LOGGING FOR KAFKA COMPONENTS	124
10.1. CONFIGURING KAFKA LOGGING PROPERTIES	124
10.2. DYNAMICALLY CHANGE LOGGING LEVELS FOR KAFKA BROKER LOGGERS	125
Resetting a broker logger	126
10.3. DYNAMICALLY CHANGE LOGGING LEVELS FOR KAFKA CONNECT AND MIRRORMAKER 2	126
CHAPTER 11. HANDLING HIGH VOLUMES OF MESSAGES	129
11.1. CONFIGURING KAFKA CONNECT FOR HIGH-VOLUME MESSAGES	130
11.2. CONFIGURING MIRRORMAKER 2 FOR HIGH-VOLUME MESSAGES	131
CHAPTER 12. SETTING LIMITS ON BROKERS USING THE KAFKA STATIC QUOTA PLUGIN	132

CHAPTER 13. ADDING AND REMOVING KAFKA BROKERS AND ZOOKEEPER NODES	134
13.1. SCALING CLUSTERS BY ADDING OR REMOVING BROKERS	134
13.2. ADDING NODES TO A ZOOKEEPER CLUSTER	135
13.3. REMOVING NODES FROM A ZOOKEEPER CLUSTER	135
CHAPTER 14. USING CRUISE CONTROL FOR CLUSTER REBALANCING	137
14.1. CRUISE CONTROL COMPONENTS AND FEATURES	138
14.2. DOWNLOADING CRUISE CONTROL	139
14.3. DEPLOYING THE CRUISE CONTROL METRICS REPORTER	139
14.4. CONFIGURING AND STARTING CRUISE CONTROL	140
Auto-created topics	142
14.5. OPTIMIZATION GOALS OVERVIEW	142
14.5.1. Goals order of priority	143
14.5.2. Goals configuration in the Cruise Control properties file	143
14.5.3. Hard and soft optimization goals	144
14.5.4. Main optimization goals	144
14.5.5. Default optimization goals	145
14.5.6. User-provided optimization goals	145
14.6. OPTIMIZATION PROPOSALS OVERVIEW	146
14.6.1. Rebalancing endpoints	146
14.6.2. Approving or rejecting an optimization proposal	147
14.6.3. Optimization proposal summary properties	148
14.6.4. Cached optimization proposal	150
14.7. REBALANCE PERFORMANCE TUNING OVERVIEW	150
Partition reassignment commands	150
Replica movement strategies	150
Rebalance tuning options	151
14.8. CRUISE CONTROL CONFIGURATION	154
Capacity configuration	154
Log cleanup policy for Cruise Control Metrics topic	155
Logging configuration	156
14.9. GENERATING OPTIMIZATION PROPOSALS	156
Asynchronous responses	159
14.10. APPROVING AN OPTIMIZATION PROPOSAL	160
14.11. STOPPING AN ACTIVE CLUSTER REBALANCE	161
CHAPTER 15. USING THE PARTITION REASSIGNMENT TOOL	163
15.1. PARTITION REASSIGNMENT TOOL OVERVIEW	163
15.1.1. Generating a partition reassignment plan	163
15.1.2. Specifying topics in a partition reassignment JSON file	164
15.1.3. Reassigning partitions between JBOD volumes	165
15.1.4. Throttling partition reassignment	166
15.2. REASSIGNING PARTITIONS AFTER ADDING BROKERS	166
15.3. REASSIGNING PARTITIONS BEFORE REMOVING BROKERS	168
15.4. CHANGING THE REPLICATION FACTOR OF TOPICS	170
CHAPTER 16. USING KERBEROS (GSSAPI) AUTHENTICATION	173
16.1. SETTING UP AMQ STREAMS TO USE KERBEROS (GSSAPI) AUTHENTICATION	173
Add service principals for authentication	173
Configure ZooKeeper to use a Kerberos Login	174
Configure the Kafka broker server to use a Kerberos login	176
Configure Kafka producer and consumer clients to use Kerberos authentication	177
CHAPTER 17. SETTING UP DISTRIBUTED TRACING	179

17.1. OUTLINE OF PROCEDURES	179
17.2. TRACING OPTIONS	180
17.3. ENVIRONMENT VARIABLES FOR TRACING	181
17.4. ENABLING TRACING FOR KAFKA CONNECT	181
17.5. ENABLING TRACING FOR MIRRORMAKER 2	182
17.6. ENABLING TRACING FOR MIRRORMAKER	183
17.7. INITIALIZING TRACING FOR KAFKA CLIENTS	184
17.8. INSTRUMENTING PRODUCERS AND CONSUMERS FOR TRACING	185
17.9. INSTRUMENTING KAFKA STREAMS APPLICATIONS FOR TRACING	186
17.10. SPECIFYING TRACING SYSTEMS WITH OPENTELEMETRY	188
17.11. CUSTOM SPAN NAMES	188
17.11.1. Specifying span names for OpenTelemetry	188
CHAPTER 18. USING KAFKA EXPORTER	190
18.1. CONSUMER LAG	190
18.2. KAFKA EXPORTER ALERTING RULE EXAMPLES	190
18.3. KAFKA EXPORTER METRICS	191
18.4. RUNNING KAFKA EXPORTER	192
18.5. PRESENTING KAFKA EXPORTER METRICS IN GRAFANA	194
CHAPTER 19. UPGRADING AMQ STREAMS AND KAFKA	195
19.1. UPGRADE PREREQUISITES	195
19.2. KAFKA VERSIONS	195
19.3. STRATEGIES FOR UPGRADING CLIENTS	196
19.4. UPGRADING KAFKA BROKERS AND ZOOKEEPER	197
19.5. UPGRADING KAFKA COMPONENTS	200
CHAPTER 20. MONITORING YOUR CLUSTER USING JMX	203
20.1. ENABLING THE JMX AGENT	203
20.2. DISABLING THE JMX AGENT	203
20.3. METRICS NAMING CONVENTIONS	204
20.4. ANALYZING KAFKA JMX METRICS FOR TROUBLESHOOTING	205
20.4.1. Checking for under-replicated partitions	205
20.4.2. Identifying performance problems in a Kafka cluster	206
20.4.3. Identifying performance problems with a Kafka controller	208
20.4.4. Identifying problems with requests	208
20.4.5. Using metrics to check the performance of clients	210
20.4.6. Using metrics to check the performance of topics and partitions	210
APPENDIX A. USING YOUR SUBSCRIPTION	212
Accessing Your Account	212
Activating a Subscription	212
Downloading Zip and Tar Files	212
Installing packages with DNF	212

MAKING OPEN SOURCE MORE INCLUSIVE

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

CHAPTER 1. OVERVIEW OF AMQ STREAMS

Red Hat AMQ Streams is a massively-scalable, distributed, and high-performance data streaming platform based on the Apache ZooKeeper and Apache Kafka projects.

The main components comprise:

Kafka Broker

Messaging broker responsible for delivering records from producing clients to consuming clients. Apache ZooKeeper is a core dependency for Kafka, providing a cluster coordination service for highly reliable distributed coordination.

Kafka Streams API

API for writing *stream processor* applications.

Producer and Consumer APIs

Java-based APIs for producing and consuming messages to and from Kafka brokers.

Kafka Bridge

AMQ Streams Kafka Bridge provides a RESTful interface that allows HTTP-based clients to interact with a Kafka cluster.

Kafka Connect

A toolkit for streaming data between Kafka brokers and other systems using *Connector* plugins.

Kafka MirrorMaker

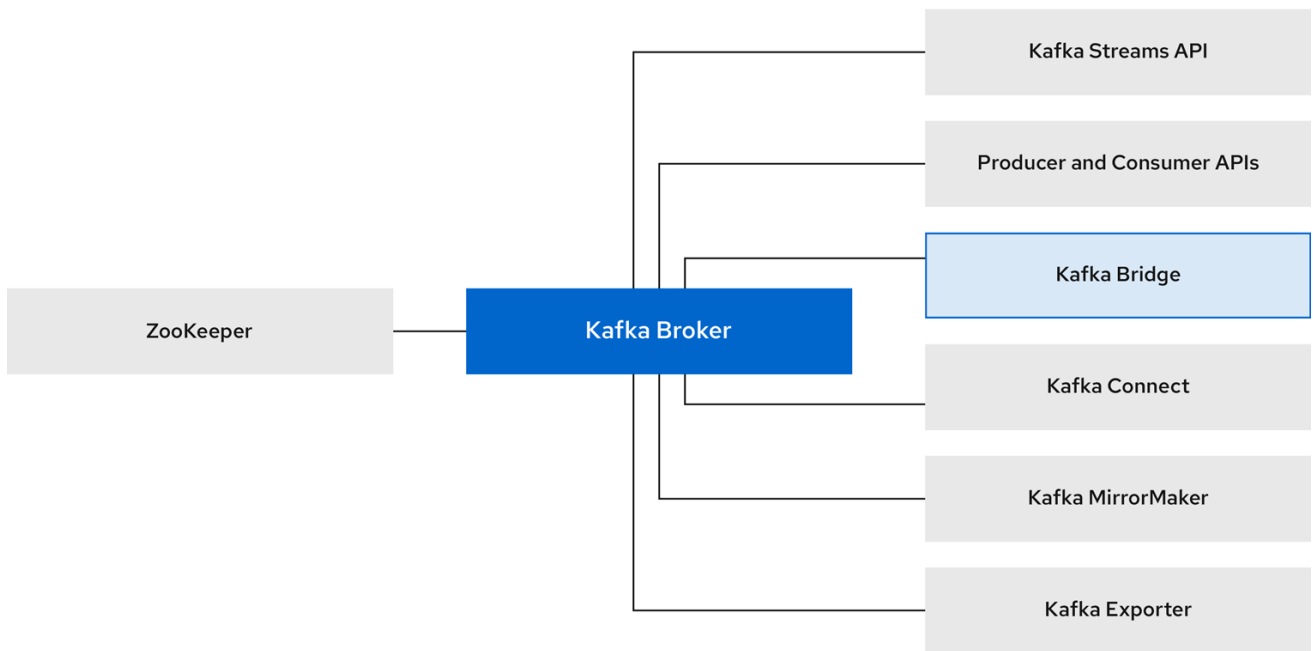
Replicates data between two Kafka clusters, within or across data centers.

Kafka Exporter

An exporter used in the extraction of Kafka metrics data for monitoring.

A cluster of Kafka brokers is the hub connecting all these components. The broker uses Apache ZooKeeper for storing configuration data and for cluster coordination. Before running Apache Kafka, an Apache ZooKeeper cluster has to be ready.

Figure 1.1. AMQ Streams architecture



222_Streams_0322

1.1. USING THE KAFKA BRIDGE TO CONNECT WITH A KAFKA CLUSTER

You can use the AMQ Streams 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 AMQ Streams Kafka Bridge](#).

1.2. DOCUMENT CONVENTIONS

User-replaced values

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

For example, in the following code, you will want to replace **<bootstrap_address>** and **<topic_name>** with your own address and topic name:

```
bin/kafka-console-consumer.sh --bootstrap-server <bootstrap_address> --topic <topic_name> --
from-beginning
```

CHAPTER 2. FIPS SUPPORT

Federal Information Processing Standards (FIPS) are standards for computer security and interoperability. To use FIPS with AMQ Streams, you must have a FIPS-compliant OpenJDK (Open Java Development Kit) installed on your system. If your RHEL system is FIPS-enabled, OpenJDK automatically switches to FIPS mode when running AMQ Streams. This ensures that AMQ Streams uses the FIPS-compliant security libraries provided by OpenJDK.

Minimum password length

When running in the FIPS mode, SCRAM-SHA-512 passwords need to be at least 32 characters long. 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.

Additional resources

- [What are Federal Information Processing Standards \(FIPS\)](#)

2.1. INSTALLING AMQ STREAMS WITH FIPS MODE ENABLED

Enable FIPS mode before you install AMQ Streams on RHEL. Red Hat recommends installing RHEL with FIPS mode enabled, as opposed to enabling FIPS mode later. Enabling FIPS mode during the installation ensures that the system generates all keys with FIPS-approved algorithms and continuous monitoring tests in place.

With RHEL running in FIPS mode, you must ensure that the AMQ Streams configuration is FIPS-compliant. Additionally, your Java implementation must also be FIPS-compliant.



NOTE

Running AMQ Streams on RHEL in FIPS mode requires a FIPS-compliant JDK. We recommend using OpenJDK 17.0.6 or newer.

Procedure

1. Install RHEL in FIPS mode.
For further information, see the information on security hardening in the [RHEL documentation](#).
2. Proceed with the installation of AMQ Streams.
3. Configure AMQ Streams to use FIPS-compliant algorithms and protocols.
If used, ensure that the following configuration is compliant:
 - SSL cipher suites and TLS versions must be supported by the JDK framework.
 - SCRAM-SHA-512 passwords must be at least 32 characters long.



IMPORTANT

Make sure that your installation environment and AMQ Streams configuration remains compliant as FIPS requirements change.

CHAPTER 3. GETTING STARTED

AMQ Streams is distributed in a ZIP file. The ZIP files contain installation artifacts for the following components:

- Apache Kafka
- Apache ZooKeeper
- Apache Kafka Connect
- Apache Kafka MirrorMaker



NOTE

The Kafka Bridge has separate installation files. For information on installing and using the Kafka Bridge, see [Using the AMQ Streams Kafka Bridge](#).

3.1. INSTALLATION ENVIRONMENT

AMQ Streams runs in Red Hat Enterprise Linux. The host (node) can be a physical or virtual machine (VM). Use the installation files provided with AMQ Streams to install Kafka components. You can install Kafka in a single-node or multi-node environment.

Single-node environment

A single-node Kafka cluster runs one or more Kafka brokers and ZooKeeper instances on a single host. This configuration is not suitable for a production environment.

Multi-node environment

A multi-node Kafka cluster runs one or more Kafka brokers and ZooKeeper instances on a multiple hosts.

We recommend that you run Kafka and other Kafka components, such as Kafka Connect, on separate hosts. By running the components in this way, it's easier to maintain and upgrade each component.

Kafka clients establish a connection to the Kafka cluster using the **bootstrap.servers** configuration property. If you are using Kafka Connect, for example, the Kafka Connect configuration properties must include a **bootstrap.servers** value that specifies the hostname and port of the hosts where the Kafka brokers are running. If the Kafka cluster is running on more than one host with multiple Kafka brokers, you specify a hostname and port for each broker. Each Kafka broker is identified by a **broker.id**.

3.1.1. Supported environment versions

AMQ Streams must be running in a supported version of Red Hat Enterprise Linux. The environment must also be using a supported JVM version. For more information, see [Supported Configurations](#).

3.1.2. Data storage considerations

An efficient data storage infrastructure is essential to the optimal performance of AMQ Streams.

Block storage is required. File storage, such as NFS, does not work with Kafka.

Choose from one of the following options for your block storage:

- Cloud-based block storage solutions, such as [Amazon Elastic Block Store \(EBS\)](#)

- Local storage
- Storage Area Network (SAN) volumes accessed by a protocol such as *Fibre Channel* or *iSCSI*

3.1.3. File systems

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

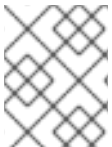
For more information, refer to [Filesystem Selection](#) in the Kafka documentation.

3.1.4. Apache Kafka and ZooKeeper storage

Use separate disks for Apache Kafka and ZooKeeper.

Kafka supports JBOD (Just a Bunch of Disks) storage, a data storage configuration of multiple disks or volumes. JBOD provides increased data storage for Kafka brokers. It can also improve performance.

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.

Additional resources

- [RHEL documentation](#)

3.2. DOWNLOADING AMQ STREAMS

A ZIP file distribution of AMQ Streams is available for download from the Red Hat website. You can download the latest version of Red Hat AMQ Streams from the [AMQ Streams software downloads page](#).

- For Kafka and other Kafka components, download the **amq-streams-<version>-bin.zip** file
- For Kafka Bridge, download the **amq-streams-<version>-bridge-bin.zip** file.
For installation instructions, see [Using the AMQ Streams Kafka Bridge](#) .

3.3. INSTALLING KAFKA

Use the AMQ Streams ZIP files to install Kafka on Red Hat Enterprise Linux. You can install Kafka in a single-node or multi-node environment. In this procedure, a single Kafka broker and ZooKeeper instance are installed on a single host (node).

The AMQ Streams installation files include the binaries for running other Kafka components, like Kafka Connect, Kafka MirrorMaker 2, and Kafka Bridge. In a single-node environment, you can run these components from the same host where you installed Kafka. However, we recommended that you add the installation files and run other Kafka components on separate hosts.



NOTE

If you are using a multi-node environment, you install Kafka brokers and ZooKeeper instances on more than one host. Repeat the installation steps for each host. To identify each ZooKeeper instance and broker, you add a unique ID in the configuration. For more information, see [Chapter 4, Running a multi-node environment](#).

Prerequisites

- You have downloaded the [installation files](#).
- You have reviewed the [supported configurations](#).
- You are logged in to Red Hat Enterprise Linux as admin (**root**) user.

Procedure

Install Kafka with ZooKeeper on your host.

1. Add a new **kafka** user and group:

```
groupadd kafka
useradd -g kafka kafka
passwd kafka
```

2. Extract and move the contents of the **amq-streams-<version>-bin.zip** file into the **/opt/kafka** directory:

```
unzip amq-streams-<version>-bin.zip -d /opt
mv /opt/kafka*redhat* /opt/kafka
```

This step requires admin privileges.

3. Change the ownership of the **/opt/kafka** directory to the **kafka** user:

```
chown -R kafka:kafka /opt/kafka
```

4. Create directory **/var/lib/zookeeper** for storing ZooKeeper data and set its ownership to the **kafka** user:

```
mkdir /var/lib/zookeeper
chown -R kafka:kafka /var/lib/zookeeper
```

5. Create directory **/var/lib/kafka** for storing Kafka data and set its ownership to the **kafka** user:

```
mkdir /var/lib/kafka
chown -R kafka:kafka /var/lib/kafka
```

You can now [run a default configuration of Kafka as a single-node cluster](#) .

You can also use the installation to run other Kafka components, like Kafka Connect, on the same host.

To run other components, specify the hostname and port to connect to the Kafka broker using the **bootstrap.servers** property in the component configuration.

Example bootstrap servers configuration pointing to a single Kafka broker on the same host

```
bootstrap.servers=localhost:9092
```

However, we recommend installing and running Kafka components on separate hosts.

6. (Optional) Install Kafka components on separate hosts.
 - a. Extract the installation files to the **/opt/kafka** directory on each host.
 - b. Change the ownership of the **/opt/kafka** directory to the **kafka** user.
 - c. Add **bootstrap.servers** configuration that connects the component to the host (or hosts in a multi-node environment) running the Kafka brokers.

Example bootstrap servers configuration pointing to Kafka brokers on different hosts

```
bootstrap.servers=kafka0.<host_ip_address>:9092,kafka1.<host_ip_address>:9092,kafka2.<host_ip_address>:9092
```

You can use this configuration for [Kafka Connect](#), [MirrorMaker 2](#), and the [Kafka Bridge](#).

3.4. RUNNING A SINGLE-NODE KAFKA CLUSTER

This procedure shows how to run a basic AMQ Streams cluster consisting of a single Apache ZooKeeper node and a single Apache Kafka node, both running on the same host. The default configuration files are used for ZooKeeper and Kafka.



WARNING

A single node AMQ Streams cluster does not provide reliability and high availability and is suitable only for development purposes.

Prerequisites

- AMQ Streams is installed on the host

Running the cluster

1. Edit the ZooKeeper configuration file **/opt/kafka/config/zookeeper.properties**. Set the **dataDir** option to **/var/lib/zookeeper/**:

```
dataDir=/var/lib/zookeeper/
```

2. Edit the Kafka configuration file **/opt/kafka/config/server.properties**. Set the **log.dirs** option to **/var/lib/kafka/**:

```
log.dirs=/var/lib/kafka/
```

3. Switch to the **kafka** user:

```
su - kafka
```

4. Start ZooKeeper:

```
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

5. Check that ZooKeeper is running:

```
jcmd | grep zookeeper
```

Returns:

```
number org.apache.zookeeper.server.quorum.QuorumPeerMain  
/opt/kafka/config/zookeeper.properties
```

6. Start Kafka:

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

7. Check that Kafka is running:

```
jcmd | grep kafka
```

Returns:

```
number kafka.Kafka /opt/kafka/config/server.properties
```

3.5. SENDING AND RECEIVING MESSAGES FROM A TOPIC

This procedure describes how to start the Kafka console producer and consumer clients and use them to send and receive several messages.

A new topic is automatically created in step one. [Topic auto-creation](#) is controlled using the **auto.create.topics.enable** configuration property (set to **true** by default). Alternatively, you can configure and create topics before using the cluster. For more information, see [Topics](#).

Prerequisites

- [AMQ Streams is installed on the host](#)
- [ZooKeeper and Kafka are running](#)

Procedure

1. Start the Kafka console producer and configure it to send messages to a new topic:

```
/opt/kafka/bin/kafka-console-producer.sh --broker-list <bootstrap_address> --topic <topic-name>
```

For example:

```
/opt/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic my-topic
```

2. Enter several messages into the console. Press **Enter** to send each individual message to your new topic:

```
>message 1
>message 2
>message 3
>message 4
```

When Kafka creates a new topic automatically, you might receive a warning that the topic does not exist:

```
WARN Error while fetching metadata with correlation id 39 :
{4-3-16-topic1=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)
```

The warning should not reappear after you send further messages.

3. In a new terminal window, start the Kafka console consumer and configure it to read messages from the beginning of your new topic.

```
/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server <bootstrap_address> --topic <topic-name> --from-beginning
```

For example:

```
/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic my-topic -from-beginning
```

The incoming messages display in the consumer console.

4. Switch to the producer console and send additional messages. Check that they display in the consumer console.
5. Stop the Kafka console producer and then the consumer by pressing **Ctrl+C**.

3.6. STOPPING THE AMQ STREAMS SERVICES

You can stop the Kafka and ZooKeeper services by running a script. All connections to the Kafka and ZooKeeper services will be terminated.

Prerequisites

- AMQ Streams is installed on the host
- ZooKeeper and Kafka are up and running

Procedure

1. Stop the Kafka broker.

```
su - kafka  
/opt/kafka/bin/kafka-server-stop.sh
```

2. Confirm that the Kafka broker is stopped.

```
jcmd | grep kafka
```

3. Stop ZooKeeper.

```
su - kafka  
/opt/kafka/bin/zookeeper-server-stop.sh
```

CHAPTER 4. RUNNING A MULTI-NODE ENVIRONMENT

A multi-node environment comprises a number of nodes that operate as a cluster. You can have a cluster of replicated ZooKeeper nodes and a cluster of broker nodes, with topic replication across the brokers.

Multi-node environments offer stability and availability.

4.1. RUNNING A MULTI-NODE ZOOKEEPER CLUSTER

Configure and run ZooKeeper as a multi-node cluster.

Prerequisites

- AMQ Streams is installed on all hosts which will be used as ZooKeeper cluster nodes.

Running the cluster

1. Create the **myid** file in **/var/lib/zookeeper/**. Enter ID **1** for the first ZooKeeper node, **2** for the second ZooKeeper node, and so on.

```
su - kafka
echo "<NodeID>" > /var/lib/zookeeper/myid
```

For example:

```
su - kafka
echo "1" > /var/lib/zookeeper/myid
```

2. Edit the ZooKeeper **/opt/kafka/config/zookeeper.properties** configuration file for the following:
 - Set the option **dataDir** to **/var/lib/zookeeper/**.
 - Configure the **initLimit** and **syncLimit** options.
 - Configure the **reconfigEnabled** and **standaloneEnabled** options.
 - Add a list of all ZooKeeper nodes. The list should include also the current node.

Example configuration for a node of ZooKeeper cluster with five members

```
tickTime=2000
dataDir=/var/lib/zookeeper/
initLimit=5
syncLimit=2
reconfigEnabled=true
standaloneEnabled=false

server.1=172.17.0.1:2888:3888:participant;172.17.0.1:2181
server.2=172.17.0.2:2888:3888:participant;172.17.0.2:2181
server.3=172.17.0.3:2888:3888:participant;172.17.0.3:2181
server.4=172.17.0.4:2888:3888:participant;172.17.0.4:2181
server.5=172.17.0.5:2888:3888:participant;172.17.0.5:2181
```

3. Start ZooKeeper with the default configuration file.

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

4. Verify that ZooKeeper is running.

```
jcmd | grep zookeeper
```

Returns:

```
number org.apache.zookeeper.server.quorum.QuorumPeerMain
/opt/kafka/config/zookeeper.properties
```

5. Repeat this procedure on all the nodes of the cluster.
6. Verify that all nodes are members of the cluster by sending a **stat** command to each of the nodes using the **ncat** utility.

Use ncat stat to check the node status

```
echo stat | ncat localhost 2181
```

To use four-letter word commands, like **stat**, you need to specify **4lw.commands.whitelist=*** in **zookeeper.properties**.

The output shows that a node is either a **leader** or **follower**.

Example output from the ncat command

```
ZooKeeper version: 3.4.13-2d71af4dbe22557fda74f9a9b4309b15a7487f03, built on
06/29/2018 00:39 GMT
Clients:
/0:0:0:0:0:0:1:59726[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0
Received: 2
Sent: 1
Connections: 1
Outstanding: 0
Zxid: 0x200000000
Mode: follower
Node count: 4
```

4.2. RUNNING A MULTI-NODE KAFKA CLUSTER

Configure and run Kafka as a multi-node cluster.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.
- A ZooKeeper cluster is [configured and running](#).

Running the cluster

For each Kafka broker in your AMQ Streams cluster:

1. Edit the `/opt/kafka/config/server.properties` Kafka configuration file as follows:
 - Set the **broker.id** field to **0** for the first broker, **1** for the second broker, and so on.
 - Configure the details for connecting to ZooKeeper in the **zookeeper.connect** option.
 - Configure the Kafka listeners.
 - Set the directories where the commit logs should be stored in the **logs.dir** directory.

Here we see an example configuration for a Kafka broker:

```
broker.id=0
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-
domain.com:2181
listeners=REPLICATION://:9091,PLAINTEXT://:9092
listener.security.protocol.map=PLAINTEXT:PLAINTEXT,REPLICATION:PLAINTEXT
inter.broker.listener.name=REPLICATION
log.dirs=/var/lib/kafka
```

In a typical installation where each Kafka broker is running on identical hardware, only the **broker.id** configuration property will differ between each broker config.

2. Start the Kafka broker with the default configuration file.

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

3. Verify that the Kafka broker is running.

```
jcmd | grep Kafka
```

Returns:

```
number kafka.Kafka /opt/kafka/config/server.properties
```

4. Verify that all nodes are members of the Kafka cluster by sending a **dump** command to one of the ZooKeeper nodes using the **ncat** utility.

Use ncat dump to check all Kafka brokers registered in ZooKeeper

```
echo dump | ncat zoo1.my-domain.com 2181
```

To use four-letter word commands, like **dump**, you need to specify **4lw.commands.whitelist=*** in **zookeeper.properties**.

The output must contain all Kafka brokers you just configured and started.

Example output from the ncat command for a Kafka cluster with 3 nodes

```
SessionTracker dump:
```



```
org.apache.zookeeper.server.quorum.LearnerSessionTracker@28848ab9
ephemeral nodes dump:
Sessions with Ephemerals (3):
0x20000015dd00000:
    /brokers/ids/1
0x10000015dc70000:
    /controller
    /brokers/ids/0
0x10000015dc70001:
    /brokers/ids/2
```

4.3. PERFORMING A GRACEFUL ROLLING RESTART OF KAFKA BROKERS

This procedure shows how to do a graceful rolling restart of brokers in a multi-node cluster. A rolling restart is usually required following an upgrade or change to the Kafka cluster configuration properties.



NOTE

Some broker configurations do not need a restart of the broker. For more information, see [Updating Broker Configs](#) in the Apache Kafka documentation.

After you perform a restart of a broker, check for under-replicated topic partitions to make sure that replica partitions have caught up.

You can only perform a graceful restart, with no loss of availability, if you are replicating topics and ensuring that at least one replica is in sync. For a multi-node cluster, the standard approach is to have a topic replication factor of at least 3 and a minimum number of in-sync replicas set to 1 less than the replication factor. If you are using **acks=all** in your producer configuration for data durability, check that the broker you restarted is in sync with all the partitions it's replicating before restarting the next broker.

Single-node clusters are unavailable during a restart, since all partitions are on the same broker.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.
- A ZooKeeper cluster is [configured and running](#).
- The Kafka cluster is operating as expected.
Check for under-replicated partitions or any other issues affecting broker operation. The steps in this procedure describe how to check for under-replicated partitions.

Procedure

Perform the following steps on each Kafka broker. Complete the steps on the first broker before moving on to the next. Perform the steps on the broker that's the active controller last. Otherwise, the active controller needs to change on more than one restart.

1. Stop the Kafka broker:

```
/opt/kafka/bin/kafka-server-stop.sh
```

2. Make any changes to the broker configuration that require a restart after completion.

For further information, see the following:

- [Configuring Kafka](#)
- [Upgrading Kafka brokers and ZooKeeper](#)

3. Restart the Kafka broker:

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

4. Check that Kafka is running:

```
jcmd | grep kafka
```

Returns:

```
number kafka.Kafka /opt/kafka/config/server.properties
```

5. Verify that all nodes are members of the Kafka cluster by sending a **dump** command to one of the ZooKeeper nodes using the **ncat** utility.

Use ncat dump to check all Kafka brokers registered in ZooKeeper

```
echo dump | ncat zoo1.my-domain.com 2181
```

To use four-letter word commands, like **dump**, you need to specify **4lw.commands.whitelist=*** in **zookeeper.properties**.

The output must contain the Kafka broker you started.

Example output from the ncat command for a Kafka cluster with 3 nodes

```
SessionTracker dump:
org.apache.zookeeper.server.quorum.LearnerSessionTracker@28848ab9
ephemeral nodes dump:
Sessions with Ephemerals (3):
0x20000015dd00000:
    /brokers/ids/1
0x10000015dc70000:
    /controller
    /brokers/ids/0
0x10000015dc70001:
    /brokers/ids/2
```

6. Wait until the broker has zero under-replicated partitions. You can check from the command line or use metrics.

- Use the **kafka-topics.sh** command with the **--under-replicated-partitions** parameter:

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <bootstrap_address> --describe --
under-replicated-partitions
```

For example:

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --under-replicated-partitions
```

The command provides a list of topics with under-replicated partitions in a cluster.

Topics with under-replicated partitions

```
Topic: topic3 Partition: 4 Leader: 2 Replicas: 2,3 Isr: 2
Topic: topic3 Partition: 5 Leader: 3 Replicas: 1,2 Isr: 1
Topic: topic1 Partition: 1 Leader: 3 Replicas: 1,3 Isr: 3
# ...
```

Under-replicated partitions are listed if the ISR (in-sync replica) count is less than the number of replicas. If a list is not returned, there are no under-replicated partitions.

- Use the **UnderReplicatedPartitions** metric:

```
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
```

The metric provides a count of partitions where replicas have not caught up. You wait until the count is zero.

TIP

Use the [Kafka Exporter](#) to create an alert when there are one or more under-replicated partitions for a topic.

Checking logs when restarting

If a broker fails to start, check the application logs for information. You can also check the status of a broker shutdown and restart in the `/opt/kafka/logs/server.log` application log.

Log for the successful shutdown of a broker

```
# ...
[2022-06-08 14:32:29,885] INFO Terminating process due to signal SIGTERM
(org.apache.kafka.common.utils.LoggingSignalHandler)
[2022-06-08 14:32:29,886] INFO [KafkaServer id=0] shutting down (kafka.server.KafkaServer)
[2022-06-08 14:32:29,887] INFO [KafkaServer id=0] Starting controlled shutdown
(kafka.server.KafkaServer)
[2022-06-08 14:32:29,896] INFO [KafkaServer id=0] Controlled shutdown request returned
successfully after 6ms (kafka.server.KafkaServer)
# ...
```

Log for the successful restart of a broker

```
# ...
[2022-06-08 14:39:35,245] INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
# ...
```

Additional resources

- [Section 20.4, “Analyzing Kafka JMX metrics for troubleshooting”](#)

- [Chapter 10, *Configuring logging for Kafka components*](#)
- [Kafka configuration tuning](#)

CHAPTER 5. RUNNING KAFKA IN KRAFT MODE (DEVELOPMENT PREVIEW)

When you run AMQ Streams in KRaft (Kafka Raft metadata) mode, Kafka clusters are managed by an internal quorum of controllers instead of ZooKeeper.

Apache Kafka is in the process of phasing out the need for ZooKeeper. KRaft mode is now available to try. You can deploy a Kafka cluster in KRaft mode without ZooKeeper.

CAUTION

KRaft mode is experimental, intended **only** for development and testing, and must not be enabled for a production environment.

Currently, the KRaft mode in AMQ Streams has the following major limitations:

- Migrating from Kafka clusters with ZooKeeper to KRaft clusters is not recommended for production.
- Downgrading from KRaft mode to using ZooKeeper is not supported.
- JBOD storage with multiple disks is not supported.
- Many configuration options are still in development.

5.1. USING AMQ STREAMS WITH KAFKA IN KRAFT MODE

If you use Kafka in KRaft mode, you do not need to use ZooKeeper for cluster coordination or storing metadata. Kafka coordinates the cluster itself using brokers that act as controllers. Kafka also stores the metadata used to track the status of brokers and partitions.

To identify a cluster, create an ID. The ID is used when creating logs for the brokers you add to the cluster.

In the configuration of each broker node, specify the following:

- A node ID
- Broker roles
- A list of brokers (or **voters**) that act as controllers

A broker performs the role of broker, controller, or both.

Broker role

A broker, sometimes referred to as a node or server, orchestrates the storage and passing of messages.

Controller role

A controller coordinates the cluster and manages the tracking metadata.

You can use combined broker and controller nodes, though you might want to separate these functions. Brokers performing combined roles can be more efficient in simpler deployments.

You specify a list of controllers, configured as **voters**, using the node ID and connection details (hostname and port) for each controller.

5.2. RUNNING A KAFKA CLUSTER IN KRAFT MODE

Configure and run Kafka in KRaft mode. You can run Kafka in KRaft mode if you are using a single-node or multi-node Kafka cluster. Run a minimum of three broker and controller nodes for stability and availability.

You set roles for brokers so that they can also be controllers. You apply broker configuration, including the setting of roles, using a configuration properties file. Broker configuration differs according to role. KRaft provides three example broker configuration properties files.

- `/opt/kafka/config/kraft/broker.properties` has example configuration for a broker role
- `/opt/kafka/config/kraft/controller.properties` has example configuration for a controller role
- `/opt/kafka/config/kraft/server.properties` has example configuration for a dual role

You can base your broker configuration on these example properties files. In this procedure, the example **server.properties** configuration is used.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.

Procedure

1. Generate an ID for the Kafka cluster using the **kafka-storage** tool:

```
/opt/kafka/bin/kafka-storage.sh random-uuid
```

The command returns an ID. A cluster ID is required in KRaft mode.

2. Create a configuration properties file for each broker in the cluster. You can base the file on the examples provided with Kafka.
 - a. Specify a role as **broker**, **controller**, or **broker, controller**. For example, specify **process.roles=broker, controller** for a dual role.
 - b. Specify a unique **node.id** for each node in the cluster starting from **0**. For example, **node.id=1**.
 - c. Specify a list of **controller.quorum.voters** in the format **<node_id>@<hostname:port>**. For example, **controller.quorum.voters=1@localhost:9093**.
3. Set up log directories for each node in your Kafka cluster:

```
/opt/kafka/bin/kafka-storage.sh format -t <uuid> -c /opt/kafka/config/kraft/server.properties
```

Returns:

```
Formatting /tmp/kraft-combined-logs
```

Replace `<uuid>` with the cluster ID you generated. Use the same ID for each node in your cluster.

Apply the broker configuration using the properties file you created for the broker.

The default log directory location specified in the **server.properties** configuration file is **/tmp/kraft-combined-logs**. You can add a comma-separated list to set up multiple log directories.

4. Start each Kafka broker.

```
/opt/kafka/bin/kafka-server-start.sh /opt/kafka/config/kraft/server.properties
```

5. Check that Kafka is running:

```
jcmd | grep kafka
```

Returns:

```
number kafka.Kafka /opt/kafka/config/kraft/server.properties
```

You can now create topics, and send and receive messages from the brokers.

For brokers passing messages, you can use topic replication across the brokers in a cluster for data durability. Configure topics to have a replication factor of at least three and a minimum number of in-sync replicas set to 1 less than the replication factor. For more information, see [Section 7.7, “Creating a topic”](#).

CHAPTER 6. CONFIGURING AN AMQ STREAMS ON RHEL DEPLOYMENT

Use the Kafka and ZooKeeper properties files to configure AMQ Streams.

ZooKeeper

`/kafka/config/zookeeper.properties`

Kafka

`/kafka/config/server.properties`

The properties files are in the Java format, with each property on separate line in the following format:

```
<option> = <value>
```

Lines starting with `#` or `!` will be treated as comments and will be ignored by AMQ Streams components.

```
# This is a comment
```

Values can be split into multiple lines by using `\` directly before the newline / carriage return.

```
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required \  
  username="bob" \  
  password="bobs-password";
```

After you save the changes in the properties files, you need to restart the Kafka broker or ZooKeeper. In a multi-node environment, you will need to repeat the process on each node in the cluster.

6.1. USING STANDARD KAFKA CONFIGURATION PROPERTIES

Use standard Kafka configuration properties to configure Kafka components.

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

- Brokers
- Topics
- Clients (producers and consumers)
- Admin client
- Kafka Connect
- Kafka Streams

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



NOTE

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

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

- [Kafka configuration properties](#)
- [Kafka configuration tuning](#)

6.2. LOADING CONFIGURATION VALUES FROM ENVIRONMENT VARIABLES

Use the Environment Variables Configuration Provider plugin to load configuration data from environment variables. You can use the Environment Variables Configuration Provider, for example, to load certificates or JAAS configuration from environment variables.

You can use the provider to load configuration data for all Kafka components, including producers and consumers. Use the provider, for example, to provide the credentials for Kafka Connect connector configuration.

Prerequisites

- AMQ Streams is downloaded and installed on the host
- Environment Variables Configuration Provider JAR file
The JAR file is available from the [AMQ Streams archive](#)

Procedure

1. Add the Environment Variables Configuration Provider JAR file to the Kafka **libs** directory.
2. Initialize the Environment Variables Configuration Provider in the configuration properties file of the Kafka component. For example, to initialize the provider for Kafka, add the configuration to the **server.properties** file.

Configuration to enable the Environment Variables Configuration Provider

```
config.providers=env
config.providers.env.class=io.strimzi.kafka.EnvVarConfigProvider
```

3. Add configuration to the properties file to load data from environment variables.

Configuration to load data from an environment variable

```
option=${env:<MY_ENV_VAR_NAME>}
```

Use capitalized or upper-case environment variable naming conventions, such as **MY_ENV_VAR_NAME**.

4. Save the changes.
5. Restart the Kafka component.
For information on restarting brokers in a multi-node cluster, see [Section 4.3, "Performing a graceful rolling restart of Kafka brokers"](#).

6.3. CONFIGURING ZOOKEEPER

Kafka uses ZooKeeper to store configuration data and for cluster coordination. It is strongly recommended to run a cluster of replicated ZooKeeper instances.

6.3.1. Basic configuration

The most important ZooKeeper configuration options are:

tickTime

ZooKeeper's basic time unit in milliseconds. It is used for heartbeats and session timeouts. For example, minimum session timeout will be two ticks.

dataDir

The directory where ZooKeeper stores its transaction logs and snapshots of its in-memory database. This should be set to the `/var/lib/zookeeper/` directory that was created during installation.

clientPort

Port number where clients can connect. Defaults to **2181**.

An example ZooKeeper configuration file named **config/zookeeper.properties** is located in the AMQ Streams installation directory. It is recommended to place the **dataDir** directory on a separate disk device to minimize the latency in ZooKeeper.

ZooKeeper configuration file should be located in `/opt/kafka/config/zookeeper.properties`. A basic example of the configuration file can be found below. The configuration file has to be readable by the **kafka** user.

```
tickTime=2000
dataDir=/var/lib/zookeeper/
clientPort=2181
```

6.3.2. ZooKeeper cluster configuration

In most production environments, we recommend you deploy a cluster of replicated ZooKeeper instances. A stable and highly available ZooKeeper cluster is important for running for a reliable ZooKeeper service. ZooKeeper clusters are also referred to as *ensembles*.

ZooKeeper clusters usually consist of an odd number of nodes. ZooKeeper requires that a majority of the nodes in the cluster are up and running. For example:

- In a cluster with three nodes, at least two of the nodes must be up and running. This means it can tolerate one node being down.
- In a cluster consisting of five nodes, at least three nodes must be available. This means it can tolerate two nodes being down.
- In a cluster consisting of seven nodes, at least four nodes must be available. This means it can tolerate three nodes being down.

Having more nodes in the ZooKeeper cluster delivers better resiliency and reliability of the whole cluster.

ZooKeeper can run in clusters with an even number of nodes. The additional node, however, does not increase the resiliency of the cluster. A cluster with four nodes requires at least three nodes to be available and can tolerate only one node being down. Therefore it has exactly the same resiliency as a

cluster with only three nodes.

Ideally, the different ZooKeeper nodes should be located in different data centers or network segments. Increasing the number of ZooKeeper nodes increases the workload spent on cluster synchronization. For most Kafka use cases, a ZooKeeper cluster with 3, 5 or 7 nodes should be sufficient.



WARNING

A ZooKeeper cluster with 3 nodes can tolerate only 1 unavailable node. This means that if a cluster node crashes while you are doing maintenance on another node your ZooKeeper cluster will be unavailable.

Replicated ZooKeeper configuration supports all configuration options supported by the standalone configuration. Additional options are added for the clustering configuration:

initLimit

Amount of time to allow followers to connect and sync to the cluster leader. The time is specified as a number of ticks (see the [tickTime option](#) for more details).

syncLimit

Amount of time for which followers can be behind the leader. The time is specified as a number of ticks (see the [tickTime option](#) for more details).

reconfigEnabled

Enables or disables dynamic reconfiguration. Must be enabled in order to add or remove servers to a ZooKeeper cluster.

standaloneEnabled

Enables or disables standalone mode, where ZooKeeper runs with only one server.

In addition to the options above, every configuration file should contain a list of servers which should be members of the ZooKeeper cluster. The server records should be specified in the format **server.id=hostname:port1:port2**, where:

id

The ID of the ZooKeeper cluster node.

hostname

The hostname or IP address where the node listens for connections.

port1

The port number used for intra-cluster communication.

port2

The port number used for leader election.

The following is an example configuration file of a ZooKeeper cluster with three nodes:

```
tickTime=2000
dataDir=/var/lib/zookeeper/
initLimit=5
syncLimit=2
```

```
reconfigEnabled=true
standaloneEnabled=false
```

```
server.1=172.17.0.1:2888:3888:participant;172.17.0.1:2181
server.2=172.17.0.2:2888:3888:participant;172.17.0.2:2181
server.3=172.17.0.3:2888:3888:participant;172.17.0.3:2181
```

TIP

To use four letter word commands, specify `4lw.commands.whitelist=*` in `zookeeper.properties`.

myid files

Each node in the ZooKeeper cluster must be assigned a unique **ID**. Each node's **ID** must be configured in a **myid** file and stored in the **dataDir** folder, like `/var/lib/zookeeper/`. The **myid** files should contain only a single line with the written **ID** as text. The **ID** can be any integer from 1 to 255. You must manually create this file on each cluster node. Using this file, each ZooKeeper instance will use the configuration from the corresponding **server.** line in the configuration file to configure its listeners. It will also use all other **server.** lines to identify other cluster members.

In the above example, there are three nodes, so each one will have a different **myid** with values **1**, **2**, and **3** respectively.

6.3.3. Authentication

By default, ZooKeeper does not use any form of authentication and allows anonymous connections. However, it supports Java Authentication and Authorization Service (JAAS) which can be used to set up authentication using Simple Authentication and Security Layer (SASL). ZooKeeper supports authentication using the DIGEST-MD5 SASL mechanism with locally stored credentials.

6.3.3.1. Authentication with SASL

JAAS is configured using a separate configuration file. It is recommended to place the JAAS configuration file in the same directory as the ZooKeeper configuration (`/opt/kafka/config/`). The recommended file name is `zookeeper-jaas.conf`. When using a ZooKeeper cluster with multiple nodes, the JAAS configuration file has to be created on all cluster nodes.

JAAS is configured using contexts. Separate parts such as the server and client are always configured with a separate *context*. The context is a *configuration* option and has the following format:

```
ContextName {
    param1
    param2;
};
```

SASL Authentication is configured separately for server-to-server communication (communication between ZooKeeper instances) and client-to-server communication (communication between Kafka and ZooKeeper). Server-to-server authentication is relevant only for ZooKeeper clusters with multiple nodes.

Server-to-Server authentication

For server-to-server authentication, the JAAS configuration file contains two parts:

- The server configuration

- The client configuration

When using DIGEST-MD5 SASL mechanism, the **QuorumServer** context is used to configure the authentication server. It must contain all the usernames to be allowed to connect together with their passwords in an unencrypted form. The second context, **QuorumLearner**, has to be configured for the client which is built into ZooKeeper. It also contains the password in an unencrypted form. An example of the JAAS configuration file for DIGEST-MD5 mechanism can be found below:

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_zookeeper="123456";
};

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    username="zookeeper"
    password="123456";
};
```

In addition to the JAAS configuration file, you must enable the server-to-server authentication in the regular ZooKeeper configuration file by specifying the following options:

```
quorum.auth.enableSasl=true
quorum.auth.learnerRequireSasl=true
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner
quorum.auth.server.loginContext=QuorumServer
quorum.cnxn.threads.size=20
```

Use the **KAFKA_OPTS** environment variable to pass the JAAS configuration file to the ZooKeeper server as a Java property:

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf";
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

For more information about server-to-server authentication, see [ZooKeeper wiki](#).

Client-to-Server authentication

Client-to-server authentication is configured in the same JAAS file as the server-to-server authentication. However, unlike the server-to-server authentication, it contains only the server configuration. The client part of the configuration has to be done in the client. For information on how to configure a Kafka broker to connect to ZooKeeper using authentication, see the [Kafka installation](#) section.

Add the Server context to the JAAS configuration file to configure client-to-server authentication. For DIGEST-MD5 mechanism it configures all usernames and passwords:

```
Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_super="123456"
    user_kafka="123456"
    user_someoneelse="123456";
};
```

After configuring the JAAS context, enable the client-to-server authentication in the ZooKeeper configuration file by adding the following line:

```
requireClientAuthScheme=sasl
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.2=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.3=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
```

You must add the **authProvider.<ID>** property for every server that is part of the ZooKeeper cluster.

Use the **KAFKA_OPTS** environment variable to pass the JAAS configuration file to the ZooKeeper server as a Java property:

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf";
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

For more information about configuring ZooKeeper authentication in Kafka brokers, see [Section 6.4.5, "ZooKeeper authentication"](#).

6.3.3.2. Enabling server-to-server authentication using DIGEST-MD5

This procedure describes how to enable authentication using the SASL DIGEST-MD5 mechanism between the nodes of the ZooKeeper cluster.

Prerequisites

- AMQ Streams is installed on the host
- ZooKeeper cluster is [configured](#) with multiple nodes.

Enabling SASL DIGEST-MD5 authentication

1. On all ZooKeeper nodes, create or edit the **/opt/kafka/config/zookeeper-jaas.conf** JAAS configuration file and add the following contexts:

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_<Username>=<Password>;
};

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    username="<Username>"
    password="<Password>";
};
```

The username and password must be the same in both JAAS contexts. For example:

```
QuorumServer {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_zookeeper="123456";
};
```

```

QuorumLearner {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    username="zookeeper"
    password="123456";
};

```

2. On all ZooKeeper nodes, edit the `/opt/kafka/config/zookeeper.properties` ZooKeeper configuration file and set the following options:

```

quorum.auth.enableSasl=true
quorum.auth.learnerRequireSasl=true
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner
quorum.auth.server.loginContext=QuorumServer
quorum.cnxn.threads.size=20

```

3. Restart all ZooKeeper nodes one by one. To pass the JAAS configuration to ZooKeeper, use the `KAFKA_OPTS` environment variable.

```

su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf"; /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties

```

6.3.3.3. Enabling Client-to-server authentication using DIGEST-MD5

This procedure describes how to enable authentication using the SASL DIGEST-MD5 mechanism between ZooKeeper clients and ZooKeeper.

Prerequisites

- AMQ Streams is installed on the host
- ZooKeeper cluster is [configured and running](#).

Enabling SASL DIGEST-MD5 authentication

1. On all ZooKeeper nodes, create or edit the `/opt/kafka/config/zookeeper-jaas.conf` JAAS configuration file and add the following context:

```

Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required
    user_super="<SuperUserPassword>"
    user<Username1>_="<Password1>" user<Username2>_="<Password2>";
};

```

The **super** automatically has administrator privileges. The file can contain multiple users, but only one additional user is required by the Kafka brokers. The recommended name for the Kafka user is **kafka**.

The following example shows the **Server** context for client-to-server authentication:

```

Server {
    org.apache.zookeeper.server.auth.DigestLoginModule required

```

```

user_super="123456"
user_kafka="123456";
};

```

- On all ZooKeeper nodes, edit the `/opt/kafka/config/zookeeper.properties` ZooKeeper configuration file and set the following options:

```

requireClientAuthScheme=sasl
authProvider.<IdOfBroker1>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.<IdOfBroker2>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.<IdOfBroker3>=org.apache.zookeeper.server.auth.SASLAuthenticationProvider

```

The `authProvider.<ID>` property has to be added for every node which is part of the ZooKeeper cluster. An example three-node ZooKeeper cluster configuration must look like the following:

```

requireClientAuthScheme=sasl
authProvider.1=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.2=org.apache.zookeeper.server.auth.SASLAuthenticationProvider
authProvider.3=org.apache.zookeeper.server.auth.SASLAuthenticationProvider

```

- Restart all ZooKeeper nodes one by one. To pass the JAAS configuration to ZooKeeper, use the `KAFKA_OPTS` environment variable.

```

su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/zookeeper-jaas.conf"; /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties

```

6.3.4. Authorization

ZooKeeper supports access control lists (ACLs) to protect data stored inside it. Kafka brokers can automatically configure the ACL rights for all ZooKeeper records they create so no other ZooKeeper user can modify them.

For more information about enabling ZooKeeper ACLs in Kafka brokers, see [Section 6.4.7, "ZooKeeper authorization"](#).

6.3.5. TLS

ZooKeeper supports TLS for encryption or authentication.

6.3.6. Additional configuration options

You can set the following additional ZooKeeper configuration options based on your use case:

maxClientCnxns

The maximum number of concurrent client connections to a single member of the ZooKeeper cluster.

autopurge.snapRetainCount

Number of snapshots of ZooKeeper's in-memory database which will be retained. Default value is **3**.

autopurge.purgeInterval

The time interval in hours for purging snapshots. The default value is **0** and this option is disabled.

All available configuration options can be found in the [ZooKeeper documentation](#).

6.4. CONFIGURING KAFKA

Kafka uses a properties file to store static configuration. The recommended location for the configuration file is `/opt/kafka/config/server.properties`. The configuration file has to be readable by the `kafka` user.

AMQ Streams ships an example configuration file that highlights various basic and advanced features of the product. It can be found under `config/server.properties` in the AMQ Streams installation directory.

This chapter explains the most important configuration options.

6.4.1. ZooKeeper

Kafka brokers need ZooKeeper to store some parts of their configuration as well as to coordinate the cluster (for example to decide which node is a leader for which partition). Connection details for the ZooKeeper cluster are stored in the configuration file. The field `zookeeper.connect` contains a comma-separated list of hostnames and ports of members of the zookeeper cluster.

For example:

```
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-
domain.com:2181
```

Kafka will use these addresses to connect to the ZooKeeper cluster. With this configuration, all Kafka **znodes** will be created directly in the root of ZooKeeper database. Therefore, such a ZooKeeper cluster could be used only for a single Kafka cluster. To configure multiple Kafka clusters to use single ZooKeeper cluster, specify a base (prefix) path at the end of the ZooKeeper connection string in the Kafka configuration file:

```
zookeeper.connect=zoo1.my-domain.com:2181,zoo2.my-domain.com:2181,zoo3.my-
domain.com:2181/my-cluster-1
```

6.4.2. Listeners

Listeners are used to connect to Kafka brokers. Each Kafka broker can be configured to use multiple listeners. Each listener requires a different configuration so it can listen on a different port or network interface.

To configure listeners, edit the `listeners` property in the configuration file (`/opt/kafka/config/server.properties`). Add listeners to the `listeners` property as a comma-separated list. Configure each property as follows:

```
<listenerName>://<hostname>:<port>
```

If `<hostname>` is empty, Kafka uses the `java.net.InetAddress.getCanonicalHostName()` class as the hostname.

Example configuration for multiple listeners

```
listeners=internal-1://:9092,internal-2://:9093,replication://:9094
```

When a Kafka client wants to connect to a Kafka cluster, it first connects to the *bootstrap server*, which is one of the cluster nodes. The bootstrap server provides the client with a list of all the brokers in the cluster, and the client connects to each one individually. The list of brokers is based on the configured **listeners**.

Advertised listeners

Optionally, you can use the **advertised.listeners** property to provide the client with a different set of listener addresses than those given in the **listeners** property. This is useful if additional network infrastructure, such as a proxy, is between the client and the broker, or an external DNS name is being used instead of an IP address.

The **advertised.listeners** property is formatted in the same way as the **listeners** property.

Example configuration for advertised listeners

```
listeners=internal-1://:9092,internal-2://:9093
advertised.listeners=internal-1://my-broker-1.my-domain.com:1234,internal-2://my-broker-1.my-domain.com:1235
```



NOTE

The names of the advertised listeners must match those listed in the **listeners** property.

Inter-broker listeners

Inter-broker listeners are used for communication between Kafka brokers. Inter-broker communication is required for:

- Coordinating workloads between different brokers
- Replicating messages between partitions stored on different brokers
- Handling administrative tasks from the controller, such as partition leadership changes

The inter-broker listener can be assigned to a port of your choice. When multiple listeners are configured, you can define the name of the inter-broker listener in the **inter.broker.listener.name** property.

Here, the inter-broker listener is named as **REPLICATION**:

```
listeners=REPLICATION://0.0.0.0:9091
inter.broker.listener.name=REPLICATION
```

Control plane listeners

By default, communication between the controller and other brokers uses the [inter-broker listener](#). The controller is responsible for coordinating administrative tasks, such as partition leadership changes.

You can enable a dedicated *control plane listener* for controller connections. The control plane listener can be assigned to a port of your choice.

To enable the control plane listener, configure the **control.plane.listener.name** property with a listener name:

```
listeners=CONTROLLER://0.0.0.0:9090,REPLICATION://0.0.0.0:9091
...
control.plane.listener.name=CONTROLLER
```

Enabling the control plane listener might improve cluster performance because controller communications are not delayed by data replication across brokers. Data replication continues through the inter-broker listener.

If **control.plane.listener** is not configured, controller connections use the [inter-broker listener](#).

6.4.3. Commit logs

Apache Kafka stores all records it receives from producers in commit logs. The commit logs contain the actual data, in the form of records, that Kafka needs to deliver. These are not the application log files which record what the broker is doing.

Log directories

You can configure log directories using the **log.dirs** property file to store commit logs in one or multiple log directories. It should be set to **/var/lib/kafka** directory created during installation:

```
log.dirs=/var/lib/kafka
```

For performance reasons, you can configure **log.dirs** to multiple directories and place each of them on a different physical device to improve disk I/O performance. For example:

```
log.dirs=/var/lib/kafka1,/var/lib/kafka2,/var/lib/kafka3
```

6.4.4. Broker ID

Broker ID is a unique identifier for each broker in the cluster. You can assign an integer greater than or equal to 0 as broker ID. The broker ID is used to identify the brokers after restarts or crashes and it is therefore important that the id is stable and does not change over time. The broker ID is configured in the broker properties file:

```
broker.id=1
```

6.4.5. ZooKeeper authentication

By default, connections between ZooKeeper and Kafka are not authenticated. However, Kafka and ZooKeeper support Java Authentication and Authorization Service (JAAS) which can be used to set up authentication using Simple Authentication and Security Layer (SASL). ZooKeeper supports authentication using the DIGEST-MD5 SASL mechanism with locally stored credentials.

6.4.5.1. JAAS Configuration

SASL authentication for ZooKeeper connections has to be configured in the JAAS configuration file. By default, Kafka will use the JAAS context named **Client** for connecting to ZooKeeper. The **Client** context should be configured in the **/opt/kafka/config/jass.conf** file. The context has to enable the **PLAIN** SASL authentication, as in the following example:

```
Client {
    org.apache.kafka.common.security.plain.PlainLoginModule required
```

```
username="kafka"
password="123456";
};
```

6.4.5.2. Enabling ZooKeeper authentication

This procedure describes how to enable authentication using the SASL DIGEST-MD5 mechanism when connecting to ZooKeeper.

Prerequisites

- Client-to-server authentication is [enabled](#) in ZooKeeper

Enabling SASL DIGEST-MD5 authentication

1. On all Kafka broker nodes, create or edit the `/opt/kafka/config/jaas.conf` JAAS configuration file and add the following context:

```
Client {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  username="<Username>"
  password="<Password>";
};
```

The username and password should be the same as configured in ZooKeeper.

Following example shows the **Client** context:

```
Client {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  username="kafka"
  password="123456";
};
```

2. Restart all Kafka broker nodes one by one. To pass the JAAS configuration to Kafka brokers, use the **KAFKA_OPTS** environment variable.

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

For information on restarting brokers in a multi-node cluster, see [Section 4.3, "Performing a graceful rolling restart of Kafka brokers"](#).

Additional resources

- [Authentication](#)

6.4.6. Authorization

Authorization in Kafka brokers is implemented using authorizer plugins.

In this section we describe how to use the **AclAuthorizer** plugin provided with Kafka.

Alternatively, you can use your own authorization plugins. For example, if you are using [OAuth 2.0 token-based authentication](#), you can use [OAuth 2.0 authorization](#).

6.4.6.1. Simple ACL authorizer

Authorizer plugins, including **AclAuthorizer**, are enabled through the **authorizer.class.name** property:

```
authorizer.class.name=kafka.security.auth.AclAuthorizer
```

A fully-qualified name is required for the chosen authorizer. For **AclAuthorizer**, the fully-qualified name is **kafka.security.auth.AclAuthorizer**.

6.4.6.1.1. ACL rules

AclAuthorizer uses ACL rules to manage access to Kafka brokers.

ACL rules are defined in the format:

Principal **P** is allowed / denied operation **O** on Kafka resource **R** from host **H**

For example, a rule might be set so that user:

John can **view** the topic **comments** from host **127.0.0.1**

Host is the IP address of the machine that John is connecting from.

In most cases, the user is a producer or consumer application:

Consumer01 can **write** to the consumer group **accounts** from host **127.0.0.1**

If ACL rules are not present

If ACL rules are not present for a given resource, all actions are denied. This behavior can be changed by setting the property **allow.everyone.if.no.acl.found** to **true** in the Kafka configuration file **/opt/kafka/config/server.properties**.

6.4.6.1.2. Principals

A *principal* represents the identity of a user. The format of the ID depends on the authentication mechanism used by clients to connect to Kafka:

- **User:ANONYMOUS** when connected without authentication.
- **User:<username>** when connected using simple authentication mechanisms, such as PLAIN or SCRAM.
For example **User:admin** or **User:user1**.
- **User:<DistinguishedName>** when connected using TLS client authentication.
For example **User:CN=user1,O=MyCompany,L=Prague,C=CZ**.
- **User:<Kerberos username>** when connected using Kerberos.

The *DistinguishedName* is the distinguished name from the client certificate.

The *Kerberos username* is the primary part of the Kerberos principal, which is used by default when connecting using Kerberos. You can use the **sasl.kerberos.principal.to.local.rules** property to configure how the Kafka principal is built from the Kerberos principal.

6.4.6.1.3. Authentication of users

To use authorization, you need to have authentication enabled and used by your clients. Otherwise, all connections will have the principal **User:ANONYMOUS**.

For more information on methods of authentication, see [Encryption and authentication](#).

6.4.6.1.4. Super users

Super users are allowed to take all actions regardless of the ACL rules.

Super users are defined in the Kafka configuration file using the property **super.users**.

For example:

```
super.users=User:admin,User:operator
```

6.4.6.1.5. Replica broker authentication

When authorization is enabled, it is applied to all listeners and all connections. This includes the inter-broker connections used for replication of data between brokers. If enabling authorization, therefore, ensure that you use authentication for inter-broker connections and give the users used by the brokers sufficient rights. For example, if authentication between brokers uses the **kafka-broker** user, then super user configuration must include the username **super.users=User:kafka-broker**.

6.4.6.1.6. Supported resources

You can apply Kafka ACLs to these types of resource:

- Topics
- Consumer groups
- The cluster
- TransactionId
- DelegationToken

6.4.6.1.7. Supported operations

AclAuthorizer authorizes operations on resources.

Fields with **X** in the following table mark the supported operations for each resource.

Table 6.1. Supported operations for a resource

	Topics	Consumer Groups	Cluster
Read	X	X	

	Topics	Consumer Groups	Cluster
Write	X		
Create			X
Delete	X		
Alter	X		
Describe	X	X	X
ClusterAction			X
All	X	X	X

6.4.6.1.8. ACL management options

ACL rules are managed using the **bin/kafka-acls.sh** utility, which is provided as part of the Kafka distribution package.

Use **kafka-acls.sh** parameter options to add, list and remove ACL rules, and perform other functions.

The parameters require a double-hyphen convention, such as **--add**.

Option	Type	Description	Default
add	Action	Add ACL rule.	
remove	Action	Remove ACL rule.	
list	Action	List ACL rules.	
authorizer	Action	Fully-qualified class name of the authorizer.	kafka.security.auth.AclAuthorizer
authorizer-properties	Configuration	Key/value pairs passed to the authorizer for initialization. For AclAuthorizer , the example values are: zookeeper.connect=zoo1.my-domain.com:2181 .	
bootstrap-server	Resource	Host/port pairs to connect to the Kafka cluster.	Use this option or the authorizer option, not both.

Option	Type	Description	Default
command-config	Resource	Configuration property file to pass to the Admin Client, which is used in conjunction with the bootstrap-server parameter.	
cluster	Resource	Specifies a cluster as an ACL resource.	
topic	Resource	Specifies a topic name as an ACL resource. An asterisk (*) used as a wildcard translates to <i>all topics</i> . A single command can specify multiple --topic options.	
group	Resource	Specifies a consumer group name as an ACL resource. A single command can specify multiple --group options.	
transactional-id	Resource	Specifies a transactional ID as an ACL resource. Transactional delivery means that all messages sent by a producer to multiple partitions must be successfully delivered or none of them. An asterisk (*) used as a wildcard translates to <i>all IDs</i> .	
delegation-token	Resource	Specifies a delegation token as an ACL resource. An asterisk (*) used as a wildcard translates to <i>all tokens</i> .	

Option	Type	Description	Default
resource-pattern-type	Configuration	<p>Specifies a type of resource pattern for the add parameter or a resource pattern filter value for the list or remove parameters.</p> <p>Use literal or prefixed as the resource pattern type for a resource name.</p> <p>Use any or match as resource pattern filter values, or a specific pattern type filter.</p>	literal
allow-principal	Principal	<p>Principal added to an allow ACL rule.</p> <p>A single command can specify multiple --allow-principal options.</p>	
deny-principal	Principal	<p>Principal added to a deny ACL rule.</p> <p>A single command can specify multiple --deny-principal options.</p>	
principal	Principal	<p>Principal name used with the list parameter to return a list of ACLs for the principal.</p> <p>A single command can specify multiple --principal options.</p>	
allow-host	Host	<p>IP address that allows access to the principals listed in --allow-principal.</p> <p>Hostnames or CIDR ranges are not supported.</p>	If --allow-principal is specified, defaults to * meaning "all hosts".

Option	Type	Description	Default
deny-host	Host	<p>IP address that denies access to the principals listed in --deny-principal.</p> <p>Hostnames or CIDR ranges are not supported.</p>	if --deny-principal is specified, defaults to * meaning "all hosts".
operation	Operation	<p>Allows or denies an operation.</p> <p>A single command can specify multiple --operation options can be specified in single command.</p>	All
producer	Shortcut	A shortcut to allow or deny all operations needed by a message producer (WRITE and DESCRIBE on topic, CREATE on cluster).	
consumer	Shortcut	A shortcut to allow or deny all operations needed by a message consumer (READ and DESCRIBE on topic, READ on consumer group).	
idempotent	Shortcut	<p>A shortcut to enable idempotence when used with the --producer parameter, so that messages are delivered exactly once to a partition.</p> <p>Idempotence is enabled automatically if the producer is authorized to send messages based on a specific transactional ID.</p>	
force	Shortcut	A shortcut to accept all queries and do not prompt.	

6.4.6.2. Enabling authorization

This procedure describes how to enable the **AclAuthorizer** plugin for authorization in Kafka brokers.

Prerequisites

- [AMQ Streams is installed](#) on all hosts used as Kafka brokers.

Procedure

1. Edit the `/opt/kafka/config/server.properties` Kafka configuration file to use the **AclAuthorizer**.

```
authorizer.class.name=kafka.security.auth.AclAuthorizer
```

2. (Re)start the Kafka brokers.

6.4.6.3. Adding ACL rules

When using the **AclAuthorizer** plugin to control access to Kafka brokers based on Access Control Lists (ACLs), you can add new ACL rules using the **kafka-acls.sh** utility.

Prerequisites

- Users have been created and granted appropriate permissions to access Kafka resources.
- [AMQ Streams is installed](#) on all hosts used as Kafka brokers.
- [Authorization is enabled](#) in Kafka brokers.

Procedure

- Run **kafka-acls.sh** with the **--add** option.
Examples:
- Allow **user1** and **user2** access to read from **myTopic** using the **MyConsumerGroup** consumer group.

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Read --topic myTopic --allow-principal User:user1 --allow-principal User:user2
```

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Describe --topic myTopic --allow-principal User:user1 --allow-principal User:user2
```

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Read --operation Describe --group MyConsumerGroup --allow-principal User:user1 --allow-principal User:user2
```

- Deny **user1** access to read **myTopic** from IP address host **127.0.0.1**.

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --operation Describe --operation Read --topic myTopic --group MyConsumerGroup --deny-principal User:user1 --deny-host 127.0.0.1
```

- Add **user1** as the consumer of **myTopic** with **MyConsumerGroup**.

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --add --consumer --topic myTopic --group MyConsumerGroup --allow-principal User:user1
```

Additional resources

- [kafka-acls.sh options](#)

6.4.6.4. Listing ACL rules

When using the **AclAuthorizer** plugin to control access to Kafka brokers based on Access Control Lists (ACLs), you can list existing ACL rules using the **kafka-acls.sh** utility.

Prerequisites

- [ACLs have been added.](#)

Procedure

- Run **kafka-acls.sh** with the **--list** option.
For example:

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --list --topic myTopic
```

```
Current ACLs for resource `Topic:myTopic`:
```

```
User:user1 has Allow permission for operations: Read from hosts: *
User:user2 has Allow permission for operations: Read from hosts: *
User:user2 has Deny permission for operations: Read from hosts: 127.0.0.1
User:user1 has Allow permission for operations: Describe from hosts: *
User:user2 has Allow permission for operations: Describe from hosts: *
User:user2 has Deny permission for operations: Describe from hosts: 127.0.0.1
```

Additional resources

- [kafka-acls.sh options](#)

6.4.6.5. Removing ACL rules

When using the **AclAuthorizer** plugin to control access to Kafka brokers based on Access Control Lists (ACLs), you can remove existing ACL rules using the **kafka-acls.sh** utility.

Prerequisites

- [ACLs have been added.](#)

Procedure

- Run **kafka-acls.sh** with the **--remove** option.
Examples:
- Remove the ACL allowing Allow **user1** and **user2** access to read from **myTopic** using the **MyConsumerGroup** consumer group.

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Read --
topic myTopic --allow-principal User:user1 --allow-principal User:user2
```

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Describe -
-topic myTopic --allow-principal User:user1 --allow-principal User:user2
```

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Read --
operation Describe --group MyConsumerGroup --allow-principal User:user1 --allow-principal
User:user2
```

- Remove the ACL adding **user1** as the consumer of **myTopic** with **MyConsumerGroup**.

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --consumer --topic
myTopic --group MyConsumerGroup --allow-principal User:user1
```

- Remove the ACL denying **user1** access to read **myTopic** from IP address host **127.0.0.1**.

```
opt/kafka/bin/kafka-acls.sh --bootstrap-server localhost:9092 --remove --operation Describe -
-operation Read --topic myTopic --group MyConsumerGroup --deny-principal User:user1 --
deny-host 127.0.0.1
```

Additional resources

- [kafka-acls.sh options](#)

6.4.7. ZooKeeper authorization

When authentication is enabled between Kafka and ZooKeeper, you can use ZooKeeper Access Control List (ACL) rules to automatically control access to Kafka's metadata stored in ZooKeeper.

6.4.7.1. ACL Configuration

Enforcement of ZooKeeper ACL rules is controlled by the **zookeeper.set.acl** property in the **config/server.properties** Kafka configuration file.

The property is disabled by default and enabled by setting to **true**:

```
zookeeper.set.acl=true
```

If ACL rules are enabled, when a **znode** is created in ZooKeeper only the Kafka user who created it can modify or delete it. All other users have read-only access.

Kafka sets ACL rules only for newly created ZooKeeper **znodes**. If the ACLs are only enabled after the first start of the cluster, the **zookeeper-security-migration.sh** tool can set ACLs on all existing **znodes**.

Confidentiality of data in ZooKeeper

Data stored in ZooKeeper includes:

- Topic names and their configuration
- Salted and hashed user credentials when SASL SCRAM authentication is used.

But ZooKeeper does not store any records sent and received using Kafka. The data stored in ZooKeeper is assumed to be non-confidential.

If the data is to be regarded as confidential (for example because topic names contain customer IDs), the only option available for protection is isolating ZooKeeper on the network level and allowing access only to Kafka brokers.

6.4.7.2. Enabling ZooKeeper ACLs for a new Kafka cluster

This procedure describes how to enable ZooKeeper ACLs in Kafka configuration for a new Kafka cluster. Use this procedure only before the first start of the Kafka cluster. For enabling ZooKeeper ACLs in a cluster that is already running, see [Section 6.4.7.3, “Enabling ZooKeeper ACLs in an existing Kafka cluster”](#).

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.
- ZooKeeper cluster is [configured and running](#).
- Client-to-server authentication is [enabled](#) in ZooKeeper.
- ZooKeeper authentication is [enabled](#) in the Kafka brokers.
- Kafka brokers have not yet been started.

Procedure

1. Edit the `/opt/kafka/config/server.properties` Kafka configuration file to set the `zookeeper.set.acl` field to `true` on all cluster nodes.

```
zookeeper.set.acl=true
```

2. Start the Kafka brokers.

6.4.7.3. Enabling ZooKeeper ACLs in an existing Kafka cluster

This procedure describes how to enable ZooKeeper ACLs in Kafka configuration for a Kafka cluster that is running. Use the `zookeeper-security-migration.sh` tool to set ZooKeeper ACLs on all existing `znodes`. The `zookeeper-security-migration.sh` is available as part of AMQ Streams, and can be found in the `bin` directory.

Prerequisites

- Kafka cluster is [configured and running](#).

Enabling the ZooKeeper ACLs

1. Edit the `/opt/kafka/config/server.properties` Kafka configuration file to set the `zookeeper.set.acl` field to `true` on all cluster nodes.

```
zookeeper.set.acl=true
```

2. Restart all Kafka brokers one by one.

For information on restarting brokers in a multi-node cluster, see [Section 4.3, “Performing a graceful rolling restart of Kafka brokers”](#).

3. Set the ACLs on all existing ZooKeeper **znodes** using the **zookeeper-security-migration.sh** tool.

```
su - kafka
cd /opt/kafka
KAFKA_OPTS="-Djava.security.auth.login.config=./config/jaas.conf"; ./bin/zookeeper-
security-migration.sh --zookeeper.acl=secure --zookeeper.connect=<ZooKeeperURL>
exit
```

For example:

```
su - kafka
cd /opt/kafka
KAFKA_OPTS="-Djava.security.auth.login.config=./config/jaas.conf"; ./bin/zookeeper-
security-migration.sh --zookeeper.acl=secure --zookeeper.connect=zoo1.my-
domain.com:2181
exit
```

6.4.8. Encryption and authentication

AMQ Streams supports encryption and authentication, which is configured as part of the listener configuration.

6.4.8.1. Listener configuration

Encryption and authentication in Kafka brokers is configured per listener. For more information about Kafka listener configuration, see [Section 6.4.2, “Listeners”](#).

Each listener in the Kafka broker is configured with its own security protocol. The configuration property **listener.security.protocol.map** defines which listener uses which security protocol. It maps each listener name to its security protocol. Supported security protocols are:

PLAINTEXT

Listener without any encryption or authentication.

SSL

Listener using TLS encryption and, optionally, authentication using TLS client certificates.

SASL_PLAINTEXT

Listener without encryption but with SASL-based authentication.

SASL_SSL

Listener with TLS-based encryption and SASL-based authentication.

Given the following **listeners** configuration:

```
listeners=INT1://:9092,INT2://:9093,REPLICATION://:9094
```

the **listener.security.protocol.map** might look like this:

```
listener.security.protocol.map=INT1:SASL_PLAINTEXT,INT2:SASL_SSL,REPLICATION:SSL
```

This would configure the listener **INT1** to use unencrypted connections with SASL authentication, the listener **INT2** to use encrypted connections with SASL authentication and the **REPLICATION** interface to use TLS encryption (possibly with TLS client authentication). The same security protocol can be used multiple times. The following example is also a valid configuration:

```
listener.security.protocol.map=INT1:SSL,INT2:SSL,REPLICATION:SSL
```

Such a configuration would use TLS encryption and TLS authentication for all interfaces. The following chapters will explain in more detail how to configure TLS and SASL.

6.4.8.2. TLS Encryption

Kafka supports TLS for encrypting communication with Kafka clients.

In order to use TLS encryption and server authentication, a keystore containing private and public keys has to be provided. This is usually done using a file in the Java Keystore (JKS) format. A path to this file is set in the **ssl.keystore.location** property. The **ssl.keystore.password** property should be used to set the password protecting the keystore. For example:

```
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456
```

In some cases, an additional password is used to protect the private key. Any such password can be set using the **ssl.key.password** property.

Kafka is able to use keys signed by certification authorities as well as self-signed keys. Using keys signed by certification authorities should always be the preferred method. In order to allow clients to verify the identity of the Kafka broker they are connecting to, the certificate should always contain the advertised hostname(s) as its Common Name (CN) or in the Subject Alternative Names (SAN).

It is possible to use different SSL configurations for different listeners. All options starting with **ssl.** can be prefixed with **listener.name.<NameOfTheListener>.**, where the name of the listener has to be always in lower case. This will override the default SSL configuration for that specific listener. The following example shows how to use different SSL configurations for different listeners:

```
listeners=INT1://:9092,INT2://:9093,REPLICATION://:9094
listener.security.protocol.map=INT1:SSL,INT2:SSL,REPLICATION:SSL

# Default configuration - will be used for listeners INT1 and INT2
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456

# Different configuration for listener REPLICATION
listener.name.replication.ssl.keystore.location=/path/to/keystore/server-1.jks
listener.name.replication.ssl.keystore.password=123456
```

Additional TLS configuration options

In addition to the main TLS configuration options described above, Kafka supports many options for fine-tuning the TLS configuration. For example, to enable or disable TLS / SSL protocols or cipher suites:

ssl.cipher.suites

List of enabled cipher suites. Each cipher suite is a combination of authentication, encryption, MAC and key exchange algorithms used for the TLS connection. By default, all available cipher suites are enabled.

ssl.enabled.protocols

List of enabled TLS / SSL protocols. Defaults to **TLSv1.2,TLSv1.1,TLSv1**.

6.4.8.3. Enabling TLS encryption

This procedure describes how to enable encryption in Kafka brokers.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.

Procedure

1. Generate TLS certificates for all Kafka brokers in your cluster. The certificates should have their advertised and bootstrap addresses in their Common Name or Subject Alternative Name.
2. Edit the **/opt/kafka/config/server.properties** Kafka configuration file on all cluster nodes for the following:
 - Change the **listener.security.protocol.map** field to specify the **SSL** protocol for the listener where you want to use TLS encryption.
 - Set the **ssl.keystore.location** option to the path to the JKS keystore with the broker certificate.
 - Set the **ssl.keystore.password** option to the password you used to protect the keystore. For example:

```
listeners=UNENCRYPTED://:9092,ENCRYPTED://:9093,REPLICATION://:9094
listener.security.protocol.map=UNENCRYPTED:PLAINTEXT,ENCRYPTED:SSL,REPLICATION:PLAINTEXT
ssl.keystore.location=/path/to/keystore/server-1.jks
ssl.keystore.password=123456
```

3. (Re)start the Kafka brokers

6.4.8.4. Authentication

For authentication, you can use:

- TLS client authentication based on X.509 certificates on encrypted connections
- A supported Kafka SASL (Simple Authentication and Security Layer) mechanism
- [OAuth 2.0 token based authentication](#)

6.4.8.4.1. TLS client authentication

TLS client authentication can be used only on connections which are already using TLS encryption. To use TLS client authentication, a truststore with public keys can be provided to the broker. These keys can be used to authenticate clients connecting to the broker. The truststore should be provided in Java Keystore (JKS) format and should contain public keys of the certification authorities. All clients with

public and private keys signed by one of the certification authorities included in the truststore will be authenticated. The location of the truststore is set using field **ssl.truststore.location**. In case the truststore is password protected, the password should be set in the **ssl.truststore.password** property. For example:

```
ssl.truststore.location=/path/to/keystore/server-1.jks
ssl.truststore.password=123456
```

Once the truststore is configured, TLS client authentication has to be enabled using the **ssl.client.auth** property. This property can be set to one of three different values:

none

TLS client authentication is switched off. (Default value)

requested

TLS client authentication is optional. Clients will be asked to authenticate using TLS client certificate but they can choose not to.

required

Clients are required to authenticate using TLS client certificate.

When a client authenticates using TLS client authentication, the authenticated principal name is the distinguished name from the authenticated client certificate. For example, a user with a certificate which has a distinguished name **CN=someuser** will be authenticated with the following principal **CN=someuser,OU=Unknown,O=Unknown,L=Unknown,ST=Unknown,C=Unknown**. When TLS client authentication is not used and SASL is disabled, the principal name will be **ANONYMOUS**.

6.4.8.4.2. SASL authentication

SASL authentication is configured using Java Authentication and Authorization Service (JAAS). JAAS is also used for authentication of connections between Kafka and ZooKeeper. JAAS uses its own configuration file. The recommended location for this file is **/opt/kafka/config/jaas.conf**. The file has to be readable by the **kafka** user. When running Kafka, the location of this file is specified using Java system property **java.security.auth.login.config**. This property has to be passed to Kafka when starting the broker nodes:

```
KAFKA_OPTS="-Djava.security.auth.login.config=/path/to/my/jaas.config"; bin/kafka-server-start.sh
```

SASL authentication is supported both through plain unencrypted connections as well as through TLS connections. SASL can be enabled individually for each listener. To enable it, the security protocol in **listener.security.protocol.map** has to be either **SASL_PLAINTEXT** or **SASL_SSL**.

SASL authentication in Kafka supports several different mechanisms:

PLAIN

Implements authentication based on username and passwords. Usernames and passwords are stored locally in Kafka configuration.

SCRAM-SHA-256 and SCRAM-SHA-512

Implements authentication using Salted Challenge Response Authentication Mechanism (SCRAM). SCRAM credentials are stored centrally in ZooKeeper. SCRAM can be used in situations where ZooKeeper cluster nodes are running isolated in a private network.

GSSAPI

Implements authentication against a Kerberos server.

**WARNING**

The **PLAIN** mechanism sends the username and password over the network in an unencrypted format. It should be therefore only be used in combination with TLS encryption.

The SASL mechanisms are configured via the JAAS configuration file. Kafka uses the JAAS context named **KafkaServer**. After they are configured in JAAS, the SASL mechanisms have to be enabled in the Kafka configuration. This is done using the **sasl.enabled.mechanisms** property. This property contains a comma-separated list of enabled mechanisms:

```
sasl.enabled.mechanisms=PLAIN,SCRAM-SHA-256,SCRAM-SHA-512
```

In case the listener used for inter-broker communication is using SASL, the property **sasl.mechanism.inter.broker.protocol** has to be used to specify the SASL mechanism which it should use. For example:

```
sasl.mechanism.inter.broker.protocol=PLAIN
```

The username and password which will be used for the inter-broker communication has to be specified in the **KafkaServer** JAAS context using the field **username** and **password**.

SASL PLAIN

To use the PLAIN mechanism, the usernames and password which are allowed to connect are specified directly in the JAAS context. The following example shows the context configured for SASL PLAIN authentication. The example configures three different users:

- **admin**
- **user1**
- **user2**

```
KafkaServer {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  user_admin="123456"
  user_user1="123456"
  user_user2="123456";
};
```

The JAAS configuration file with the user database should be kept in sync on all Kafka brokers.

When SASL PLAIN is also used for inter-broker authentication, the **username** and **password** properties should be included in the JAAS context:

```
KafkaServer {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  username="admin"
  password="123456"
```

```

user_admin="123456"
user_user1="123456"
user_user2="123456";
};

```

SASL SCRAM

SCRAM authentication in Kafka consists of two mechanisms: **SCRAM-SHA-256** and **SCRAM-SHA-512**. These mechanisms differ only in the hashing algorithm used - SHA-256 versus stronger SHA-512. To enable SCRAM authentication, the JAAS configuration file has to include the following configuration:

```

KafkaServer {
    org.apache.kafka.common.security.scram.ScramLoginModule required;
};

```

When enabling SASL authentication in the Kafka configuration file, both SCRAM mechanisms can be listed. However, only one of them can be chosen for the inter-broker communication. For example:

```

sasl.enabled.mechanisms=SCRAM-SHA-256,SCRAM-SHA-512
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512

```

User credentials for the SCRAM mechanism are stored in ZooKeeper. The **kafka-configs.sh** tool can be used to manage them. For example, run the following command to add user user1 with password 123456:

```

bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'SCRAM-SHA-256=[password=123456],SCRAM-SHA-512=[password=123456]' --entity-type users --entity-name user1

```

To delete a user credential use:

```

bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --delete-config 'SCRAM-SHA-512' --entity-type users --entity-name user1

```

SASL GSSAPI

The SASL mechanism used for authentication using Kerberos is called **GSSAPI**. To configure Kerberos SASL authentication, the following configuration should be added to the JAAS configuration file:

```

KafkaServer {
    com.sun.security.auth.module.Krb5LoginModule required
    useKeyTab=true
    storeKey=true
    keyTab="/etc/security/keytabs/kafka_server.keytab"
    principal="kafka/kafka1.hostname.com@EXAMPLE.COM";
};

```

The domain name in the Kerberos principal has to be always in upper case.

In addition to the JAAS configuration, the Kerberos service name needs to be specified in the **sasl.kerberos.service.name** property in the Kafka configuration:

```

sasl.enabled.mechanisms=GSSAPI
sasl.mechanism.inter.broker.protocol=GSSAPI
sasl.kerberos.service.name=kafka

```

Multiple SASL mechanisms

Kafka can use multiple SASL mechanisms at the same time. The different JAAS configurations can be all added to the same context:

```
KafkaServer {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  user_admin="123456"
  user_user1="123456"
  user_user2="123456";

  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  storeKey=true
  keyTab="/etc/security/keytabs/kafka_server.keytab"
  principal="kafka/kafka1.hostname.com@EXAMPLE.COM";

  org.apache.kafka.common.security.scram.ScramLoginModule required;
};
```

When multiple mechanisms are enabled, clients will be able to choose the mechanism which they want to use.

6.4.8.5. Enabling TLS client authentication

This procedure describes how to enable TLS client authentication in Kafka brokers.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.
- TLS encryption is [enabled](#).

Procedure

1. Prepare a JKS truststore containing the public key of the certification authority used to sign the user certificates.
2. Edit the `/opt/kafka/config/server.properties` Kafka configuration file on all cluster nodes for the following:
 - Set the **ssl.truststore.location** option to the path to the JKS truststore with the certification authority of the user certificates.
 - Set the **ssl.truststore.password** option to the password you used to protect the truststore.
 - Set the **ssl.client.auth** option to **required**.
For example:

```
ssl.truststore.location=/path/to/truststore.jks
ssl.truststore.password=123456
ssl.client.auth=required
```

3. (Re)start the Kafka brokers

6.4.8.6. Enabling SASL PLAIN authentication

This procedure describes how to enable SASL PLAIN authentication in Kafka brokers.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.

Procedure

1. Edit or create the `/opt/kafka/config/jaas.conf` JAAS configuration file. This file should contain all your users and their passwords. Make sure this file is the same on all Kafka brokers.

For example:

```
KafkaServer {
  org.apache.kafka.common.security.plain.PlainLoginModule required
  user_admin="123456"
  user_user1="123456"
  user_user2="123456";
};
```

2. Edit the `/opt/kafka/config/server.properties` Kafka configuration file on all cluster nodes for the following:

- Change the `listener.security.protocol.map` field to specify the **SASL_PLAINTEXT** or **SASL_SSL** protocol for the listener where you want to use SASL PLAIN authentication.

- Set the `sasl.enabled.mechanisms` option to **PLAIN**.

For example:

```
listeners=INSECURE://:9092,AUTHENTICATED://:9093,REPLICATION://:9094
listener.security.protocol.map=INSECURE:PLAINTEXT,AUTHENTICATED:SASL_PLAINTEXT,REPLICATION:PLAINTEXT
sasl.enabled.mechanisms=PLAIN
```

3. (Re)start the Kafka brokers using the `KAFKA_OPTS` environment variable to pass the JAAS configuration to Kafka brokers.

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

6.4.8.7. Enabling SASL SCRAM authentication

This procedure describes how to enable SASL SCRAM authentication in Kafka brokers.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.

Procedure

1. Edit or create the `/opt/kafka/config/jaas.conf` JAAS configuration file. Enable the **ScramLoginModule** for the **KafkaServer** context. Make sure this file is the same on all Kafka brokers.

For example:

```
KafkaServer {
    org.apache.kafka.common.security.scram.ScramLoginModule required;
};
```

2. Edit the `/opt/kafka/config/server.properties` Kafka configuration file on all cluster nodes for the following:

- Change the **listener.security.protocol.map** field to specify the **SASL_PLAINTEXT** or **SASL_SSL** protocol for the listener where you want to use SASL SCRAM authentication.

- Set the **sasl.enabled.mechanisms** option to **SCRAM-SHA-256** or **SCRAM-SHA-512**.

For example:

```
listeners=INSECURE://:9092,AUTHENTICATED://:9093,REPLICATION://:9094
listener.security.protocol.map=INSECURE:PLAINTEXT,AUTHENTICATED:SASL_PLAINTEXT,REPLICATION:PLAINTEXT
sasl.enabled.mechanisms=SCRAM-SHA-512
```

3. (Re)start the Kafka brokers using the `KAFKA_OPTS` environment variable to pass the JAAS configuration to Kafka brokers.

```
su - kafka
export KAFKA_OPTS="-Djava.security.auth.login.config=/opt/kafka/config/jaas.conf";
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

Additional resources

- [Adding SASL SCRAM users](#)
- [Deleting SASL SCRAM users](#)

6.4.8.8. Adding SASL SCRAM users

This procedure describes how to add new users for authentication using SASL SCRAM.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.
- SASL SCRAM authentication is [enabled](#).

Procedure

- Use the **kafka-configs.sh** tool to add new SASL SCRAM users.

```
bin/kafka-configs.sh --bootstrap-server <broker_address> --alter --add-config 'SCRAM-SHA-512=[password=<Password>]' --entity-type users --entity-name <Username>
```

For example:

```
bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config 'SCRAM-SHA-512=[password=123456]' --entity-type users --entity-name user1
```

6.4.8.9. Deleting SASL SCRAM users

This procedure describes how to remove users when using SASL SCRAM authentication.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.
- SASL SCRAM authentication is [enabled](#).

Procedure

- Use the **kafka-configs.sh** tool to delete SASL SCRAM users.

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --alter --delete-config 'SCRAM-SHA-512' --entity-type users --entity-name <Username>
```

For example:

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --delete-config 'SCRAM-SHA-512' --entity-type users --entity-name user1
```

6.4.9. Using OAuth 2.0 token-based authentication

AMQ Streams supports the use of [OAuth 2.0 authentication](#) using the *OAUTHBEARER* and *PLAIN* mechanisms.

OAuth 2.0 enables standardized token-based authentication and authorization between applications, using a central authorization server to issue tokens that grant limited access to resources.

You can configure OAuth 2.0 authentication, then [OAuth 2.0 authorization](#).

Kafka brokers and clients both need to be configured to use OAuth 2.0. OAuth 2.0 authentication can also be used in conjunction with **simple** or OPA-based Kafka authorization.

Using OAuth 2.0 authentication, application clients can access resources on application servers (called *resource servers*) without exposing account credentials.

The application client passes an access token as a means of authenticating, which application servers can also use to determine the level of access to grant. The authorization server handles the granting of access and inquiries about access.

In the context of AMQ Streams:

- Kafka brokers act as OAuth 2.0 resource servers
- Kafka clients act as OAuth 2.0 application clients

Kafka clients authenticate to Kafka brokers. The brokers and clients communicate with the OAuth 2.0 authorization server, as necessary, to obtain or validate access tokens.

For a deployment of AMQ Streams, OAuth 2.0 integration provides:

- Server-side OAuth 2.0 support for Kafka brokers
- Client-side OAuth 2.0 support for Kafka MirrorMaker, Kafka Connect, and the Kafka Bridge

AMQ Streams on RHEL includes two OAuth 2.0 libraries:

kafka-oauth-client

Provides a custom login callback handler class named **io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler**. To handle the **OAUTHBEARER** authentication mechanism, use the login callback handler with the **OAuthBearerLoginModule** provided by Apache Kafka.

kafka-oauth-common

A helper library that provides some of the functionality needed by the **kafka-oauth-client** library.

The provided client libraries also have dependencies on some additional third-party libraries, such as: **keycloak-core**, **jackson-databind**, and **slf4j-api**.

We recommend using a Maven project to package your client to ensure that all the dependency libraries are included. Dependency libraries might change in future versions.

Additional resources

- [OAuth 2.0 site](#)

6.4.9.1. OAuth 2.0 authentication mechanisms

AMQ Streams supports the OAUTHBEARER and PLAIN mechanisms for OAuth 2.0 authentication. Both mechanisms allow Kafka clients to establish authenticated sessions with Kafka brokers. The authentication flow between clients, the authorization server, and Kafka brokers is different for each mechanism.

We recommend that you configure clients to use OAUTHBEARER whenever possible. OAUTHBEARER provides a higher level of security than PLAIN because client credentials are *never* shared with Kafka brokers. Consider using PLAIN only with Kafka clients that do not support OAUTHBEARER.

You configure Kafka broker listeners to use OAuth 2.0 authentication for connecting clients. If necessary, you can use the OAUTHBEARER and PLAIN mechanisms on the same **oauth** listener. The properties to support each mechanism must be explicitly specified in the **oauth** listener configuration.

OAUTHBEARER overview

To use OAUTHBEARER, set **sasl.enabled.mechanisms** to **OAUTHBEARER** in the OAuth authentication listener configuration for the Kafka broker. For detailed configuration, see [Section 6.4.9.2, "OAuth 2.0 Kafka broker configuration"](#).

```
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
```

Many Kafka client tools use libraries that provide basic support for OAUTHBEARER at the protocol level. To support application development, AMQ Streams provides an *OAuth callback handler* for the upstream Kafka Client Java libraries (but not for other libraries). Therefore, you do not need to write your own callback handlers. An application client can use the callback handler to provide the access token. Clients written in other languages, such as Go, must use custom code to connect to the authorization server and obtain the access token.

With OAUTHBEARER, the client initiates a session with the Kafka broker for credentials exchange, where credentials take the form of a bearer token provided by the callback handler. Using the callbacks, you can configure token provision in one of three ways:

- Client ID and Secret (by using the *OAuth 2.0 client credentials* mechanism)
- A long-lived access token, obtained manually at configuration time
- A long-lived refresh token, obtained manually at configuration time



NOTE

OAUTHBEARER authentication can only be used by Kafka clients that support the OAUTHBEARER mechanism at the protocol level.

PLAIN overview

To use PLAIN, add **PLAIN** to the value of **sasl.enabled.mechanisms**.

```
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER,PLAIN
```

PLAIN is a simple authentication mechanism used by all Kafka client tools. To enable PLAIN to be used with OAuth 2.0 authentication, AMQ Streams provides *OAuth 2.0 over PLAIN* server-side callbacks.

With the AMQ Streams implementation of PLAIN, the client credentials are not stored in ZooKeeper. Instead, client credentials are handled centrally behind a compliant authorization server, similar to when OAUTHBEARER authentication is used.

When used with the OAuth 2.0 over PLAIN callbacks, Kafka clients authenticate with Kafka brokers using either of the following methods:

- Client ID and secret (by using the *OAuth 2.0 client credentials* mechanism)
- A long-lived access token, obtained manually at configuration time

For both methods, the client must provide the PLAIN **username** and **password** properties to pass credentials to the Kafka broker. The client uses these properties to pass a client ID and secret or username and access token.

Client IDs and secrets are used to obtain access tokens.

Access tokens are passed as **password** property values. You pass the access token with or without an **\$accessToken:** prefix.

- If you configure a token endpoint (**oauth.token.endpoint.uri**) in the listener configuration, you need the prefix.
- If you don't configure a token endpoint (**oauth.token.endpoint.uri**) 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 **oauth.username.claim**, **oauth.fallback.username.claim**,

oauth.fallback.username.prefix, and **oauth.userinfo.endpoint.uri** properties. The username extraction process also depends on your authorization server; in particular, how it maps client IDs to account names.



NOTE

OAuth over PLAIN does not support passing a username and password (password grants) using the (deprecated) OAuth 2.0 password grant mechanism.

6.4.9.1.1. Configuring OAuth 2.0 with properties or variables

You can configure OAuth 2.0 settings using Java Authentication and Authorization Service (JAAS) properties or environment variables.

- JAAS properties are configured in the **server.properties** configuration file, and passed as key-values pairs of the **listener.name.LISTENER-NAME.oauthbearer.sasl.jaas.config** property.
- If using environment variables, you still need to provide the **listener.name.LISTENER-NAME.oauthbearer.sasl.jaas.config** property in the **server.properties** file, but you can omit the other JAAS properties.
You can use capitalized or upper-case environment variable naming conventions.

The AMQ Streams OAuth 2.0 libraries use properties that start with:

- **oauth.** to configure authentication
- **strimzi.** to [configure OAuth 2.0 authorization](#)

Additional resources

- [OAuth 2.0 Kafka broker configuration](#)

6.4.9.2. OAuth 2.0 Kafka broker configuration

Kafka broker configuration for OAuth 2.0 authentication involves:

- Creating the OAuth 2.0 client in the authorization server
- Configuring OAuth 2.0 authentication in the Kafka cluster



NOTE

In relation to the authorization server, Kafka brokers and Kafka clients are both regarded as OAuth 2.0 clients.

6.4.9.2.1. OAuth 2.0 client configuration on an authorization server

To configure a Kafka broker to validate the token received during session initiation, the recommended approach is to create an OAuth 2.0 *client* definition in an authorization server, configured as *confidential*, with the following client credentials enabled:

- Client ID of **kafka-broker** (for example)
- Client ID and secret as the authentication mechanism

**NOTE**

You only need to use a client ID and secret when using a non-public introspection endpoint of the authorization server. The credentials are not typically required when using public authorization server endpoints, as with fast local JWT token validation.

6.4.9.2.2. OAuth 2.0 authentication configuration in the Kafka cluster

To use OAuth 2.0 authentication in the Kafka cluster, you enable an OAuth authentication listener configuration for your Kafka cluster, in the Kafka **server.properties** file. A minimum configuration is required. You can also configure a TLS listener, where TLS is used for inter-broker communication.

You can configure the broker for token validation by the authorization server using one of the following methods:

- Fast local token validation: a *JWKS* endpoint in combination with signed JWT-formatted access tokens
- *Introspection* endpoint

You can configure OAUTHBEARER or PLAIN authentication, or both.

The following example shows a minimum configuration that applies a *global* listener configuration, which means that inter-broker communication goes through the same listener as application clients.

The example also shows an OAuth 2.0 configuration for a specific listener, where you specify **listener.name.LISTENER-NAME.sasl.enabled.mechanisms** instead of **sasl.enabled.mechanisms**. *LISTENER-NAME* is the case-insensitive name of the listener. Here, we name the listener **CLIENT**, so the property name is **listener.name.client.sasl.enabled.mechanisms**.

The example uses OAUTHBEARER authentication.

Example: Minimum listener configuration for OAuth 2.0 authentication using a JWKS endpoint

```

sasl.enabled.mechanisms=OAUTHBEARER 1
listeners=CLIENT://0.0.0.0:9092 2
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT 3
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER 4
sasl.mechanism.inter.broker.protocol=OAUTHBEARER 5
inter.broker.listener.name=CLIENT 6
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasServerOAuthValidatorCallbackHandler 7
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \ 8
  oauth.valid.issuer.uri="https://<oauth_server_address>" \ 9
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \ 10
  oauth.username.claim="preferred_username" \ 11
  oauth.client.id="kafka-broker" \ 12
  oauth.client.secret="kafka-secret" \ 13
  oauth.token.endpoint.uri="https://<oauth_server_address>/token" ; 14
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler 15
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000 16

```

- 1 Enables the `OAUTHBEARER` mechanism for credentials exchange over SASL.
- 2 Configures a listener for client applications to connect to. The system `hostname` is used as an advertised hostname, which clients must resolve in order to reconnect. The listener is named `CLIENT` in this example.
- 3 Specifies the channel protocol for the listener. `SASL_SSL` is for TLS. `SASL_PLAINTEXT` is used for an unencrypted connection (no TLS), but there is risk of eavesdropping and interception at the TCP connection layer.
- 4 Specifies the `OAUTHBEARER` mechanism for the `CLIENT` listener. The client name (`CLIENT`) is usually specified in uppercase in the `listeners` property, in lowercase for `listener.name` properties (`listener.name.client`), and in lowercase when part of a `listener.name.client.*` property.
- 5 Specifies the `OAUTHBEARER` mechanism for inter-broker communication.
- 6 Specifies the listener for inter-broker communication. The specification is required for the configuration to be valid.
- 7 Configures OAuth 2.0 authentication on the client listener.
- 8 Configures authentication settings for client and inter-broker communication. The `oauth.client.id`, `oauth.client.secret`, and `auth.token.endpoint.uri` properties relate to inter-broker configuration.
- 9 A valid issuer URI. Only access tokens issued by this issuer will be accepted. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME`.
- 10 The JWKS endpoint URL. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs`.
- 11 The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The value will depend on the authentication flow and the authorization server used. If required, you can use a JsonPath expression like `"['user.info']`. `['user.id']"` to retrieve the username from nested JSON attributes within a token.
- 12 Client ID of the Kafka broker, which is the same for all brokers. This is the [client registered with the authorization server as kafka-broker](#).
- 13 Secret for the Kafka broker, which is the same for all brokers.
- 14 The OAuth 2.0 token endpoint URL to your authorization server. For production, always use `https://` urls. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token`.
- 15 Enables (and is only required for) OAuth 2.0 authentication for inter-broker communication.
- 16 (Optional) Enforces session expiry when a token expires, and also activates the [Kafka re-authentication mechanism](#). 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.

The following example shows a minimum configuration for a TLS listener, where TLS is used for inter-broker communication.

Example: TLS listener configuration for OAuth 2.0 authentication

-

```

listeners=REPLICATION://kafka:9091,CLIENT://kafka:9092 1
listener.security.protocol.map=REPLICATION:SSL,CLIENT:SASL_PLAINTEXT 2
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
inter.broker.listener.name=REPLICATION
listener.name.replication.ssl.keystore.password=<keystore_password> 3
listener.name.replication.ssl.truststore.password=<truststore_password>
listener.name.replication.ssl.keystore.type=JKS
listener.name.replication.ssl.truststore.type=JKS
listener.name.replication.ssl.secure.random.implementation=SHA1PRNG 4
listener.name.replication.ssl.endpoint.identification.algorithm=HTTPS 5
listener.name.replication.ssl.keystore.location=<path_to_keystore> 6
listener.name.replication.ssl.truststore.location=<path_to_truststore> 7
listener.name.replication.ssl.client.auth=required 8
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasServerOAuthValidatorCallbackHandler
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \ 9
  oauth.valid.issuer.uri="https://<oauth_server_address>" \
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \
  oauth.username.claim="preferred_username" ;

```

- 1 Separate configurations are required for inter-broker communication and client applications.
- 2 Configures the *REPLICATION* listener to use TLS, and the *CLIENT* listener to use SASL over an unencrypted channel. The client could use an encrypted channel (**SASL_SSL**) in a production environment.
- 3 The **ssl**. properties define the TLS configuration.
- 4 Random number generator implementation. If not set, the Java platform SDK default is used.
- 5 Hostname verification. If set to an empty string, the hostname verification is turned off. If not set, the default value is **HTTPS**, which enforces hostname verification for server certificates.
- 6 Path to the keystore for the listener.
- 7 Path to the truststore for the listener.
- 8 Specifies that clients of the *REPLICATION* listener have to authenticate with a client certificate when establishing a TLS connection (used for inter-broker connectivity).
- 9 Configures the *CLIENT* listener for OAuth 2.0. Connectivity with the authorization server should use secure HTTPS connections.

The following example shows a minimum configuration for OAuth 2.0 authentication using the PLAIN authentication mechanism for credentials exchange over SASL. Fast local token validation is used.

Example: Minimum listener configuration for PLAIN authentication

```

listeners=CLIENT://0.0.0.0:9092 1
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT 2
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER,PLAIN 3
sasl.mechanism.inter.broker.protocol=OAUTHBEARER 4

```

```

inter.broker.listener.name=CLIENT 5
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasServerOAuthValidatorCallbackHandler 6
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \ 7
  oauth.valid.issuer.uri="http://<auth_server>/auth/realms/<realm>" \ 8
  oauth.jwks.endpoint.uri="https://<auth_server>/auth/realms/<realm>/protocol/openid-connect/certs" \
9
  oauth.username.claim="preferred_username" \ 10
  oauth.client.id="kafka-broker" \ 11
  oauth.client.secret="kafka-secret" \ 12
  oauth.token.endpoint.uri="https://<oauth_server_address>/token" ; 13
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler 14
listener.name.client.plain.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.plain.JaasServerOAuthOverPlainValidatorCallbackHandler 15
listener.name.client.plain.sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule
required \ 16
  oauth.valid.issuer.uri="https://<oauth_server_address>" \ 17
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \ 18
  oauth.username.claim="preferred_username" \ 19
  oauth.token.endpoint.uri="http://<auth_server>/auth/realms/<realm>/protocol/openid-connect/token"
; 20
connections.max.reauth.ms=3600000 21

```

- 1 Configures a listener (named **CLIENT** in this example) for client applications to connect to. The system **hostname** is used as an advertised hostname, which clients must resolve in order to reconnect. Because this is the only configured listener, it is also used for inter-broker communication.
- 2 Configures the example **CLIENT** listener to use SASL over an unencrypted channel. In a production environment, the client should use an encrypted channel (**SASL_SSL**) in order to guard against eavesdropping and interception at the TCP connection layer.
- 3 Enables the *PLAIN* authentication mechanism for credentials exchange over SASL as well as *OAUTHBEARER*. *OAUTHBEARER* is also specified because it is required for inter-broker communication. Kafka clients can choose which mechanism to use to connect.
- 4 Specifies the *OAUTHBEARER* authentication mechanism for inter-broker communication.
- 5 Specifies the listener (named **CLIENT** in this example) for inter-broker communication. Required for the configuration to be valid.
- 6 Configures the server callback handler for the *OAUTHBEARER* mechanism.
- 7 Configures authentication settings for client and inter-broker communication using the *OAUTHBEARER* mechanism. The **oauth.client.id**, **oauth.client.secret**, and **oauth.token.endpoint.uri** properties relate to inter-broker configuration.
- 8 A valid issuer URI. Only access tokens from this issuer are accepted. For example, *https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME*
- 9 The JWKS endpoint URL. For example, *https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs*

- 10 The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The value will depend on the authentication flow and the
- 11 Client ID of the Kafka broker, which is the same for all brokers. This is the [client registered with the authorization server as kafka-broker](#).
- 12 Secret for the Kafka broker (the same for all brokers).
- 13 The OAuth 2.0 token endpoint URL to your authorization server. For production, always use **https://** urls. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token`
- 14 Enables OAuth 2.0 authentication for inter-broker communication.
- 15 Configures the server callback handler for *PLAIN* authentication.
- 16 Configures authentication settings for client communication using *PLAIN* authentication.

oauth.token.endpoint.uri is an optional property that enables OAuth 2.0 over PLAIN using the *OAuth 2.0 client credentials mechanism*.

- 17 A valid issuer URI. Only access tokens from this issuer are accepted. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME`
- 18 The JWKS endpoint URL. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs`
- 19 The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The value will depend on the authentication flow and the authorization server used. If required, you can use a JsonPath expression like `"['user.info'].['user.id']"` to retrieve the username from nested JSON attributes within a token.
- 20 The OAuth 2.0 token endpoint URL to your authorization server. 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. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token`.

- 21 (Optional) Enforces session expiry when a token expires, and also activates the [Kafka re-authentication mechanism](#). 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.

6.4.9.2.3. Fast local JWT token validation configuration

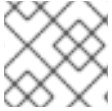
Fast local JWT token validation checks a JWT token signature locally.

The local check ensures that a token:

- Conforms to type by containing a (*typ*) claim value of **Bearer** for an access token
- Is valid (not expired)
- Has an issuer that matches a **validIssuerURI**

You specify a *valid issuer URI* 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 *JWKS endpoint URI* exposed by the OAuth 2.0 authorization server. The endpoint contains the public keys used to validate signed JWT tokens, which are sent as credentials by Kafka clients.



NOTE

All communication with the authorization server should be performed using HTTPS.

For a TLS listener, you can configure a certificate *truststore* and point to the truststore file.

Example properties for fast local JWT token validation

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://<oauth_server_address>" \ 1
  oauth.jwks.endpoint.uri="https://<oauth_server_address>/jwks" \ 2
  oauth.jwks.refresh.seconds="300" \ 3
  oauth.jwks.refresh.min.pause.seconds="1" \ 4
  oauth.jwks.expiry.seconds="360" \ 5
  oauth.username.claim="preferred_username" \ 6
  oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \ 7
  oauth.ssl.truststore.password="<truststore_password>" \ 8
  oauth.ssl.truststore.type="PKCS12" ; 9
```

- 1 A valid issuer URI. Only access tokens issued by this issuer will be accepted. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME`.
- 2 The JWKS endpoint URL. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/certs`.
- 3 The period between endpoint refreshes (default 300).
- 4 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 **oauth.jwks.refresh.seconds**. The default value is 1.
- 5 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.
- 6 The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The value will depend on the authentication flow and the authorization server used. If required, you can use a JsonPath expression like `"['user.info'].['user.id']"` to retrieve the username from nested JSON attributes within a token.
- 7 The location of the truststore used in the TLS configuration.
- 8 Password to access the truststore.

- 9 The truststore type in PKCS #12 format.

6.4.9.2.4. OAuth 2.0 introspection endpoint configuration

Token validation using an OAuth 2.0 introspection endpoint treats a received access token as opaque. The Kafka broker sends an access token to the introspection endpoint, which responds with the token information necessary for validation. Importantly, it returns up-to-date information if the specific access token is valid, and also information about when the token expires.

To configure OAuth 2.0 introspection-based validation, you specify an *introspection endpoint URI* rather than the *JWKS endpoint URI* specified for fast local JWT token validation. Depending on the authorization server, you typically have to specify a *client ID* and *client secret*, because the introspection endpoint is usually protected.

Example properties for an introspection endpoint

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.introspection.endpoint.uri="https://<oauth_server_address>/introspection" \ 1
  oauth.client.id="kafka-broker" \ 2
  oauth.client.secret="kafka-broker-secret" \ 3
  oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \ 4
  oauth.ssl.truststore.password="<truststore_password>" \ 5
  oauth.ssl.truststore.type="PKCS12" \ 6
  oauth.username.claim="preferred_username" ; 7
```

- 1 The OAuth 2.0 introspection endpoint URI. For example, `https://AUTH-SERVER-ADDRESS/auth/realms/REALM-NAME/protocol/openid-connect/token/introspect`.
- 2 Client ID of the Kafka broker.
- 3 Secret for the Kafka broker.
- 4 The location of the truststore used in the TLS configuration.
- 5 Password to access the truststore.
- 6 The truststore type in PKCS #12 format.
- 7 The token claim (or key) that contains the actual user name in the token. The user name is the *principal* used to identify the user. The value will depend on the authentication flow and the authorization server used. If required, you can use a `JsonPath` expression like `"[user.info].[user.id]"` to retrieve the username from nested JSON attributes within a token.

6.4.9.3. Session re-authentication for Kafka brokers

You can configure OAuth listeners to use Kafka *session re-authentication* for OAuth 2.0 sessions between Kafka clients and Kafka brokers. This mechanism enforces the expiry of an authenticated session between the client and the broker after a defined period of time. When a session expires, the client immediately starts a new session by reusing the existing connection rather than dropping it.

Session re-authentication is disabled by default. You can enable it in the **server.properties** file. Set the **connections.max.reauth.ms** property for a TLS listener with OAUTHBEARER or PLAIN enabled as the SASL mechanism.

You can specify session re-authentication per listener. For example:

```
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000
```

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 over the existing connection.

Session expiry for OAUTHBEARER and PLAIN

When session re-authentication is configured, session expiry works differently for OAUTHBEARER and PLAIN authentication.

For OAUTHBEARER and PLAIN, using the *client ID and secret* method:

- The broker's authenticated session will expire at the configured **connections.max.reauth.ms**.
- 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 **connections.max.reauth.ms**.
- 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 **connections.max.reauth.ms** is *not* configured, OAUTHBEARER and PLAIN clients can remain connected to brokers indefinitely, without needing to re-authenticate. Authenticated sessions do not end with access token expiry. However, this can be considered when configuring authorization, for example, by using **keycloak** authorization or installing a custom authorizer.

Additional resources

- [OAuth 2.0 Kafka broker configuration](#)
- [Configuring OAuth 2.0 support for Kafka brokers](#)
- [KIP-368: Allow SASL Connections to Periodically Re-Authenticate](#)

6.4.9.4. OAuth 2.0 Kafka client configuration

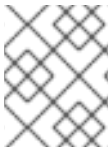
A Kafka client is configured with either:

- The credentials required to obtain a valid access token from an authorization server (client ID and Secret)
- A valid long-lived access token or refresh token, obtained using tools provided by an authorization server

The only information ever sent to the Kafka broker is an access token. The credentials used to authenticate with the authorization server to obtain the access token are never sent to the broker.

When a client obtains an access token, no further communication with the authorization server is needed.

The simplest mechanism is authentication with a client ID and Secret. Using a long-lived access token, or a long-lived refresh token, adds more complexity because there is an additional dependency on authorization server tools.



NOTE

If you are using long-lived access tokens, you may need to configure the client in the authorization server to increase the maximum lifetime of the token.

If the Kafka client is not configured with an access token directly, the client exchanges credentials for an access token during Kafka session initiation by contacting the authorization server. The Kafka client exchanges either:

- Client ID and Secret
- Client ID, refresh token, and (optionally) a secret
- Username and password, with client ID and (optionally) a secret

6.4.9.5. OAuth 2.0 client authentication flows

OAuth 2.0 authentication flows depend on the underlying Kafka client and Kafka broker configuration. The flows must also be supported by the authorization server used.

The Kafka broker listener configuration determines how clients authenticate using an access token. The client can pass a client ID and secret to request an access token.

If a listener is configured to use PLAIN authentication, the client can authenticate with a client ID and secret or username and access token. These values are passed as the **username** and **password** properties of the PLAIN mechanism.

Listener configuration supports the following token validation options:

- You can use fast local token validation based on JWT signature checking and local token introspection, without contacting an authorization server. The authorization server provides a JWKS endpoint with public certificates that are used to validate signatures on the tokens.
- You can use a call to a token introspection endpoint provided by an authorization server. Each time a new Kafka broker connection is established, the broker passes the access token received from the client to the authorization server. The Kafka broker checks the response to confirm

whether or not the token is valid.



NOTE

An authorization server might only allow the use of opaque access tokens, which means that local token validation is not possible.

Kafka client credentials can also be configured for the following types of authentication:

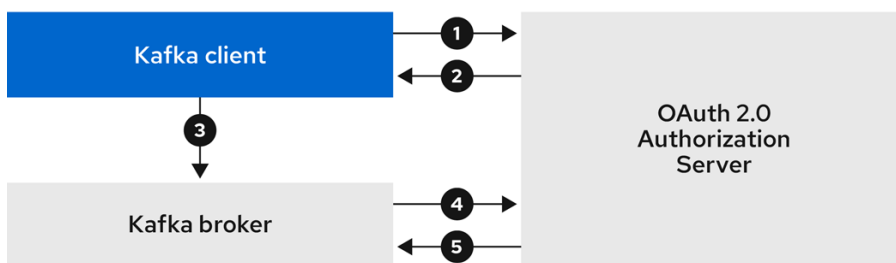
- Direct local access using a previously generated long-lived access token
- Contact with the authorization server for a new access token to be issued (using a client ID and a secret, or a refresh token, or a username and a password)

6.4.9.5.1. Example client authentication flows using the SASL OAUTHBEARER mechanism

You can use the following communication flows for Kafka authentication using the SASL OAUTHBEARER mechanism.

- [Client using client ID and secret, with broker delegating validation to authorization server](#)
- [Client using client ID and secret, with broker performing fast local token validation](#)
- [Client using long-lived access token, with broker delegating validation to authorization server](#)
- [Client using long-lived access token, with broker performing fast local validation](#)

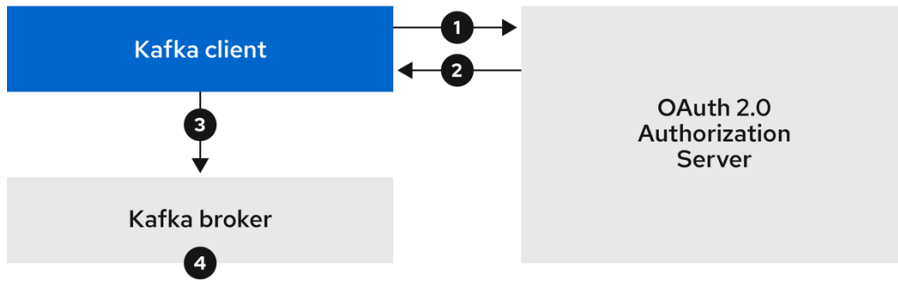
Client using client ID and secret, with broker delegating validation to authorization server



222_Streams_0322

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.

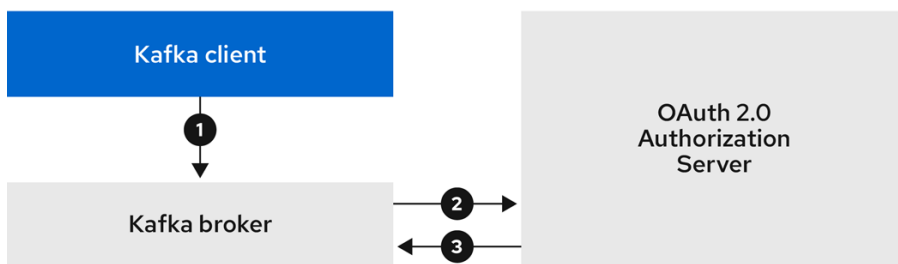
Client using client ID and secret, with broker performing fast local token validation



222_Streams_0322

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



222_Streams_0322

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



222_Streams_0322

1. The Kafka client authenticates with the Kafka broker using the SASL OAUTHBEARER mechanism to pass the long-lived access token.

- The Kafka broker validates the access token locally using a JWT token signature check and local token introspection.



WARNING

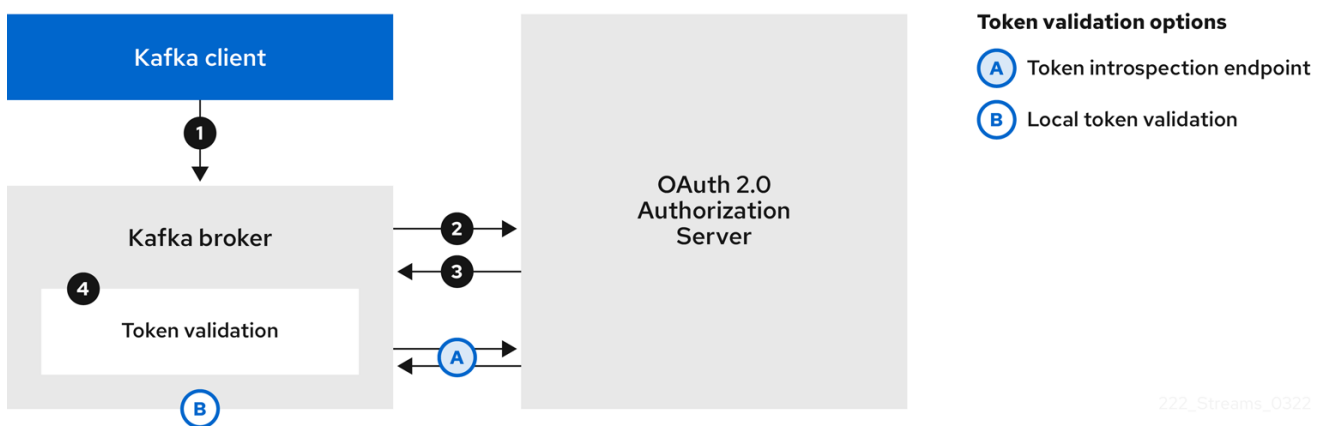
Fast local JWT token signature validation is suitable only for short-lived tokens as there is no check with the authorization server if a token has been revoked. Token expiration is written into the token, but revocation can happen at any time, so cannot be accounted for without contacting the authorization server. Any issued token would be considered valid until it expires.

6.4.9.5.2. Example client authentication flows using the SASL PLAIN mechanism

You can use the following communication flows for Kafka authentication using the OAuth PLAIN mechanism.

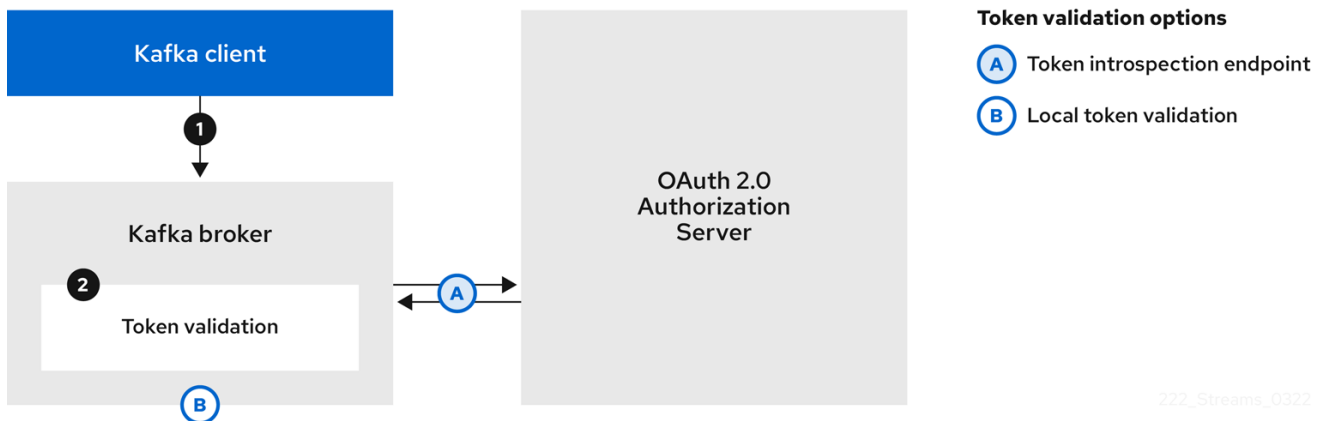
- Client using a client ID and secret, with the broker obtaining the access token for the client
- Client using a long-lived access token without a client ID and secret

Client using a client ID and secret, with the broker obtaining the access token for the client



- The Kafka client passes a **clientId** as a username and a **secret** as a password.
- The Kafka broker uses a token endpoint to pass the **clientId** and **secret** to the authorization server.
- The authorization server returns a fresh access token or an error if the client credentials are not valid.
- The Kafka broker validates the token in one of the following ways:
 - 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.
 - If local token introspection is used, a request is not made to the authorization server. The Kafka broker validates the access token locally using a JWT token signature check.

Client using a long-lived access token without a client ID and secret



1. The Kafka client passes a username and password. The password provides the value of an access token that was obtained manually and configured before running the client.
2. The password is passed with or without an **\$accessToken:** string prefix depending on whether or not the Kafka broker listener is configured with a token endpoint for authentication.
 - a. If the token endpoint is configured, the password should be prefixed by **\$accessToken:** to let the broker know that the password parameter contains an access token rather than a client secret. The Kafka broker interprets the username as the account username.
 - b. If the token endpoint is not configured on the Kafka broker listener (enforcing a **no-client-credentials mode**), the password should provide the access token without the prefix. The Kafka broker interprets the username as the account username. In this mode, the client doesn't use a client ID and secret, and the **password** parameter is always interpreted as a raw access token.
3. The Kafka broker validates the token in one of the following ways:
 - a. If a token introspection endpoint is specified, the Kafka broker validates the access token by calling the endpoint on the authorization server. A session is established if token validation is successful.
 - b. If local token introspection is used, there is no request made to the authorization server. Kafka broker validates the access token locally using a JWT token signature check.

6.4.9.6. Configuring OAuth 2.0 authentication

OAuth 2.0 is used for interaction between Kafka clients and AMQ Streams components.

In order to use OAuth 2.0 for AMQ Streams, you must:

1. [Configure an OAuth 2.0 authorization server for the AMQ Streams cluster and Kafka clients](#)
2. [Deploy or update the Kafka cluster with Kafka broker listeners configured to use OAuth 2.0](#)
3. [Update your Java-based Kafka clients to use OAuth 2.0](#)

6.4.9.6.1. Configuring Red Hat Single Sign-On as an OAuth 2.0 authorization server

This procedure describes how to deploy Red Hat Single Sign-On as an authorization server and configure it for integration with AMQ Streams.

The authorization server provides a central point for authentication and authorization, and management of users, clients, and permissions. Red Hat Single Sign-On has a concept of realms where a *realm* represents a separate set of users, clients, permissions, and other configuration. You can use a default *master realm*, or create a new one. Each realm exposes its own OAuth 2.0 endpoints, which means that application clients and application servers all need to use the same realm.

To use OAuth 2.0 with AMQ Streams, you use a deployment of Red Hat Single Sign-On to create and manage authentication realms.



NOTE

If you already have Red Hat Single Sign-On deployed, you can skip the deployment step and use your current deployment.

Before you begin

You will need to be familiar with using Red Hat Single Sign-On.

For installation and administration instructions, see:

- [Server Installation and Configuration Guide](#)
- [Server Administration Guide](#)

Prerequisites

- AMQ Streams and Kafka are running

For the Red Hat Single Sign-On deployment:

- Check the [Red Hat Single Sign-On Supported Configurations](#)

Procedure

1. Install Red Hat Single Sign-On.
You can install from a ZIP file or by using an RPM.
2. Log in to the Red Hat Single Sign-On Admin Console to create the OAuth 2.0 policies for AMQ Streams.
Login details are provided when you deploy Red Hat Single Sign-On.
3. Create and enable a realm.
You can use an existing master realm.
4. Adjust the session and token timeouts for the realm, if required.
5. Create a client called **kafka-broker**.
6. From the **Settings** tab, set:
 - **Access Type** to **Confidential**
 - **Standard Flow Enabled** to **OFF** to disable web login for this client
 - **Service Accounts Enabled** to **ON** to allow this client to authenticate in its own name

7. Click **Save** before continuing.
8. From the **Credentials** tab, take a note of the secret for using in your AMQ Streams Kafka cluster configuration.
9. Repeat the client creation steps for any application client that will connect to your Kafka brokers.
Create a definition for each new client.

You will use the names as client IDs in your configuration.

What to do next

After deploying and configuring the authorization server, [configure the Kafka brokers to use OAuth 2.0](#) .

6.4.9.6.2. Configuring OAuth 2.0 support for Kafka brokers

This procedure describes how to configure Kafka brokers so that the broker listeners are enabled to use OAuth 2.0 authentication using an authorization server.

We advise use of OAuth 2.0 over an encrypted interface through configuration of TLS listeners. Plain listeners are not recommended.

Configure the Kafka brokers using properties that support your chosen authorization server, and the type of authorization you are implementing.

Before you start

For more information on the configuration and authentication of Kafka broker listeners, see:

- [Listeners](#)
- [OAuth 2.0 authentication mechanisms](#)

For a description of the properties used in the listener configuration, see:

- [OAuth 2.0 Kafka broker configuration](#)

Prerequisites

- AMQ Streams and Kafka are running
- An OAuth 2.0 authorization server is deployed

Procedure

1. Configure the Kafka broker listener configuration in the **server.properties** file.
For example, using the OAUTHBEARER mechanism:

```
sasl.enabled.mechanisms=OAUTHBEARER
listeners=CLIENT://0.0.0.0:9092
listener.security.protocol.map=CLIENT:SASL_PLAINTEXT
listener.name.client.sasl.enabled.mechanisms=OAUTHBEARER
sasl.mechanism.inter.broker.protocol=OAUTHBEARER
inter.broker.listener.name=CLIENT
listener.name.client.oauthbearer.sasl.server.callback.handler.class=io.strimzi.kafka.oauth.server.JaasServerOauthValidatorCallbackHandler
```

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required ;
listener.name.client.oauthbearer.sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JasClientOAuthLoginCallbackHandler
```

2. Configure broker connection settings as part of the **listener.name.client.oauthbearer.sasl.jaas.config**.

The examples here show connection configuration options.

Example 1: Local token validation using a JWKS endpoint configuration

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
  oauth.valid.issuer.uri="https://<oauth_server_address>/auth/realms/<realm_name>" \

oauth.jwks.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_name>/protocol/
openid-connect/certs" \
  oauth.jwks.refresh.seconds="300" \
  oauth.jwks.refresh.min.pause.seconds="1" \
  oauth.jwks.expiry.seconds="360" \
  oauth.username.claim="preferred_username" \
  oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \
  oauth.ssl.truststore.password="<truststore_password>" \
  oauth.ssl.truststore.type="PKCS12" ;
listener.name.client.oauthbearer.connections.max.reauth.ms=3600000
```

Example 2: Delegating token validation to the authorization server through the OAuth 2.0 introspection endpoint

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \

oauth.introspection.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_name>/
protocol/openid-connect/introspection" \
# ...
```

3. If required, configure access to the authorization server.

This step is normally required for a production environment, unless a technology like *service mesh* is used to configure secure channels outside containers.

 - a. Provide a custom truststore for connecting to a secured authorization server. SSL is always required for access to the authorization server. Set properties to configure the truststore.

For example:

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
# ...
oauth.client.id="kafka-broker" \
oauth.client.secret="kafka-broker-secret" \
oauth.ssl.truststore.location="<path_to_truststore_p12_file>" \
oauth.ssl.truststore.password="<truststore_password>" \
oauth.ssl.truststore.type="PKCS12" ;
```

- b. If the certificate hostname does not match the access URL hostname, you can turn off certificate hostname validation:

```
oauth.ssl.endpoint.identification.algorithm=""
```

The check ensures that client connection to the authorization server is authentic. You may wish to turn off the validation in a non-production environment.

4. Configure additional properties according to your chosen authentication flow:

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
# ...

oauth.token.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_name>/protocol/
/openid-connect/token" \ 1
oauth.custom.claim.check="@.custom == 'custom-value'" \ 2
oauth.scope="<scope>" \ 3
oauth.check.audience="true" \ 4
oauth.audience="<audience>" \ 5
oauth.valid.issuer.uri="https://https://<oauth_server_address>/auth/<realm_name>" \ 6
oauth.client.id="kafka-broker" \ 7
oauth.client.secret="kafka-broker-secret" \ 8
oauth.connect.timeout.seconds=60 \ 9
oauth.read.timeout.seconds=60 \ 10
oauth.http.retries=2 \ 11
oauth.http.retry.pause.millis=300 \ 12
oauth.groups.claim="$.groups" \ 13
oauth.groups.claim.delimiter="," \ 14
oauth.include.accept.header="false" ; 15
```

- 1 The OAuth 2.0 token endpoint URL to your authorization server. For production, always use **https://** urls. Required when **KeycloakAuthorizer** is used, or an OAuth 2.0 enabled listener is used for inter-broker communication.
- 2 (Optional) **Custom claim checking**. A JsonPath filter query that applies additional custom rules to the JWT access token during validation. If the access token does not contain the necessary data, it is rejected. When using the *introspection* endpoint method, the custom check is applied to the introspection endpoint response JSON.
- 3 (Optional) 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.
- 4 (Optional) **Audience checking**. If your authorization server provides an **aud** (audience) claim, and you want to enforce an audience check, set **oauth.check.audience** 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** claims. Default is **false**.
- 5 (Optional) 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.

- 6 A valid issuer URI. Only access tokens issued by this issuer will be accepted. (Always required.)
- 7 The configured client ID of the Kafka broker, which is the same for all brokers. This is the [client registered with the authorization server as kafka-broker](#). Required when an introspection endpoint is used for token validation, or when **KeycloakAuthorizer** is used.
- 8 The configured secret for the Kafka broker, which is the same for all brokers. When the broker must authenticate to the authorization server, either a client secret, access token or a refresh token has to be specified.
- 9 (Optional) The connect timeout in seconds when connecting to the authorization server. The default value is 60.
- 10 (Optional) The read timeout in seconds when connecting to the authorization server. The default value is 60.
- 11 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 **oauth.connect.timeout.seconds** and **oauth.read.timeout.seconds** 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.
- 12 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.
- 13 A JsonPath query used to extract groups information from JWT token or introspection endpoint response. Not set by default. This can be used by a custom authorizer to make authorization decisions based on user groups.
- 14 A delimiter used to parse groups information when returned as a single delimited string. The default value is ',' (comma).
- 15 (Optional) Sets **oauth.include.accept.header** to **false** to remove the **Accept** header from requests. You can use this setting if including the header is causing issues when communicating with the authorization server.

5. Depending on how you apply OAuth 2.0 authentication, and the type of authorization server being used, add additional configuration settings:

```
listener.name.client.oauthbearer.sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
# ...
oauth.check.issuer=false \ 1
oauth.fallback.username.claim="<client_id>" \ 2
oauth.fallback.username.prefix="<client_account>" \ 3
oauth.valid.token.type="bearer" \ 4
```

`oauth.userinfo.endpoint.uri="https://<oauth_server_address>/auth/realms/<realm_name>/protocol/openid-connect/userinfo" ; 5`

- 1 If your authorization server does not provide an **iss** claim, it is not possible to perform an issuer check. In this situation, set **oauth.check.issuer** to **false** and do not specify a **oauth.valid.issuer.uri**. Default is **true**.
- 2 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 required, you can use a JsonPath expression like **"['client.info'].['client.id']"** to retrieve the fallback username from nested JSON attributes within a token.
- 3 In situations where **oauth.fallback.username.claim** 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.
- 4 (Only applicable when using **oauth.introspection.endpoint.uri**) 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.
- 5 (Only applicable when using **oauth.introspection.endpoint.uri**) 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 **oauth.fallback.username.claim**, **oauth.fallback.username.claim**, and **oauth.fallback.username.prefix** settings are applied to the response of the **userinfo** endpoint.

What to do next

- [Configure your Kafka clients to use OAuth 2.0](#)

6.4.9.6.3. Configuring Kafka Java clients to use OAuth 2.0

Configure Kafka producer and consumer APIs to use OAuth 2.0 for interaction with Kafka brokers. Add a callback plugin to your client **pom.xml** file, then configure your client for OAuth 2.0.

Specify the following in your client configuration:

- A SASL (Simple Authentication and Security Layer) security protocol:
 - **SASL_SSL** for authentication over TLS encrypted connections
 - **SASL_PLAINTEXT** for authentication over unencrypted connections

Use **SASL_SSL** for production and **SASL_PLAINTEXT** for local development only. When using **SASL_SSL**, additional **ssl.truststore** configuration is needed. The truststore configuration is required for secure connection (**https://**) to the OAuth 2.0 authorization

server. To verify the OAuth 2.0 authorization server, add the CA certificate for the authorization server to the truststore in your client configuration. You can configure a truststore in PEM or PKCS #12 format.

- A Kafka SASL mechanism:
 - **OAUTHBEARER** for credentials exchange using a bearer token
 - **PLAIN** to pass client credentials (clientId + secret) or an access token
- A JAAS (Java Authentication and Authorization Service) module that implements the SASL mechanism:
 - **org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule** implements the OAuthbearer mechanism
 - **org.apache.kafka.common.security.plain.PlainLoginModule** implements the plain mechanism

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.

- SASL authentication properties, which support the following authentication methods:
 - OAuth 2.0 client credentials
 - OAuth 2.0 password grant (deprecated)
 - Access token
 - Refresh token

Add the SASL authentication properties as JAAS configuration (**sasl.jaas.config** and **sasl.login.callback.handler.class**). How you configure the authentication properties depends on the authentication method you are using to access the OAuth 2.0 authorization server. In this procedure, the properties are specified in a properties file, then loaded into the client configuration.



NOTE

You can also specify authentication properties as environment variables, or as Java system properties. For Java system properties, you can set them using **setProperty** and pass them on the command line using the **-D** option.

Prerequisites

- AMQ Streams and Kafka are running
- An OAuth 2.0 authorization server is deployed and configured for OAuth access to Kafka brokers
- Kafka brokers are configured for OAuth 2.0

Procedure

1. Add the client library with OAuth 2.0 support to the **pom.xml** file for the Kafka client:

```
<dependency>
  <groupId>io.strimzi</groupId>
  <artifactId>kafka-oauth-client</artifactId>
  <version>0.14.0.redhat-00006</version>
</dependency>
```

2. Configure the client properties by specifying the following configuration in a properties file:

- The security protocol
- The SASL mechanism
- The JAAS module and authentication properties according to the method being used
For example, we can add the following to a **client.properties** file:

Client credentials mechanism properties

```
security.protocol=SASL_SSL 1
sasl.mechanism=OAUTHBEARER 2
ssl.truststore.location=/tmp/truststore.p12 3
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \ 4
  oauth.client.id="<client_id>" \ 5
  oauth.client.secret="<client_secret>" \ 6
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \ 7
  oauth.ssl.truststore.password="$STOREPASS" \ 8
  oauth.ssl.truststore.type="PKCS12" \ 9
  oauth.scope="<scope>" \ 10
  oauth.audience="<audience>" ; 11
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallback
Handler
```

- 1 **SASL_SSL** security protocol for TLS-encrypted connections. Use **SASL_PLAINTEXT** over unencrypted connections for local development only.
- 2 The SASL mechanism specified as **OAUTHBEARER** or **PLAIN**.
- 3 The truststore configuration for secure access to the Kafka cluster.
- 4 URI of the authorization server token endpoint.
- 5 Client ID, which is the name used when creating the *client* in the authorization server.
- 6 Client secret created when creating the *client* in the authorization server.
- 7 The location contains the public key certificate (**truststore.p12**) for the authorization server.

- 8 The password for accessing the truststore.
- 9 The truststore type.
- 10 (Optional) The **scope** for requesting the token from the token endpoint. An authorization server may require a client to specify the scope.
- 11 (Optional) The **audience** for requesting the token from the token endpoint. An authorization server may require a client to specify the audience.

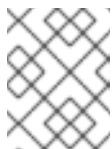
Password grants mechanism properties

```

security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.password.grant.username="<username>" \ 3
  oauth.password.grant.password="<password>" \ 4
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.scope="<scope>" \
  oauth.audience="<audience>" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallback
Handler

```

- 1 Client ID, which is the name used when creating the *client* in the authorization server.
- 2 (Optional) Client secret created when creating the *client* in the authorization server.
- 3 Username for password grant authentication. OAuth password grant configuration (username and password) uses the OAuth 2.0 password grant method. To use password grants, create a user account for a client on your authorization server with limited permissions. The account should act like a service account. Use in environments where user accounts are required for authentication, but consider using a refresh token first.
- 4 Password for password grant authentication.



NOTE

SASL PLAIN does not support passing a username and password (password grants) using the OAuth 2.0 password grant method.

Access token properties

```

security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.access.token="<access_token>" \ 1
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallback
Handler

```

- 1 Long-lived access token for Kafka clients.

Refresh token properties

```

security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.token.endpoint.uri="<token_endpoint_url>" \
  oauth.client.id="<client_id>" \ 1
  oauth.client.secret="<client_secret>" \ 2
  oauth.refresh.token="<refresh_token>" \ 3
  oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallback
Handler

```

- 1 Client ID, which is the name used when creating the *client* in the authorization server.
- 2 (Optional) Client secret created when creating the *client* in the authorization server.
- 3 Long-lived refresh token for Kafka clients.

3. Input the client properties for OAUTH 2.0 authentication into the Java client code.

Example showing input of client properties

```

Properties props = new Properties();
try (FileReader reader = new FileReader("client.properties", StandardCharsets.UTF_8)) {
    props.load(reader);
}

```

4. Verify that the Kafka client can access the Kafka brokers.

6.4.10. Using OAuth 2.0 token-based authorization

If you are using OAuth 2.0 with Red Hat Single Sign-On for token-based authentication, you can also use Red Hat Single Sign-On to configure authorization rules to constrain client access to Kafka brokers. Authentication establishes the identity of a user. Authorization decides the level of access for that user.

AMQ Streams supports the use of OAuth 2.0 token-based authorization through Red Hat Single Sign-On [Authorization Services](#), which allows you to manage security policies and permissions centrally.

Security policies and permissions defined in Red Hat Single Sign-On are used to grant access to resources on Kafka brokers. Users and clients are matched against policies that permit access to perform specific actions on Kafka brokers.

Kafka allows all users full access to brokers by default, and also provides the **AclAuthorizer** 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 the *username*, and these rules are stored within the Kafka cluster itself. However, OAuth 2.0 token-based authorization with Red Hat Single Sign-On offers far greater flexibility on how you wish to implement access control to Kafka brokers. In addition, you can configure your Kafka brokers to use OAuth 2.0 authorization and ACLs.

Additional resources

- [Using OAuth 2.0 token-based authentication](#)
- [Kafka Authorization](#)
- [Red Hat Single Sign-On documentation](#)

6.4.10.1. OAuth 2.0 authorization mechanism

OAuth 2.0 authorization in AMQ Streams uses Red Hat Single Sign-On server Authorization Services REST endpoints to extend token-based authentication with Red Hat Single Sign-On by applying defined security policies on a particular user, and providing a list of permissions granted on different resources for that user. Policies use roles and groups to match permissions to users. OAuth 2.0 authorization enforces permissions locally based on the received list of grants for the user from Red Hat Single Sign-On Authorization Services.

6.4.10.1.1. Kafka broker custom authorizer

A Red Hat Single Sign-On *authorizer* (**KeycloakAuthorizer**) is provided with AMQ Streams. To be able to use the Red Hat Single Sign-On REST endpoints for Authorization Services provided by Red Hat Single Sign-On, you configure a custom authorizer on the Kafka broker.

The authorizer fetches a list of granted permissions from the authorization server as needed, and enforces authorization locally on the Kafka Broker, making rapid authorization decisions for each client request.

6.4.10.2. Configuring OAuth 2.0 authorization support

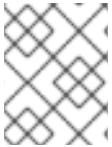
This procedure describes how to configure Kafka brokers to use OAuth 2.0 authorization using Red Hat Single Sign-On Authorization Services.

Before you begin

Consider the access you require or want to limit for certain users. You can use a combination of Red Hat Single Sign-On *groups*, *roles*, *clients*, and *users* to configure access in Red Hat Single Sign-On.

Typically, groups are used to match users based on organizational departments or geographical locations. And roles are used to match users based on their function.

With Red Hat Single Sign-On, you can store users and groups in LDAP, whereas clients and roles cannot be stored this way. Storage and access to user data may be a factor in how you choose to configure authorization policies.



NOTE

[Super users](#) always have unconstrained access to a Kafka broker regardless of the authorization implemented on the Kafka broker.

Prerequisites

- AMQ Streams must be configured to use OAuth 2.0 with Red Hat Single Sign-On for [token-based authentication](#). You use the same Red Hat Single Sign-On server endpoint when you set up authorization.
- You need to understand how to manage policies and permissions for Red Hat Single Sign-On Authorization Services, as described in the [Red Hat Single Sign-On documentation](#).

Procedure

1. Access the Red Hat Single Sign-On Admin Console or use the Red Hat Single Sign-On Admin CLI to enable Authorization Services for the Kafka broker client you created when setting up OAuth 2.0 authentication.
2. Use Authorization Services to define resources, authorization scopes, policies, and permissions for the client.
3. Bind the permissions to users and clients by assigning them roles and groups.
4. Configure the Kafka brokers to use Red Hat Single Sign-On authorization. Add the following to the Kafka **server.properties** configuration file to install the authorizer in Kafka:

```
authorizer.class.name=io.strimzi.kafka.oauth.server.authorizer.KeycloakAuthorizer
principal.builder.class=io.strimzi.kafka.oauth.server.OAuthKafkaPrincipalBuilder
```

5. Add configuration for the Kafka brokers to access the authorization server and Authorization Services.

Here we show example configuration added as additional properties to **server.properties**, but you can also define them as environment variables using capitalized or upper-case naming conventions.

```
strimzi.authorization.token.endpoint.uri="https://<auth_server_address>/auth/realms/REALM-NAME/protocol/openid-connect/token" 1
strimzi.authorization.client.id="kafka" 2
```

- 1** The OAuth 2.0 token endpoint URL to Red Hat Single Sign-On. For production, always use **https://** urls.

- 2 The client ID of the OAuth 2.0 client definition in Red Hat Single Sign-On that has Authorization Services enabled. Typically, **kafka** is used as the ID.

6. (Optional) Add configuration for specific Kafka clusters.
For example:

```
strimzi.authorization.kafka.cluster.name="kafka-cluster" 1
```

- 1 The name of a specific Kafka cluster. Names are used to target permissions, making it possible to manage multiple clusters within the same Red Hat Single Sign-On realm. The default value is **kafka-cluster**.

7. (Optional) Delegate to simple authorization:

```
strimzi.authorization.delegate.to.kafka.acl="true" 1
```

- 1 Delegate authorization to Kafka **AclAuthorizer** if access is denied by Red Hat Single Sign-On Authorization Services policies. The default is **false**.

8. (Optional) Add configuration for TLS connection to the authorization server.
For example:

```
strimzi.authorization.ssl.truststore.location=<path_to_truststore> 1
strimzi.authorization.ssl.truststore.password=<my_truststore_password> 2
strimzi.authorization.ssl.truststore.type=JKS 3
strimzi.authorization.ssl.secure.random.implementation=SHA1PRNG 4
strimzi.authorization.ssl.endpoint.identification.algorithm=HTTPS 5
```

- 1 The path to the truststore that contain the certificates.
- 2 The password for the truststore.
- 3 The truststore type. If not set, the default Java keystore type is used.
- 4 Random number generator implementation. If not set, the Java platform SDK default is used.
- 5 Hostname verification. If set to an empty string, the hostname verification is turned off. If not set, the default value is **HTTPS**, which enforces hostname verification for server certificates.

9. (Optional) Configure the refresh of grants from the authorization server. The grants refresh job works by enumerating the active tokens and requesting the latest grants for each.
For example:

```
strimzi.authorization.grants.refresh.period.seconds="120" 1
strimzi.authorization.grants.refresh.pool.size="10" 2
strimzi.authorization.grants.max.idle.time.seconds="300" 3
strimzi.authorization.grants.gc.period.seconds="300" 4
strimzi.authorization.reuse.grants="false" 5
```

-
- 1 Specifies how often the list of grants from the authorization server is refreshed (once per minute by default). To turn grants refresh off for debugging purposes, set to **"0"**.
- 2 Specifies the size of the thread pool (the degree of parallelism) used by the grants refresh job. The default value is **"5"**.
- 3 The time, in seconds, after which an idle grant in the cache can be evicted. The default value is 300.
- 4 The time, in seconds, between consecutive runs of a job that cleans stale grants from the cache. The default value is 300.
- 5 Controls whether the latest grants are fetched for a new session. When disabled, grants are retrieved from Red Hat Single Sign-On and cached for the user. The default value is **true**.

10. (Optional) Configure network timeouts when communicating with the authorization server. For example:

```
strimzi.authorization.connect.timeout.seconds="60" 1
strimzi.authorization.read.timeout.seconds="60" 2
strimzi.authorization.http.retries="2" 3
```

- 1 The connect timeout in seconds when connecting to the Red Hat Single Sign-On token endpoint. The default value is **60**.
- 2 The read timeout in seconds when connecting to the Red Hat Single Sign-On token endpoint. The default value is **60**.
- 3 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 **strimzi.authorization.connect.timeout.seconds** and **strimzi.authorization.read.timeout.seconds** 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.

11. (Optional) Enable OAuth 2.0 metrics for token validation and authorization:

```
oauth.enable.metrics="true" 1
```

- 1 Controls whether to enable or disable OAuth metrics. The default value is **false**.

12. (Optional) Remove the **Accept** header from requests:

```
oauth.include.accept.header="false" 1
```

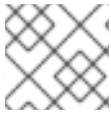
- 1 Set to **false** if including the header is causing issues when communicating with the authorization server. The default value is **true**.

13. Verify the configured permissions by accessing Kafka brokers as clients or users with specific roles, making sure they have the necessary access, or do not have the access they are not supposed to have.

6.4.11. Using OPA policy-based authorization

Open Policy Agent (OPA) is an open-source policy engine. You can integrate OPA with AMQ Streams to act as a policy-based authorization mechanism for permitting client operations on Kafka brokers.

When a request is made from a client, OPA will evaluate the request against policies defined for Kafka access, then allow or deny the request.



NOTE

Red Hat does not support the OPA server.

Additional resources

- [Open Policy Agent website](#)

6.4.11.1. Defining OPA policies

Before integrating OPA with AMQ Streams, consider how you will define policies to provide fine-grained access controls.

You can define access control for Kafka clusters, consumer groups and topics. For instance, you can define an authorization policy that allows write access from a producer client to a specific broker topic.

For this, the policy might specify the:

- **User principal** and **host address** associated with the producer client
- **Operations** allowed for the client
- **Resource type (topic)** and **resource name** the policy applies to

Allow and deny decisions are written into the policy, and a response is provided based on the request and client identification data provided.

In our example the producer client would have to satisfy the policy to be allowed to write to the topic.

6.4.11.2. Connecting to the OPA

To enable Kafka to access the OPA policy engine to query access control policies, you configure a custom OPA authorizer plugin (**kafka-authorizer-*OPA-VERSION*.jar**) in your Kafka **server.properties** file.

When a request is made by a client, the OPA policy engine is queried by the plugin using a specified URL address and a REST endpoint, which must be the name of the defined policy.

The plugin provides the details of the client request – user principal, operation, and resource – in JSON format to be checked against the policy. The details will include the unique identity of the client; for example, taking the distinguished name from the client certificate if TLS authentication is used.

OPA uses the data to provide a response – either *true* or *false* – to the plugin to allow or deny the request.

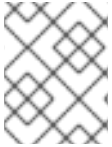
6.4.11.3. Configuring OPA authorization support

This procedure describes how to configure Kafka brokers to use OPA authorization.

Before you begin

Consider the access you require or want to limit for certain users. You can use a combination of *users* and Kafka *resources* to define OPA policies.

It is possible to set up OPA to load user information from an LDAP data source.



NOTE

[Super users](#) always have unconstrained access to a Kafka broker regardless of the authorization implemented on the Kafka broker.

Prerequisites

- An OPA server must be available for connection.
- [OPA authorizer plugin for Kafka](#)

Procedure

1. Write the OPA policies required for authorizing client requests to perform operations on the Kafka brokers.
See [Defining OPA policies](#).

Now configure the Kafka brokers to use OPA.

2. Install the [OPA authorizer plugin for Kafka](#) .
See [Connecting to the OPA](#).

Make sure that the plugin files are included in the Kafka classpath.

3. Add the following to the Kafka **server.properties** configuration file to enable the OPA plugin:

```
authorizer.class.name: com.bisnode.kafka.authorization.OpaAuthorizer
```

4. Add further configuration to **server.properties** for the Kafka brokers to access the OPA policy engine and policies.
For example:

```
opa.authorizer.url=https://OPA-ADDRESS/allow 1
opa.authorizer.allow.on.error=false 2
opa.authorizer.cache.initial.capacity=50000 3
opa.authorizer.cache.maximum.size=50000 4
opa.authorizer.cache.expire.after.seconds=600000 5
super.users=User:alice;User:bob 6
```


- 1 (Required) The OAuth 2.0 token endpoint URL for the policy the authorizer plugin will query. In this example, the policy is called **allow**.
- 2 Flag to specify whether a client is allowed or denied access by default if the authorizer plugin fails to connect with the OPA policy engine.
- 3 Initial capacity in bytes of the local cache. The cache is used so that the plugin does not have to query the OPA policy engine for every request.
- 4 Maximum capacity in bytes of the local cache.
- 5 Time in milliseconds that the local cache is refreshed by reloading from the OPA policy engine.
- 6 A list of user principals treated as super users, so that they are always allowed without querying the Open Policy Agent policy.

Refer to the [Open Policy Agent website](#) for information on authentication and authorization options.

5. Verify the configured permissions by accessing Kafka brokers using clients that have and do not have the correct authorization.

CHAPTER 7. CREATING AND MANAGING TOPICS

Messages in Kafka are always sent to or received from a topic. This chapter describes how to create and manage Kafka topics.

7.1. PARTITIONS AND REPLICAS

Messages in Kafka are always sent to or received from a topic. A topic is always split into one or more partitions. Partitions act as shards. That means that every message sent by a producer is always written only into a single partition. Thanks to the sharding of messages into different partitions, topics are easy to scale horizontally.

Each partition can have one or more replicas, which will be stored on different brokers in the cluster. When creating a topic you can configure the number of replicas using the *replication factor*. *Replication factor* defines the number of copies which will be held within the cluster. One of the replicas for given partition will be elected as a leader. The leader replica will be used by the producers to send new messages and by the consumers to consume messages. The other replicas will be follower replicas. The followers replicate the leader.

If the leader fails, one of the followers will automatically become the new leader. Each server acts as a leader for some of its partitions and a follower for others so the load is well balanced within the cluster.



NOTE

The replication factor determines the number of replicas including the leader and the followers. For example, if you set the replication factor to **3**, then there will one leader and two follower replicas.

7.2. MESSAGE RETENTION

The message retention policy defines how long the messages will be stored on the Kafka brokers. It can be defined based on time, partition size or both.

For example, you can define that the messages should be kept:

- For 7 days
- Until the partition has 1GB of messages. Once the limit is reached, the oldest messages will be removed.
- For 7 days or until the 1GB limit has been reached. Whatever limit comes first will be used.



WARNING

Kafka brokers store messages in log segments. The messages which are past their retention policy will be deleted only when a new log segment is created. New log segments are created when the previous log segment exceeds the configured log segment size. Additionally, users can request new segments to be created periodically.

Additionally, Kafka brokers support a compacting policy.

For a topic with the compacted policy, the broker will always keep only the last message for each key. The older messages with the same key will be removed from the partition. Because compacting is a periodically executed action, it does not happen immediately when the new message with the same key are sent to the partition. Instead it might take some time until the older messages are removed.

For more information about the message retention configuration options, see [Section 7.5, “Topic configuration”](#).

7.3. TOPIC AUTO-CREATION

When a producer or consumer tries to send messages to or receive messages from a topic that does not exist, Kafka will, by default, automatically create that topic. This behavior is controlled by the **auto.create.topics.enable** configuration property which is set to **true** by default.

To disable it, set **auto.create.topics.enable** to **false** in the Kafka broker configuration file:

```
auto.create.topics.enable=false
```

7.4. TOPIC DELETION

Kafka offers the possibility to disable deletion of topics. This is configured through the **delete.topic.enable** property, which is set to **true** by default (that is, deleting topics is possible). When this property is set to **false** it will be not possible to delete topics and all attempts to delete topic will return success but the topic will not be deleted.

```
delete.topic.enable=false
```

7.5. TOPIC CONFIGURATION

Auto-created topics will use the default topic configuration which can be specified in the broker properties file. However, when creating topics manually, their configuration can be specified at creation time. It is also possible to change a topic’s configuration after it has been created. The main topic configuration options for manually created topics are:

cleanup.policy

Configures the retention policy to **delete** or **compact**. The **delete** policy will delete old records. The **compact** policy will enable log compaction. The default value is **delete**. For more information about log compaction, see [Kafka website](#).

compression.type

Specifies the compression which is used for stored messages. Valid values are **gzip**, **snappy**, **lz4**, **uncompressed** (no compression) and **producer** (retain the compression codec used by the producer). The default value is **producer**.

max.message.bytes

The maximum size of a batch of messages allowed by the Kafka broker, in bytes. The default value is **100012**.

min.insync.replicas

The minimum number of replicas which must be in sync for a write to be considered successful. The default value is **1**.

retention.ms

Maximum number of milliseconds for which log segments will be retained. Log segments older than this value will be deleted. The default value is **604800000** (7 days).

retention.bytes

The maximum number of bytes a partition will retain. Once the partition size grows over this limit, the oldest log segments will be deleted. Value of **-1** indicates no limit. The default value is **-1**.

segment.bytes

The maximum file size of a single commit log segment file in bytes. When the segment reaches its size, a new segment will be started. The default value is **1073741824** bytes (1 gibibyte).

The defaults for auto-created topics can be specified in the Kafka broker configuration using similar options:

log.cleanup.policy

See **cleanup.policy** above.

compression.type

See **compression.type** above.

message.max.bytes

See **max.message.bytes** above.

min.insync.replicas

See **min.insync.replicas** above.

log.retention.ms

See **retention.ms** above.

log.retention.bytes

See **retention.bytes** above.

log.segment.bytes

See **segment.bytes** above.

default.replication.factor

Default replication factor for automatically created topics. Default value is **1**.

num.partitions

Default number of partitions for automatically created topics. Default value is **1**.

7.6. INTERNAL TOPICS

Internal topics are created and used internally by the Kafka brokers and clients. Kafka has several internal topics. These are used to store consumer offsets (**__consumer_offsets**) or transaction state (**__transaction_state**). These topics can be configured using dedicated Kafka broker configuration options starting with prefix **offsets.topic.** and **transaction.state.log.**. The most important configuration options are:

offsets.topic.replication.factor

Number of replicas for **__consumer_offsets** topic. The default value is **3**.

offsets.topic.num.partitions

Number of partitions for **__consumer_offsets** topic. The default value is **50**.

transaction.state.log.replication.factor

Number of replicas for **__transaction_state** topic. The default value is **3**.

transaction.state.log.num.partitions

Number of partitions for `__transaction_state` topic. The default value is **50**.

transaction.state.log.min.isr

Minimum number of replicas that must acknowledge a write to `__transaction_state` topic to be considered successful. If this minimum cannot be met, then the producer will fail with an exception. The default value is **2**.

7.7. CREATING A TOPIC

Use the **kafka-topics.sh** tool to manage topics. **kafka-topics.sh** is part of the AMQ Streams distribution and is found in the **bin** directory.

Prerequisites

- AMQ Streams cluster is installed and running

Creating a topic

1. Create a topic using the **kafka-topics.sh** utility and specify the following:

- Host and port of the Kafka broker in the **--bootstrap-server** option.
- The new topic to be created in the **--create** option.
- Topic name in the **--topic** option.
- The number of partitions in the **--partitions** option.
- Topic replication factor in the **--replication-factor** option.
You can also override some of the default topic configuration options using the option **--config**. This option can be used multiple times to override different options.

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --create --topic
<TopicName> --partitions <NumberOfPartitions> --replication-factor <ReplicationFactor>
--config <Option1>=<Value1> --config <Option2>=<Value2>
```

Example of the command to create a topic named **mytopic**

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --create --topic mytopic -
-partitions 50 --replication-factor 3 --config cleanup.policy=compact --config
min.insync.replicas=2
```

2. Verify that the topic exists using **kafka-topics.sh**.

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --describe --topic
<TopicName>
```

Example of the command to describe a topic named **mytopic**

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic mytopic
```

Additional resources

- [Topic configuration](#)

7.8. LISTING AND DESCRIBING TOPICS

The **kafka-topics.sh** tool can be used to list and describe topics. **kafka-topics.sh** is part of the AMQ Streams distribution and can be found in the **bin** directory.

Prerequisites

- AMQ Streams cluster is installed and running
- Topic **mytopic** exists

Describing a topic

1. Describe a topic using the **kafka-topics.sh** utility and specify the following:
 - Host and port of the Kafka broker in the **--bootstrap-server** option.
 - Use the **--describe** option to specify that you want to describe a topic.
 - Topic name must be specified in the **--topic** option.
 - When the **--topic** option is omitted, it will describe all available topics.

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --describe --topic  
<TopicName>
```

Example of the command to describe a topic named **mytopic**

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe --topic  
mytopic
```

The command lists all partitions and replicas which belong to this topic. It also lists all topic configuration options.

Additional resources

- [Topic configuration](#)
- [Creating a topic](#)

7.9. MODIFYING A TOPIC CONFIGURATION

The **kafka-configs.sh** tool can be used to modify topic configurations. **kafka-configs.sh** is part of the AMQ Streams distribution and can be found in the **bin** directory.

Prerequisites

- AMQ Streams cluster is installed and running
- Topic **mytopic** exists

Modify topic configuration

1. Use the **kafka-configs.sh** tool to get the current configuration.
 - Specify the host and port of the Kafka broker in the **--bootstrap-server** option.
 - Set the **--entity-type** as **topic** and **--entity-name** to the name of your topic.
 - Use **--describe** option to get the current configuration.

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --entity-type topics -
-entity-name <TopicName> --describe
```

Example of the command to get configuration of a topic named **mytopic**

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --
entity-name mytopic --describe
```

2. Use the **kafka-configs.sh** tool to change the configuration.
 - Specify the host and port of the Kafka broker in the **--bootstrap-server** option.
 - Set the **--entity-type** as **topic** and **--entity-name** to the name of your topic.
 - Use **--alter** option to modify the current configuration.
 - Specify the options you want to add or change in the option **--add-config**.

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --entity-type topics -
-entity-name <TopicName> --alter --add-config <Option>=<Value>
```

Example of the command to change configuration of a topic named **mytopic**

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --
entity-name mytopic --alter --add-config min.insync.replicas=1
```

3. Use the **kafka-configs.sh** tool to delete an existing configuration option.
 - Specify the host and port of the Kafka broker in the **--bootstrap-server** option.
 - Set the **--entity-type** as **topic** and **--entity-name** to the name of your topic.
 - Use **--delete-config** option to remove existing configuration option.
 - Specify the options you want to remove in the option **--remove-config**.

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --entity-type topics -
-entity-name <TopicName> --alter --delete-config <Option>
```

Example of the command to change configuration of a topic named **mytopic**

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --entity-type topics --
entity-name mytopic --alter --delete-config min.insync.replicas
```

Additional resources

- [Topic configuration](#)
- [Creating a topic](#)

7.10. DELETING A TOPIC

The **kafka-topics.sh** tool can be used to manage topics. **kafka-topics.sh** is part of the AMQ Streams distribution and can be found in the **bin** directory.

Prerequisites

- AMQ Streams cluster is installed and running
- Topic **mytopic** exists

Deleting a topic

1. Delete a topic using the **kafka-topics.sh** utility.
 - Host and port of the Kafka broker in the **--bootstrap-server** option.
 - Use the **--delete** option to specify that an existing topic should be deleted.
 - Topic name must be specified in the **--topic** option.

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --delete --topic <TopicName>
```

Example of the command to create a topic named **mytopic**

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic mytopic
```

2. Verify that the topic was deleted using **kafka-topics.sh**.

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --list
```

Example of the command to list all topics

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --list
```

Additional resources

- [Creating a topic](#)

CHAPTER 8. USING AMQ STREAMS WITH KAFKA CONNECT

Use Kafka Connect to stream data between Kafka and external systems. Kafka Connect provides a framework for moving large amounts of data while maintaining scalability and reliability. Kafka Connect is typically used to integrate Kafka with database, storage, and messaging systems that are external to your Kafka cluster.

Kafka Connect runs in standalone or distributed modes.

Standalone mode

In standalone mode, Kafka Connect runs on a single node. Standalone mode is intended for development and testing.

Distributed mode

In distributed mode, Kafka Connect runs across one or more worker nodes and the workloads are distributed among them. Distributed mode is intended for production.

Kafka Connect uses connector plugins that implement connectivity for different types of external systems. There are two types of connector plugins: sink and source. Sink connectors stream data from Kafka to external systems. Source connectors stream data from external systems into Kafka.

You can also use the Kafka Connect REST API to create, manage, and monitor connector instances.

Connector configuration specifies details such as the source or sink connectors and the Kafka topics to read from or write to. How you manage the configuration depends on whether you are running Kafka Connect in standalone or distributed mode.

- In standalone mode, you can provide the connector configuration as JSON through the Kafka Connect REST API or you can use properties files to define the configuration.
- In distributed mode, you can only provide the connector configuration as JSON through the Kafka Connect REST API.

Handling high volumes of messages

You can tune the configuration to handle high volumes of messages. For more information, see [Chapter 11, Handling high volumes of messages](#).

8.1. USING KAFKA CONNECT IN STANDALONE MODE

In Kafka Connect standalone mode, connectors run on the same node as the Kafka Connect worker process, which runs as a single process in a single JVM. This means that the worker process and connectors share the same resources, such as CPU, memory, and disk.

8.1.1. Configuring Kafka Connect in standalone mode

To configure Kafka Connect in standalone mode, edit the **config/connect-standalone.properties** configuration file. The following options are the most important.

bootstrap.servers

A list of Kafka broker addresses used as bootstrap connections to Kafka. For example, **kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092**.

key.converter

The class used to convert message keys to and from Kafka format. For example, **org.apache.kafka.connect.json.JsonConverter**.

value.converter

The class used to convert message payloads to and from Kafka format. For example, **org.apache.kafka.connect.json.JsonConverter**.

offset.storage.file.filename

Specifies the file in which the offset data is stored.

Connector plugins open client connections to the Kafka brokers using the bootstrap address. To configure these connections, use the standard Kafka producer and consumer configuration options prefixed by **producer.** or **consumer.**

8.1.2. Running Kafka Connect in standalone mode

Configure and run Kafka Connect in standalone mode.

Prerequisites

- AMQ Streams is installed and a Kafka cluster is running.
- You have specified connector configuration in properties files.
You can also use the Kafka Connect REST API to [manage connectors](#).

Procedure

1. Edit the **/opt/kafka/config/connect-standalone.properties** Kafka Connect configuration file and set **bootstrap.server** to point to your Kafka brokers. For example:

```
bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092
```

2. Start Kafka Connect with the configuration file and specify one or more connector configurations.

```
su - kafka
/opt/kafka/bin/connect-standalone.sh /opt/kafka/config/connect-standalone.properties
connector1.properties
[connector2.properties ...]
```

3. Verify that Kafka Connect is running.

```
jcmd | grep ConnectStandalone
```

8.2. USING KAFKA CONNECT IN DISTRIBUTED MODE

In distributed mode, Kafka Connect runs as a cluster of worker processes, with each worker running on a separate node. Connectors can run on any worker in the cluster, allowing for greater scalability and fault tolerance. The connectors are managed by the workers, which coordinate with each other to distribute the work and ensure that each connector is running on a single node at any given time.

8.2.1. Configuring Kafka Connect in distributed mode

To configure Kafka Connect in distributed mode, edit the **config/connect-distributed.properties** configuration file. The following options are the most important.

bootstrap.servers

A list of Kafka broker addresses used as bootstrap connections to Kafka. For example, **kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-domain.com:9092**.

key.converter

The class used to convert message keys to and from Kafka format. For example, **org.apache.kafka.connect.json.JsonConverter**.

value.converter

The class used to convert message payloads to and from Kafka format. For example, **org.apache.kafka.connect.json.JsonConverter**.

group.id

The name of the distributed Kafka Connect cluster. This must be unique and must not conflict with another consumer group ID. The default value is **connect-cluster**.

config.storage.topic

The Kafka topic used to store connector configurations. The default value is **connect-configs**.

offset.storage.topic

The Kafka topic used to store offsets. The default value is **connect-offset**.

status.storage.topic

The Kafka topic used for worker node statuses. The default value is **connect-status**.

AMQ Streams includes an example configuration file for Kafka Connect in distributed mode – see **config/connect-distributed.properties** in the AMQ Streams installation directory.

Connector plugins open client connections to the Kafka brokers using the bootstrap address. To configure these connections, use the standard Kafka producer and consumer configuration options prefixed by **producer.** or **consumer.**

8.2.2. Running Kafka Connect in distributed mode

Configure and run Kafka Connect in distributed mode.

Prerequisites

- AMQ Streams is installed and a Kafka cluster is running.

Running the cluster

1. Edit the **/opt/kafka/config/connect-distributed.properties** Kafka Connect configuration file on all Kafka Connect worker nodes.
 - Set the **bootstrap.server** option to point to your Kafka brokers.
 - Set the **group.id** option.
 - Set the **config.storage.topic** option.
 - Set the **offset.storage.topic** option.
 - Set the **status.storage.topic** option.
For example:

```
bootstrap.servers=kafka0.my-domain.com:9092,kafka1.my-domain.com:9092,kafka2.my-
```

```
domain.com:9092
group.id=my-group-id
config.storage.topic=my-group-id-configs
offset.storage.topic=my-group-id-offsets
status.storage.topic=my-group-id-status
```

2. Start the Kafka Connect workers with the `/opt/kafka/config/connect-distributed.properties` configuration file on all Kafka Connect nodes.

```
su - kafka
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

3. Verify that Kafka Connect is running.

```
jcmd | grep ConnectDistributed
```

4. Use the Kafka Connect REST API to [manage connectors](#).

8.3. MANAGING CONNECTORS

The Kafka Connect REST API provides endpoints for creating, updating, and deleting connectors directly. You can also use the API to check the status of connectors or change logging levels. When you create a connector through the API, you provide the configuration details for the connector as part of the API call.

You can also add and manage connectors as plugins. Plugins are packaged as JAR files that contain the classes to implement the connectors through the Kafka Connect API. You just need to specify the plugin in the classpath or add it to a plugin path for Kafka Connect to run the connector plugin on startup.

In addition to using the Kafka Connect REST API or plugins to manage connectors, you can also add connector configuration using properties files when running Kafka Connect in standalone mode. To do this, you simply specify the location of the properties file when starting the Kafka Connect worker process. The properties file should contain the configuration details for the connector, including the connector class, source and destination topics, and any required authentication or serialization settings.

8.3.1. Limiting access to the Kafka Connect API

The Kafka Connect REST API can be accessed by anyone who has authenticated access and knows the endpoint URL, which includes the hostname/IP address and port number. It is crucial to restrict access to the Kafka Connect API only to trusted users to prevent unauthorized actions and potential security issues.

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

- (Kafka 3.4 or later) **org.apache.kafka.disallowed.login.modules** to specifically exclude insecure login modules
- **connector.client.config.override.policy** set to **NONE** to prevent connector configurations from overriding the Kafka Connect configuration and the consumers and producers it uses

8.3.2. Configuring connectors

Use the Kafka Connect REST API or properties files to create, manage, and monitor connector instances. You can use the REST API when using Kafka Connect in standalone or distributed mode. You can use properties files when using Kafka Connect in standalone mode.

8.3.2.1. Using the Kafka Connect REST API to manage connectors

When using the Kafka Connect REST API, you can create connectors dynamically by sending **PUT** or **POST** HTTP requests to the Kafka Connect REST API, specifying the connector configuration details in the request body.

TIP

When you use the **PUT** command, it's the same command for starting and updating connectors.

The REST interface listens on port 8083 by default and supports the following endpoints:

GET /connectors

Return a list of existing connectors.

POST /connectors

Create a connector. The request body has to be a JSON object with the connector configuration.

GET /connectors/<connector_name>

Get information about a specific connector.

GET /connectors/<connector_name>/config

Get configuration of a specific connector.

PUT /connectors/<connector_name>/config

Update the configuration of a specific connector.

GET /connectors/<connector_name>/status

Get the status of a specific connector.

GET /connectors/<connector_name>/tasks

Get a list of tasks for a specific connector

GET /connectors/<connector_name>/tasks/<task_id>/status

Get the status of a task for a specific connector

PUT /connectors/<connector_name>/pause

Pause the connector and all its tasks. The connector will stop processing any messages.

PUT /connectors/<connector_name>/stop

Stop the connector and all its tasks. The connector will stop processing any messages. Stopping a connector from running may be more suitable for longer durations than just pausing.

PUT /connectors/<connector_name>/resume

Resume a paused connector.

POST /connectors/<connector_name>/restart

Restart a connector in case it has failed.

POST /connectors/<connector_name>/tasks/<task_id>/restart

Restart a specific task.

DELETE /connectors/<connector_name>

Delete a connector.

GET /connectors/<connector_name>/topics

Get the topics for a specific connector.

PUT /connectors/<connector_name>/topics/reset

Empty the set of active topics for a specific connector.

GET /connectors/<connector_name>/offsets

Get the current offsets for a connector.

DELETE /connectors/<connector_name>/offsets

Reset the offsets for a connector, which must be in a stopped state.

PATCH /connectors/<connector_name>/offsets

Adjust the offsets (using an **offset** property in the request) for a connector, which must be in a stopped state.

GET /connector-plugins

Get a list of all supported connector plugins.

GET /connector-plugins/<connector_plugin_type>/config

Get the configuration for a connector plugin.

PUT /connector-plugins/<connector_type>/config/validate

Validate connector configuration.

8.3.2.2. Specifying connector configuration properties

To configure a Kafka Connect connector, you need to specify the configuration details for source or sink connectors. There are two ways to do this: through the Kafka Connect REST API, using JSON to provide the configuration, or by using properties files to define the configuration properties. The specific configuration options available for each type of connector may differ, but both methods provide a flexible way to specify the necessary settings.

The following options apply to all connectors:

name

The name of the connector, which must be unique within the current Kafka Connect instance.

connector.class

The class of the connector plug-in. For example, **org.apache.kafka.connect.file.FileStreamSinkConnector**.

tasks.max

The maximum number of tasks that the specified connector can use. Tasks enable the connector to perform work in parallel. The connector might create fewer tasks than specified.

key.converter

The class used to convert message keys to and from Kafka format. This overrides the default value set by the Kafka Connect configuration. For example, **org.apache.kafka.connect.json.JsonConverter**.

value.converter

The class used to convert message payloads to and from Kafka format. This overrides the default value set by the Kafka Connect configuration. For example, **org.apache.kafka.connect.json.JsonConverter**.

You must set at least one of the following options for sink connectors:

topics

A comma-separated list of topics used as input.

topics.regex

A Java regular expression of topics used as input.

For all other options, see the connector properties in the [Apache Kafka documentation](#).



NOTE

AMQ Streams includes the example connector configuration files **config/connect-file-sink.properties** and **config/connect-file-source.properties** in the AMQ Streams installation directory.

Additional resources

- [Kafka Connect REST API OpenAPI documentation](#)

8.3.3. Creating connectors using the Kafka Connect API

Use the Kafka Connect REST API to create a connector to use with Kafka Connect.

Prerequisites

- A Kafka Connect installation.

Procedure

1. Prepare a JSON payload with the connector configuration. For example:

```
{
  "name": "my-connector",
  "config": {
    "connector.class": "org.apache.kafka.connect.file.FileStreamSinkConnector",
    "tasks.max": "1",
    "topics": "my-topic-1,my-topic-2",
    "file": "/tmp/output-file.txt"
  }
}
```

2. Send a POST request to **<KafkaConnectAddress>:8083/connectors** to create the connector. The following example uses **curl**:

```
curl -X POST -H "Content-Type: application/json" --data @sink-connector.json
http://connect0.my-domain.com:8083/connectors
```

3. Verify that the connector was deployed by sending a GET request to **<KafkaConnectAddress>:8083/connectors**. The following example uses **curl**:

```
curl http://connect0.my-domain.com:8083/connectors
```

8.3.4. Deleting connectors using the Kafka Connect API

Use the Kafka Connect REST API to delete a connector from Kafka Connect.

Prerequisites

- A Kafka Connect installation.

Deleting connectors

1. Verify that the connector exists by sending a **GET** request to **<KafkaConnectAddress>:8083/connectors/<ConnectorName>**. The following example uses **curl**:

```
curl http://connect0.my-domain.com:8083/connectors
```

2. To delete the connector, send a **DELETE** request to **<KafkaConnectAddress>:8083/connectors**. The following example uses **curl**:

```
curl -X DELETE http://connect0.my-domain.com:8083/connectors/my-connector
```

3. Verify that the connector was deleted by sending a GET request to **<KafkaConnectAddress>:8083/connectors**. The following example uses **curl**:

```
curl http://connect0.my-domain.com:8083/connectors
```

8.3.5. Adding connector plugins

Kafka provides example connectors to use as a starting point for developing connectors. The following example connectors are included with AMQ Streams:

FileStreamSink

Reads data from Kafka topics and writes the data to a file.

FileStreamSource

Reads data from a file and sends the data to Kafka topics.

Both connectors are contained in the **libs/connect-file-<kafka_version>.redhat-<build>.jar** plugin.

To use the connector plugins in Kafka Connect, you can add them to the classpath or specify a plugin path in the Kafka Connect properties file and copy the plugins to the location.

Specifying the example connectors in the classpath

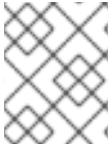
```
CLASSPATH=/opt/kafka/libs/connect-file-<kafka_version>.redhat-<build>.jar opt/kafka/bin/connect-distributed.sh
```

Setting a plugin path

```
plugin.path=/opt/kafka/connector-plugins,/opt/connectors
```

The **plugin.path** configuration option can contain a comma-separated list of paths.

You can add more connector plugins if needed. Kafka Connect searches for and runs connector plugins at startup.

**NOTE**

When running Kafka Connect in distributed mode, plugins must be made available on all worker nodes.

CHAPTER 9. USING AMQ STREAMS WITH MIRRORMAKER 2

Use MirrorMaker 2 to replicate data between two or more active Kafka clusters, within or across data centers.

To configure MirrorMaker 2, edit the **config/connect-mirror-maker.properties** configuration file. If required, you can [enable distributed tracing for MirrorMaker 2](#).

Handling high volumes of messages

You can tune the configuration to handle high volumes of messages. For more information, see [Chapter 11, Handling high volumes of messages](#).



NOTE

MirrorMaker 2 has features not supported by the previous version of MirrorMaker. However, you can [configure MirrorMaker 2 to be used in legacy mode](#).

9.1. 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.

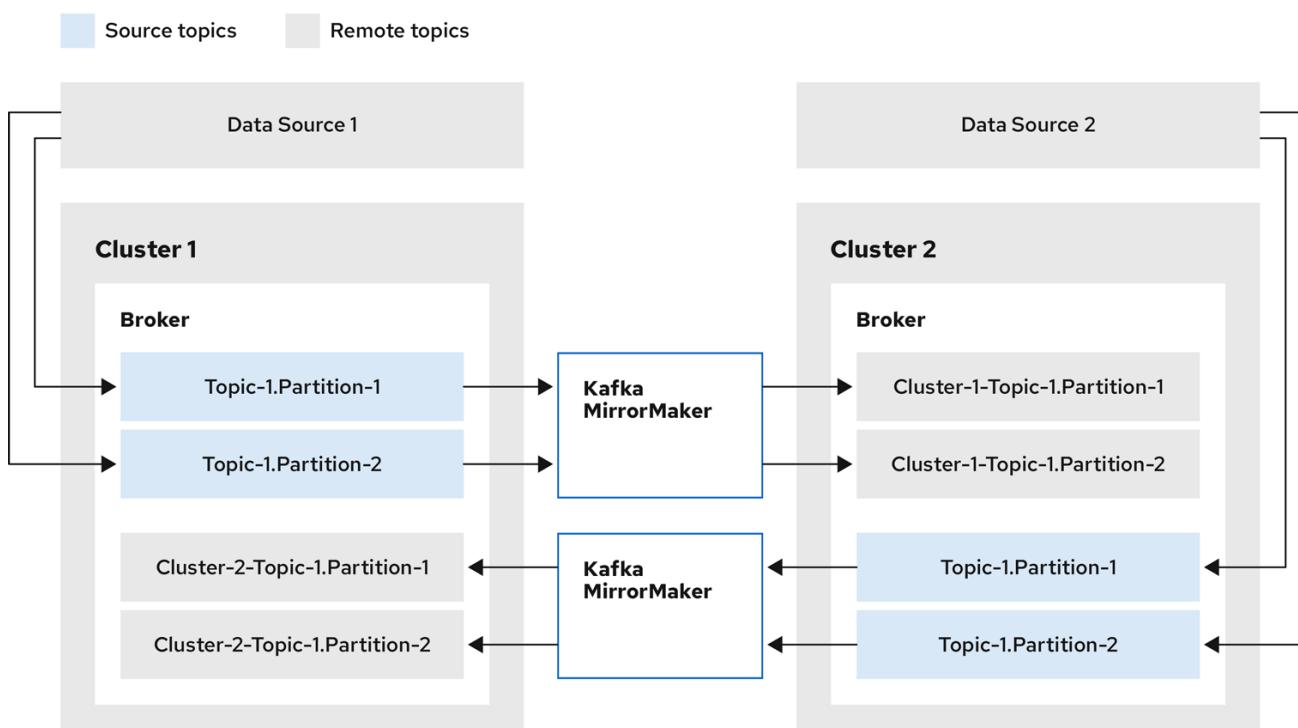
The expectation is that producers and consumers connect to active clusters only. A MirrorMaker 2 cluster is required at each target destination.

9.1.1. 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.

Figure 9.1. Topic renaming



222_Streams_0322

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.

9.1.2. 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.

9.2. 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 9.1. 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.	✓	✓	

Property	sourceConnector	checkpointConnector	heartbeatConnector
topic.filter.class Topic filter to select the topics to replicate. Default is org.apache.kafka.connect.mirror.DefaultTopicFilter .	✓	✓	
config.property.filter.class Topic filter to select the topic configuration properties to replicate. Default is org.apache.kafka.connect.mirror.DefaultConfigPropertyFilter .	✓		
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 100 .	✓		
offset-syncs.topic.replication.factor Replication factor for the internal offset-syncs topic. Default is 3 .	✓		
refresh.topics.enabled Enables check for new topics and partitions. Default is true .	✓		
refresh.topics.interval.seconds Frequency of topic refresh. Default is 600 (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 refresh.topics.interval.seconds to the source connector configuration.	✓		

Property	sourceConnector	checkpointConnector	heartbeatConnector
replication.factor The replication factor for new topics. Default is 2 .	✓		
sync.topic.acls.enabled Enables synchronization of ACLs from the source cluster. Default is true . For more information, see Section 9.5, "ACL rules synchronization" .	✓		
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).		✓	

Property	sourceConnector	checkpointConnector	heartbeatConnector
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 .		✓	
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 .			✓

9.2.1. 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.

9.2.2. 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.

9.2.3. 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.

9.2.4. 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.

9.3. CONNECTOR PRODUCER AND CONSUMER CONFIGURATION

MirrorMaker 2 connectors use internal producers and consumers. If needed, you can configure these producers and consumers to override the default settings.

**IMPORTANT**

Producer and consumer configuration options depend on the MirrorMaker 2 implementation, and may be subject to change.

Producer and consumer configuration applies to **all** connectors. You specify the configuration in the **config/connect-mirror-maker.properties** file.

Use the properties file to override any default configuration for the producers and consumers in the following format:

- `<source_cluster_name>.consumer.<property>`
- `<source_cluster_name>.producer.<property>`
- `<target_cluster_name>.consumer.<property>`
- `<target_cluster_name>.producer.<property>`

The following example shows how you configure the producers and consumers. Though the properties are set for all connectors, some configuration properties are only relevant to certain connectors.

Example configuration for connector producers and consumers

```
clusters=cluster-1,cluster-2

# ...
cluster-1.consumer.fetch.max.bytes=52428800
cluster-2.producer.batch.size=327680
cluster-2.producer.linger.ms=100
cluster-2.producer.request.timeout.ms=30000
```

9.4. SPECIFYING A MAXIMUM NUMBER OF TASKS

Connectors create the tasks that are responsible for moving data in and out of Kafka. Each connector comprises one or more tasks that are distributed across a group of worker pods that run the tasks. Increasing the number of tasks can help with performance issues when replicating a large number of partitions or synchronizing the offsets of a large number of consumer groups.

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

You can specify the maximum number of connector tasks in your MirrorMaker configuration using the **tasks.max** 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 **tasks.max**. 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.

tasks.max configuration for MirrorMaker connectors

```
clusters=cluster-1,cluster-2
# ...
tasks.max = 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.

9.5. ACL RULES SYNCHRONIZATION

If **AclAuthorizer** is being used, ACL rules that manage access to brokers also apply to remote topics. Users that can read a source topic can read its remote equivalent.



NOTE

OAuth 2.0 authorization does not support access to remote topics in this way.

9.6. RUNNING MIRRORMAKER 2 IN DEDICATED MODE

Use MirrorMaker 2 to synchronize data between Kafka clusters through configuration. This procedure shows how to configure and run a dedicated single-node MirrorMaker 2 cluster. Dedicated clusters use Kafka Connect worker nodes to mirror data between Kafka clusters. At present, MirrorMaker 2 in dedicated mode only works with a single worker node.



NOTE

It is also possible to run MirrorMaker 2 in distributed mode. In distributed mode, MirrorMaker 2 runs as connectors in a Kafka Connect cluster. Kafka provides MirrorMaker source connectors for data replication. If you wish to use the connectors instead of running a dedicated MirrorMaker cluster, the connectors must be configured in the Kafka Connect cluster. For more information, refer to the [Apache Kafka documentation](#).

The previous version of MirrorMaker continues to be supported, by [running MirrorMaker 2 in legacy mode](#).

The configuration must specify:

- Each Kafka cluster
- Connection information for each cluster, including TLS authentication
- The replication flow and direction
 - Cluster to cluster
 - Topic to topic
- Replication rules
- Committed offset tracking intervals

This procedure describes how to implement MirrorMaker 2 by creating the configuration in a properties file, then passing the properties when using the MirrorMaker script file to set up the connections.

You can specify the topics and consumer groups you wish to replicate from a source cluster. You specify the names of the source and target clusters, then specify the topics and consumer groups to replicate.

In the following example, topics and consumer groups are specified for replication from cluster 1 to 2.

Example configuration to replicate specific topics and consumer groups

```
clusters=cluster-1,cluster-2
cluster-1->cluster-2.topics = topic-1, topic-2
cluster-1->cluster-2.groups = group-1, group-2
```

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 these 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.

Before you begin

A sample configuration properties file is provided in `./config/connect-mirror-maker.properties`.

Prerequisites

- You need AMQ Streams installed on the hosts of each Kafka cluster node you are replicating.

Procedure

- Open the sample properties file in a text editor, or create a new one, and edit the file to include connection information and the replication flows for each Kafka cluster. The following example shows a configuration to connect two clusters, `cluster-1` and `cluster-2`, bidirectionally. Cluster names are configurable through the `clusters` property.

Example MirrorMaker 2 configuration

```
clusters=cluster-1,cluster-2 1

cluster-1.bootstrap.servers=<cluster_name>-kafka-bootstrap-<project_name_one>:443 2
cluster-1.security.protocol=SSL 3
cluster-1.ssl.truststore.password=<truststore_name>
cluster-1.ssl.truststore.location=<path_to_truststore>/truststore.cluster-1.jks_
cluster-1.ssl.keystore.password=<keystore_name>
cluster-1.ssl.keystore.location=<path_to_keystore>/user.cluster-1.p12

cluster-2.bootstrap.servers=<cluster_name>-kafka-bootstrap-<project_name_two>:443 4
cluster-2.security.protocol=SSL 5
cluster-2.ssl.truststore.password=<truststore_name>
cluster-2.ssl.truststore.location=<path_to_truststore>/truststore.cluster-2.jks_
cluster-2.ssl.keystore.password=<keystore_name>
cluster-2.ssl.keystore.location=<path_to_keystore>/user.cluster-2.p12

cluster-1->cluster-2.enabled=true 6
cluster-2->cluster-1.enabled=true 7
cluster-1->cluster-2.topics=.* 8
cluster-2->cluster-1.topics=topic-1, topic-2 9
cluster-1->cluster-2.groups=.* 10
cluster-2->cluster-1.groups=group-1, group-2 11

replication.policy.separator=- 12
```

```
sync.topic.acls.enabled=false 13
refresh.topics.interval.seconds=60 14
refresh.groups.interval.seconds=60 15
```

- 1** Each Kafka cluster is identified with its alias.
 - 2** Connection information for *cluster-1*, using the *bootstrap* address and port 443. Both clusters use port 443 to connect to Kafka using OpenShift *Routes*.
 - 3** The **ssl.** properties define TLS configuration for *cluster-1*.
 - 4** Connection information for *cluster-2*.
 - 5** The **ssl.** properties define the TLS configuration for *cluster-2*.
 - 6** Replication flow enabled from *cluster-1* to *cluster-2*.
 - 7** Replication flow enabled from *cluster-2* to *cluster-1*.
 - 8** Replication of all topics from *cluster-1* to *cluster-2*. The source connector replicates the specified topics. The checkpoint connector tracks offsets for the specified topics.
 - 9** Replication of specific topics from *cluster-2* to *cluster-1*.
 - 10** Replication of all consumer groups from *cluster-1* to *cluster-2*. The checkpoint connector replicates the specified consumer groups.
 - 11** Replication of specific consumer groups from *cluster-2* to *cluster-1*.
 - 12** Defines the separator used for the renaming of remote topics.
 - 13** When enabled, ACLs are applied to synchronized topics. The default is **false**.
 - 14** The period between checks for new topics to synchronize.
 - 15** The period between checks for new consumer groups to synchronize.
2. OPTION: If required, add 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 used for active/passive backups and data migration.

```
replication.policy.class=org.apache.kafka.connect.mirror.IdentityReplicationPolicy
```

3. OPTION: If you want to synchronize consumer group offsets, add configuration to enable and manage the synchronization:

```
refresh.groups.interval.seconds=60
sync.group.offsets.enabled=true 1
sync.group.offsets.interval.seconds=60 2
emit.checkpoints.interval.seconds=60 3
```

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

- 2 If the synchronization of consumer group offsets is enabled, you can adjust the frequency of the synchronization.
- 3 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.

4. Start ZooKeeper and Kafka in the target clusters:

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon \
/opt/kafka/config/zookeeper.properties
```

```
/opt/kafka/bin/kafka-server-start.sh -daemon \
/opt/kafka/config/server.properties
```

5. Start MirrorMaker with the cluster connection configuration and replication policies you defined in your properties file:

```
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

MirrorMaker sets up connections between the clusters.

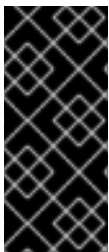
6. For each target cluster, verify that the topics are being replicated:

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --list
```

9.7. USING MIRRORMAKER 2 IN LEGACY MODE

This procedure describes how to configure MirrorMaker 2 to use it in legacy mode. Legacy mode supports the previous version of MirrorMaker.

The MirrorMaker script `/opt/kafka/bin/kafka-mirror-maker.sh` can run MirrorMaker 2 in legacy mode.



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, Kafka MirrorMaker 1 has been deprecated in AMQ Streams as well. Kafka MirrorMaker 1 will be removed from AMQ Streams when we adopt Apache Kafka 4.0.0. As a replacement, use MirrorMaker 2 with the [IdentityReplicationPolicy](#).

Prerequisites

You need the properties files you currently use with the legacy version of MirrorMaker.

- `/opt/kafka/config/consumer.properties`
- `/opt/kafka/config/producer.properties`

Procedure

1. Edit the MirrorMaker **consumer.properties** and **producer.properties** files to turn off MirrorMaker 2 features.

For example:

```
replication.policy.class=org.apache.kafka.mirror.LegacyReplicationPolicy 1
refresh.topics.enabled=false 2
refresh.groups.enabled=false
emit.checkpoints.enabled=false
emit.heartbeats.enabled=false
sync.topic.configs.enabled=false
sync.topic.acls.enabled=false
```

- 1 Emulate the previous version of MirrorMaker.
- 2 MirrorMaker 2 features disabled, including the internal *checkpoint* and *heartbeat* topics

2. Save the changes and restart MirrorMaker with the properties files you used with the previous version of MirrorMaker:

```
su - kafka /opt/kafka/bin/kafka-mirror-maker.sh \
--consumer.config /opt/kafka/config/consumer.properties \
--producer.config /opt/kafka/config/producer.properties \
--num.streams=2
```

The **consumer** properties provide the configuration for the source cluster and the **producer** properties provide the target cluster configuration.

MirrorMaker sets up connections between the clusters.

3. Start ZooKeeper and Kafka in the target cluster:

```
su - kafka
/opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties
```

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

4. For the target cluster, verify that the topics are being replicated:

```
/opt/kafka/bin/kafka-topics.sh --bootstrap-server <broker_address> --list
```

CHAPTER 10. CONFIGURING LOGGING FOR KAFKA COMPONENTS

Configure the logging levels of Kafka components directly in the configuration properties. You can also change the broker levels dynamically for Kafka brokers, Kafka Connect, and MirrorMaker 2.

Increasing the log level detail, such as from INFO to DEBUG, can aid in troubleshooting a Kafka cluster. However, more verbose logs may also negatively impact performance and make it more difficult to diagnose issues.

10.1. CONFIGURING KAFKA LOGGING PROPERTIES

Kafka components use the Log4j framework for error logging. By default, logging configuration is read from the classpath or **config** directory using the following properties files:

- **log4j.properties** for Kafka and ZooKeeper
- **connect-log4j.properties** for Kafka Connect and MirrorMaker 2

If they are not set explicitly, loggers inherit the **log4j.rootLogger** logging level configuration in each file. You can change the logging level in these files. You can also add and set logging levels for other loggers.

You can change the location and name of logging properties file using the **KAFKA_LOG4J_OPTS** environment variable, which is used by the start script for the component.

Passing the name and location of the logging properties file used by Kafka brokers

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/log4j.properties"; \
/opt/kafka/bin/kafka-server-start.sh \
/opt/kafka/config/server.properties
```

Passing the name and location of the logging properties file used by ZooKeeper

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/log4j.properties"; \
/opt/kafka/bin/zookeeper-server-start.sh -daemon \
/opt/kafka/config/zookeeper.properties
```

Passing the name and location of the logging properties file used by Kafka Connect

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/connect-log4j.properties"; \
/opt/kafka/bin/connect-distributed.sh \
/opt/kafka/config/connect-distributed.properties
```

Passing the name and location of the logging properties file used by MirrorMaker 2

```
su - kafka
export KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/my/path/to/connect-log4j.properties"; \
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```


10.2. DYNAMICALLY CHANGE LOGGING LEVELS FOR KAFKA BROKER LOGGERS

Kafka broker logging is provided by broker loggers in each broker. Dynamically change the logging level for broker loggers at runtime without having to restart the broker.

You can also reset broker loggers dynamically to their default logging levels.

Prerequisites

- [AMQ Streams is installed on the host](#)
- [ZooKeeper and Kafka are running](#)

Procedure

1. Switch to the **kafka** user:

```
su - kafka
```

2. List all the broker loggers for a broker by using the **kafka-configs.sh** tool:

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --describe --entity-type broker-loggers --entity-name BROKER-ID
```

For example, for broker **0**:

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --describe --entity-type broker-loggers --entity-name 0
```

This returns the logging level for each logger: **TRACE**, **DEBUG**, **INFO**, **WARN**, **ERROR**, or **FATAL**.

For example:

```
#...
kafka.controller.ControllerChannelManager=INFO sensitive=false synonyms={}
kafka.log.TimeIndex=INFO sensitive=false synonyms={}
```

3. Change the logging level for one or more broker loggers. Use the **--alter** and **--add-config** options and specify each logger and its level as a comma-separated list in double quotes.

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --alter --add-config "LOGGER-ONE=NEW-LEVEL,LOGGER-TWO=NEW-LEVEL" --entity-type broker-loggers --entity-name BROKER-ID
```

For example, for broker **0**:

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --add-config "kafka.controller.ControllerChannelManager=WARN,kafka.log.TimeIndex=WARN" --entity-type broker-loggers --entity-name 0
```

If successful this returns:

Completed updating config for broker: 0.

Resetting a broker logger

You can reset one or more broker loggers to their default logging levels by using the `kafka-configs.sh` tool. Use the `--alter` and `--delete-config` options and specify each broker logger as a comma-separated list in double quotes:

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server localhost:9092 --alter --delete-config "LOGGER-ONE,LOGGER-TWO" --entity-type broker-loggers --entity-name BROKER-ID
```

Additional resources

- [Updating Broker Configs](#) in the Apache Kafka documentation

10.3. DYNAMICALLY CHANGE LOGGING LEVELS FOR KAFKA CONNECT AND MIRRORMAKER 2

Dynamically change logging levels for Kafka Connect workers or MirrorMaker 2 connectors at runtime without having to restart.

Use the Kafka Connect API to change the log level temporarily for a worker or connector logger. The Kafka Connect API provides an `admin/loggers` endpoint to get or modify logging levels. When you change the log level using the API, the logger configuration in the `connect-log4j.properties` configuration file does not change. If required, you can permanently change the logging levels in the configuration file.



NOTE

You can only change the logging level of MirrorMaker 2 at runtime when in distributed or standalone mode. Dedicated MirrorMaker 2 clusters have no Kafka Connect REST API, so changing the logging level is not possible.

The default listener for the Kafka Connect API is on port 8083, which is used in this procedure. You can change or add more listeners, and also enable TLS authentication, using `admin.listeners` configuration.

Example listener configuration for the admin endpoint

```
admin.listeners=https://localhost:8083
admin.listeners.https.ssl.truststore.location=/path/to/truststore.jks
admin.listeners.https.ssl.truststore.password=123456
admin.listeners.https.ssl.keystore.location=/path/to/keystore.jks
admin.listeners.https.ssl.keystore.password=123456
```

If you do not want the `admin` endpoint to be available, you can disable it in the configuration by specifying an empty string.

Example listener configuration to disable the admin endpoint

```
admin.listeners=
```

Prerequisites

- [AMQ Streams is installed on the host](#)
- [ZooKeeper and Kafka are running](#)
- Kafka Connect or MirrorMaker 2 is running

Procedure

1. Switch to the **kafka** user:

```
su - kafka
```

2. Check the current logging level for the loggers configured in the **connect-log4j.properties** file:

```
$ cat /opt/kafka/config/connect-log4j.properties
# ...
log4j.rootLogger=INFO, stdout, connectAppender
# ...
log4j.logger.org.apache.zookeeper=ERROR
log4j.logger.org.reflections=ERROR
```

Use a curl command to check the logging levels from the **admin/loggers** endpoint of the Kafka Connect API:

```
curl -s http://localhost:8083/admin/loggers/ | jq
{
  "org.apache.zookeeper": {
    "level": "ERROR"
  },
  "org.reflections": {
    "level": "ERROR"
  },
  "root": {
    "level": "INFO"
  }
}
```

jq prints the output in JSON format. The list shows standard **org** and **root** level loggers, plus any specific loggers with modified logging levels.

If you configure TLS (Transport Layer Security) authentication for the **admin.listeners** configuration in Kafka Connect, then the address of the loggers endpoint is the value specified for **admin.listeners** with the protocol as https, such as **https://localhost:8083**.

You can also get the log level of a specific logger:

```
curl -s
http://localhost:8083/admin/loggers/org.apache.kafka.connect.mirror.MirrorCheckpointConnector | jq
{
  "level": "INFO"
}
```

-
3. Use a PUT method to change the log level for a logger:

```
curl -Ss -X PUT -H 'Content-Type: application/json' -d '{"level": "TRACE"}'  
http://localhost:8083/admin/loggers/root
```

```
{  
  # ...  
  
  "org.reflections": {  
    "level": "TRACE"  
  },  
  "org.reflections.Reflections": {  
    "level": "TRACE"  
  },  
  "root": {  
    "level": "TRACE"  
  }  
}
```

If you change the **root** logger, the logging level for loggers that used the root logging level by default are also changed.

CHAPTER 11. HANDLING HIGH VOLUMES OF MESSAGES

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

Kafka 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:

- [Optimizing producers for throughput and latency](#)
- [Optimizing consumers for throughput and latency](#)

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 consumer configuration (**consumer.***), 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 producer configuration (**producer.***), 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.

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 **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 **tasks.max** property
 - Increasing the number of nodes for the workers that run tasks

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.

11.1. 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 a Kafka Connect source connector.

Example source connector configuration for handling high volumes of messages

```
# ...
producer.batch.size=327680
producer.linger.ms=100
# ...
tasks.max = 2
```

Consumer configuration is added for the sink connector.

Example sink connector configuration for handling high volumes of messages

```
# ...
consumer.fetch.max.bytes=52428800
consumer.max.partition.fetch.bytes=1048576
consumer.max.poll.records=500
# ...
tasks.max = 2
```

11.2. 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. The configuration relates to the consumer that fetches messages from the source and the producer that sends messages to the target Kafka cluster.

Example MirrorMaker 2 configuration for handling high volumes of messages

```
clusters=cluster-1,cluster-2
# ...
cluster-2.producer.batch.size=327680
cluster-2.producer.linger.ms=100
cluster-1.consumer.fetch.max.bytes=52428800
cluster-1.consumer.max.partition.fetch.bytes=1048576
cluster-1.consumer.max.poll.records=500
# ...
tasks.max = 2
```

CHAPTER 12. SETTING LIMITS ON BROKERS USING THE KAFKA STATIC QUOTA PLUGIN



IMPORTANT

The Kafka Static Quota plugin is a Technology Preview only. Technology Preview features are not supported with Red Hat production service-level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend implementing any Technology Preview features in production environments. This Technology Preview feature provides early access to upcoming product innovations, enabling you to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see [Technology Preview Features Support Scope](#).

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 adding properties to the Kafka configuration file. You can set a byte-rate threshold and storage quotas to put limits on the clients interacting with your brokers.

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

Storage quotas throttle Kafka disk storage limits between a soft limit and hard limit. The limits apply to all available disk space. Producers are slowed gradually between the soft and hard limit. The limits prevent disks filling up too quickly and exceeding their capacity. Full disks can lead to issues that are hard to rectify. The hard limit is the maximum storage limit.



NOTE

For JBOD storage, the limit applies across all disks. If a broker is using two 1 TB disks and the quota is 1.1 TB, one disk might fill and the other disk will be almost empty.

Prerequisites

- AMQ Streams is [installed](#) on all hosts which will be used as Kafka brokers.
- A ZooKeeper cluster is [configured and running](#).

Procedure

1. Edit the `/opt/kafka/config/server.properties` Kafka configuration file. The plugin properties are shown in this example configuration.

Example Kafka Static Quota plugin configuration

```
# ...
client.quota.callback.class=io.strimzi.kafka.quotas.StaticQuotaCallback 1
client.quota.callback.static.produce=1000000 2
client.quota.callback.static.fetch=1000000 3
client.quota.callback.static.storage.soft=400000000000 4
```



```
client.quota.callback.static.storage.hard=500000000000 5  
client.quota.callback.static.storage.check-interval=5 6  
# ...
```

- 1 Loads the Kafka Static Quota plugin.
 - 2 Sets the producer byte-rate threshold. 1 MBps in this example.
 - 3 Sets the consumer byte-rate threshold. 1 MBps in this example.
 - 4 Sets the lower soft limit for storage. 400 GB in this example.
 - 5 Sets the higher hard limit for storage. 500 GB in this example.
 - 6 Sets the interval in seconds between checks on storage. 5 seconds in this example. You can set this to 0 to disable the check.
2. Start the Kafka broker with the default configuration file.

```
su - kafka  
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

3. Verify that the Kafka broker is running.

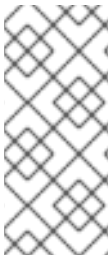
```
jcmd | grep Kafka
```

CHAPTER 13. ADDING AND REMOVING KAFKA BROKERS AND ZOOKEEPER NODES

In a Kafka cluster, managing the addition and removal of brokers and ZooKeeper nodes is critical to maintaining a stable and scalable system. When you add to the number of available brokers, you can configure the default replication factor and minimum in-sync replicas for topics across the brokers. You can use dynamic reconfiguration to add and remove ZooKeeper nodes from an ensemble without disruption.

13.1. 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.

Adding brokers when running a multi-node Kafka cluster 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

```
default.replication.factor = 3
min.insync.replicas = 2
```

When you add or remove brokers, Kafka does not automatically reassign 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.



NOTE

When scaling down brokers, you cannot specify which specific pod to remove from the cluster. Instead, the broker removal process starts from the highest numbered pod.

13.2. ADDING NODES TO A ZOOKEEPER CLUSTER

Use [dynamic reconfiguration](#) to add nodes from a ZooKeeper cluster without stopping the entire cluster. Dynamic Reconfiguration allows ZooKeeper to change the membership of a set of nodes that make up the ZooKeeper cluster without interruption.

Prerequisites

- Dynamic reconfiguration is enabled in the ZooKeeper configuration file (**reconfigEnabled=true**).
- ZooKeeper authentication is enabled and you can access the new server using the authentication mechanism.

Procedure

Perform the following steps for each ZooKeeper server you are adding, one at a time:

1. Add a server to the ZooKeeper cluster as described in [Section 4.1, “Running a multi-node ZooKeeper cluster”](#) and then start ZooKeeper.
2. Note the IP address and configured access ports of the new server.
3. Start a **zookeeper-shell** session for the server. Run the following command from a machine that has access to the cluster (this might be one of the ZooKeeper nodes or your local machine, if it has access).

```
su - kafka
/opt/kafka/bin/zookeeper-shell.sh <ip-address>:<zk-port>
```

4. In the shell session, with the ZooKeeper node running, enter the following line to add the new server to the quorum as a voting member:

```
reconfig -add server.<positive-id> = <address1>:<port1>:<port2>[:role];[<client-port-
address>:]<client-port>
```

For example:

```
reconfig -add server.4=172.17.0.4:2888:3888:participant;172.17.0.4:2181
```

Where **<positive-id>** is the new server ID **4**.

For the two ports, **<port1>** 2888 is for communication between ZooKeeper servers, and **<port2>** 3888 is for leader election.

The new configuration propagates to the other servers in the ZooKeeper cluster; the new server is now a full member of the quorum.

13.3. REMOVING NODES FROM A ZOOKEEPER CLUSTER

Use [dynamic reconfiguration](#) to remove nodes from a ZooKeeper cluster without stopping the entire cluster. Dynamic Reconfiguration allows ZooKeeper to change the membership of a set of nodes that make up the ZooKeeper cluster without interruption.

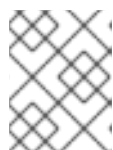
Prerequisites

- Dynamic reconfiguration is enabled in the ZooKeeper configuration file (**reconfigEnabled=true**).
- ZooKeeper authentication is enabled and you can access the new server using the authentication mechanism.

Procedure

Perform the following steps, one at a time, for each ZooKeeper server you remove:

1. Log in to the **zookeeper-shell** on one of the servers that will be **retained** after the scale down (for example, server 1).



NOTE

Access the server using the authentication mechanism configured for the ZooKeeper cluster.

2. Remove a server, for example server 5.

```
reconfig -remove 5
```

3. Deactivate the server that you removed.

CHAPTER 14. USING CRUISE CONTROL FOR CLUSTER REBALANCING

Cruise Control is an open source system for automating Kafka operations, such as monitoring cluster workload, rebalancing a cluster based on predefined constraints, and detecting and fixing anomalies. It consists of four main components—the Load Monitor, the Analyzer, the Anomaly Detector, and the Executor—and a REST API for client interactions.

You can use [Cruise Control](#) to *rebalance* a Kafka cluster. Cruise Control for AMQ Streams on Red Hat Enterprise Linux is provided as a separate zipped distribution.

AMQ Streams utilizes the REST API to support the following Cruise Control features:

- Generating optimization proposals from optimization goals.
- Rebalancing a Kafka cluster based on an optimization proposal.

Optimization goals

An optimization goal describes a specific objective to achieve from a rebalance. For example, a goal might be to distribute topic replicas across brokers more evenly. You can change what goals to include through configuration. A goal is defined as a hard goal or soft goal. You can add hard goals through Cruise Control deployment configuration. You also have main, default, and user-provided goals that fit into each of these categories.

- **Hard goals** are preset and must be satisfied for an optimization proposal to be successful.
- **Soft goals** do not need to be satisfied for an optimization proposal to be successful. They can be set aside if it means that all hard goals are met.
- **Main goals** are inherited from Cruise Control. Some are preset as hard goals. Main goals are used in optimization proposals by default.
- **Default goals** are the same as the main goals by default. You can specify your own set of default goals.
- **User-provided** goals are a subset of default goals that are configured for generating a specific optimization proposal.

Optimization proposals

Optimization proposals comprise the goals you want to achieve from a rebalance. You generate an optimization proposal to create a summary of proposed changes and the results that are possible with the rebalance. The goals are assessed in a specific order of priority. You can then choose to approve or reject the proposal. You can reject the proposal to run it again with an adjusted set of goals.

You can generate and approve an optimization proposal by making a request to one of the following API endpoints.

- **/rebalance** endpoint to run a full rebalance.
- **/add_broker** endpoint to rebalance after adding brokers when scaling up a Kafka cluster.
- **/remove_broker** endpoint to rebalance before removing brokers when scaling down a Kafka cluster.

You configure optimization goals through a configuration properties file. AMQ Streams provides example properties files for Cruise Control.

Other Cruise Control features are not currently supported, including self healing, notifications, write-your-own goals, and changing the topic replication factor.

14.1. CRUISE CONTROL COMPONENTS AND FEATURES

Cruise Control consists of four main components—the Load Monitor, the Analyzer, the Anomaly Detector, and the Executor—and a REST API for client interactions. AMQ Streams utilizes the REST API to support the following Cruise Control features:

- Generating optimization proposals from optimization goals.
- Rebalancing a Kafka cluster based on an optimization proposal.

Optimization goals

An optimization goal describes a specific objective to achieve from a rebalance. For example, a goal might be to distribute topic replicas across brokers more evenly. You can change what goals to include through configuration. A goal is defined as a hard goal or soft goal. You can add hard goals through Cruise Control deployment configuration. You also have main, default, and user-provided goals that fit into each of these categories.

- **Hard goals** are preset and must be satisfied for an optimization proposal to be successful.
- **Soft goals** do not need to be satisfied for an optimization proposal to be successful. They can be set aside if it means that all hard goals are met.
- **Main goals** are inherited from Cruise Control. Some are preset as hard goals. Main goals are used in optimization proposals by default.
- **Default goals** are the same as the main goals by default. You can specify your own set of default goals.
- **User-provided goals** are a subset of default goals that are configured for generating a specific optimization proposal.

Optimization proposals

Optimization proposals comprise the goals you want to achieve from a rebalance. You generate an optimization proposal to create a summary of proposed changes and the results that are possible with the rebalance. The goals are assessed in a specific order of priority. You can then choose to approve or reject the proposal. You can reject the proposal to run it again with an adjusted set of goals.

You can generate an optimization proposal in one of three modes.

- **full** is the default mode and runs a full rebalance.
- **add-brokers** is the mode you use after adding brokers when scaling up a Kafka cluster.
- **remove-brokers** is the mode you use before removing brokers when scaling down a Kafka cluster.

Other Cruise Control features are not currently supported, including self healing, notifications, write-your-own goals, and changing the topic replication factor.

Additional resources

- [Cruise Control documentation](#)

14.2. DOWNLOADING CRUISE CONTROL

A ZIP file distribution of Cruise Control is available for download from the Red Hat website. You can download the latest version of Red Hat AMQ Streams from the [AMQ Streams software downloads page](#).

Procedure

1. Download the latest version of the **Red Hat AMQ Streams Cruise Control** archive from the [Red Hat Customer Portal](#).
2. Create the **/opt/cruise-control** directory:

```
sudo mkdir /opt/cruise-control
```

3. Extract the contents of the Cruise Control ZIP file to the new directory:

```
unzip amq-streams-<version>-cruise-control-bin.zip -d /opt/cruise-control
```

4. Change the ownership of the **/opt/cruise-control** directory to the **kafka** user:

```
sudo chown -R kafka:kafka /opt/cruise-control
```

14.3. DEPLOYING THE CRUISE CONTROL METRICS REPORTER

Before starting Cruise Control, you must configure the Kafka brokers to use the provided Cruise Control Metrics Reporter. The file for the Metrics Reporter is supplied with the AMQ Streams installation artifacts.

When loaded at runtime, the Metrics Reporter sends metrics to the **__CruiseControlMetrics** topic, one of three [auto-created topics](#). Cruise Control uses these metrics to create and update the workload model and to calculate optimization proposals.

Prerequisites

- You are logged in to Red Hat Enterprise Linux as the **kafka** user.
- Kafka and ZooKeeper are running.

Procedure

For each broker in the Kafka cluster and one at a time:

1. Stop the Kafka broker:

```
/opt/kafka/bin/kafka-server-stop.sh
```

2. In the Kafka configuration file (**/opt/kafka/config/server.properties**) configure the Cruise Control Metrics Reporter:

- a. Add the **CruiseControlMetricsReporter** class to the **metric.reporters** configuration option. Do not remove any existing Metrics Reporters.

```
metric.reporters=com.linkedin.kafka.cruisecontrol.metricsreporter.CruiseControlMetricsReporter
```

- b. Add the following configuration options and values:

```
cruise.control.metrics.topic.auto.create=true
cruise.control.metrics.topic.num.partitions=1
cruise.control.metrics.topic.replication.factor=1
```

These options enable the Cruise Control Metrics Reporter to create the **__CruiseControlMetrics** topic with a log cleanup policy of **DELETE**. For more information, see [Auto-created topics](#) and [Log cleanup policy for Cruise Control Metrics topic](#).

3. Configure SSL, if required.

- a. In the Kafka configuration file (**/opt/kafka/config/server.properties**) configure SSL between the Cruise Control Metrics Reporter and the Kafka broker by setting the relevant client configuration properties.

The Metrics Reporter accepts all standard producer-specific configuration properties with the **cruise.control.metrics.reporter** prefix. For example:
cruise.control.metrics.reporter.ssl.truststore.password.

- b. In the Cruise Control properties file (**/opt/cruise-control/config/cruisecontrol.properties**) configure SSL between the Kafka broker and the Cruise Control server by setting the relevant client configuration properties.

Cruise Control inherits SSL client property options from Kafka and uses those properties for all Cruise Control server clients.

4. Restart the Kafka broker:

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

For information on restarting brokers in a multi-node cluster, see [Section 4.3, "Performing a graceful rolling restart of Kafka brokers"](#).

5. Repeat steps 1-5 for the remaining brokers.

14.4. CONFIGURING AND STARTING CRUISE CONTROL

Configure the properties used by Cruise Control and then start the Cruise Control server using the **kafka-cruise-control-start.sh** script. The server is hosted on a single machine for the whole Kafka cluster.

Three topics are auto-created when Cruise Control starts. For more information, see [Auto-created topics](#).

Prerequisites

- You are logged in to Red Hat Enterprise Linux as the **kafka** user.
- [Section 14.2, "Downloading Cruise Control"](#)

- [Section 14.3, “Deploying the Cruise Control Metrics Reporter”](#)

Procedure

1. Edit the Cruise Control properties file (`/opt/cruise-control/config/cruisecontrol.properties`).
2. Configure the properties shown in the following example configuration:

```
# The Kafka cluster to control.
bootstrap.servers=localhost:9092 1

# The replication factor of Kafka metric sample store topic
sample.store.topic.replication.factor=2 2

# The configuration for the BrokerCapacityConfigFileResolver (supports JBOD, non-JBOD,
and heterogeneous CPU core capacities)
#capacity.config.file=config/capacity.json
#capacity.config.file=config/capacityCores.json
capacity.config.file=config/capacityJBOD.json 3

# The list of goals to optimize the Kafka cluster for with pre-computed proposals
default.goals={List of default optimization goals} 4

# The list of supported goals
goals={list of main optimization goals} 5

# The list of supported hard goals
hard.goals={List of hard goals} 6

# How often should the cached proposal be expired and recalculated if necessary
proposal.expiration.ms=60000 7

# The zookeeper connect of the Kafka cluster
zookeeper.connect=localhost:2181 8
```

- 1 Host and port numbers of the Kafka broker (always port 9092).
- 2 Replication factor of the Kafka metric sample store topic. If you are evaluating Cruise Control in a single-node Kafka and ZooKeeper cluster, set this property to 1. For production use, set this property to 2 or more.
- 3 The configuration file that sets the maximum capacity limits for broker resources. Use the file that applies to your Kafka deployment configuration. For more information, see [Capacity configuration](#).
- 4 Comma-separated list of default optimization goals, using fully-qualified domain names (FQDNs). A number of main optimization goals (see 5) are already set as default optimization goals; you can add or remove goals if desired. For more information, see [Section 14.5, “Optimization goals overview”](#).
- 5 Comma-separated list of main optimization goals, using FQDNs. To completely exclude goals from being used to generate optimization proposals, remove them from the list. For more information, see [Section 14.5, “Optimization goals overview”](#).
- 6 Comma-separated list of hard goals, using FQDNs. Seven of the main optimization goals

are already set as hard goals; you can add or remove goals if desired. For more information, see [Section 14.5, “Optimization goals overview”](#).

- 7 The interval, in milliseconds, for refreshing the cached optimization proposal that is generated from the default optimization goals. For more information, see [Section 14.6, “Optimization proposals overview”](#).
 - 8 Host and port numbers of the ZooKeeper connection (always port 2181).
3. Start the Cruise Control server. The server starts on port 9092 by default; optionally, specify a different port.

```
cd /opt/cruise-control/
./kafka-cruise-control-start.sh config/cruisecontrol.properties <port_number>
```

4. To verify that Cruise Control is running, send a GET request to the **/state** endpoint of the Cruise Control server:

```
curl 'http://HOST:PORT/kafkacruisecontrol/state'
```

Auto-created topics

The following table shows the three topics that are automatically created when Cruise Control starts. These topics are required for Cruise Control to work properly and must not be deleted or changed.

Table 14.1. Auto-created topics

Auto-created topic	Created by	Function
__CruiseControlMetrics	Cruise Control Metrics Reporter	Stores the raw metrics from the Metrics Reporter in each Kafka broker.
__KafkaCruiseControlPartitionMetricSamples	Cruise Control	Stores the derived metrics for each partition. These are created by the Metric Sample Aggregator .
__KafkaCruiseControlModelTrainingSamples	Cruise Control	Stores the metrics samples used to create the Cluster Workload Model .

To ensure that log compaction is *disabled* in the auto-created topics, make sure that you configure the Cruise Control Metrics Reporter as described in [Section 14.3, “Deploying the Cruise Control Metrics Reporter”](#). Log compaction can remove records that are needed by Cruise Control and prevent it from working properly.

Additional resources

- [Log cleanup policy for Cruise Control Metrics topic](#)

14.5. 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](#).

14.5.1. Goals order of priority

AMQ Streams on Red Hat Enterprise Linux supports all 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: Disk capacity, Network inbound capacity, Network outbound capacity
5. CPU capacity
6. Replica distribution
7. Potential network output
8. Resource distribution: Disk utilization distribution, Network inbound utilization distribution, Network outbound utilization distribution
9. Leader bytes-in rate distribution
10. Topic replica distribution
11. CPU usage distribution
12. Leader replica distribution
13. Preferred leader election
14. Kafka Assigner disk usage distribution
15. Intra-broker disk capacity
16. Intra-broker disk usage

For more information on each optimization goal, see [Goals](#) in the [Cruise Control Wiki](#).

14.5.2. Goals configuration in the Cruise Control properties file

You configure optimization goals in the **cruisecontrol.properties** file in the **cruise-control/config/** directory. 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 the following types of optimization goal in the following configuration:

- **Main goals** – **cruisecontrol.properties** file
- **Hard goals** – **cruisecontrol.properties** file
- **Default goals** – **cruisecontrol.properties** file

- **User-provided goals** – runtime parameters

Optionally, [user-provided](#) optimization goals are set at runtime as parameters in requests to the **/rebalance** endpoint.

Optimization goals are subject to any [capacity limits](#) on broker resources.

14.5.3. Hard and soft optimization goals

Hard goals are goals that *must* be satisfied in optimization proposals. Goals that are not configured as hard goals are known as *soft goals*. You can think of soft goals as *best effort* goals: they do not need to be satisfied in optimization proposals, but are included in optimization calculations.

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 the Analyzer and is not sent to the user.



NOTE

For example, you might have a soft goal to distribute a topic's replicas evenly across the cluster (the topic replica distribution goal). Cruise Control will ignore this goal if doing so enables all the configured hard goals to be met.

In Cruise Control, the following [main optimization goals](#) are preset as hard goals:

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal
```

To change the hard goals, edit the **hard.goals** property of the **cruisecontrol.properties** file and specify the goals using their fully-qualified domain names.

Increasing the number of hard goals reduces the likelihood that Cruise Control will calculate and generate valid optimization proposals.

14.5.4. 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.

The following main optimization goals are preset in the **goals** property of the **cruisecontrol.properties** file in descending priority order:

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; ReplicaDistributionGoal;
PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal;
NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal;
LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal
```

To reduce complexity, we recommend that you do not change the preset main optimization goals, unless you need to completely exclude one or more goals from being used to generate optimization proposals. The priority order of the main optimization goals can be modified, if desired, in the configuration for default optimization goals.

To modify the preset main optimization goals, specify a list of goals in the **goals** property in descending priority order. Use fully-qualified domain names as shown in the **cruisecontrol.properties** file.

You must specify at least one main goal, or Cruise Control will crash.



NOTE

If you change the preset main optimization goals, you must ensure that the configured **hard.goals** are a subset of the main optimization goals that you configured. Otherwise, errors will occur when generating optimization proposals.

14.5.5. Default optimization goals

Cruise Control uses the *default optimization goals* list to generate the *cached optimization proposal*. For more information, see [Section 14.6, "Optimization proposals overview"](#).

You can override the default optimization goals at runtime by setting [user-provided optimization goals](#).

The following default optimization goals are preset in the **default.goals** property of the **cruisecontrol.properties** file in descending priority order:

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal;
ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal;
NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal;
CpuUsageDistributionGoal; TopicReplicaDistributionGoal; LeaderReplicaDistributionGoal;
LeaderBytesInDistributionGoal
```

You must specify at least one default goal, or Cruise Control will crash.

To modify the default optimization goals, specify a list of goals in the **default.goals** property in descending priority order. Default goals must be a subset of the main optimization goals; use fully-qualified domain names.

14.5.6. 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, as parameters in HTTP requests to the **/rebalance** endpoint. For more information, see [Section 14.9, "Generating optimization proposals"](#).

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 send a request to the **/rebalance** endpoint containing a single 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 in an optimization proposal, add the **skip_hard_goals_check=true** parameter to the request.

Additional resources

- [Cruise Control configuration](#)
- [Configurations](#) in the Cruise Control Wiki

14.6. OPTIMIZATION PROPOSALS OVERVIEW

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 the optimization proposal using one of the following endpoints:

- **`/rebalance`**
- **`/add_broker`**
- **`/remove_broker`**

14.6.1. Rebalancing endpoints

You specify a rebalancing endpoint when you send a POST request to generate an optimization proposal.

`/rebalance`

The **`/rebalance`** endpoint runs a full rebalance by moving replicas across all the brokers in the cluster.

`/add_broker`

The **`add_broker`** endpoint 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_broker`** endpoint 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 **`brokerid`** list in the POST request.

`/remove_broker`

The **`/remove_broker`** endpoint 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_broker`** endpoint 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 **`brokerid`** list in the POST request.

In general, use the **`/rebalance`** endpoint to rebalance a Kafka cluster by spreading the load across brokers. Use the **`/add-broker`** endpoint and **`/remove_broker`** endpoint 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 endpoints. The only difference is with listing brokers that have been added or will be removed to the request.

14.6.2. Approving or rejecting an optimization proposal

An optimization proposal summary shows the proposed scope of changes. The summary is returned in a response to a HTTP request through the Cruise Control API.

When you make a POST request to the **/rebalance** endpoint, an optimization proposal summary is returned in the response.

Returning an optimization proposal summary

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

Use the summary to decide whether to approve or reject an optimization proposal.

Approving an optimization proposal

You approve the optimization proposal by making a POST request to the **/rebalance** endpoint and setting the **dryrun** parameter to **false** (default **true**). 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 resend a request without the **dryrun** parameter to generate a new optimization proposal.

Use the optimization proposal 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

```
Optimization has 55 inter-broker replica (12 MB) moves, 0 intra-broker
replica (0 MB) moves and 24 leadership moves with a cluster model of 5
recent windows and 100.000% of the partitions covered.
Excluded Topics: [].
Excluded Brokers For Leadership: [].
Excluded Brokers For Replica Move: [].
Counts: 3 brokers 343 replicas 7 topics.
On-demand Balancedness Score Before (78.012) After (82.912).
Provision Status: RIGHT_SIZED.
```

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.

The provision status indicates whether the current cluster configuration supports the optimization goals. Check the provision status to see if you should add or remove brokers.

Table 14.2. Optimization proposal provision status

Status	Description
RIGHT_SIZED	The cluster has an appropriate number of brokers to satisfy the optimization goals.
UNDER_PROVISIONED	The cluster is under-provisioned and requires more brokers to satisfy the optimization goals.
OVER_PROVISIONED	The cluster is over-provisioned and requires fewer brokers to satisfy the optimization goals.
UNDECIDED	The status is not relevant or it has not yet been decided.

14.6.3. Optimization proposal summary properties

The following table describes the properties contained in an optimization proposal.

Table 14.3. Properties contained in an optimization proposal summary

Property	Description
n inter-broker replica (y MB) moves	<p>n: The number of partition replicas that will be moved between separate brokers.</p> <p>Performance impact during rebalance operation Relatively high.</p> <p>y MB: The sum of the size of each partition replica that will be moved to a separate broker.</p> <p>Performance impact during rebalance operation Variable. The larger the number of MBs, the longer the cluster rebalance will take to complete.</p>

Property	Description
n intra-broker replica (y MB) moves	<p>n: The total number of partition replicas that will be transferred between the disks of the cluster's brokers.</p> <p>Performance impact during rebalance operation Relatively high, but less than inter-broker replica moves.</p> <p>y MB: The sum of the size of each partition replica that will be moved between disks on the same broker.</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 inter-broker replica moves).</p>
n excluded topics	<p>The number of topics excluded from the calculation of partition replica/leader movements in the optimization proposal.</p> <p>You can exclude topics in one of the following ways:</p> <p>In the cruisecontrol.properties file, specify a regular expression in the topics.excluded.from.partition.movement property.</p> <p>In a POST request to the /rebalance endpoint, specify a regular expression in the excluded_topics parameter.</p> <p>Topics that match the regular expression are listed in the response and will be excluded from the cluster rebalance.</p>
n leadership moves	<p>n: The number of partitions whose leaders will be switched to different replicas. This involves a change to ZooKeeper configuration.</p> <p>Performance impact during rebalance operation Relatively low.</p>
n recent windows	<p>n: The number of metrics windows upon which the optimization proposal is based.</p>
n% of the partitions covered	<p>n%: The percentage of partitions in the Kafka cluster covered by the optimization proposal.</p>
On-demand Balancedness Score Before (nn.yyy) After (nn.yyy)	<p>Measurements of the overall balance of a Kafka Cluster.</p> <p>Cruise Control assigns a Balancedness Score to every optimization goal based on several factors, including priority (the goal's position in the list of default.goals or user-provided goals). The On-demand Balancedness Score is calculated by subtracting the sum of the Balancedness Score of each violated soft goal from 100.</p> <p>The Before score is based on the current configuration of the Kafka cluster. The After score is based on the generated optimization proposal.</p>

14.6.4. 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.

The most recent cached optimization proposal is returned when the following goal configurations are used:

- The default optimization goals
- User-provided optimization goals that can be met by the current cached proposal

To change the cached optimization proposal refresh interval, edit the **proposal.expiration.ms** setting in the **cruisecontrol.properties** file. 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](#)
- [Initiating a cluster rebalance](#)

14.7. REBALANCE PERFORMANCE TUNING OVERVIEW

You can adjust several performance tuning options for cluster rebalances. These options control how partition replica and leadership movements in a rebalance are executed, as well as the bandwidth that is allocated to a rebalance operation.

Partition reassignment commands

[Optimization proposals](#) are composed of separate partition reassignment commands. When you initiate 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:** 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.

To configure partition reassignment commands, see [Rebalance tuning options](#).

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 applies the commands in the order in which they were generated. However, if there are some very large partition reassignments early in the proposal, this strategy can slow down the application of the other reassignments.

Cruise Control provides three alternative replica movement strategies that can be applied to optimization proposals:

- **PrioritizeSmallReplicaMovementStrategy**: Order reassignments in ascending size.
- **PrioritizeLargeReplicaMovementStrategy**: Order reassignments in descending size.
- **PostponeUrpReplicaMovementStrategy**: Prioritize reassignments for replicas of partitions which have no out-of-sync replicas.

These strategies can be configured as a sequence. The first strategy attempts to compare two partition reassignments using its internal logic. If the reassignments are equivalent, then it passes them to the next strategy in the sequence to decide the order, and so on.

To configure replica movement strategies, see [Rebalance tuning options](#).

Rebalance tuning options

Cruise Control provides several configuration options for tuning rebalance parameters. These options are set in the following ways:

- As properties, in the default Cruise Control configuration, in the **cruisecontrol.properties** file
- As parameters in POST requests to the **/rebalance** endpoint

The relevant configurations for both methods are summarized in the following table.

Table 14.4. Rebalance performance tuning configuration

Cruise Control properties	KafkaRebalance parameters	Default	Description
num.concurrent.partition.movement.per.broker	concurrent_partition_movements_per_broker	5	The maximum number of inter-broker partition movements in each partition reassignment batch
num.concurrent.intra.broker.partition.movements	concurrent_intra_broker_partition_movements	2	The maximum number of intra-broker partition movements in each partition reassignment batch

Cruise Control properties	KafkaRebalance parameters	Default	Description
num.concurrent.leader.movements	concurrent_leader_movements	1000	The maximum number of partition leadership changes in each partition reassignment batch
default.replication.throttle	replication_throttle	Null (no limit)	The bandwidth (in bytes per second) to assign to partition reassignment

Cruise Control properties	KafkaRebalance parameters	Default	Description
default.replica.movement.strategies	replica_movement_strategies	BaseReplicaMovementStrategy	<p>The list of strategies (in priority order) used to determine the order in which partition reassignment commands are executed for generated proposals. There are three strategies:</p> <p>PrioritizeSmallReplicaMovementStrategy, PrioritizeLargeReplicaMovementStrategy, and PostponeUrgentReplicaMovementStrategy.</p> <p>For the server setting, use a comma-separated list with the fully qualified names of the strategy class (add com.linkedin.kafka.cruisecontrol.executor.strategy. to the start of each class name). For the rebalance parameters, use a comma-separated list of the class names of the replica movement strategies.</p>

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

- [Configurations](#) in the Cruise Control Wiki
- [REST APIs](#) in the Cruise Control Wiki

14.8. CRUISE CONTROL CONFIGURATION

The **config/cruisecontrol.properties** file contains the configuration for Cruise Control. The file consists of properties in one of the following types:

- String
- Number
- Boolean

You can specify and configure all the properties listed in the [Configurations](#) section of the Cruise Control Wiki.

Capacity configuration

Cruise Control uses *capacity limits* to determine if certain resource-based optimization goals are being broken. An attempted optimization fails if one or more of these resource-based goals is set as a hard goal and then broken. This prevents the optimization from being used to generate an optimization proposal.

You specify capacity limits for Kafka broker resources in one of the following three **.json** files in **cruise-control/config**:

- **capacityJBOD.json**: For use in JBOD Kafka deployments (the default file).
- **capacity.json**: For use in non-JBOD Kafka deployments where each broker has the same number of CPU cores.
- **capacityCores.json**: For use in non-JBOD Kafka deployments where each broker has varying numbers of CPU cores.

Set the file in the **capacity.config.file** property in **cruisecontrol.properties**. The selected file will be used for broker capacity resolution. For example:

```
capacity.config.file=config/capacityJBOD.json
```

Capacity limits can be set for the following broker resources in the described units:

- **DISK**: Disk storage in MB
- **CPU**: CPU utilization as a percentage (0-100) or as a number of cores
- **NW_IN**: Inbound network throughput in KB per second
- **NW_OUT**: Outbound network throughput in KB per second

To apply the same capacity limits to every broker monitored by Cruise Control, set capacity limits for broker ID **-1**. To set different capacity limits for individual brokers, specify each broker ID and its capacity configuration.

Example capacity limits configuration

```
{
  "brokerCapacities":[
    {
      "brokerId": "-1",
      "capacity": {
        "DISK": "100000",
        "CPU": "100",
        "NW_IN": "10000",
        "NW_OUT": "10000"
      },
      "doc": "This is the default capacity. Capacity unit used for disk is in MB, cpu is in percentage,
network throughput is in KB."
    },
    {
      "brokerId": "0",
      "capacity": {
        "DISK": "500000",
        "CPU": "100",
        "NW_IN": "50000",
        "NW_OUT": "50000"
      },
      "doc": "This overrides the capacity for broker 0."
    }
  ]
}
```

For more information, see [Populating the Capacity Configuration File](#) in the Cruise Control Wiki.

Log cleanup policy for Cruise Control Metrics topic

It is important that the auto-created `__CruiseControlMetrics` topic (see [auto-created topics](#)) has a log cleanup policy of **DELETE** rather than **COMPACT**. Otherwise, records that are needed by Cruise Control might be removed.

As described in [Section 14.3, “Deploying the Cruise Control Metrics Reporter”](#), setting the following options in the Kafka configuration file ensures that the **COMPACT** log cleanup policy is correctly set:

- `cruise.control.metrics.topic.auto.create=true`
- `cruise.control.metrics.topic.num.partitions=1`
- `cruise.control.metrics.topic.replication.factor=1`

If topic auto-creation is *disabled* in the Cruise Control Metrics Reporter (`cruise.control.metrics.topic.auto.create=false`), but *enabled* in the Kafka cluster, then the `__CruiseControlMetrics` topic is still automatically created by the broker. In this case, you must change the log cleanup policy of the `__CruiseControlMetrics` topic to **DELETE** using the `kafka-configs.sh` tool.

1. Get the current configuration of the `__CruiseControlMetrics` topic:

```
opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --entity-type topics --
entity-name __CruiseControlMetrics --describe
```

2. Change the log cleanup policy in the topic configuration:

```
/opt/kafka/bin/kafka-configs.sh --bootstrap-server <broker_address> --entity-type topics --
entity-name __CruiseControlMetrics --alter --add-config cleanup.policy=delete
```

If topic auto-creation is *disabled* in both the Cruise Control Metrics Reporter *and* the Kafka cluster, you must create the **__CruiseControlMetrics** topic manually and then configure it to use the **DELETE** log cleanup policy using the **kafka-configs.sh** tool.

For more information, see [Section 7.9, “Modifying a topic configuration”](#).

Logging configuration

Cruise Control uses **log4j1** for all server logging. To change the default configuration, edit the **log4j.properties** file in **/opt/cruise-control/config/log4j.properties**.

You must restart the Cruise Control server before the changes take effect.

14.9. GENERATING OPTIMIZATION PROPOSALS

When you make a POST request to the **/rebalance** endpoint, Cruise Control generates an optimization proposal to rebalance the Kafka cluster based on the optimization goals provided. You can use the results of the optimization proposal to rebalance your Kafka cluster.

You can run the optimization proposal using one of the following endpoints:

- **/rebalance**
- **/add_broker**
- **/remove_broker**

The endpoint 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 endpoints with broker scaling](#).

The optimization proposal is generated as a *dry run*, unless the **dryrun** parameter is supplied and set to **false**. In “dry run mode”, Cruise Control generates the optimization proposal and the estimated result, but doesn’t initiate the proposal by rebalancing the cluster.

You can analyze the information returned in the optimization proposal and decide whether to approve it.

Use the following parameters to make requests to the endpoints:

dryrun

type: boolean, default: true

Informs Cruise Control whether you want to generate an optimization proposal only (**true**), or generate an optimization proposal and perform a cluster rebalance (**false**).

When **dryrun=true** (the default), you can also pass the **verbose** parameter to return more detailed information about the state of the Kafka cluster. This includes metrics for the load on each Kafka broker before and after the optimization proposal is applied, and the differences between the before and after values.

excluded_topics

type: regex

A regular expression that matches the topics to exclude from the calculation of the optimization proposal.

goals

type: list of strings, default: the configured **default.goals** list

List of user-provided optimization goals to use to prepare the optimization proposal. If goals are not supplied, the configured **default.goals** list in the **cruisecontrol.properties** file is used.

skip_hard_goals_check

type: boolean, default: **false**

By default, Cruise Control checks that the user-provided optimization goals (in the **goals** parameter) contain all the configured hard goals (in **hard.goals**). A request fails if you supply goals that are not a subset of the configured **hard.goals**.

Set **skip_hard_goals_check** to **true** if you want to generate an optimization proposal with user-provided optimization goals that do not include all the configured **hard.goals**.

json

type: boolean, default: **false**

Controls the type of response returned by the Cruise Control server. If not supplied, or set to **false**, then Cruise Control returns text formatted for display on the command line. If you want to extract elements of the returned information programmatically, set **json=true**. This will return JSON formatted text that can be piped to tools such as **jq**, or parsed in scripts and programs.

verbose

type: boolean, default: **false**

Controls the level of detail in responses that are returned by the Cruise Control server. Can be used with **dryrun=true**.



NOTE

Other parameters are available. For more information, see [REST APIs](#) in the Cruise Control Wiki.

Prerequisites

- Kafka and ZooKeeper are running
- Cruise Control is running
- (Optional for scaling up) You have [installed new brokers on hosts](#) to include in the rebalance

Procedure

1. Generate an optimization proposal using a POST request to the **/rebalance**, **/add_broker**, or **/remove_broker** endpoint.

Example request to /rebalance using default goals

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

The cached optimization proposal is immediately returned.



NOTE

If **NotEnoughValidWindows** is returned, Cruise Control has not yet recorded enough metrics data to generate an optimization proposal. Wait a few minutes and then resend the request.

Example request to /rebalance using specified goals

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?goals=RackAwareGoal,ReplicaCapacityGoal'
```

If the request satisfies the supplied goals, the cached optimization proposal is immediately returned. Otherwise, a new optimization proposal is generated using the supplied goals; this takes longer to calculate. You can enforce this behavior by adding the **ignore_proposal_cache=true** parameter to the request.

Example request to /rebalance using specified goals without hard goals

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?goals=RackAwareGoal,ReplicaCapacityGoal,ReplicaDistributionGoal&skip_hard_goal_check=true'
```

Example request to /add_broker that includes specified brokers

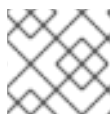
```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/add_broker?brokerid=3,4'
```

The request includes the IDs of the new brokers only. For example, this request adds brokers with the IDs **3** and **4**. Replicas are moved to the new brokers from existing brokers when rebalancing.

Example request to /remove_broker that excludes specified brokers

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/remove_broker?brokerid=3,4'
```

The request includes the IDs of the brokers being excluded only. For example, this request excludes brokers with the IDs **3** and **4**. Replicas are moved from the brokers being removed to other existing brokers when rebalancing.



NOTE

If a broker that is being removed has excluded topics, replicas are still moved.

2. Review the optimization proposal contained in the response. The properties describe the pending cluster rebalance operation.

The proposal contains a high level summary of the proposed optimization, followed by summaries for each default optimization goal, and the expected cluster state after the proposal has executed.

Pay particular attention to the following information:

- The **Cluster load after rebalance** summary. If it meets your requirements, you should assess the impact of the proposed changes using the high level summary.
- **n inter-broker replica (y MB) moves** indicates how much data will be moved across the network between brokers. The higher the value, the greater the potential performance impact on the Kafka cluster during the rebalance.
- **n intra-broker replica (y MB) moves** indicates how much data will be moved within the brokers themselves (between disks). The higher the value, the greater the potential performance impact on individual brokers (although less than that of **n inter-broker replica (y MB) moves**).
- The number of leadership moves. This has a negligible impact on the performance of the cluster during the rebalance.

Asynchronous responses

The Cruise Control REST API endpoints timeout after 10 seconds by default, although proposal generation continues on the server. A timeout might occur if the most recent cached optimization proposal is not ready, or if user-provided optimization goals were specified with **ignore_proposal_cache=true**.

To allow you to retrieve the optimization proposal at a later time, take note of the request's unique identifier, which is given in the header of responses from the **/rebalance** endpoint.

To obtain the response using **curl**, specify the verbose (**-v**) option:

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

Here is an example header:

```
* Connected to cruise-control-server (:::1) port 9090 (#0)
> POST /kafkacruisecontrol/rebalance HTTP/1.1
> Host: cc-host:9090
> User-Agent: curl/7.70.0
> Accept: /
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Mon, 01 Jun 2020 15:19:26 GMT
< Set-Cookie: JSESSIONID=node01wk6vjzjj12go13m81o7no5p7h9.node0; Path=/
< Expires: Thu, 01 Jan 1970 00:00:00 GMT
< User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201
< Content-Type: text/plain;charset=utf-8
< Cruise-Control-Version: 2.0.103.redhat-00002
< Cruise-Control-Commit_Id: 58975c9d5d0a78dd33cd67d4bcb497c9fd42ae7c
< Content-Length: 12368
< Server: Jetty(9.4.26.v20200117-redhat-00001)
```

If an optimization proposal is not ready within the timeout, you can re-submit the POST request, this time including the **User-Task-ID** of the original request in the header:

```
curl -v -X POST -H 'User-Task-ID: 274b8095-d739-4840-85b9-f4cfaaf5c201' 'cruise-control-server:9090/kafkacruisecontrol/rebalance'
```

What to do next

[Section 14.10, "Approving an optimization proposal"](#)

14.10. APPROVING AN OPTIMIZATION PROPOSAL

If you are satisfied with your most recently generated optimization proposal, you can instruct Cruise Control to initiate a cluster rebalance and begin reassigning partitions.

Leave as little time as possible between generating an optimization proposal and initiating the cluster rebalance. If some time has passed since you generated the original optimization proposal, the cluster state might have changed. Therefore, the cluster rebalance that is initiated might be different to the one you reviewed. If in doubt, first generate a new optimization proposal.

Only one cluster rebalance, with a status of "Active", can be in progress at a time.

Prerequisites

- You have [generated an optimization proposal](#) from Cruise Control.

Procedure

- Send a POST request to the `/rebalance`, `/add_broker`, or `/remove_broker` endpoint with the `dryrun=false` parameter:
If you used the `/add_broker` or `/remove_broker` endpoint to generate a proposal that included or excluded brokers, use the same endpoint to perform the rebalance with or without the specified brokers.

Example request to `/rebalance`

```
curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/rebalance?dryrun=false'
```

Example request to `/add_broker`

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/add_broker?dryrun=false&brokerid=3,4'
```

Example request to `/remove_broker`

```
curl -v -X POST 'cruise-control-server:9090/kafkacruisecontrol/remove_broker?dryrun=false&brokerid=3,4'
```

Cruise Control initiates the cluster rebalance and returns the optimization proposal.

- Check the changes that are summarized in the optimization proposal. If the changes are not what you expect, you can [stop the rebalance](#).
- Check the progress of the cluster rebalance using the `/user_tasks` endpoint. The cluster rebalance in progress has a status of "Active".
To view all cluster rebalance tasks executed on the Cruise Control server:

```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks'
```

USER TASK ID	CLIENT ADDRESS	START TIME	STATUS	REQUEST URL
c459316f-9eb5-482f-9d2d-97b5a4cd294d	0:0:0:0:0:0:1	2020-06-01_16:10:29 UTC	Active	POST /kafkacruisecontrol/rebalance?dryrun=false
445e2fc3-6531-4243-b0a6-36ef7c5059b4	0:0:0:0:0:0:1	2020-06-01_14:21:26 UTC	Completed	GET /kafkacruisecontrol/state?json=true
05c37737-16d1-4e33-8e2b-800dee9f1b01	0:0:0:0:0:0:1	2020-06-01_14:36:11 UTC	Completed	GET /kafkacruisecontrol/state?json=true
aebae987-985d-4871-8cfb-6134ecd504ab	0:0:0:0:0:0:1	2020-06-01_16:10:04 UTC		

- To view the status of a particular cluster rebalance task, supply the **user-task-ids** parameter and the task ID:

```
curl 'cruise-control-server:9090/kafkacruisecontrol/user_tasks?user_task_ids=c459316f-9eb5-482f-9d2d-97b5a4cd294d'
```

(Optional) Removing brokers when scaling down

After a successful rebalance you can stop any brokers you excluded in order to scale down the Kafka cluster.

- Check that each broker being removed does not have any live partitions in its log (**log.dirs**).

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

If a log directory does not match the regular expression `\.[a-z0-9]-delete$`, active partitions are still present. If you have active partitions, check the rebalance has finished or the configuration for the optimization proposal. You can run the proposal again. Make sure that there are no active partitions before moving on to the next step.

- Stop the broker.

```
su - kafka
/opt/kafka/bin/kafka-server-stop.sh
```

- Confirm that the broker has stopped.

```
jcmd | grep kafka
```

14.11. STOPPING AN ACTIVE CLUSTER REBALANCE

You can stop the cluster rebalance that is currently in progress.

This instructs Cruise Control to finish the current batch of partition reassignments and then stop the rebalance. When the rebalance has stopped, completed partition reassignments have already been applied; therefore, the state of the Kafka cluster is different when compared to before 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

- A cluster rebalance is in progress (indicated by a status of "Active").

Procedure

- Send a POST request to the **/stop_proposal_execution** endpoint:

```
curl -X POST 'cruise-control-server:9090/kafkacruisecontrol/stop_proposal_execution'
```

Additional resources

- [Generating optimization proposals](#)

CHAPTER 15. USING THE PARTITION REASSIGNMENT TOOL

When scaling a Kafka cluster, you may need to add or remove brokers and update the distribution of partitions or the replication factor of topics. To update partitions and topics, you can use the **kafka-reassign-partitions.sh** tool.

You can change the replication factor of a topic using the **kafka-reassign-partitions.sh** tool. The tool can also be used to reassign partitions and balance the distribution of partitions across brokers to improve performance. However, it is recommended to use [Cruise Control for automated partition reassignments 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. The AMQ Streams Cruise Control integration does not support changing the replication factor of a topic.

15.1. 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.

15.1.1. 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 reassignments should be broken down into a number of smaller reassignments in case there is a need to stop in-progress reassignment.

15.1.2. 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" }
  ]
}

```

15.1.3. 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"]
  }
]
}

```

15.1.4. 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.

15.2. REASSIGNING 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.



NOTE

Though you can use the `kafka-reassign-partitions.sh` tool, Cruise Control is recommended [for automated partition reassignments 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

- An existing Kafka cluster.
- A new machine with the additional AMQ broker [installed](#).
- You have created a JSON file to specify how partitions should be reassigned to brokers in the enlarged cluster.
In this procedure, we are reassigning all partitions for a topic called **my-topic**. A JSON file named **topics.json** specifies the topic, and is used to generate a **reassignment.json** file.

Example JSON file specifies my-topic

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

Procedure

1. Create a configuration file for the new broker using the same settings as for the other brokers in your cluster, except for **broker.id**, which should be a number that is not already used by any of the other brokers.
2. Start the new Kafka broker passing the configuration file you created in the previous step as the argument to the **kafka-server-start.sh** script:

```
su - kafka
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

3. Verify that the Kafka broker is running.

```
jcmd | grep Kafka
```

4. Repeat the above steps for each new broker.
5. If you haven't done so, generate a reassignment JSON file named **reassignment.json** using the **kafka-reassign-partitions.sh** tool.

Example command to generate the reassignment JSON file

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --topics-to-move-json-file topics.json \ 1
  --broker-list 0,1,2,3,4 \ 2
  --generate
```

- 1** The JSON file that specifies the topic.
- 2** Brokers IDs in the kafka cluster to include in the operation. This assumes broker **4** has been added.

Example reassignment JSON file showing the current and proposed replica assignment

Current partition replica assignment

```
{ "version": 1, "partitions": [ { "topic": "my-topic", "partition": 0, "replicas": [ 0, 1, 2 ], "log_dirs": [ "any", "any", "any" ] }, { "topic": "my-topic", "partition": 1, "replicas": [ 1, 2, 3 ], "log_dirs": [ "any", "any", "any" ] }, { "topic": "my-topic", "partition": 2, "replicas": [ 2, 3, 0 ], "log_dirs": [ "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.

- Run the partition reassignment using the **--execute** option.

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```

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:

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --throttle 5000000 \
  --execute
```

- Verify that the reassignment has completed using the **--verify** option.

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file 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.

15.3. REASSIGNING 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.



NOTE

Though you can use the **kafka-reassign-partitions.sh** tool, Cruise Control is recommended [for automated partition reassignments 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

- An existing Kafka cluster.
- You have created a JSON file to specify how partitions should be reassigned to brokers in the reduced cluster.

In this procedure, we are reassigning all partitions for a topic called **my-topic**. A JSON file named **topics.json** specifies the topic, and is used to generate a **reassignment.json** file.

Example JSON file specifies my-topic

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

Procedure

1. If you haven't done so, generate a reassignment JSON file named **reassignment.json** using the **kafka-reassign-partitions.sh** tool.

Example command to generate the reassignment JSON file

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --topics-to-move-json-file topics.json \ 1
  --broker-list 0,1,2,3 \ 2
  --generate
```

- 1 The JSON file that specifies the topic.
- 2 Brokers IDs in the kafka cluster to include in the operation. This assumes broker **4** has been removed.

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"]},{"topic":"my-topic","partition":1,"replicas":[0,2,3,1],"log_dirs":["any","any","any","any"]},{"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],"log_dirs":
      ["any","any","any"]},
    {"topic":"my-topic","partition":1,"replicas":[1,2,3],"log_dirs":
      ["any","any","any"]},
    {"topic":"my-topic","partition":2,"replicas":[2,3,0],"log_dirs":
      ["any","any","any"]}
  ]
}
```

Save a copy of this file locally in case you need to revert the changes later on.

2. Run the partition reassignment using the **--execute** option.

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --execute
```

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:

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file reassignment.json \
  --throttle 5000000 \
  --execute
```

3. Verify that the reassignment has completed using the **--verify** option.

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file 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.

4. Check that each broker being removed does not have any live partitions in its log (**log.dirs**).

```
ls -l <LogDir> | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'
```

If a log directory does not match the regular expression `\.[a-z0-9]-delete$`, active partitions are still present. If you have active partitions, check the reassignment has finished or the configuration in the reassignment JSON file. You can run the reassignment again. Make sure that there are no active partitions before moving on to the next step.

5. Stop the broker.

```
su - kafka
/opt/kafka/bin/kafka-server-stop.sh
```

6. Confirm that the Kafka broker has stopped.

```
jcmd | grep kafka
```

15.4. CHANGING THE REPLICATION FACTOR OF TOPICS

Use the **kafka-reassign-partitions.sh** tool to change the replication factor of topics in a Kafka cluster. This can be done using a reassignment file to describe how the topic replicas should be changed.

Prerequisites

- An existing Kafka cluster.
- You have created a JSON file to specify the topics to include in the operation. 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 is used to generate a **reassignment.json** file.

Example JSON file specifies my-topic

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

Procedure

1. If you haven't done so, generate a reassignment JSON file named **reassignment.json** using the **kafka-reassign-partitions.sh** tool.

Example command to generate the reassignment JSON file

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
--bootstrap-server localhost:9092 \
--topics-to-move-json-file topics.json \ 1
--broker-list 0,1,2,3,4 \ 2
--generate
```

- 1** The JSON file that specifies the topic.
- 2** Brokers IDs in the kafka cluster to include in the operation.

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"]},{"topic":"my-topic","partition":1,"replicas":[0,2,3,1],"log_dirs":["any","any","any","any"]},{"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 **jq** 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. Make the topic replica change using the **--execute** option.

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file 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.

4. Verify that the change to the topic replicas has completed using the **--verify** option.

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --reassignment-json-file 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.

5. Run the **bin/kafka-topics.sh** command with the **--describe** option to see the results of the change to the topics.

```
/opt/kafka/bin/kafka-reassign-partitions.sh \
  --bootstrap-server localhost:9092 \
  --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
```


CHAPTER 16. USING KERBEROS (GSSAPI) AUTHENTICATION

AMQ Streams supports the use of the Kerberos (GSSAPI) authentication protocol for secure single sign-on access to your Kafka cluster. GSSAPI is an API wrapper for Kerberos functionality, insulating applications from underlying implementation changes.

Kerberos is a network authentication system that allows clients and servers to authenticate to each other by using symmetric encryption and a trusted third party, the Kerberos Key Distribution Centre (KDC).

16.1. SETTING UP AMQ STREAMS TO USE KERBEROS (GSSAPI) AUTHENTICATION

This procedure shows how to configure AMQ Streams so that Kafka clients can access Kafka and ZooKeeper using Kerberos (GSSAPI) authentication.

The procedure assumes that a Kerberos *krb5* resource server has been set up on a Red Hat Enterprise Linux host.

The procedure shows, with examples, how to configure:

1. Service principals
2. Kafka brokers to use the Kerberos login
3. ZooKeeper to use Kerberos login
4. Producer and consumer clients to access Kafka using Kerberos authentication

The instructions describe Kerberos set up for a single ZooKeeper and Kafka installation on a single host, with additional configuration for a producer and consumer client.

Prerequisites

To be able to configure Kafka and ZooKeeper to authenticate and authorize Kerberos credentials, you will need:

- Access to a Kerberos server
- A Kerberos client on each Kafka broker host

For more information on the steps to set up a Kerberos server, and clients on broker hosts, see the [example Kerberos on RHEL set up configuration](#).

Add service principals for authentication

From your Kerberos server, create service principals (users) for ZooKeeper, Kafka brokers, and Kafka producer and consumer clients.

Service principals must take the form *SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-REALM*.

1. Create the service principals, and keytabs that store the principal keys, through the Kerberos KDC.

For example:

- **zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM**

- **kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
- **producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
- **consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM**
The ZooKeeper service principal must have the same hostname as the **zookeeper.connect** configuration in the Kafka **config/server.properties** file:

```
zookeeper.connect=node1.example.redhat.com:2181
```

If the hostname is not the same, *localhost* is used and authentication will fail.

2. Create a directory on the host and add the keytab files:
For example:

```
/opt/kafka/krb5/zookeeper-node1.keytab
/opt/kafka/krb5/kafka-node1.keytab
/opt/kafka/krb5/kafka-producer1.keytab
/opt/kafka/krb5/kafka-consumer1.keytab
```

3. Ensure the **kafka** user can access the directory:

```
chown kafka:kafka -R /opt/kafka/krb5
```

Configure ZooKeeper to use a Kerberos Login

Configure ZooKeeper to use the Kerberos Key Distribution Center (KDC) for authentication using the user principals and keytabs previously created for **zookeeper**.

1. Create or modify the **opt/kafka/config/jaas.conf** file to support ZooKeeper client and server operations:

```
Client {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true 1
  storeKey=true 2
  useTicketCache=false 3
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab" 4
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM"; 5
};

Server {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  useTicketCache=false
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumServer {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
```

```

principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

QuorumLearner {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/zookeeper-node1.keytab"
  principal="zookeeper/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};

```

- 1 Set to **true** to get the principal key from the keytab.
- 2 Set to **true** to store the principal key.
- 3 Set to **true** to obtain the Ticket Granting Ticket (TGT) from the ticket cache.
- 4 The **keyTab** property points to the location of the keytab file copied from the Kerberos KDC. The location and file must be readable by the **kafka** user.
- 5 The **principal** property is configured to match the fully-qualified principal name created on the KDC host, which follows the format **SERVICE-NAME/FULLY-QUALIFIED-HOST-NAME@DOMAIN-NAME**.

2. Edit **opt/kafka/config/zookeeper.properties** to use the updated JAAS configuration:

```

# ...

requireClientAuthScheme=sasl
jaasLoginRenew=3600000 1
kerberos.removeHostFromPrincipal=false 2
kerberos.removeRealmFromPrincipal=false 3
quorum.auth.enableSasl=true 4
quorum.auth.learnerRequireSasl=true 5
quorum.auth.serverRequireSasl=true
quorum.auth.learner.loginContext=QuorumLearner 6
quorum.auth.server.loginContext=QuorumServer
quorum.auth.kerberos.servicePrincipal=zookeeper/_HOST 7
quorum.cnxn.threads.size=20

```

- 1 Controls the frequency for login renewal in milliseconds, which can be adjusted to suit ticket renewal intervals. Default is one hour.
- 2 Dictates whether the hostname is used as part of the login principal name. If using a single keytab for all nodes in the cluster, this is set to **true**. However, it is recommended to generate a separate keytab and fully-qualified principal for each broker host for troubleshooting.
- 3 Controls whether the realm name is stripped from the principal name for Kerberos negotiations. It is recommended that this setting is set as **false**.
- 4 Enables SASL authentication mechanisms for the ZooKeeper server and client.
- 5 The **RequireSasl** properties controls whether SASL authentication is required for quorum events, such as master elections.

- 6 The **loginContext** properties identify the name of the login context in the JAAS configuration used for authentication configuration of the specified component. The
- 7 Controls the naming convention to be used to form the principal name used for identification. The placeholder **_HOST** is automatically resolved to the hostnames defined by the **server.1** properties at runtime.

3. Start ZooKeeper with JVM parameters to specify the Kerberos login configuration:

```
su - kafka
export EXTRA_ARGS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/zookeeper-server-
start.sh -daemon /opt/kafka/config/zookeeper.properties
```

If you are not using the default service name (**zookeeper**), add the name using the **-Dzookeeper.sasl.client.username=NAME** parameter.



NOTE

If you are using the **/etc/krb5.conf** location, you do not need to specify **-Djava.security.krb5.conf=/etc/krb5.conf** when starting ZooKeeper, Kafka, or the Kafka producer and consumer.

Configure the Kafka broker server to use a Kerberos login

Configure Kafka to use the Kerberos Key Distribution Center (KDC) for authentication using the user principals and keytabs previously created for **kafka**.

1. Modify the **opt/kafka/config/jaas.conf** file with the following elements:

```
KafkaServer {
  com.sun.security.auth.module.Krb5LoginModule required
  useKeyTab=true
  storeKey=true
  keyTab="/opt/kafka/krb5/kafka-node1.keytab"
  principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
KafkaClient {
  com.sun.security.auth.module.Krb5LoginModule required debug=true
  useKeyTab=true
  storeKey=true
  useTicketCache=false
  keyTab="/opt/kafka/krb5/kafka-node1.keytab"
  principal="kafka/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
};
```

2. Configure each broker in the Kafka cluster by modifying the listener configuration in the **config/server.properties** file so the listeners use the SASL/GSSAPI login. Add the SASL protocol to the map of security protocols for the listener, and remove any unwanted protocols.

For example:

```
# ...
broker.id=0
```

```
# ...
listeners=SECURE://:9092,REPLICATION://:9094 ❶
inter.broker.listener.name=REPLICATION
# ...
listener.security.protocol.map=SECURE:SASL_PLAINTEXT,REPLICATION:SASL_PLAINTEXT ❷
# ..
sasl.enabled.mechanisms=GSSAPI ❸
sasl.mechanism.inter.broker.protocol=GSSAPI ❹
sasl.kerberos.service.name=kafka ❺
...
```

- ❶ Two listeners are configured: a secure listener for general-purpose communications with clients (supporting TLS for communications), and a replication listener for inter-broker communications.
- ❷ For TLS-enabled listeners, the protocol name is SASL_PLAINTEXT. For non-TLS-enabled connectors, the protocol name is SASL_PLAINTEXT. If SSL is not required, you can remove the **ssl.*** properties.
- ❸ SASL mechanism for Kerberos authentication is **GSSAPI**.
- ❹ Kerberos authentication for inter-broker communication.
- ❺ The name of the service used for authentication requests is specified to distinguish it from other services that may also be using the same Kerberos configuration.

3. Start the Kafka broker, with JVM parameters to specify the Kerberos login configuration:

```
su - kafka
export KAFKA_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Djava.security.auth.login.config=/opt/kafka/config/jaas.conf"; /opt/kafka/bin/kafka-server-
start.sh -daemon /opt/kafka/config/server.properties
```

If the broker and ZooKeeper cluster were previously configured and working with a non-Kerberos-based authentication system, it is possible to start the ZooKeeper and broker cluster and check for configuration errors in the logs.

After starting the broker and Zookeeper instances, the cluster is now configured for Kerberos authentication.

Configure Kafka producer and consumer clients to use Kerberos authentication

Configure Kafka producer and consumer clients to use the Kerberos Key Distribution Center (KDC) for authentication using the user principals and keytabs previously created for **producer1** and **consumer1**.

1. Add the Kerberos configuration to the producer or consumer configuration file.
For example:

/opt/kafka/config/producer.properties

```
# ...
sasl.mechanism=GSSAPI ❶
security.protocol=SASL_PLAINTEXT ❷
sasl.kerberos.service.name=kafka ❸
```

```
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \ 4
  useKeyTab=true \
  useTicketCache=false \
  storeKey=true \
  keyTab="/opt/kafka/krb5/producer1.keytab" \
  principal="producer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

- 1 Configuration for Kerberos (GSSAPI) authentication.
- 2 Kerberos uses the SASL plaintext (username/password) security protocol.
- 3 The service principal (user) for Kafka that was configured in the Kerberos KDC.
- 4 Configuration for the JAAS using the same properties defined in **jaas.conf**.

/opt/kafka/config/consumer.properties

```
# ...
sasl.mechanism=GSSAPI
security.protocol=SASL_PLAINTEXT
sasl.kerberos.service.name=kafka
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required \
  useKeyTab=true \
  useTicketCache=false \
  storeKey=true \
  keyTab="/opt/kafka/krb5/consumer1.keytab" \
  principal="consumer1/node1.example.redhat.com@EXAMPLE.REDHAT.COM";
# ...
```

2. Run the clients to verify that you can send and receive messages from the Kafka brokers.
Producer client:

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-producer.sh --producer.config
/opt/kafka/config/producer.properties --topic topic1 --bootstrap-server
node1.example.redhat.com:9094
```

Consumer client:

```
export KAFKA_HEAP_OPTS="-Djava.security.krb5.conf=/etc/krb5.conf -
Dsun.security.krb5.debug=true"; /opt/kafka/bin/kafka-console-consumer.sh --
consumer.config /opt/kafka/config/consumer.properties --topic topic1 --bootstrap-server
node1.example.redhat.com:9094
```

Additional resources

- Kerberos man pages: [krb5.conf\(5\)](#), [kinit\(1\)](#), [klist\(1\)](#), and [kdestroy\(1\)](#)
- [Example Kerberos server on RHEL set up configuration](#)
- [Example client application to authenticate with a Kafka cluster using Kerberos tickets](#)

CHAPTER 17. SETTING UP DISTRIBUTED TRACING

Distributed tracing allows you to track 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 AMQ Streams, tracing facilitates the end-to-end tracking of messages: from source systems to Kafka, and then from Kafka to target systems and applications. It complements the metrics that are available to view in [JMX metrics](#), as well as the component loggers.

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

- Kafka Connect
- MirrorMaker
- MirrorMaker 2
- AMQ Streams Kafka Bridge

Tracing is not supported for Kafka brokers.

You add tracing configuration to the properties file of the component.

To enable tracing, you set environment variables and add the library of the tracing system to the Kafka classpath. For Jaeger tracing, you can add tracing artifacts for OpenTelemetry with the Jaeger Exporter.



NOTE

AMQ Streams no longer supports OpenTracing. If you were previously using OpenTracing with Jaeger, we encourage you to transition to using OpenTelemetry instead.

To enable tracing in Kafka producers, consumers, and Kafka Streams API applications, you *instrument* application code. When instrumented, clients generate trace data; for example, when producing messages or writing offsets to the log.



NOTE

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

17.1. OUTLINE OF PROCEDURES

To set up tracing for AMQ Streams, follow these procedures in order:

- Set up tracing for Kafka Connect, MirrorMaker 2, and MirrorMaker:
 - [Enable tracing for Kafka Connect](#)
 - [Enable tracing for MirrorMaker 2](#)
 - [Enable tracing for MirrorMaker](#)

- 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](#)



NOTE

For information on enabling tracing for the Kafka Bridge, see [Using the AMQ Streams Kafka Bridge](#).

17.2. 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

The screenshot displays the Jaeger web interface. On the left, there is a search sidebar with the following settings:

- Search:** JSON File
- Service (4):** hello-world-producer
- Operation (1):** all
- Tags:** http.status_code=200 error=true
- Lookback:** Last Hour
- Min Duration:** e.g. 1.2s, 100ms, 500us
- Max Duration:** e.g. 1.2s, 100ms, 500us

The main area shows a scatter plot of trace durations over time, with a y-axis labeled 'Duration' ranging from 107ms to 109ms and an x-axis labeled 'Time' showing timestamps from 09:41:25 pm to 09:41:30 pm. Below the plot, it indicates '20 Traces' and a 'Sort: Most Recent' dropdown.

Below the plot, there is a section titled 'Compare traces by selecting result items' with two trace entries:

- Trace 1:** hello-world-producer: To_my-topic fc47c0db, 107.01ms. It consists of 2 spans: hello-world-consumer (1) and hello-world-producer (1). It was recorded today at 9:41:42 pm.
- Trace 2:** hello-world-producer: To_my-topic 4e1361b, 107.01ms. It consists of 4 spans: hello-world-consumer (1), hello-world-producer (1), and hello-world-streams (2). It was recorded today at 9:41:41 pm.

Additional resources

- [Jaeger documentation](#)

- [OpenTelemetry documentation](#)

17.3. 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 17.1. 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 otlp by default. If using Jaeger tracing, you need to set this environment variable as jaeger . If you are using another tracing implementation, specify the exporter used .

17.4. ENABLING TRACING FOR KAFKA CONNECT

Enable distributed tracing for Kafka Connect using configuration properties. Only messages produced and consumed by Kafka Connect itself are traced. To trace messages sent between Kafka Connect and external systems, you must configure tracing in the connectors for those systems.

You can enable tracing that uses OpenTelemetry.

Procedure

1. Add the tracing artifacts to the **opt/kafka/libs** directory.
2. Configure producer and consumer tracing in the relevant Kafka Connect configuration file.
 - If you are running Kafka Connect in standalone mode, edit the **/opt/kafka/config/connect-standalone.properties** file.
 - If you are running Kafka Connect in distributed mode, edit the **/opt/kafka/config/connect-distributed.properties** file.

Add the following tracing interceptor properties to the configuration file:

Properties for OpenTelemetry

```
producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInter
ceptor
consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerI
nterceptor
```

With tracing enabled, you initialize tracing when you run the Kafka Connect script.

3. Save the configuration file.
4. Set the [environment variables](#) for tracing.
5. Start Kafka Connect in standalone or distributed mode with the configuration file as a parameter (plus any connector properties):

Running Kafka Connect in standalone mode

```
su - kafka
/opt/kafka/bin/connect-standalone.sh \
/opt/kafka/config/connect-standalone.properties \
connector1.properties \
[connector2.properties ...]
```

Running Kafka Connect in distributed mode

```
su - kafka
/opt/kafka/bin/connect-distributed.sh /opt/kafka/config/connect-distributed.properties
```

The internal consumers and producers of Kafka Connect are now enabled for tracing.

17.5. ENABLING TRACING FOR MIRRORMAKER 2

Enable distributed tracing for MirrorMaker 2 by defining the Interceptor properties in the MirrorMaker 2 properties file. Messages are traced between Kafka clusters. The trace data records messages entering and leaving the MirrorMaker 2 component.

You can enable tracing that uses OpenTelemetry.

Procedure

1. Add the tracing artifacts to the **opt/kafka/libs** directory.
2. Configure producer and consumer tracing in the **opt/kafka/config/connect-mirror-maker.properties** file.

Add the following tracing interceptor properties to the configuration file:

Properties for OpenTelemetry

```
header.converter=org.apache.kafka.connect.converters.ByteArrayConverter
producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInter
ceptor
consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerI
nterceptor
```

ByteArrayConverter prevents Kafka Connect from converting message headers (containing trace IDs) to base64 encoding. This ensures that messages are the same in both the source and the target clusters.

With tracing enabled, you initialize tracing when you run the Kafka MirrorMaker 2 script.

3. Save the configuration file.
4. Set the [environment variables](#) for tracing.
5. Start MirrorMaker 2 with the producer and consumer configuration files as parameters:

```
su - kafka
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

The internal consumers and producers of MirrorMaker 2 are now enabled for tracing.

17.6. ENABLING TRACING FOR MIRRORMAKER

Enable distributed tracing for MirrorMaker by passing the Interceptor properties as consumer and producer configuration parameters. Messages are traced from the source cluster to the target cluster. The trace data records messages entering and leaving the MirrorMaker component.

You can enable tracing that uses OpenTelemetry.

Procedure

1. Add the tracing artifacts to the **opt/kafka/libs** directory.
2. Configure producer tracing in the **/opt/kafka/config/producer.properties** file. Add the following tracing interceptor property:

Producer property for OpenTelemetry

```
producer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingProducerInterceptor
```

3. Save the configuration file.
4. Configure consumer tracing in the **/opt/kafka/config/consumer.properties** file. Add the following tracing interceptor property:

Consumer property for OpenTelemetry

```
consumer.interceptor.classes=io.opentelemetry.instrumentation.kafkaclients.TracingConsumerInterceptor
```

With tracing enabled, you initialize tracing when you run the Kafka MirrorMaker script.

5. Save the configuration file.
6. Set the [environment variables](#) for tracing.
7. Start MirrorMaker with the producer and consumer configuration files as parameters:

```

su - kafka
/opt/kafka/bin/kafka-mirror-maker.sh \
--producer.config /opt/kafka/config/producer.properties \
--consumer.config /opt/kafka/config/consumer.properties \
--num.streams=2

```

The internal consumers and producers of MirrorMaker are now enabled for tracing.

17.7. 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</groupId>
  <artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>
  <version>1.19.0.redhat-00002</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry.instrumentation</groupId>
  <artifactId>opentelemetry-kafka-clients-2.6</artifactId>
  <version>1.19.0.redhat-00002</version>
</dependency>
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>opentelemetry-exporter-otlp</artifactId>
  <version>1.19.0.redhat-00002</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:

- [Section 17.8, "Instrumenting producers and consumers for tracing"](#)

- [Section 17.9, “Instrumenting Kafka Streams applications for tracing”](#)

17.8. 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);
```

17.9. 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());
```

17.10. SPECIFYING TRACING SYSTEMS WITH OPENTELEMETRY

Instead of the default Jaeger system, you can specify other tracing systems that are supported by OpenTelemetry.

If you want to use another tracing system with OpenTelemetry, do the following:

1. Add the library of the tracing system to the Kafka classpath.
2. Add the name of the tracing system as an additional exporter environment variable.

Additional environment variable when not using Jaeger

```
OTEL_SERVICE_NAME=my-tracing-service
OTEL_TRACES_EXPORTER=zipkin 1
OTEL_EXPORTER_ZIPKIN_ENDPOINT=http://localhost:9411/api/v2/spans 2
```

- 1** The name of the tracing system. In this example, Zipkin is specified.
- 2** The endpoint of the specific selected exporter that listens for spans. In this example, a Zipkin endpoint is specified.

Additional resources

- [OpenTelemetry exporter values](#)

17.11. CUSTOM SPAN NAMES

A tracing *span* is a logical unit of work in Jaeger, with an operation name, start time, and duration. Spans have built-in names, but you can specify custom span names in your Kafka client instrumentation where used.

Specifying custom span names is optional and only applies when using a decorator pattern [in producer and consumer client instrumentation](#) or [Kafka Streams instrumentation](#).

17.11.1. Specifying span names for OpenTelemetry

Custom span names cannot be specified directly with OpenTelemetry. Instead, you retrieve span names by adding code to your client application to extract additional tags and attributes.

Example code to extract attributes

```
//Defines attribute extraction for a producer
private static class ProducerAttribExtractor implements AttributesExtractor < ProducerRecord < ? , ? >
, Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord) {
        set(attributes, AttributeKey.stringKey("prod_start"), "prod1");
    }
}
```



```

    }
    @Override
    public void onEnd(AttributesBuilder attributes, ProducerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("prod_end"), "prod2");
    }
}
//Defines attribute extraction for a consumer
private static class ConsumerAttribExtractor implements AttributesExtractor < ConsumerRecord < ? ,
? > , Void > {
    @Override
    public void onStart(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord) {
        set(attributes, AttributeKey.stringKey("con_start"), "con1");
    }
    @Override
    public void onEnd(AttributesBuilder attributes, ConsumerRecord < ? , ? > producerRecord,
@Nullable Void unused, @Nullable Throwable error) {
        set(attributes, AttributeKey.stringKey("con_end"), "con2");
    }
}
//Extracts the attributes
public static void main(String[] args) throws Exception {
    Map < String, Object > configs = new HashMap < >
(Collections.singletonMap(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092"));
    System.setProperty("otel.traces.exporter", "jaeger");
    System.setProperty("otel.service.name", "myapp1");
    KafkaTracing tracing = KafkaTracing.newBuilder(GlobalOpenTelemetry.get())
        .addProducerAttributesExtractors(new ProducerAttribExtractor())
        .addConsumerAttributesExtractors(new ConsumerAttribExtractor())
        .build();
}

```

CHAPTER 18. USING KAFKA EXPORTER

[Kafka Exporter](#) is an open source project to enhance monitoring of Apache Kafka brokers and clients.

Kafka Exporter is provided with AMQ Streams for deployment with a Kafka cluster to extract 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.

If you are already using Prometheus and Grafana for monitoring of built-in Kafka metrics, you can configure Prometheus to also scrape the Kafka Exporter Prometheus endpoint.

Kafka exposes metrics through JMX, which can then be exported as Prometheus metrics. For more information, see [Monitoring your cluster using JMX](#).

18.1. CONSUMER LAG

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.

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

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 AMQ Streams 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.

18.2. KAFKA EXPORTER ALERTING RULE EXAMPLES

The sample alert notification rules specific to Kafka Exporter are as follows:

UnderReplicatedPartition

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

TooLargeConsumerGroupLag

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

NoMessageForTooLong

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

You can adapt alerting rules according to your specific needs.

Additional resources

For more information about setting up alerting rules, see [Configuration](#) in the Prometheus documentation.

18.3. KAFKA EXPORTER METRICS

Lag information is exposed by Kafka Exporter as Prometheus metrics for presentation in Grafana.

Kafka Exporter exposes metrics data for brokers, topics, and consumer groups.

Table 18.1. Broker metrics output

Name	Information
kafka_brokers	Number of brokers in the Kafka cluster

Table 18.2. Topic metrics output

Name	Information
kafka_topic_partitions	Number of partitions for a topic
kafka_topic_partition_current_offset	Current topic partition offset for a broker
kafka_topic_partition_oldest_offset	Oldest topic partition offset for a broker
kafka_topic_partition_in_sync_replica	Number of in-sync replicas for a topic partition
kafka_topic_partition_leader	Leader broker ID of a topic partition

Name	Information
kafka_topic_partition_leader_is_preferred	Shows 1 if a topic partition is using the preferred broker
kafka_topic_partition_replicas	Number of replicas for this topic partition
kafka_topic_partition_under_replicated_partition	Shows 1 if a topic partition is under-replicated

Table 18.3. Consumer group metrics output

Name	Information
kafka_consumergroup_current_offset	Current topic partition offset for a consumer group
kafka_consumergroup_lag	Current approximate lag for a consumer group at a topic partition

18.4. RUNNING KAFKA EXPORTER

Run Kafka Exporter to expose Prometheus metrics for presentation in a Grafana dashboard.

Download and install the Kafka Exporter package to use the Kafka Exporter with AMQ Streams. You need an AMQ Streams subscription to be able to download and install the package.

Prerequisites

- [AMQ Streams is installed on the host](#)
- [You have a subscription to AMQ Streams](#)

This procedure assumes you already have access to a Grafana user interface and Prometheus is deployed and added as a data source.

Procedure

1. Install the Kafka Exporter package:

```
dnf install kafka_exporter
```

2. Verify the package has installed:

```
dnf info kafka_exporter
```

3. Run the Kafka Exporter using appropriate configuration parameter values:

```
kafka_exporter --kafka.server=<kafka_bootstrap_address>:9092 --kafka.version=3.6.0 -  
-<my_other_parameters>
```

The parameters require a double-hyphen convention, such as **--kafka.server**.

Table 18.4. Kafka Exporter configuration parameters

Option	Description	Default
kafka.server	Host/post address of the Kafka server.	kafka:9092
kafka.version	Kafka broker version.	1.0.0
group.filter	A regular expression to specify the consumer groups to include in the metrics.	.* (all)
topic.filter	A regular expression to specify the topics to include in the metrics.	.* (all)
sasl.<parameter>	Parameters to enable and connect to the Kafka cluster using SASL/PLAIN authentication, with user name and password.	false
tls.<parameter>	Parameters to enable connect to the Kafka cluster using TLS authentication, with optional certificate and key.	false
web.listen-address	Port address to expose the metrics.	:9308
web.telemetry-path	Path for the exposed metrics.	/metrics
log.level	Logging configuration, to log messages with a given severity (debug, info, warn, error, fatal) or above.	info
log.enable-sarama	Boolean to enable Sarama logging, a Go client library used by the Kafka Exporter.	false
legacy.partitions	Boolean to enable metrics to be fetched from inactive topic partitions as well as from active partitions. If you want Kafka Exporter to return metrics for inactive partitions, set to true .	false

You can use `kafka_exporter --help` for information on the properties.

4. Configure Prometheus to monitor the Kafka Exporter metrics.
For more information on configuring Prometheus, see the [Prometheus documentation](#).
5. Enable Grafana to present the Kafka Exporter metrics data exposed by Prometheus.
For more information, see [Presenting Kafka Exporter metrics in Grafana](#).

Updating Kafka Exporter

Use the latest version of Kafka Exporter with your AMQ Streams installation.

To check for updates, use:

```
dnf check-update
```

To update Kafka Exporter, use:

```
dnf update kafka_exporter
```

18.5. PRESENTING KAFKA EXPORTER METRICS IN GRAFANA

Using Kafka Exporter Prometheus metrics as a data source, you can create a dashboard of Grafana charts.

For example, from the metrics you can create the following Grafana charts:

- Message in per second (from topics)
- Message in per minute (from topics)
- Lag by consumer group
- Messages consumed per minute (by consumer groups)

When metrics data has been collected for some time, the Kafka Exporter charts are populated.

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.

Additional resources

- [Example dashboard for Kafka Exporter](#)
- [Grafana documentation](#)

CHAPTER 19. UPGRADING AMQ STREAMS AND KAFKA

Upgrade your Kafka cluster with no downtime. Releases of AMQ Streams for Red Hat Enterprise Linux support one version of Apache Kafka. You upgrade to the latest supported version of Kafka when you install the latest version of AMQ Streams.

19.1. UPGRADE PREREQUISITES

Before you begin the upgrade process, make sure you are familiar with any upgrade changes described in the [AMQ Streams 2.6 on Red Hat Enterprise Linux Release Notes](#) .



NOTE

Refer to the documentation supporting a specific version of AMQ Streams for information on how to upgrade to that version.

19.2. KAFKA VERSIONS

Kafka's log message format version and inter-broker protocol version specify, respectively, the log format version appended to messages and the version of the Kafka protocol used in a cluster. To ensure the correct versions are used, the upgrade process involves making configuration changes to existing Kafka brokers and code changes to client applications (consumers and producers).

The following table shows the differences between Kafka versions:

Table 19.1. Kafka version differences

AMQ Streams version	Kafka version	Inter-broker protocol version	Log message format version	ZooKeeper version
2.6	3.6.0	3.6	3.6	3.8.3
2.5	3.5.0	3.5	3.5	3.6.4

- Kafka 3.6.0 is supported for production use.
- Kafka 3.5.0 is supported only for the purpose of upgrading to AMQ Streams 2.6.

Inter-broker protocol version

In Kafka, the network protocol used for inter-broker communication is called the *inter-broker protocol*. Each version of Kafka has a compatible version of the inter-broker protocol. The minor version of the protocol typically increases to match the minor version of Kafka, as shown in the preceding table.

The inter-broker protocol version is set cluster wide in the **Kafka** resource. To change it, you edit the **inter.broker.protocol.version** property in **Kafka.spec.kafka.config**.

Log message format version

When a producer sends a message to a Kafka broker, the message is encoded using a specific format. The format can change between Kafka releases, so messages specify which version of the message format they were encoded with.

The properties used to set a specific message format version are as follows:

- **message.format.version** property for topics
- **log.message.format.version** property for Kafka brokers

From Kafka 3.0.0, the message format version values are assumed to match the **inter.broker.protocol.version** and don't need to be set. The values reflect the Kafka version used.

When upgrading to Kafka 3.0.0 or higher, you can remove these settings when you update the **inter.broker.protocol.version**. Otherwise, set the message format version based on the Kafka version you are upgrading to.

The default value of **message.format.version** for a topic is defined by the **log.message.format.version** that is set on the Kafka broker. You can manually set the **message.format.version** of a topic by modifying its topic configuration.

19.3. 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.

Upgrading clients when using Kafka versions older than Kafka 3.0

Before Kafka 3.0, you would 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 support older Kafka clients that were using an outdated message format. Otherwise, the brokers would 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>**

TIP

The following Kafka broker metrics help monitor the performance of message down-conversion:

- **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.

19.4. UPGRADING KAFKA BROKERS AND ZOOKEEPER

Upgrade Kafka brokers and ZooKeeper on a host machine to use the latest version of AMQ Streams. You update the installation files, then configure and restart all Kafka brokers to use a new inter-broker protocol version. After performing these steps, data is transmitted between the Kafka brokers using the new inter-broker protocol version.

**NOTE**

From Kafka 3.0.0, 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.

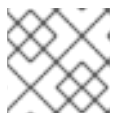
Prerequisites

- You are logged in to Red Hat Enterprise Linux as the **kafka** user.
- You have installed Kafka and other Kafka components you are using on separate hosts. For more information, see [Section 3.1, "Installation environment"](#).
- You have downloaded the [installation files](#).

Procedure

For each Kafka broker in your AMQ Streams cluster and one at a time:

1. Download the AMQ Streams archive from the [AMQ Streams software downloads page](#).

**NOTE**

If prompted, log in to your Red Hat account.

2. On the command line, create a temporary directory and extract the contents of the **amq-streams-<version>-bin.zip** file.

```
mkdir /tmp/kafka
unzip amq-streams-<version>-bin.zip -d /tmp/kafka
```

3. If running, stop ZooKeeper and the Kafka broker running on the host.

```
/opt/kafka/bin/zookeeper-server-stop.sh
/opt/kafka/bin/kafka-server-stop.sh
jcmd | grep zookeeper
```

```
│ jcmd | grep kafka
```

If you are running Kafka on a multi-node cluster, see [Section 4.3, “Performing a graceful rolling restart of Kafka brokers”](#).

4. Delete the **libs** and **bin** directories from your existing installation:

```
│ rm -rf /opt/kafka/libs /opt/kafka/bin
```

5. Copy the **libs** and **bin** directories from the temporary directory:

```
│ cp -r /tmp/kafka/kafka_<version>/libs /opt/kafka/  
│ cp -r /tmp/kafka/kafka_<version>/bin /opt/kafka/
```

6. If required, update the configuration files in the **config** directory to reflect any changes in the new versions.
7. Delete the temporary directory.

```
│ rm -r /tmp/kafka
```

8. Edit the **/opt/kafka/config/server.properties** properties file.
Set the **inter.broker.protocol.version** and **log.message.format.version** properties to the *current* version.

For example, the current version is 3.5 if upgrading from Kafka version 3.5.0 to 3.6.0:

```
│ inter.broker.protocol.version=3.5  
│ log.message.format.version=3.5
```

Use the correct version for the Kafka version you are upgrading from (**3.4**, **3.5**, and so on). 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.

If the properties are not configured, add them with the current version.

If you are upgrading from Kafka 3.0.0 or later, you only need to set the **inter.broker.protocol.version**.

9. Restart the updated ZooKeeper and Kafka broker:

```
│ /opt/kafka/bin/zookeeper-server-start.sh -daemon /opt/kafka/config/zookeeper.properties  
│ /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

The Kafka broker and ZooKeeper start using the binaries for the latest Kafka version.

For information on restarting brokers in a multi-node cluster, see [Section 4.3, “Performing a graceful rolling restart of Kafka brokers”](#).

10. Verify that the restarted Kafka broker has caught up with the partition replicas it is following.
Use the **kafka-topics.sh** tool to ensure that all replicas contained in the broker are back in sync.
For instructions, see [Listing and describing topics](#).

In the next steps, update your Kafka brokers to use the new inter-broker protocol version.

Update each broker, one at a time.



WARNING

Downgrading AMQ Streams is not possible after completing the following steps.

11. Depending on your chosen [strategy for upgrading clients](#), upgrade all client applications to use the new version of the client binaries.
12. Set the **inter.broker.protocol.version** property to **3.6** in the **/opt/kafka/config/server.properties** file:

```
inter.broker.protocol.version=3.6
```

13. On the command line, stop the Kafka broker that you modified:

```
/opt/kafka/bin/kafka-server-stop.sh
```

14. Check that Kafka is not running:

```
jcmd | grep kafka
```

15. Restart the Kafka broker that you modified:

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

16. Check that Kafka is running:

```
jcmd | grep kafka
```

17. If you are upgrading from a version earlier than Kafka 3.0.0, set the **log.message.format.version** property to **3.6** in the **/opt/kafka/config/server.properties** file:

```
log.message.format.version=3.6
```

18. On the command line, stop the Kafka broker that you modified:

```
/opt/kafka/bin/kafka-server-stop.sh
```

19. Check that Kafka is not running:

```
jcmd | grep kafka
```

20. Restart the Kafka broker that you modified:

```
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties
```

21. Check that Kafka is running:

```
jcmd | grep kafka
```

22. Verify that the restarted Kafka broker has caught up with the partition replicas it is following. Use the **kafka-topics.sh** tool to ensure that all replicas contained in the broker are back in sync. For instructions, see [Listing and describing topics](#).
23. If it was used in the upgrade, remove the legacy **log.message.format.version** configuration from the **server.properties** file.

19.5. UPGRADING KAFKA COMPONENTS

Upgrade Kafka components on a host machine to use the latest version of AMQ Streams. You can use the AMQ Streams installation files to upgrade the following components:

- Kafka Connect
- MirrorMaker
- Kafka Bridge (separate ZIP file)

Prerequisites

- You are logged in to Red Hat Enterprise Linux as the **kafka** user.
- You have downloaded the [installation files](#).
- You have installed Kafka and other Kafka components you are using on separate hosts. For more information, see [Section 3.1, "Installation environment"](#).
- You have [upgraded Kafka and ZooKeeper](#).
If a Kafka component is running on the same host as Kafka and ZooKeeper, you'll also need to stop and start Kafka and ZooKeeper when upgrading.

Procedure

For each host running an instance of the Kafka component:

1. Download the AMQ Streams or Kafka Bridge installation files from the [AMQ Streams software downloads page](#).



NOTE

If prompted, log in to your Red Hat account.

2. On the command line, create a temporary directory and extract the contents of the **amq-streams-<version>-bin.zip** file.

```
mkdir /tmp/kafka
unzip amq-streams-<version>-bin.zip -d /tmp/kafka
```

For Kafka Bridge, extract the **amq-streams-<version>-bridge-bin.zip** file.

3. If running, stop the Kafka component running on the host.

4. Delete the **libs** and **bin** directories from your existing installation:

```
rm -rf /opt/kafka/libs /opt/kafka/bin
```

5. Copy the **libs** and **bin** directories from the temporary directory:

```
cp -r /tmp/kafka/kafka_<version>/libs /opt/kafka/
cp -r /tmp/kafka/kafka_<version>/bin /opt/kafka/
```

6. If required, update the configuration files in the **config** directory to reflect any changes in the new versions.
7. Delete the temporary directory.

```
rm -r /tmp/kafka
```

8. Start the Kafka component using the appropriate script and properties files.

Starting Kafka Connect in standalone mode

```
/opt/kafka/bin/connect-standalone.sh \
/opt/kafka/config/connect-standalone.properties <connector1>.properties
[<connector2>.properties ...]
```

Starting Kafka Connect in distributed mode

```
/opt/kafka/bin/connect-distributed.sh \
/opt/kafka/config/connect-distributed.properties
```

Starting MirrorMaker 2 in dedicated mode

```
/opt/kafka/bin/connect-mirror-maker.sh \
/opt/kafka/config/connect-mirror-maker.properties
```

Starting Kafka Bridge

```
su - kafka
./bin/kafka_bridge_run.sh \
--config-file=<path>/application.properties
```

9. Verify that the Kafka component is running, and producing or consuming data as expected.

Verifying Kafka Connect in standalone mode is running

```
jcmd | grep ConnectStandalone
```

Verifying Kafka Connect in distributed mode is running

```
jcmd | grep ConnectDistributed
```

Verifying MirrorMaker 2 in dedicated mode is running

`jqcmd | grep mirrorMaker`

Verifying Kafka Bridge is running by checking the log

HTTP-Kafka Bridge started and listening on port 8080
HTTP-Kafka Bridge bootstrap servers localhost:9092

CHAPTER 20. MONITORING YOUR CLUSTER USING JMX

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.

Kafka brokers, ZooKeeper, Kafka Connect, and Kafka clients use Java Management Extensions (JMX) to actively expose management information. This information primarily consists of metrics that help monitor the performance and condition of the Kafka cluster. Kafka, like other Java applications, relies on managed beans or MBeans to provide this information to monitoring tools and dashboards.

JMX operates at the JVM level, allowing external tools to connect and retrieve management information from the ZooKeeper, Kafka broker, and so on. To connect to the JVM, these tools must be running on the same machine and as the same user by default.

20.1. ENABLING THE JMX AGENT

Enable JMX monitoring of Kafka components using JVM system properties. Use the **KAFKA_JMX_OPTS** environment variable to set the JMX system properties required for enabling JMX monitoring. The scripts that run the Kafka component use these properties.

Procedure

1. Set the **KAFKA_JMX_OPTS** environment variable with the JMX properties for enabling JMX monitoring.

```
export KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote=true
-Dcom.sun.management.jmxremote.port=<port>
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

Replace <port> with the name of the port on which you want the Kafka component to listen for JMX connections.

2. Add **org.apache.kafka.common.metrics.JmxReporter** to **metric.reporters** in the **server.properties** file.

```
metric.reporters=org.apache.kafka.common.metrics.JmxReporter
```

3. Start the Kafka component using the appropriate script, such as **bin/kafka-server-start.sh** for a broker or **bin/connect-distributed.sh** for Kafka Connect.



IMPORTANT

It is recommended that you configure authentication and SSL to secure a remote JMX connection. For more information about the system properties needed to do this, see the [Oracle documentation](#).

20.2. DISABLING THE JMX AGENT

Disable JMX monitoring for Kafka components by updating the **KAFKA_JMX_OPTS** environment variable.

Procedure

1. Set the **KAFKA_JMX_OPTS** environment variable to disable JMX monitoring.

```
export KAFKA_JMX_OPTS=-Dcom.sun.management.jmxremote=false
```

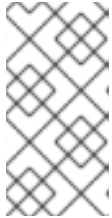


NOTE

Other JMX properties, like port, authentication, and SSL properties do not need to be specified when disabling JMX monitoring.

2. Set **auto.include.jmx.reporter** to **false** in the Kafka **server.properties** file.

```
auto.include.jmx.reporter=false
```



NOTE

The **auto.include.jmx.reporter** property is deprecated. From Kafka 4, the JMXReporter is only enabled if **org.apache.kafka.common.metrics.JmxReporter** is added to the **metric.reporters** configuration in the properties file.

3. Start the Kafka component using the appropriate script, such as **bin/kafka-server-start.sh** for a broker or **bin/connect-distributed.sh** for Kafka Connect.

20.3. METRICS NAMING CONVENTIONS

When working with Kafka JMX metrics, it's important to understand the naming conventions used to identify and retrieve specific metrics. Kafka JMX metrics use the following format:

Metrics format

```
<metric_group>:type=<type_name>,name=<metric_name><other_attribute>=<value>
```

- <metric_group> is the name of the metric group
- <type_name> is the name of the type of metric
- <metric_name> is the name of the specific metric
- <other_attribute> represents zero or more additional attributes

For example, the **BytesInPerSec** metric is a **BrokerTopicMetrics** type in the **kafka.server** group:

```
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec
```

In some cases, metrics may include the ID of an entity. For instance, when monitoring a specific client, the metric format includes the client ID:

Metrics for a specific client

```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<client_id>
```

Similarly, a metric can be further narrowed down to a specific client and topic:

Metrics for a specific client and topic

```
kafka.consumer:type=consumer-fetch-manager-metrics,client-id=<client_id>,topic=<topic_id>
```

Understanding these naming conventions will allow you to accurately specify the metrics you want to monitor and analyze.



NOTE

To view the full list of available JMX metrics for a Strimzi installation, you can use a graphical tool like JConsole. JConsole is a Java Monitoring and Management Console that allows you to monitor and manage Java applications, including Kafka. By connecting to the JVM running the Kafka component using its process ID, the tool's user interface allows you to view the list of metrics.

20.4. ANALYZING KAFKA JMX METRICS FOR TROUBLESHOOTING

JMX provides a way to gather metrics about Kafka brokers for monitoring and managing their performance and resource usage. By analyzing these metrics, common broker issues such as high CPU usage, memory leaks, thread contention, and slow response times can be diagnosed and resolved. Certain metrics can pinpoint the root cause of these issues.

JMX metrics also provide insights into the overall health and performance of a Kafka cluster. They help monitor the system's throughput, latency, and availability, diagnose issues, and optimize performance. This section explores the use of JMX metrics to help identify common issues and provides insights into the performance of a Kafka cluster.

Collecting and graphing these metrics using tools like Prometheus and Grafana allows you to visualize the information returned. This can be particularly helpful in detecting issues or optimizing performance. Graphing metrics over time can also help with identifying trends and forecasting resource consumption.

20.4.1. Checking for under-replicated partitions

A balanced Kafka cluster is important for optimal performance. In a balanced cluster, partitions and leaders are evenly distributed across all brokers, and I/O metrics reflect this. As well as using metrics, you can use the **kafka-topics.sh** tool to get a list of under-replicated partitions and identify the problematic brokers. If the number of under-replicated partitions is fluctuating or many brokers show high request latency, this typically indicates a performance issue in the cluster that requires investigation. On the other hand, a steady (unchanging) number of under-replicated partitions reported by many of the brokers in a cluster normally indicates that one of the brokers in the cluster is offline.

Use the **describe --under-replicated-partitions** option from the **kafka-topics.sh** tool to show information about partitions that are currently under-replicated in the cluster. These are the partitions that have fewer replicas than the configured replication factor.

If the output is blank, the Kafka cluster has no under-replicated partitions. Otherwise, the output shows replicas that are not in sync or available.

In the following example, only 2 of the 3 replicas are in sync for each partition, with a replica missing from the ISR (in-sync replica).

Returning information on under-replicated partitions from the command line

```
bin/kafka-topics.sh --bootstrap-server :9092 --describe --under-replicated-partitions
```

```
Topic: topic-1 Partition: 0 Leader: 4 Replicas: 4,2,3 Isr: 4,3
```

```
Topic: topic-1 Partition: 1 Leader: 3 Replicas: 2,3,4 Isr: 3,4
```

```
Topic: topic-1 Partition: 2 Leader: 3 Replicas: 3,4,2 Isr: 3,4
```

Here are some metrics to check for I/O and under-replicated partitions:

Metrics to check for under-replicated partitions

```
kafka.server:type=ReplicaManager,name=PartitionCount 1
kafka.server:type=ReplicaManager,name=LeaderCount 2
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec 3
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec 4
kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions 5
kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount 6
```

- 1** Total number of partitions across all topics in the cluster.
- 2** Total number of leaders across all topics in the cluster.
- 3** Rate of incoming bytes per second for each broker.
- 4** Rate of outgoing bytes per second for each broker.
- 5** Number of under-replicated partitions across all topics in the cluster.
- 6** Number of partitions below the minimum ISR.

If topic configuration is set for high availability, with a replication factor of at least 3 for topics and a minimum number of in-sync replicas being 1 less than the replication factor, under-replicated partitions can still be usable. Conversely, partitions below the minimum ISR have reduced availability. You can monitor these using the **kafka.server:type=ReplicaManager,name=UnderMinIsrPartitionCount** metric and the **under-min-isr-partitions** option from the **kafka-topics.sh** tool.

TIP

Use Cruise Control to automate the task of monitoring and rebalancing a Kafka cluster to ensure that the partition load is evenly distributed. For more information, see [Chapter 14, Using Cruise Control for cluster rebalancing](#).

20.4.2. Identifying performance problems in a Kafka cluster

Spikes in cluster metrics may indicate a broker issue, which is often related to slow or failing storage devices or compute restraints from other processes. If there is no issue at the operating system or hardware level, an imbalance in the load of the Kafka cluster is likely, with some partitions receiving disproportionate traffic compared to others in the same Kafka topic.

To anticipate performance problems in a Kafka cluster, it's useful to monitor the **RequestHandlerAvgIdlePercent** metric. **RequestHandlerAvgIdlePercent** provides a good overall indicator of how the cluster is behaving. The value of this metric is between 0 and 1. A value below 0.7 indicates that threads are busy 30% of the time and performance is starting to degrade. If the value drops below 50%, problems are likely to occur, especially if the cluster needs to scale or rebalance. At 30%, a cluster is barely usable.

Another useful metric is **kafka.network:type=Processor,name=IdlePercent**, which you can use to monitor the extent (as a percentage) to which network processors in a Kafka cluster are idle. The metric helps identify whether the processors are over or underutilized.

To ensure optimal performance, set the **num.io.threads** property equal to the number of processors in the system, including hyper-threaded processors. If the cluster is balanced, but a single client has changed its request pattern and is causing issues, reduce the load on the cluster or increase the number of brokers.

It's important to note that a single disk failure on a single broker can severely impact the performance of an entire cluster. Since producer clients connect to all brokers that lead partitions for a topic, and those partitions are evenly spread over the entire cluster, a poorly performing broker will slow down produce requests and cause back pressure in the producers, slowing down requests to all brokers. A RAID (Redundant Array of Inexpensive Disks) storage configuration that combines multiple physical disk drives into a single logical unit can help prevent this issue.

Here are some metrics to check the performance of a Kafka cluster:

Metrics to check the performance of a Kafka cluster

```
kafka.server:type=KafkaRequestHandlerPool,name=RequestHandlerAvgIdlePercent 1
# attributes: OneMinuteRate, FifteenMinuteRate
kafka.server:type=socket-server-metrics,listener=([-.\w]+),networkProcessor=([\d]+) 2
# attributes: connection-creation-rate
kafka.network:type=RequestChannel,name=RequestQueueSize 3
kafka.network:type=RequestChannel,name=ResponseQueueSize 4
kafka.network:type=Processor,name=IdlePercent,networkProcessor=([-.\w]+) 5
kafka.server:type=KafkaServer,name=TotalDiskReadBytes 6
kafka.server:type=KafkaServer,name=TotalDiskWriteBytes 7
```

- 1 Average idle percentage of the request handler threads in the Kafka broker's thread pool. The **OneMinuteRate** and **FifteenMinuteRate** attributes show the request rate of the last one minute and fifteen minutes, respectively.
- 2 Rate at which new connections are being created on a specific network processor of a specific listener in the Kafka broker. The **listener** attribute refers to the name of the listener, and the **networkProcessor** attribute refers to the ID of the network processor. The **connection-creation-rate** attribute shows the rate of connection creation in connections per second.
- 3 Current size of the request queue.
- 4 Current sizes of the response queue.
- 5 Percentage of time the specified network processor is idle. The **networkProcessor** specifies the ID of the network processor to monitor.
- 6 Total number of bytes read from disk by a Kafka server.

- 7 Total number of bytes written to disk by a Kafka server.

20.4.3. Identifying performance problems with a Kafka controller

The Kafka controller is responsible for managing the overall state of the cluster, such as broker registration, partition reassignment, and topic management. Problems with the controller in the Kafka cluster are difficult to diagnose and often fall into the category of bugs in Kafka itself. Controller issues might manifest as broker metadata being out of sync, offline replicas when the brokers appear to be fine, or actions on topics like topic creation not happening correctly.

There are not many ways to monitor the controller, but you can monitor the active controller count and the controller queue size. Monitoring these metrics gives a high-level indicator if there is a problem. Although spikes in the queue size are expected, if this value continuously increases, or stays steady at a high value and does not drop, it indicates that the controller may be stuck. If you encounter this problem, you can move the controller to a different broker, which requires shutting down the broker that is currently the controller.

Here are some metrics to check the performance of a Kafka controller:

Metrics to check the performance of a Kafka controller

```
kafka.controller:type=KafkaController,name=ActiveControllerCount 1  
kafka.controller:type=KafkaController,name=OfflinePartitionsCount 2  
kafka.controller:type=ControllerEventManager,name=EventQueueSize 3
```

- 1 Number of active controllers in the Kafka cluster. A value of 1 indicates that there is only one active controller, which is the desired state.
- 2 Number of partitions that are currently offline. If this value is continuously increasing or stays at a high value, there may be a problem with the controller.
- 3 Size of the event queue in the controller. Events are actions that must be performed by the controller, such as creating a new topic or moving a partition to a new broker. If the value continuously increases or stays at a high value, the controller may be stuck and unable to perform the required actions.

20.4.4. Identifying problems with requests

You can use the **RequestHandlerAvgIdlePercent** metric to determine if requests are slow. Additionally, request metrics can identify which specific requests are experiencing delays and other issues.

To effectively monitor Kafka requests, it is crucial to collect two key metrics: count and 99th percentile latency, also known as tail latency.

The count metric represents the number of requests processed within a specific time interval. It provides insights into the volume of requests handled by your Kafka cluster and helps identify spikes or drops in traffic.

The 99th percentile latency metric measures the request latency, which is the time taken for a request to be processed. It represents the duration within which 99% of requests are handled. However, it does not provide information about the exact duration for the remaining 1% of requests. In other words, the 99th percentile latency metric tells you that 99% of the requests are handled within a certain duration,

and the remaining 1% may take even longer, but the precise duration for this remaining 1% is not known. The choice of the 99th percentile is primarily to focus on the majority of requests and exclude outliers that can skew the results.

This metric is particularly useful for identifying performance issues and bottlenecks related to the majority of requests, but it does not give a complete picture of the maximum latency experienced by a small fraction of requests.

By collecting and analyzing both count and 99th percentile latency metrics, you can gain an understanding of the overall performance and health of your Kafka cluster, as well as the latency of the requests being processed.

Here are some metrics to check the performance of Kafka requests:

Metrics to check the performance of requests

```
# requests: EndTxn, Fetch, FetchConsumer, FetchFollower, FindCoordinator, Heartbeat,
InitProducerId,
# JoinGroup, LeaderAndIsr, LeaveGroup, Metadata, Produce, SyncGroup, UpdateMetadata 1
kafka.network:type=RequestMetrics,name=RequestsPerSec,request=(\w+) 2
kafka.network:type=RequestMetrics,name=RequestQueueTimeMs,request=(\w+) 3
kafka.network:type=RequestMetrics,name=TotalTimeMs,request=(\w+) 4
kafka.network:type=RequestMetrics,name=LocalTimeMs,request=(\w+) 5
kafka.network:type=RequestMetrics,name=RemoteTimeMs,request=(\w+) 6
kafka.network:type=RequestMetrics,name=ThrottleTimeMs,request=(\w+) 7
kafka.network:type=RequestMetrics,name=ResponseQueueTimeMs,request=(\w+) 8
kafka.network:type=RequestMetrics,name=ResponseSendTimeMs,request=(\w+) 9
# attributes: Count, 99thPercentile 10
```

- 1 Request types to break down the request metrics.
- 2 Rate at which requests are being processed by the Kafka broker per second.
- 3 Time (in milliseconds) that a request spends waiting in the broker's request queue before being processed.
- 4 Total time (in milliseconds) that a request takes to complete, from the time it is received by the broker to the time the response is sent back to the client.
- 5 Time (in milliseconds) that a request spends being processed by the broker on the local machine.
- 6 Time (in milliseconds) that a request spends being processed by other brokers in the cluster.
- 7 Time (in milliseconds) that a request spends being throttled by the broker. Throttling occurs when the broker determines that a client is sending too many requests too quickly and needs to be slowed down.
- 8 Time (in milliseconds) that a response spends waiting in the broker's response queue before being sent back to the client.
- 9 Time (in milliseconds) that a response takes to be sent back to the client after it has been generated by the broker.
- 10 For all of the requests metrics, the **Count** and **99thPercentile** attributes show the total number of requests that have been processed and the time it takes for the slowest 1% of requests to complete, respectively.

complete, respectively.

20.4.5. Using metrics to check the performance of clients

By analyzing client metrics, you can monitor the performance of the Kafka clients (producers and consumers) connected to a broker. This can help identify issues highlighted in broker logs, such as consumers being frequently kicked off their consumer groups, high request failure rates, or frequent disconnections.

Here are some metrics to check the performance of Kafka clients:

Metrics to check the performance of client requests

```
kafka.consumer:type=consumer-metrics,client-id=[-.\w]+ 1
# attributes: time-between-poll-avg, time-between-poll-max
kafka.consumer:type=consumer-coordinator-metrics,client-id=[-.\w]+ 2
# attributes: heartbeat-response-time-max, heartbeat-rate, join-time-max, join-rate, rebalance-rate-per-hour
kafka.producer:type=producer-metrics,client-id=[-.\w]+ 3
# attributes: buffer-available-bytes, bufferpool-wait-time, request-latency-max, requests-in-flight
# attributes: txn-init-time-ns-total, txn-begin-time-ns-total, txn-send-offsets-time-ns-total, txn-commit-time-ns-total, txn-abort-time-ns-total
# attributes: record-error-total, record-queue-time-avg, record-queue-time-max, record-retry-rate, record-retry-total, record-send-rate, record-send-total
```

- 1 (Consumer) Average and maximum time between poll requests, which can help determine if the consumers are polling for messages frequently enough to keep up with the message flow. The **time-between-poll-avg** and **time-between-poll-max** attributes show the average and maximum time in milliseconds between successive polls by a consumer, respectively.
- 2 (Consumer) Metrics to monitor the coordination process between Kafka consumers and the broker coordinator. Attributes relate to the heartbeat, join, and rebalance process.
- 3 (Producer) Metrics to monitor the performance of Kafka producers. Attributes relate to buffer usage, request latency, in-flight requests, transactional processing, and record handling.

20.4.6. Using metrics to check the performance of topics and partitions

Metrics for topics and partitions can also be helpful in diagnosing issues in a Kafka cluster. You can also use them to debug issues with a specific client when you are unable to collect client metrics.

Here are some metrics to check the performance of a specific topic and partition:

Metrics to check the performance of topics and partitions

```
#Topic metrics
kafka.server:type=BrokerTopicMetrics,name=BytesInPerSec,topic=[-.\w]+ 1
kafka.server:type=BrokerTopicMetrics,name=BytesOutPerSec,topic=[-.\w]+ 2
kafka.server:type=BrokerTopicMetrics,name=FailedFetchRequestPerSec,topic=[-.\w]+ 3
kafka.server:type=BrokerTopicMetrics,name=FailedProduceRequestsPerSec,topic=[-.\w]+ 4
kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec,topic=[-.\w]+ 5
kafka.server:type=BrokerTopicMetrics,name=TotalFetchRequestPerSec,topic=[-.\w]+ 6
```

```

kafka.server:type=BrokerTopicMetrics,name=TotalProduceRequestsPerSec,topic=([-.\w]+) 7
#Partition metrics
kafka.log:type=Log,name=Size,topic=([-.\w]+),partition=(\d+) 8
kafka.log:type=Log,name=NumLogSegments,topic=([-.\w]+),partition=(\d+) 9
kafka.log:type=Log,name=LogEndOffset,topic=([-.\w]+),partition=(\d+) 10
kafka.log:type=Log,name=LogStartOffset,topic=([-.\w]+),partition=(\d+) 11

```

- 1 Rate of incoming bytes per second for a specific topic.
- 2 Rate of outgoing bytes per second for a specific topic.
- 3 Rate of fetch requests that failed per second for a specific topic.
- 4 Rate of produce requests that failed per second for a specific topic.
- 5 Incoming message rate per second for a specific topic.
- 6 Total rate of fetch requests (successful and failed) per second for a specific topic.
- 7 Total rate of fetch requests (successful and failed) per second for a specific topic.
- 8 Size of a specific partition's log in bytes.
- 9 Number of log segments in a specific partition.
- 10 Offset of the last message in a specific partition's log.
- 11 Offset of the first message in a specific partition's log

Additional resources

- [Apache Kafka documentation](#) for a full list of available metrics
- [Prometheus documentation](#)
- [Grafana documentation](#)

APPENDIX A. USING YOUR SUBSCRIPTION

AMQ Streams is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing Your Account

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

Activating a Subscription

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading Zip and Tar Files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Locate the **AMQ Streams for Apache Kafka** entries in the **INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ Streams product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Installing packages with DNF

To install a package and all the package dependencies, use:

```
dnf install <package_name>
```

To install a previously-downloaded package from a local directory, use:

```
dnf install <path_to_download_package>
```

Revised on 2024-04-16 07:43:45 UTC