Red Hat AMQ Streams 1.0

Using AMQ Streams on OpenShift Container Platform

Last Updated: 2018-10-25

Red Hat AMQ Streams 1.0 Using AMQ Streams on OpenShift Container Platform

Legal Notice

Copyright © 2018 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, 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 Software Collections 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

This guide describes how to install, configure, and manage Red Hat AMQ Streams to build a large-scale messaging network.

Table of Contents

CHAPTER 1. OVERVIEW OF AMQ STREAMS **11**
1.1. KAFKA KEY FEATURES 11
1.2. DOCUMENT CONVENTIONS 11

CHAPTER 2. GETTING STARTED WITH AMQ STREAMS **12**
2.1. INSTALLING AMQ STREAMS AND DEPLOYING COMPONENTS 12
2.2. CLUSTER OPERATOR 12
2.2.1. Overview of the Cluster Operator component 12
2.2.2. Deploying the Cluster Operator to OpenShift 13
2.2.3. Deploying the Cluster Operator to watch multiple namespaces 14
2.3. KAFKA CLUSTER 15
2.3.1. Deploying the Kafka cluster to OpenShift 16
2.4. KAFKA CONNECT 16
2.4.1. Deploying Kafka Connect to OpenShift 16
2.4.2. Using Kafka Connect with plugins 17
2.4.2.1. Create a new image based on our base image 17
2.4.2.2. Using OpenShift builds and S2I to create new images 18
2.5. KAFKA MIRROR MAKER 19
2.5.1. Deploying Kafka Mirror Maker to OpenShift 19
2.6. DEPLOYING EXAMPLE CLIENTS 19
2.7. TOPIC OPERATOR 20
2.7.1. Overview of the Topic Operator component 20
2.7.2. Deploying the Topic Operator using the Cluster Operator 21
2.8. USER OPERATOR 22
2.8.1. Overview of the User Operator component 22
2.8.2. Deploying the User Operator using the Cluster Operator 23

CHAPTER 3. DEPLOYMENT CONFIGURATION **24**
3.1. KAFKA CLUSTER CONFIGURATION 24
3.1.1. Kafka and Zookeeper storage 24
3.1.1.1. Ephemeral storage 24
3.1.1.2. Persistent storage 25
3.1.2. Replicas 26
3.1.2.1. Configuring the number of broker nodes 27
3.1.3. Kafka broker configuration 27
3.1.3.1. Kafka broker configuration 28
3.1.3.2. Configuring Kafka brokers 29
3.1.4. Kafka broker listeners 30
3.1.4.1. Mutual TLS authentication for clients 30
3.1.4.1.1. Mutual TLS authentication 30
3.1.4.1.2. When to use mutual TLS authentication for clients 30
3.1.4.2. SCRAM-SHA authentication 31
3.1.4.2.1. Supported SCRAM credentials 31
3.1.4.2.2. When to use SCRAM-SHA authentication for clients 31
3.1.4.3. Kafka listeners 31
3.1.4.3.1. External listener 32
3.1.4.3.2. Listener authentication 33
3.1.4.4. Configuring Kafka listeners 33
3.1.4.5. Accessing Kafka using OpenShift routes 34
3.1.4.6. Accessing Kafka using loadbalancers routes 35
3.1.4.7. Accessing Kafka using node ports routes 36
3.1.5. Authentication and Authorization 38

3.1.5.1. Authentication 38
3.1.5.1.1. TLS client authentication 38
3.1.5.2. Configuring authentication in Kafka brokers 38
3.1.5.3. Authorization 39
3.1.5.3.1. Simple authorization 39
3.1.5.4. Configuring authorization in Kafka brokers 39
3.1.6. Replicas 40
3.1.6.1. Number of Zookeeper nodes 41
3.1.6.2. Changing number of replicas 41
3.1.7. Zookeeper configuration 41
3.1.7.1. Zookeeper configuration 42
3.1.7.2. Configuring Zookeeper 43
3.1.8. Entity Operator 44
3.1.8.1. Configuration 44
3.1.8.1.1. Topic Operator 45
3.1.8.1.2. User Operator 46
3.1.8.2. Configuring Entity Operator 46
3.1.9. CPU and memory resources 47
3.1.9.1. Resource limits and requests 47
3.1.9.1.1. Resource requests 48
3.1.9.1.2. Resource limits 49
3.1.9.1.3. Supported CPU formats 49
3.1.9.1.4. Supported memory formats 50
3.1.9.1.5. Additional resources 50
3.1.9.2. Configuring resource requests and limits 50
3.1.10. Logging 51
3.1.10.1. Using inline logging setting 51
3.1.10.2. Using external ConfigMap for logging setting 52
3.1.10.3. Loggers 52
3.1.11. Kafka rack awareness 53
3.1.11.1. Configuring rack awareness in Kafka brokers 53
3.1.12. Healthchecks 54
3.1.12.1. Healthcheck configurations 55
3.1.12.2. Configuring healthchecks 55
3.1.13. Prometheus metrics 56
3.1.13.1. Metrics configuration 56
3.1.13.2. Configuring Prometheus metrics 57
3.1.14. JVM Options 57
3.1.14.1. JVM configuration 58
3.1.14.2. Configuring JVM options 60
3.1.15. Container images 61
3.1.15.1. Container image configurations 61
3.1.15.2. Configuring container images 63
3.1.16. TLS sidecar 64
3.1.16.1. TLS sidecar configuration 64
3.1.16.2. Configuring TLS sidecar 65
3.1.17. Configuring pod scheduling 66
3.1.17.1. Scheduling pods based on other applications 66
3.1.17.1.1. Avoid critical applications to share the node 66
3.1.17.1.2. Affinity 66
3.1.17.1.3. Configuring pod anti-affinity in Kafka components 66
3.1.17.2. Scheduling pods to specific nodes 67
3.1.17.2.1. Node scheduling 67

3.1.17.2.2. Affinity 67
3.1.17.2.3. Configuring node affinity in Kafka components 68
3.1.17.3. Using dedicated nodes 69
3.1.17.3.1. Dedicated nodes 69
3.1.17.3.2. Affinity 69
3.1.17.3.3. Tolerations 69
3.1.17.3.4. Setting up dedicated nodes and scheduling pods on them 70
3.1.18. Performing a rolling update of a Kafka cluster 71
3.1.19. Performing a rolling update of a Zookeeper cluster 71
3.1.20. Scaling clusters 72
3.1.20.1. Scaling Kafka clusters 72
3.1.20.1.1. Adding brokers to a cluster 72
3.1.20.1.2. Removing brokers from a cluster 72
3.1.20.2. Partition reassignment 73
3.1.20.2.1. Reassignment JSON file 73
3.1.20.3. Generating reassignment JSON files 74
3.1.20.4. Creating reassignment JSON files manually 75
3.1.20.5. Reassignment throttles 75
3.1.20.6. Scaling up a Kafka cluster 76
3.1.20.7. Scaling down a Kafka cluster 77
3.1.21. Deleting Kafka nodes manually 79
3.1.22. Deleting Zookeeper nodes manually 80
3.1.23. List of resources created as part of Kafka cluster 81
3.2. KAFKA CONNECT CLUSTER CONFIGURATION 83
3.2.1. Replicas 83
3.2.1.1. Configuring the number of nodes 83
3.2.2. Bootstrap servers 84
3.2.2.1. Configuring bootstrap servers 84
3.2.3. Connecting to Kafka brokers using TLS 85
3.2.3.1. TLS support in Kafka Connect 85
3.2.3.2. Configuring TLS in Kafka Connect 86
3.2.4. Connecting to Kafka brokers with Authentication 86
3.2.4.1. Authentication support in Kafka Connect 86
3.2.4.1.1. TLS Client Authentication 87
3.2.4.1.2. SCRAM-SHA-512 authentication 87
3.2.4.2. Configuring TLS client authentication in Kafka Connect 88
3.2.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect 88
3.2.5. Kafka Connect configuration 89
3.2.5.1. Kafka Connect configuration 89
3.2.5.2. Configuring Kafka Connect 91
3.2.6. CPU and memory resources 92
3.2.6.1. Resource limits and requests 92
3.2.6.1.1. Resource requests 93
3.2.6.1.2. Resource limits 93
3.2.6.1.3. Supported CPU formats 94
3.2.6.1.4. Supported memory formats 95
3.2.6.1.5. Additional resources 95
3.2.6.2. Configuring resource requests and limits 95
3.2.7. Logging 96
3.2.7.1. Using inline logging setting 96
3.2.7.2. Using external ConfigMap for logging setting 97
3.2.7.3. Loggers 97
3.2.8. Healthchecks 98

3.2.8.1. Healthcheck configurations 98
3.2.8.2. Configuring healthchecks 99
3.2.9. Prometheus metrics 100
3.2.9.1. Metrics configuration 100
3.2.9.2. Configuring Prometheus metrics 101
3.2.10. JVM Options 101
3.2.10.1. JVM configuration 102
3.2.10.2. Configuring JVM options 104
3.2.11. Container images 104
3.2.11.1. Container image configurations 105
3.2.11.2. Configuring container images 107
3.2.12. Configuring pod scheduling 107
3.2.12.1. Scheduling pods based on other applications 107
3.2.12.1.1. Avoid critical applications to share the node 108
3.2.12.1.2. Affinity 108
3.2.12.1.3. Configuring pod anti-affinity in Kafka components 108
3.2.12.2. Scheduling pods to specific nodes 109
3.2.12.2.1. Node scheduling 109
3.2.12.2.2. Affinity 109
3.2.12.2.3. Configuring node affinity in Kafka components 109
3.2.12.3. Using dedicated nodes 110
3.2.12.3.1. Dedicated nodes 110
3.2.12.3.2. Affinity 111
3.2.12.3.3. Tolerations 111
3.2.12.3.4. Setting up dedicated nodes and scheduling pods on them 111
3.2.13. List of resources created as part of Kafka Connect cluster 112
3.3. KAFKA CONNECT CLUSTER WITH SOURCE2IMAGE SUPPORT 113
3.3.1. Replicas 113
3.3.1.1. Configuring the number of nodes 113
3.3.2. Bootstrap servers 113
3.3.2.1. Configuring bootstrap servers 114
3.3.3. Connecting to Kafka brokers using TLS 114
3.3.3.1. TLS support in Kafka Connect 114
3.3.3.2. Configuring TLS in Kafka Connect 115
3.3.4. Connecting to Kafka brokers with Authentication 116
3.3.4.1. Authentication support in Kafka Connect 116
3.3.4.1.1. TLS Client Authentication 116
3.3.4.1.2. SCRAM-SHA-512 authentication 117
3.3.4.2. Configuring TLS client authentication in Kafka Connect 117
3.3.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect 118
3.3.5. Kafka Connect configuration 119
3.3.5.1. Kafka Connect configuration 119
3.3.5.2. Configuring Kafka Connect 121
3.3.6. CPU and memory resources 122
3.3.6.1. Resource limits and requests 122
3.3.6.1.1. Resource requests 122
3.3.6.1.2. Resource limits 123
3.3.6.1.3. Supported CPU formats 124
3.3.6.1.4. Supported memory formats 124
3.3.6.1.5. Additional resources 125
3.3.6.2. Configuring resource requests and limits 125
3.3.7. Logging 125
3.3.7.1. Using inline logging setting 126

3.3.7.2. Using external ConfigMap for logging setting 126
3.3.7.3. Loggers 127
3.3.8. Healthchecks 128
3.3.8.1. Healthcheck configurations 128
3.3.8.2. Configuring healthchecks 129
3.3.9. Prometheus metrics 129
3.3.9.1. Metrics configuration 129
3.3.9.2. Configuring Prometheus metrics 130
3.3.10. JVM Options 131
3.3.10.1. JVM configuration 131
3.3.10.2. Configuring JVM options 134
3.3.11. Container images 134
3.3.11.1. Container image configurations 134
3.3.11.2. Configuring container images 136
3.3.12. Configuring pod scheduling 137
3.3.12.1. Scheduling pods based on other applications 137
3.3.12.1.1. Avoid critical applications to share the node 137
3.3.12.1.2. Affinity 137
3.3.12.1.3. Configuring pod anti-affinity in Kafka components 138
3.3.12.2. Scheduling pods to specific nodes 138
3.3.12.2.1. Node scheduling 138
3.3.12.2.2. Affinity 139
3.3.12.2.3. Configuring node affinity in Kafka components 139
3.3.12.3. Using dedicated nodes 140
3.3.12.3.1. Dedicated nodes 140
3.3.12.3.2. Affinity 140
3.3.12.3.3. Tolerations 141
3.3.12.3.4. Setting up dedicated nodes and scheduling pods on them 141
3.3.13. List of resources created as part of Kafka Connect cluster with Source2Image support 142
3.3.14. Using OpenShift builds and S2I to create new images 142
3.4. KAFKA MIRROR MAKER CONFIGURATION 144
3.4.1. Replicas 144
3.4.1.1. Configuring the number of replicas 144
3.4.2. Bootstrap servers 145
3.4.2.1. Configuring bootstrap servers 145
3.4.3. Whitelist 146
3.4.3.1. Configuring the topics whitelist 146
3.4.4. Consumer group identifier 146
3.4.4.1. Configuring the consumer group identifier 146
3.4.5. Number of consumer streams 147
3.4.5.1. Configuring the number of consumer streams 147
3.4.6. Connecting to Kafka brokers using TLS 148
3.4.6.1. TLS support in Kafka Mirror Maker 148
3.4.6.2. Configuring TLS encryption in Kafka Mirror Maker 149
3.4.7. Connecting to Kafka brokers with Authentication 150
3.4.7.1. Authentication support in Kafka Mirror Maker 150
3.4.7.1.1. TLS Client Authentication 150
3.4.7.1.2. SCRAM-SHA-512 authentication 151
3.4.7.2. Configuring TLS client authentication in Kafka Mirror Maker 151
3.4.7.3. Configuring SCRAM-SHA-512 authentication in Kafka Mirror Maker 152
3.4.8. Kafka Mirror Maker configuration 153
3.4.8.1. Kafka Mirror Maker configuration 153
3.4.8.2. Configuring Kafka Mirror Maker 154

3.4.9. CPU and memory resources 155
3.4.9.1. Resource limits and requests 155
3.4.9.1.1. Resource requests 156
3.4.9.1.2. Resource limits 157
3.4.9.1.3. Supported CPU formats 157
3.4.9.1.4. Supported memory formats 158
3.4.9.1.5. Additional resources 158
3.4.9.2. Configuring resource requests and limits 158
3.4.10. Logging 159
3.4.10.1. Using inline logging setting 159
3.4.10.2. Using external ConfigMap for logging setting 160
3.4.10.3. Loggers 160
3.4.11. Prometheus metrics 161
3.4.11.1. Metrics configuration 161
3.4.11.2. Configuring Prometheus metrics 162
3.4.12. JVM Options 163
3.4.12.1. JVM configuration 163
3.4.12.2. Configuring JVM options 165
3.4.13. Container images 166
3.4.13.1. Container image configurations 166
3.4.13.2. Configuring container images 168
3.4.14. Configuring pod scheduling 169
3.4.14.1. Scheduling pods based on other applications 169
3.4.14.1.1. Avoid critical applications to share the node 169
3.4.14.1.2. Affinity 169
3.4.14.1.3. Configuring pod anti-affinity in Kafka components 169
3.4.14.2. Scheduling pods to specific nodes 170
3.4.14.2.1. Node scheduling 170
3.4.14.2.2. Affinity 170
3.4.14.2.3. Configuring node affinity in Kafka components 171
3.4.14.3. Using dedicated nodes 172
3.4.14.3.1. Dedicated nodes 172
3.4.14.3.2. Affinity 172
3.4.14.3.3. Tolerations 172
3.4.14.3.4. Setting up dedicated nodes and scheduling pods on them 173
3.4.15. List of resources created as part of Kafka Mirror Maker 174

CHAPTER 4. OPERATORS 175
4.1. CLUSTER OPERATOR 175
4.1.1. Overview of the Cluster Operator component 175
4.1.2. Deploying the Cluster Operator to OpenShift 176
4.1.3. Deploying the Cluster Operator to watch multiple namespaces 176
4.1.4. Reconciliation 177
4.1.5. Cluster Operator Configuration 177
4.1.6. Role-Based Access Control (RBAC) 179
4.1.6.1. Provisioning Role-Based Access Control (RBAC) for the Cluster Operator 179
4.1.6.2. Delegated privileges 179
4.1.6.3. ServiceAccount 180
4.1.6.4. ClusterRoles 180
4.1.6.5. ClusterRoleBindings 187
4.2. TOPIC OPERATOR 188
4.2.1. Overview of the Topic Operator component 188
4.2.2. Understanding the Topic Operator 189

4.2.3. Deploying the Topic Operator using the Cluster Operator 190
4.2.4. Configuring the Topic Operator with resource requests and limits 191
4.2.5. Deploying the standalone Topic Operator 191
4.2.6. Topic Operator environment 192
4.3. USER OPERATOR 193
4.3.1. Overview of the User Operator component 193
4.3.2. Deploying the User Operator using the Cluster Operator 194
4.3.3. Deploying the standalone User Operator 194

CHAPTER 5. USING THE TOPIC OPERATOR 196
5.1. TOPIC OPERATOR USAGE RECOMMENDATIONS 196
5.2. CREATING A TOPIC 196
5.3. CHANGING A TOPIC 197
5.4. DELETING A TOPIC 198

CHAPTER 6. USING THE USER OPERATOR 200
6.1. OVERVIEW OF THE USER OPERATOR COMPONENT 200
6.2. MUTUAL TLS AUTHENTICATION FOR CLIENTS 200
6.2.1. Mutual TLS authentication 200
6.2.2. When to use mutual TLS authentication for clients 200
6.3. CREATING A KAFKA USER WITH MUTUAL TLS AUTHENTICATION 201
6.4. SCRAM-SHA AUTHENTICATION 202
6.4.1. Supported SCRAM credentials 202
6.4.2. When to use SCRAM-SHA authentication for clients 202
6.5. CREATING A KAFKA USER WITH SCRAM SHA AUTHENTICATION 202
6.6. EDITING A KAFKA USER 203
6.7. DELETING A KAFKA USER 205
6.8. KAFKA USER RESOURCE 205
6.8.1. Authentication 205
6.8.1.1. TLS Client Authentication 205
6.8.1.2. SCRAM-SHA-512 Authentication 206
6.8.2. Authorization 207
6.8.2.1. Simple Authorization 207
6.8.3. Additional resources 209

CHAPTER 7. SECURITY 210
7.1. CERTIFICATE AUTHORITIES 210
7.1.1. CA certificates 210
7.2. CERTIFICATES AND SECRETS 210
7.2.1. Cluster CA Secrets 210
7.2.2. Client CA Secrets 211
7.2.3. User Secrets 212
7.3. INSTALLING YOUR OWN CA CERTIFICATES 212
7.4. CERTIFICATE RENEWAL 213
7.4.1. Renewal process with generated CAs 214
7.4.2. Client applications 214
7.5. TLS CONNECTIONS 215
7.5.1. Zookeeper communication 215
7.5.2. Kafka interbroker communication 215
7.5.3. Topic and User Operators 215
7.5.4. Kafka Client connections 215
7.6. CONFIGURING INTERNAL CLIENTS TO TRUST THE CLUSTER CA 215
7.7. CONFIGURING EXTERNAL CLIENTS TO TRUST THE CLUSTER CA 216

APPENDIX A. FREQUENTLY ASKED QUESTIONS **218**
A.1. CLUSTER OPERATOR 218
A.1.1. Log contains warnings about failing to acquire lock 218

APPENDIX B. CUSTOM RESOURCE API REFERENCE **219**
B.1. KAFKA SCHEMA REFERENCE 219
B.2. KAFKASPEC SCHEMA REFERENCE 219
B.3. KAFKACLUSTERSPEC SCHEMA REFERENCE 219
B.4. EPHEMERALSTORAGE SCHEMA REFERENCE 221
B.5. PERSISTENTCLAIMSTORAGE SCHEMA REFERENCE 221
B.6. KAFKALISTENERS SCHEMA REFERENCE 222
B.7. KAFKALISTENERPLAIN SCHEMA REFERENCE 223
B.8. KAFKALISTENERAUTHENTICATIONTLS SCHEMA REFERENCE 223
B.9. KAFKALISTENERAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE 223
B.10. KAFKALISTENERTLS SCHEMA REFERENCE 224
B.11. KAFKALISTENEREXTERNALROUTE SCHEMA REFERENCE 224
B.12. KAFKALISTENEREXTERNALLOADBALANCER SCHEMA REFERENCE 224
B.13. KAFKALISTENEREXTERNALNODEPORT SCHEMA REFERENCE 225
B.14. KAFKAAUTHORIZATIONSIMPLE SCHEMA REFERENCE 225
B.15. RACK SCHEMA REFERENCE 226
B.16. PROBE SCHEMA REFERENCE 226
B.17. JVMOPTIONS SCHEMA REFERENCE 226
B.18. RESOURCES SCHEMA REFERENCE 227
B.19. CPUMEMORY SCHEMA REFERENCE 227
B.20. INLINELOGGING SCHEMA REFERENCE 228
B.21. EXTERNALLOGGING SCHEMA REFERENCE 228
B.22. TLSSIDECAR SCHEMA REFERENCE 228
B.23. ZOOKEEPERCLUSTERSPEC SCHEMA REFERENCE 229
B.24. TOPICOPERATORSPEC SCHEMA REFERENCE 230
B.25. ENTITYOPERATORSPEC SCHEMA REFERENCE 231
B.26. ENTITYTOPICOPERATORSPEC SCHEMA REFERENCE 232
B.27. ENTITYUSEROPERATORSPEC SCHEMA REFERENCE 232
B.28. CERTIFICATEAUTHORITY SCHEMA REFERENCE 233
B.29. KAFKACONNECT SCHEMA REFERENCE 234
B.30. KAFKACONNECTSPEC SCHEMA REFERENCE 234
B.31. KAFKACONNECTAUTHENTICATIONTLS SCHEMA REFERENCE 235
B.32. CERTANDKEYSECRETSOURCE SCHEMA REFERENCE 236
B.33. KAFKACONNECTAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE 236
B.34. PASSWORDSECRETSOURCE SCHEMA REFERENCE 237
B.35. KAFKACONNECTTLS SCHEMA REFERENCE 237
B.36. CERTSECRETSOURCE SCHEMA REFERENCE 237
B.37. KAFKACONNECTS2I SCHEMA REFERENCE 238
B.38. KAFKACONNECTS2ISPEC SCHEMA REFERENCE 238
B.39. KAFKATOPIC SCHEMA REFERENCE 239
B.40. KAFKATOPICSPEC SCHEMA REFERENCE 240
B.41. KAFKAUSER SCHEMA REFERENCE 240
B.42. KAFKAUSERSPEC SCHEMA REFERENCE 240
B.43. KAFKAUSERTLSCLIENTAUTHENTICATION SCHEMA REFERENCE 241
B.44. KAFKAUSERSCRAMSHA512CLIENTAUTHENTICATION SCHEMA REFERENCE 241
B.45. KAFKAUSERAUTHORIZATIONSIMPLE SCHEMA REFERENCE 241
B.46. ACLRULE SCHEMA REFERENCE 242
B.47. ACLRULETOPICRESOURCE SCHEMA REFERENCE 242
B.48. ACLRULEGROUPRESOURCE SCHEMA REFERENCE 243

B.49. ACLRULECLUSTERRESOURCE SCHEMA REFERENCE 243
B.50. KAFKAMIRRORMAKER SCHEMA REFERENCE 244
B.51. KAFKAMIRRORMAKERSPEC SCHEMA REFERENCE 244
B.52. KAFKAMIRRORMAKERCONSUMERSPEC SCHEMA REFERENCE 245
B.53. KAFKAMIRRORMAKERAUTHENTICATIONTLS SCHEMA REFERENCE 246
B.54. KAFKAMIRRORMAKERAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE 246
B.55. KAFKAMIRRORMAKERTLS SCHEMA REFERENCE 247
B.56. KAFKAMIRRORMAKERPRODUCERSPEC SCHEMA REFERENCE 247

APPENDIX C. USING YOUR SUBSCRIPTION 249
Accessing Your Account 249
Activating a Subscription 249
Downloading Zip and Tar Files 249
Registering Your System for Packages 249

CHAPTER 1. OVERVIEW OF AMQ STREAMS

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

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

Cluster Operator

Responsible for deploying and managing Apache Kafka clusters within OpenShift cluster.

Topic Operator

Responsible for managing Kafka topics within a Kafka cluster running within OpenShift cluster.

User Operator

Responsible for managing Kafka users within a Kafka cluster running within OpenShift cluster.

This guide describes how to install and use Red Hat AMQ Streams.

1.1. KAFKA KEY FEATURES

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

1.2. DOCUMENT CONVENTIONS

Replaceables

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

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

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

CHAPTER 2. GETTING STARTED WITH AMQ STREAMS

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

AMQ Streams is based on Strimzi 0.8.0. This chapter describes the procedures to deploy AMQ Streams on OpenShift 3.9 and later.

NOTE

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

For more information about OpenShift and setting up OpenShift cluster, see OpenShift documentation.

2.1. INSTALLING AMQ STREAMS AND DEPLOYING COMPONENTS

To install AMQ Streams, download and extract the **install_and_examples.zip** file from the AMQ Streams download site.

The folder contains several YAML files to help you deploy the components of AMQ Streams to OpenShift, perform common operations, and configure your Kafka cluster. The YAML files are referenced throughout this documentation.

The remainder of this chapter provides an overview of each component and instructions for deploying the components to OpenShift using the YAML files provided.

NOTE

Although container images for AMQ Streams are available in the Red Hat Container Catalog, we recommend that you use the YAML files provided instead.

2.2. CLUSTER OPERATOR

AMQ Streams uses the Cluster Operator to deploy and manage Kafka (including Zookeeper) and Kafka Connect clusters. The Cluster Operator is deployed inside of the OpenShift cluster. To deploy a Kafka cluster, a **Kafka** resource with the cluster configuration has to be created within the OpenShift cluster. Based on what is declared inside of the **Kafka** resource, the Cluster Operator deploys a corresponding Kafka cluster. For more information about the different configuration options supported by the **Kafka** resource, see Section 3.1, “Kafka cluster configuration”

NOTE

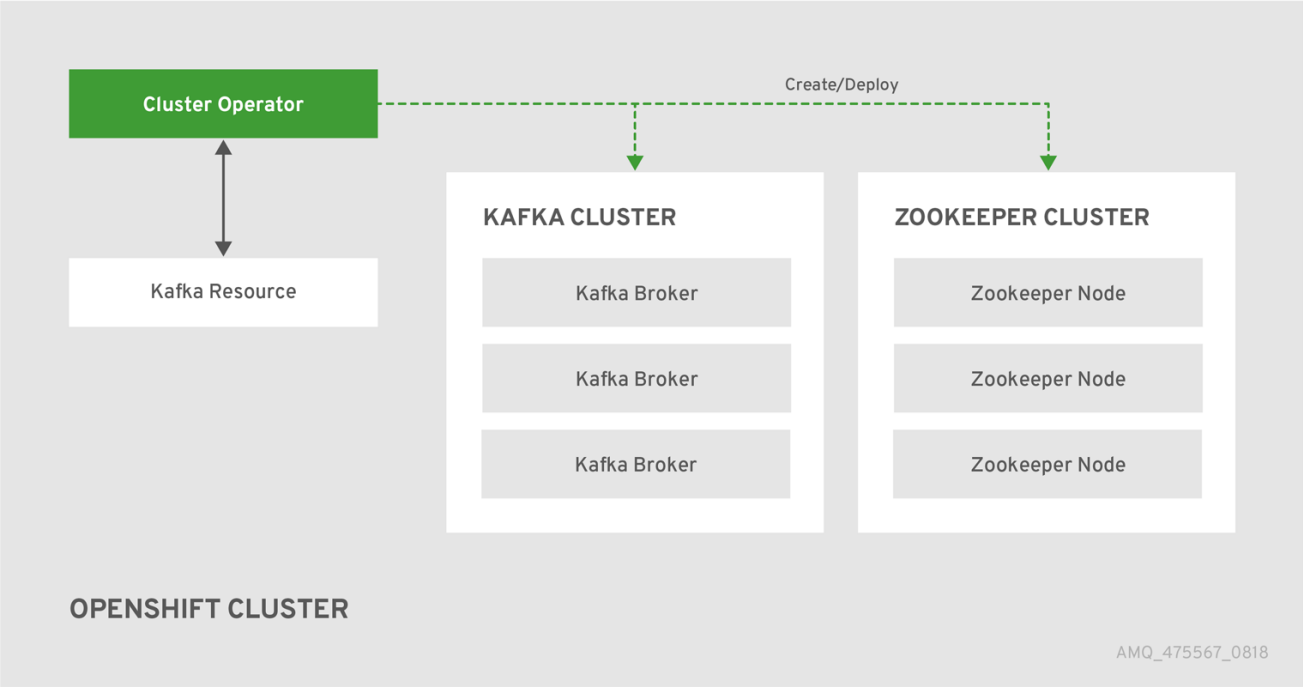
AMQ Streams contains example YAML files, which make deploying a Cluster Operator easier.

2.2.1. Overview of the Cluster Operator component

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

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

Figure 2.1. Example Architecture diagram of the Cluster Operator.



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

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

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

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

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

2.2.2. Deploying the Cluster Operator to OpenShift

Prerequisites

- A user with **`cluster-admin`** role needs to be used, for example, **`system:admin`**.

- Modify the installation files according to the namespace the Cluster Operator is going to be installed in.
  On Linux, use:

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

  On MacOS, use:

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

Procedure

1. Deploy the Cluster Operator

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

2.2.3. Deploying the Cluster Operator to watch multiple namespaces

Prerequisites

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

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

  On MacOS, use:

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

Procedure

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

   ```
   apiVersion: extensions/v1beta1
   kind: Deployment
   spec:
     template:
       spec:
         serviceAccountName: strimzi-cluster-operator
         containers:
         - name: strimzi-cluster-operator
           image: strimzi/cluster-operator:latest
           imagePullPolicy: IfNotPresent
   ```

```
        env:
        - name: STRIMZI_NAMESPACE
          value: myproject,myproject2,myproject3
```

2. For all namespaces or projects which should be watched by the Cluster Operator, install the **RoleBindings**. Replace the ***my-namespace*** or ***my-project*** with the OpenShift project or Kubernetes namespace used in the previous step.
   On OpenShift this can be done using **oc apply**:

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

3. Deploy the Cluster Operator
   On OpenShift this can be done using **oc apply**:

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

2.3. KAFKA CLUSTER

You can use AMQ Streams to deploy an ephemeral or persistent Kafka cluster to OpenShift. When installing Kafka, AMQ Streams also installs a Zookeeper cluster and adds the necessary configuration to connect Kafka with Zookeeper.

Ephemeral cluster

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

Persistent cluster

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

AMQ Streams includes two templates for deploying a Kafka cluster:

- **kafka-ephemeral.yaml** deploys an ephemeral cluster, named **my-cluster** by default.
- **kafka-persistent.yaml** deploys a persistent cluster, named **my-cluster** by default.

The cluster name is defined by the name of the resource and cannot be changed after the cluster has been deployed. To change the cluster name before you deploy the cluster, edit the **Kafka.metadata.name** property of the resource in the relevant YAML file.

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
```

```
  name: my-cluster
# ...
```

2.3.1. Deploying the Kafka cluster to OpenShift

The following procedure describes how to deploy an ephemeral or persistent Kafka cluster to OpenShift on the command line. You can also deploy clusters in the OpenShift console.

Prerequisites

- The Cluster Operator is deployed.

Procedure

1. If you plan to use the cluster for development or testing purposes, create and deploy an ephemeral cluster using **`oc apply`**.

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

2. If you plan to use the cluster in production, create and deploy a persistent cluster using **`oc apply`**.

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

Additional resources

- For more information on deploying the Cluster Operator, see Section 2.2, “Cluster Operator”. For more information on the different configuration options supported by the **`Kafka`** resource, see Section 3.1, “Kafka cluster configuration”.

2.4. KAFKA CONNECT

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

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

2.4.1. Deploying Kafka Connect to OpenShift

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

Prerequisites

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

Procedure

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”

2.4.2. Using Kafka Connect with plugins

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

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

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

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

2.4.2.1. Create a new image based on our base image

AMQ Streams provides its own Docker image for running Kafka Connect, which can be found on Red Hat Container Catalog as **`registry.access.redhat.com/amqstreams-1/amqstreams10-kafkaconnect-openshift:1.0.0`**. This image can be used as a base image for building a new custom image with additional plugins.

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

Procedure

1. Create a new **`Dockerfile`** using **`registry.access.redhat.com/amqstreams-1/amqstreams10-kafkaconnect-openshift:1.0.0`** as the base image:

```
FROM registry.access.redhat.com/amqstreams-1/amqstreams10-
kafkaconnect-openshift:1.0.0
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER kafka:kafka
```

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

Additional resources

- For more information about the **`STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE`** variable, see Section 4.1.5, “Cluster Operator Configuration”.

- For more information about the **`KafkaConnect.spec.image property`**, see Section 3.2.11, “Container images”.

2.4.2.2. Using OpenShift builds and S2I to create new images

OpenShift supports builds, which can be used together with the Source-to-Image (S2I) framework to create new container images. An OpenShift build takes a builder image with S2I support together with source code and binaries provided by the user and uses them to build a new container image. The newly created container image is stored in OpenShift’s local container image repository and can be used in deployments. AMQ Streams provides a Kafka Connect builder image, which can be found on Red Hat Container Catalog as **`registry.access.redhat.com/amqstreams-1/amqstreams10-kafkaconnects2i-openshift:1.0.0`** with this S2I support. It takes user-provided binaries (with plugins and connectors) and creates a new Kafka Connect image. This enhanced Kafka Connect image can be used with the Kafka Connect deployment.

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

Procedure

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

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

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

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

```
        ├── debezium-connector-postgres-0.7.1.jar
        ├── debezium-core-0.7.1.jar
        ├── LICENSE.txt
        ├── postgresql-42.0.0.jar
        ├── protobuf-java-2.6.1.jar
        └── README.md
```

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

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

NOTE

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

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

2.5. KAFKA MIRROR MAKER

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

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

2.5.1. Deploying Kafka Mirror Maker to OpenShift

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

Prerequisites

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

Procedure

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”

2.6. DEPLOYING EXAMPLE CLIENTS

Prerequisites

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

Procedure

1. Deploy the producer.
   On OpenShift, use **`oc run`**:

   ```
   oc run kafka-producer -ti --
   image=registry.access.redhat.com/amqstreams-1/amqstreams10-kafka-
   openshift:1.0.0 --rm=true --restart=Never -- bin/kafka-console-
   producer.sh --broker-list cluster-name-kafka-bootstrap:9092 --topic
   my-topic
   ```

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

3. Press Enter to send the message.

4. Deploy the consumer.
   On OpenShift, use **`oc run`**:

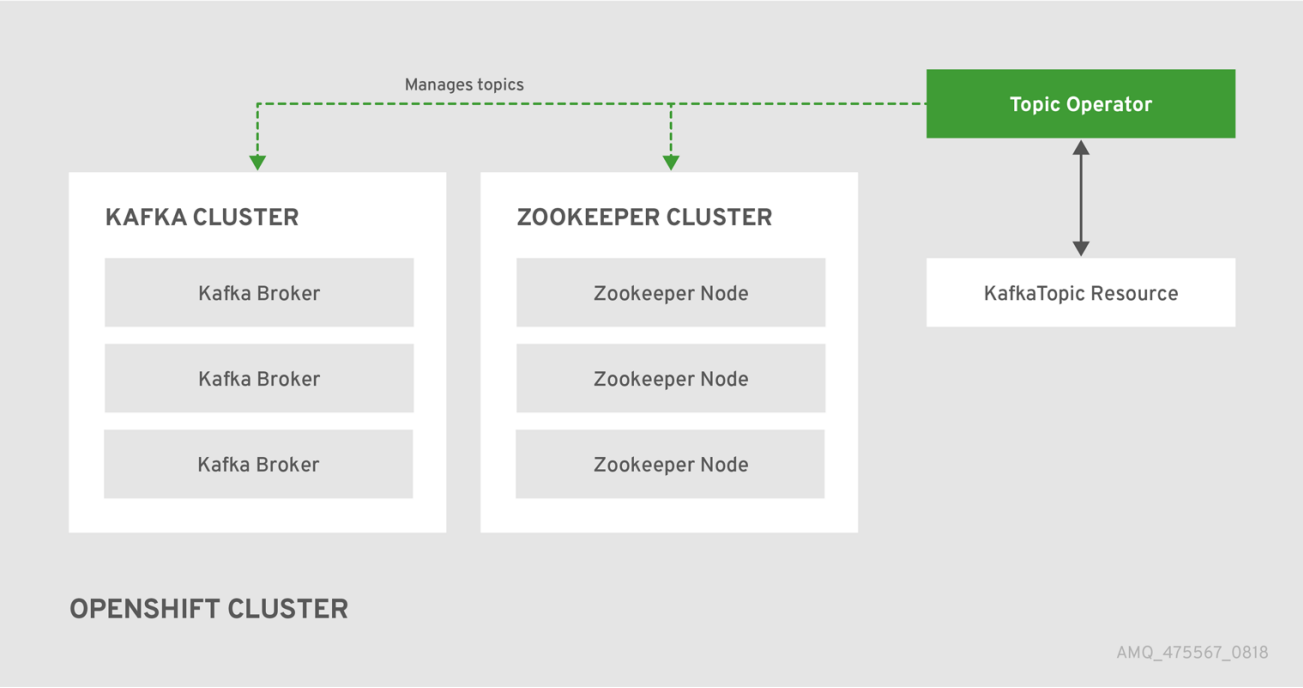
   ```
   oc run kafka-consumer -ti --
   image=registry.access.redhat.com/amqstreams-1/amqstreams10-kafka-
   openshift:1.0.0 --rm=true --restart=Never -- bin/kafka-console-
   consumer.sh --bootstrap-server cluster-name-kafka-bootstrap:9092 --
   topic my-topic --from-beginning
   ```

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

2.7. TOPIC OPERATOR

2.7.1. Overview of the Topic Operator component

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



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

Specifically:

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

And also, in the other direction:

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

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

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

For more details about creating, modifying and deleting topics, see Chapter 5, *Using the Topic Operator*.

2.7.2. Deploying the Topic Operator using the Cluster Operator

This procedure describes how to deploy the Topic Operator using the Cluster Operator. If you want to use the Topic Operator with a Kafka cluster that is not managed by AMQ Streams, you must deploy the Topic Operator as a standalone component. For more information, see Section 4.2.5, "Deploying the standalone Topic Operator".

Prerequisites

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

Procedure

1. Ensure that the **`Kafka.spec.entityOperator`** object exists in the **`Kafka`** resource. This configures the Entity Operator.

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

2. Configure the Topic Operator using the fields described in Section B.26, "**`EntityTopicOperatorSpec`** schema reference".

3. Create or update the Kafka resource in OpenShift.
   On OpenShift, use **`oc apply`**:

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, "Cluster Operator".
- For more information about deploying the Entity Operator, see Section 3.1.8, "Entity Operator".
- For more information about the **`Kafka.spec.entityOperator`** object used to configure the Topic Operator when deployed by the Cluster Operator, see Section B.25, "**`EntityOperatorSpec`** schema reference".

2.8. USER OPERATOR

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

2.8.1. Overview of the User Operator component

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

- if a **`KafkaUser`** is created, the User Operator will create the user it describes
- if a **`KafkaUser`** is deleted, the User Operator will delete the user it describes
- if a **`KafkaUser`** is changed, the User Operator will update the user it describes

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

The User Operator allows you to declare a **`KafkaUser`** as part of your application's deployment. When the user is created, the credentials will be created in a **`Secret`**. Your application needs to use the user and its credentials for authentication and to produce or consume messages.

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

2.8.2. Deploying the User Operator using the Cluster Operator

Prerequisites

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

Procedure

1. Edit the **`Kafka`** resource ensuring it has a **`Kafka.spec.entityOperator.userOperator`** object that configures the User Operator how you want.
2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, "Cluster Operator".
- For more information about the **`Kafka.spec.entityOperator`** object used to configure the User Operator when deployed by the Cluster Operator, see **`EntityOperatorSpec`** schema reference.

CHAPTER 3. DEPLOYMENT CONFIGURATION

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

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

3.1. KAFKA CLUSTER CONFIGURATION

The full schema of the **`Kafka`** resource is described in the Section B.1, "**`Kafka`** schema reference". All labels that are applied to the desired **`Kafka`** resource will also be applied to the OpenShift resources making up the Kafka cluster. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.1.1. Kafka and Zookeeper storage

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

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

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

AMQ Streams supports two types of storage:

- Ephemeral
- Persistent

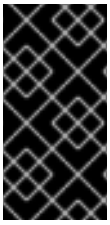
The type of storage is specified in the **`type`** field.

IMPORTANT

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

3.1.1.1. Ephemeral storage

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



IMPORTANT

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

An example of Ephemeral storage

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

3.1.1.2. Persistent storage

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

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

`size` (required)

Defines the size of the persistent volume claim, for example, "1000Gi".

`class` (optional)

The OpenShift Storage Class to use for dynamic volume provisioning.

`selector` (optional)

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

`delete-claim` (optional)

Boolean value which specifies if the Persistent Volume Claim has to be deleted when the cluster is undeployed. Default is **`false`**.

> **WARNING**
>
> Resizing persistent storage for existing AMQ Streams clusters is not currently supported. You must decide the necessary storage size before deploying the cluster.

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

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

The following example demonstrates the use of a storage class.

Example fragment of persistent storage configuration with specific Storage Class

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

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

Example fragment of persistent storage configuration with selector

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

Persistent Volume Claim naming

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

`data-`*`cluster-name`*`-kafka-`*`idx`*

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

`data-`*`cluster-name`*`-zookeeper-`*`idx`*

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

Additional resources

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

3.1.2. Replicas

A Kafka cluster can run with many brokers. You can configure the number of brokers used for the Kafka cluster in **`Kafka.spec.kafka.replicas`**. The best number of brokers for your cluster has to be determined based on your specific use case.

3.1.2.1. Configuring the number of broker nodes

This procedure describes how to configure the number of Kafka broker nodes in a new cluster. It only applies to new clusters, with no partitions. If your cluster already has topics defined you should see Section 3.1.20, “Scaling clusters”.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- A Kafka cluster with no topics defined yet

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

Additional resources

If your cluster already has topics defined see Section 3.1.20, “Scaling clusters”.

3.1.3. Kafka broker configuration

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

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

- Security (Encryption, Authentication, and Authorization)
- Listener configuration
- Broker ID configuration

- Configuration of log data directories
- Inter-broker communication
- Zookeeper connectivity

These options are automatically configured by AMQ Streams.

3.1.3.1. Kafka broker configuration

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

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

- String
- Number
- Boolean

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

- **`listeners`**
- **`advertised.`**
- **`broker.`**
- **`listener.`**
- **`host.name`**
- **`port`**
- **`inter.broker.listener.name`**
- **`sasl.`**
- **`ssl.`**
- **`security.`**
- **`password.`**
- **`principal.builder.class`**
- **`log.dir`**
- **`zookeeper.connect`**
- **`zookeeper.set.acl`**
- **`authorizer.`**

- **super.user**

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

IMPORTANT

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

An example showing Kafka broker configuration

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

3.1.3.2. Configuring Kafka brokers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
```

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.4. Kafka broker listeners

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

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

3.1.4.1. Mutual TLS authentication for clients

3.1.4.1.1. Mutual TLS authentication

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

NOTE

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

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

3.1.4.1.2. When to use mutual TLS authentication for clients

Mutual TLS authentication is recommended for authenticating Kafka clients when:

- The client supports authentication using mutual TLS authentication

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

3.1.4.2. SCRAM-SHA authentication

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

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

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

3.1.4.2.1. Supported SCRAM credentials

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

3.1.4.2.2. When to use SCRAM-SHA authentication for clients

SCRAM-SHA is recommended for authenticating Kafka clients when:

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

3.1.4.3. Kafka listeners

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

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

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

An example of `listeners` property with all listeners enabled

```
# ...
listeners:
```

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

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

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

3.1.4.3.1. External listener

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

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

Exposing Kafka using OpenShift Routes

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

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

For more information on using **`Routes`** to access Kafka, see Section 3.1.4.5, “Accessing Kafka using OpenShift routes”.

Exposing Kafka using loadbalancers

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

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

For more information on using loadbalancers to access Kafka, see Section 3.1.4.6, “Accessing Kafka using loadbalancers routes”.

Exposing Kafka using node ports

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

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

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

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

NOTE

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

For more information on using node ports to access Kafka, see Section 3.1.4.7, “Accessing Kafka using node ports routes”.

3.1.4.3.2. Listener authentication

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

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

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

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

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

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

3.1.4.4. Configuring Kafka listeners

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

+

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

1. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

Additional resources

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

3.1.4.5. Accessing Kafka using OpenShift routes

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type **`route`**.
   An example configuration with an external listener configured to use **`Routes`**:

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

```
        type: route
        # ...
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.

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

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

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

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

4. Extract the public certificate of the broker certification authority

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

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

Additional resources

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

3.1.4.6. Accessing Kafka using loadbalancers routes

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type **loadbalancer**.
   An example configuration with an external listener configured to use loadbalancers:

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3. Find the hostname of the bootstrap loadbalancer.
   On OpenShift this can be done using **oc get**:

```
oc get service cluster-name-kafka-external-bootstrap -
o=jsonpath='{.status.loadBalancer.ingress[0].hostname}{"\n"}'
```

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

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

```
oc get service cluster-name-kafka-external-bootstrap -
o=jsonpath='{.status.loadBalancer.ingress[0].ip}{"\n"}'
```

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

4. Unless TLS encryption was disabled, extract the public certificate of the broker certification authority.
   On OpenShift this can be done using **oc extract**:

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

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

Additional resources

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

3.1.4.7. Accessing Kafka using node ports routes

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Deploy Kafka cluster with an external listener enabled and configured to the type **nodeport**.
   An example configuration with an external listener configured to use node ports:

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3. Find the port number of the bootstrap service.
   On OpenShift this can be done using **oc get**:

```
oc get service cluster-name-kafka-external-bootstrap -
o=jsonpath='{.spec.ports[0].nodePort}{"\n"}'
```

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

4. Find the address of the OpenShift node.
   On OpenShift this can be done using **oc get**:

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

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

   a. ExternalDNS

   b. ExternalIP

   c. Hostname

   d. InternalDNS

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

5. Unless TLS encryption was disabled, extract the public certificate of the broker certification authority.
   On OpenShift this can be done using **oc extract**:

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

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

Additional resources

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

3.1.5. Authentication and Authorization

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

3.1.5.1. Authentication

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

The supported authentication mechanisms are:

- TLS client authentication
- SASL SCRAM-SHA-512

3.1.5.1.1. TLS client authentication

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

An example of `authentication` with type `tls`

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

3.1.5.2. Configuring authentication in Kafka brokers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

```
      # ...
      listeners:
        tls:
          authentication:
            type: tls
      # ...
    zookeeper:
      # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

Additional resources

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

3.1.5.3. Authorization

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

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

3.1.5.3.1. Simple authorization

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

An example of Simple authorization

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

3.1.5.4. Configuring authorization in Kafka brokers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

Additional resources

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

3.1.6. Replicas

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

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

Three nodes

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

Five nodes

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

Seven nodes

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

NOTE

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

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

3.1.6.1. Number of Zookeeper nodes

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

An example showing replicas configuration

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

3.1.6.2. Changing number of replicas

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.1.7. Zookeeper configuration

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

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

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

These options are automatically configured by AMQ Streams.

3.1.7.1. Zookeeper configuration

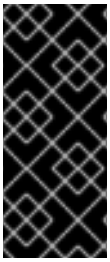
Zookeeper nodes can be configured using the **`config`** property in **`Kafka.spec.zookeeper`**. This property should contain the Zookeeper configuration options as keys. The values could be in one of the following JSON types:

- String
- Number
- Boolean

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

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

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



IMPORTANT

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

Selected options have default values:

- **`timeTick`** with default value **`2000`**

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

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

An example showing Zookeeper configuration

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

3.1.7.2. Configuring Zookeeper

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.8. Entity Operator

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

Kafka topics

managed by the Topic Operator.

Kafka users

managed by the User Operator

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

For more information about Topic Operator, see Section 4.2, “Topic Operator”. For more information about how to use Topic Operator to create or delete topics, see Chapter 5, *Using the Topic Operator*.

3.1.8.1. Configuration

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

The **`entityOperator`** property supports several sub-properties:

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

The **`tlsSidecar`** property can be used to configure the TLS sidecar container which is used to communicate with Zookeeper. For more details about configuring the TLS sidecar, see Section 3.1.16, “TLS sidecar”.

The **`affinity`** and **`tolerations`** properties can be used to configure how OpenShift schedules the Entity Operator pod. For more details about pod scheduling, see Section 3.1.17, “Configuring pod scheduling”.

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

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

Example of basic configuration enabling both operators

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

```
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

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

3.1.8.1.1. Topic Operator

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

watchedNamespace

The OpenShift namespace in which the topic operator watches for **KafkaTopics**. Default is the namespace where the Kafka cluster is deployed.

reconciliationIntervalSeconds

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

zookeeperSessionTimeoutSeconds

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

topicMetadataMaxAttempts

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

image

The **image** property can be used to configure the container image which will be used. For more details about configuring custom container images, see Section 3.1.15, “Container images”.

resources

The **resources** property configures the amount of resources allocated to the Topic Operator For more details about resource request and limit configuration, see Section 3.1.9, “CPU and memory resources”.

logging

The **logging** property configures the logging of the Topic Operator For more details about logging configuration, see Section 3.1.10, “Logging”.

Example of Topic Operator configuration

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

```
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

3.1.8.1.2. User Operator

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

`watchedNamespace`

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

`reconciliationIntervalSeconds`

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

`zookeeperSessionTimeoutSeconds`

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

`image`

The **`image`** property can be used to configure the container image which will be used. For more details about configuring custom container images, see Section 3.1.15, “Container images”.

`resources`

The **`resources`** property configures the amount of resources allocated to the User Operator. For more details about resource request and limit configuration, see Section 3.1.9, “CPU and memory resources”.

`logging`

The **`logging`** property configures the logging of the User Operator. For more details about logging configuration, see Section 3.1.10, “Logging”.

Example of Topic Operator configuration

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

3.1.8.2. Configuring Entity Operator

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.9. CPU and memory resources

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

- Memory
- CPU

AMQ Streams is using the OpenShift syntax for specifying CPU and memory resources.

3.1.9.1. Resource limits and requests

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

- **`Kafka.spec.kafka`**
- **`Kafka.spec.kafka.tlsSidecar`**
- **`Kafka.spec.zookeeper`**
- **`Kafka.spec.zookeeper.tlsSidecar`**

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

3.1.9.1.1. Resource requests

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

IMPORTANT

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

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

An example showing resource request configuration

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

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

An example showing resource request configuration with memory request only

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

Or:

An example showing resource request configuration with CPU request only

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

3.1.9.1.2. Resource limits

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

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

An example showing resource limits configuration

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

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

An example showing resource limit configuration with memory request only

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

Or:

An example showing resource limits configuration with CPU request only

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

3.1.9.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

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

An example of using different CPU units

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

```
  limits:
    cpu: 2.5
# ...
```

NOTE

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

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

3.1.9.1.4. Supported memory formats

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

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

An example of using different memory units

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

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

3.1.9.1.5. Additional resources

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

3.1.9.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

Additional resources

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

3.1.10. Logging

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

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

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

3.1.10.1. Using inline logging setting

Procedure

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

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

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

2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.10.2. Using external ConfigMap for logging setting

Procedure

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

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

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

2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.10.3. Loggers

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

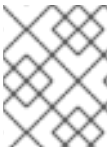
Components and their loggers are listed below.

- Kafka
  - **`kafka.root.logger.level`**
  - **`log4j.logger.org.I0Itec.zkclient.ZkClient`**
  - **`log4j.logger.org.apache.zookeeper`**
  - **`log4j.logger.kafka`**
  - **`log4j.logger.org.apache.kafka`**
  - **`log4j.logger.kafka.request.logger`**

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

3.1.11. Kafka rack awareness

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



NOTE

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

3.1.11.1. Configuring rack awareness in Kafka brokers

Kafka rack awareness can be configured in the `rack` property of `Kafka.spec.kafka`. The `rack` object has one mandatory field named `topologyKey`. This key needs to match one of the labels assigned to the OpenShift cluster nodes. The label is used by OpenShift when scheduling the Kafka broker pods to

nodes. If the OpenShift cluster is running on a cloud provider platform, that label should represent the availability zone where the node is running. Usually, the nodes are labeled with **`failure-domain.beta.kubernetes.io/zone`** that can be easily used as the **`topologyKey`** value. This has the effect of spreading the broker pods across zones, and also setting the brokers' **`broker.rack`** configuration parameter inside Kafka broker.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

Additional Resources

- For information about Configuring init container image for Kafka rack awareness, see Section 3.1.15, “Container images”.

3.1.12. Healthchecks

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

- Liveness probes
- Readiness probes

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

Users can configure selected options for liveness and readiness probes

3.1.12.1. Healthcheck configurations

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

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

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

- **initialDelaySeconds**
- **timeoutSeconds**

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

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

An example of liveness and readiness probe configuration

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

3.1.12.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.13. Prometheus metrics

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

3.1.13.1. Metrics configuration

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

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

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

Example of enabling metrics without any further configuration

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

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

Example of enabling metrics with additional Prometheus JMX Exporter configuration

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

3.1.13.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.14. JVM Options

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

3.1.14.1. JVM configuration

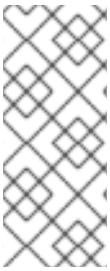
JVM options can be configured using the **`jvmOptions`** property in following resources:

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

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

-Xms and -Xmx

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



NOTE

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

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

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

IMPORTANT

Setting **-Xmx** explicitly requires some care:

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

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

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

IMPORTANT

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

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

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

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

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

-server

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

Example fragment configuring `-server`

```
# ...
```

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

NOTE

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

-XX

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

Example showing the use of the -XX object

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

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

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```

NOTE

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

3.1.14.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

```
kafka:
  # ...
  jvmOptions:
    "-Xmx": "8g"
    "-Xms": "8g"
  # ...
zookeeper:
  # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.1.15. Container images

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

3.1.15.1. Container image configurations

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

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

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

- For Kafka brokers:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.

2. **`strimzi/kafka:latest`** container image.

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

WARNING

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

Example of container image configuration

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

3.1.15.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.

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

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

3.1.16. TLS sidecar

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

The TLS sidecar is currrently being used in:

- Kafka brokers
- Zookeeper
- Entity Operator

3.1.16.1. TLS sidecar configuration

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

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

The TLS sidecar supports three additional options:

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

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

The **`image`** property can be used to configure the container image which will be used. For more details about configuring custom container images, see Section 3.1.15, “Container images”.

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

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

- info
- debug

The default value is *notice*.

Example of TLS sidecar configuration

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

3.1.16.2. Configuring TLS sidecar

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

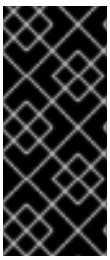
```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    tlsSidecar:
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
```

```
            memory: 128Mi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.17. Configuring pod scheduling



IMPORTANT

When two application are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.1.17.1. Scheduling pods based on other applications

3.1.17.1.1. Avoid critical applications to share the node

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

3.1.17.1.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.1.17.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.1.17.2. Scheduling pods to specific nodes

3.1.17.2.1. Node scheduling

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

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

3.1.17.2.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.1.17.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   On OpenShift this can be done using **oc label**:

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

   Alternatively, some of the existing labels might be reused.

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

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

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

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.1.17.3. Using dedicated nodes

3.1.17.3.1. Dedicated nodes

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

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

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

3.1.17.3.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.1.17.3.3. Tolerations

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

- **`Kafka.spec.kafka`**

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

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

3.1.17.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated
2. Make sure there are no workloads scheduled on these nodes
3. Set the taints on the selected nodes
   On OpenShift this can be done using **`oc adm taint`**:

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

4. Additionally, add a label to the selected nodes as well.
   On OpenShift this can be done using **`oc label`**:

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

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

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

```
                operator: In
                values:
                - Kafka
    # ...
  zookeeper:
    # ...
```

6. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.1.18. Performing a rolling update of a Kafka cluster

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

Prerequisites

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

Procedure

1. Find the name of the **StatefulSet** that controls the Kafka pods you want to manually update. For example, if your Kafka cluster is named *my-cluster*, the corresponding **StatefulSet** is named *my-cluster-kafka*.
2. Annotate a **StatefulSet** resource in OpenShift.
   + On OpenShift, use **oc annotate**:

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Kafka cluster on OpenShift, see Section 2.3.1, “Deploying the Kafka cluster to OpenShift”.

3.1.19. Performing a rolling update of a Zookeeper cluster

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

Prerequisites

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

Procedure

1. Find the name of the **`StatefulSet`** that controls the Zookeeper pods you want to manually update.
   For example, if your Kafka cluster is named *my-cluster*, the corresponding **`StatefulSet`** is named *my-cluster-zookeeper*.
2. Annotate a **`StatefulSet`** resource in OpenShift.
   + On OpenShift, use **`oc annotate`**:

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Zookeeper cluster, see Section 2.3.1, “Deploying the Kafka cluster to OpenShift”.

3.1.20. Scaling clusters

3.1.20.1. Scaling Kafka clusters

3.1.20.1.1. Adding brokers to a cluster

The primary way of increasing throughput for a topic is to increase the number of partitions for that topic. That works because the extra partitions allow the load of the topic to be shared between the different brokers in the cluster. However, in situations where every broker is constrained by a particular resource (typically I/O) using more partitions will not result in increased throughput. Instead, you need to add brokers to the cluster.

When you add an extra broker to the cluster, Kafka does not assign any partitions to it automatically. You must decide which partitions to move from the existing brokers to the new broker.

Once the partitions have been redistributed between all the brokers, the resource utilization of each broker should be reduced.

3.1.20.1.2. Removing brokers from a cluster

Because AMQ Streams uses **`StatefulSets`** to manage broker pods, you cannot remove *any* pod from the cluster. You can only remove one or more of the highest numbered pods from the cluster. For example, in a cluster of 12 brokers the pods are named ***`cluster-name`*`-kafka-0`** up to ***`cluster-`***

***name*-kafka-11**. If you decide to scale down by one broker, the ***cluster-name*-kafka-11** will be removed.

Before you remove a broker from a cluster, ensure that it is not assigned to any partitions. You should also decide which of the remaining brokers will be responsible for each of the partitions on the broker being decommissioned. Once the broker has no assigned partitions, you can scale the cluster down safely.

3.1.20.2. Partition reassignment

The Topic Operator does not currently support reassigning replicas to different brokers, so it is necessary to connect directly to broker pods to reassign replicas to brokers.

Within a broker pod, the **kafka-reassign-partitions.sh** utility allows you to reassign partitions to different brokers.

It 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 just need to reassign some of the 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 of the partitions in the file have been moved to their intended brokers. If the reassignment is complete, --verify also removes any throttles 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 need to 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.

3.1.20.2.1. Reassignment JSON file

The *reassignment JSON file* has a specific structure:

```
{
  "version": 1,
  "partitions": [
    <PartitionObjects>
  ]
}
```

Where *<PartitionObjects>* is a comma-separated list of objects like:

```
{
  "topic": <TopicName>,
  "partition": <Partition>,
  "replicas": [ <AssignedBrokerIds> ]
}
```

NOTE

Although Kafka also supports a **"log_dirs"** property this should not be used in Red Hat AMQ Streams.

The following is an example reassignment JSON file that assigns topic **topic-a**, partition **4** to brokers **2**, **4** and **7**, and topic **topic-b** partition **2** to brokers **1**, **5** and **7**:

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "topic-a",
      "partition": 4,
      "replicas": [2,4,7]
    },
    {
      "topic": "topic-b",
      "partition": 2,
      "replicas": [1,5,7]
    }
  ]
}
```

Partitions not included in the JSON are not changed.

3.1.20.3. Generating reassignment JSON files

This procedure describes how to generate a reassignment JSON file that reassigns all the partitions for a given set of topics using the **kafka-reassign-partitions.sh** tool.

Prerequisites

- A running Cluster Operator
- A **Kafka** resource
- A set of topics to reassign the partitions of

Procedure

1. Prepare a JSON file named ***topics.json*** that lists the topics to move. It must have the following structure:

```
{
  "version": 1,
  "topics": [
```

```
    <TopicObjects>
  ]
}
```

where *<TopicObjects>* is a comma-separated list of objects like:

```
{
  "topic": <TopicName>
}
```

For example if you want to reassign all the partitions of **`topic-a`** and **`topic-b`**, you would need to prepare a ***`topics.json`*** file like this:

```
{
  "version": 1,
  "topics": [
    { "topic": "topic-a"},
    { "topic": "topic-b"}
  ]
}
```

2. Copy the ***`topics.json`*** file to one of the broker pods:
   On OpenShift:

```
cat topics.json | oc rsh -c kafka <BrokerPod> /bin/bash -c \
  'cat > /tmp/topics.json'
```

3. Use the **`kafka-reassign-partitions.sh`**` command to generate the reassignment JSON.
   On OpenShift:

```
oc rsh -c kafka <BrokerPod> \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --topics-to-move-json-file /tmp/topics.json \
  --broker-list <BrokerList> \
  --generate
```

For example, to move all the partitions of **`topic-a`** and **`topic-b`** to brokers **`4`** and **`7`**

```
oc rsh -c kafka _<BrokerPod>_ \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --topics-to-move-json-file /tmp/topics.json \
  --broker-list 4,7 \
  --generate
```

3.1.20.4. Creating reassignment JSON files manually

You can manually create the reassignment JSON file if you want to move specific partitions.

3.1.20.5. Reassignment throttles

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. This might cause the reassignment to take longer to complete.

- If the throttle is too low then 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 then clients will be impacted.

For example, for producers, this could manifest as higher than normal latency waiting for acknowledgement. For consumers, this could manifest as a drop in throughput caused by higher latency between polls.

3.1.20.6. Scaling up a Kafka cluster

This procedure describes how to increase the number of brokers in a Kafka cluster.

Prerequisites

- An existing Kafka cluster.
- A *reassignment JSON file* named ***`reassignment.json`*** that describes how partitions should be reassigned to brokers in the enlarged cluster.

Procedure

1. Add as many new brokers as you need by increasing the **`Kafka.spec.kafka.replicas`** configuration option.
2. Verify that the new broker pods have started.
3. Copy the ***`reassignment.json`*** file to the broker pod on which you will later execute the commands:
   On OpenShift:

   ```
   cat reassignment.json | \
     oc rsh -c kafka broker-pod /bin/bash -c \
     'cat > /tmp/reassignment.json'
   ```

   For example:

   ```
   cat reassignment.json | \
     oc rsh -c kafka my-cluster-kafka-0 /bin/bash -c \
     'cat > /tmp/reassignment.json'
   ```

4. Execute the partition reassignment using the **`kafka-reassign-partitions.sh`** command line tool from the same broker pod.
   On OpenShift:

   ```
   oc rsh -c kafka broker-pod \
     bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
     --reassignment-json-file /tmp/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:

On OpenShift:

```
oc rsh -c kafka my-cluster-kafka-0 \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 5000000 \
  --execute
```

This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

5. If you need to change the throttle during reassignment you can use the same command line with a different throttled rate. For example:
   On OpenShift:

```
oc rsh -c kafka my-cluster-kafka-0 \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

6. Periodically verify whether the reassignment has completed using the **kafka-reassign-partitions.sh** command line tool from any of the broker pods. This is the same command as the previous step but with the **--verify** option instead of the **--execute** option.
   On OpenShift:

```
oc rsh -c kafka broker-pod \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

   For example, on {OpenShift},

```
oc rsh -c kafka my-cluster-kafka-0 \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

7. The reassignment has finished when the **--verify** command reports each of the partitions being moved as completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

3.1.20.7. Scaling down a Kafka cluster

Additional resources

This procedure describes how to decrease the number of brokers in a Kafka cluster.

Prerequisites

- An existing Kafka cluster.
- A *reassignment JSON file* named ***reassignment.json*** describing how partitions should be reassigned to brokers in the cluster once the broker(s) in the highest numbered **Pod(s)** have been removed.

Procedure

1. Copy the ***reassignment.json*** file to the broker pod on which you will later execute the commands:
   On OpenShift:

   ```
   cat reassignment.json | \
     oc rsh -c kafka broker-pod /bin/bash -c \
     'cat > /tmp/reassignment.json'
   ```

   For example:

   ```
   cat reassignment.json | \
     oc rsh -c kafka my-cluster-kafka-0 /bin/bash -c \
     'cat > /tmp/reassignment.json'
   ```

2. Execute the partition reassignment using the **kafka-reassign-partitions.sh** command line tool from the same broker pod.
   On OpenShift:

   ```
   oc rsh -c kafka broker-pod \
     bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
     --reassignment-json-file /tmp/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:

   On OpenShift:

   ```
   oc rsh -c kafka my-cluster-kafka-0 \
     bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
     --reassignment-json-file /tmp/reassignment.json \
     --throttle 5000000 \
     --execute
   ```

   This command will print out two reassignment JSON objects. The first records the current assignment for the partitions being moved. You should save this to a local file (not a file in the pod) in case you need to revert the reassignment later on. The second JSON object is the target reassignment you have passed in your reassignment JSON file.

3. If you need to change the throttle during reassignment you can use the same command line with a different throttled rate. For example:
   On OpenShift:

   ```
   oc rsh -c kafka my-cluster-kafka-0 \
   ```

```
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

4. Periodically verify whether the reassignment has completed using the **kafka-reassign-partitions.sh** command line tool from any of the broker pods. This is the same command as the previous step but with the **--verify** option instead of the **--execute** option.
   On OpenShift:

```
oc rsh -c kafka broker-pod \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

   For example, on {OpenShift},

```
oc rsh -c kafka my-cluster-kafka-0 \
  bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

5. The reassignment has finished when the **--verify** command reports each of the partitions being moved as completed successfully. This final **--verify** will also have the effect of removing any reassignment throttles. You can now delete the revert file if you saved the JSON for reverting the assignment to their original brokers.

6. Once all the partition reassignments have finished, the broker(s) being removed should not have responsibility for any of the partitions in the cluster. You can verify this by checking that the broker's data log directory does not contain any live partition logs. If the log directory on the broker contains a directory that does not match the extended regular expression **\.[a-z0-9]-delete$** then the broker still has live partitions and it should not be stopped.
   You can check this by executing the command:

```
oc rsh <BrokerN> -c kafka /bin/bash -c \
  "ls -l /var/lib/kafka/kafka-log_<N>_ | grep -E '^d' | grep -vE
'[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$'"
```

   where *N* is the number of the **Pod(s)** being deleted.

   If the above command prints any output then the broker still has live partitions. In this case, either the reassignment has not finished, or the reassignment JSON file was incorrect.

7. Once you have confirmed that the broker has no live partitions you can edit the **Kafka.spec.kafka.replicas** of your **Kafka** resource, which will scale down the **StatefulSet**, deleting the highest numbered broker **Pod(s)**.

3.1.21. Deleting Kafka nodes manually

Additional resources

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

the **Pod** and its related **PersistentVolumeClaim** are recreated automatically.

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

Prerequisites

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

Procedure

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

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

1. Annotate the **Pod** resource in OpenShift.
   + On OpenShift use **oc annotate**:

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Kafka cluster on OpenShift, see Section 2.3.1, “Deploying the Kafka cluster to OpenShift”.

3.1.22. Deleting Zookeeper nodes manually

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

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

Prerequisites

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

Procedure

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

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

1. Annotate the **`Pod`** resource in OpenShift.
   + On OpenShift use **`oc annotate`**:

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Zookeeper cluster on OpenShift, see Section 2.3.1, “Deploying the Kafka cluster to OpenShift”.

3.1.23. List of resources created as part of Kafka cluster

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

***`cluster-name`*`-kafka`**

StatefulSet which is in charge of managing the Kafka broker pods.

***`cluster-name`*`-kafka-brokers`**

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

***`cluster-name`*`-kafka-bootstrap`**

Service can be used as bootstrap servers for Kafka clients.

***`cluster-name`*`-kafka-external-bootstrap`**

Bootstrap service for clients connecting from outside of the OpenShift cluster. This resource will be created only when external listener is enabled.

cluster-name*-kafka-*pod-id

Service used to route traffic from outside of the OpenShift cluster to individual pods. This resource will be created only when external listener is enabled.

***cluster-name*-kafka-external-bootstrap**

Bootstrap route for clients connecting from outside of the OpenShift cluster. This resource will be created only when external listener is enabled and set to type **route**.

cluster-name*-kafka-*pod-id

Route for traffic from outside of the OpenShift cluster to individual pods. This resource will be created only when external listener is enabled and set to type **route**.

***cluster-name*-kafka-config**

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

***cluster-name*-kafka-brokers**

Secret with Kafka broker keys.

***cluster-name*-kafka**

Service account used by the Kafka brokers.

strimzi-*namespace-name*-*cluster-name*-kafka-init

Cluster role binding used by the Kafka brokers.

***cluster-name*-zookeeper**

StatefulSet which is in charge of managing the Zookeeper node pods.

***cluster-name*-zookeeper-nodes**

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

***cluster-name*-zookeeper-client**

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

***cluster-name*-zookeeper-config**

ConfigMap which contains the Zookeeper ancillary configuration and is mounted as a volume by the Zookeeper node pods.

***cluster-name*-zookeeper-nodes**

Secret with Zookeeper node keys.

***cluster-name*-entity-operator**

Deployment with Topic and User Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-topic-operator-config**

Configmap with ancillary configuration for Topic Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-user-operator-config**

Configmap with ancillary configuration for User Operators. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-operator-certs**

Secret with Entitiy operators keys for communication with Kafka and Zookeeper. This resource will be created only if Cluster Operator deployed Entity Operator.

***cluster-name*-entity-operator**

Service account used by the Entity Operator.

strimzi-*cluster-name*-topic-operator

Role binding used by the Entity Operator.

strimzi-*cluster-name*-user-operator

Role binding used by the Entity Operator.

***cluster-name*-cluster-ca**

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

***cluster-name*-cluster-ca-cert**

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

***cluster-name*-clients-ca**

Secret with the Clients CA used to encrypt the communication between Kafka brokers and Kafka clients.

***cluster-name*-clients-ca-cert**

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

data-*cluster-name*-kafka-*idx*

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

data-*cluster-name*-zookeeper-*idx*

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

3.2. KAFKA CONNECT CLUSTER CONFIGURATION

The full schema of the **KafkaConnect** resource is described in the Section B.29, "**KafkaConnect** schema reference". All labels that are applied to the desired **KafkaConnect** resource will also be applied to the OpenShift resources making up the Kafka Connect cluster. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.2.1. Replicas

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

3.2.1.1. Configuring the number of nodes

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

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.2.2. Bootstrap servers

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

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

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

3.2.2.1. Configuring bootstrap servers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
```

```
metadata:
  name: my-cluster
spec:
  # ...
  bootstrapServers: my-cluster-kafka-bootstrap:9092
  # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.3. Connecting to Kafka brokers using TLS

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

3.2.3.1. TLS support in Kafka Connect

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

An example showing TLS configuration with multiple certificates

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

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

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

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

```
      - secretName: my-secret
        certificate: ca2.crt
# ...
```

3.2.3.2. Configuring TLS in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Find out the name of the secret with the certificate which should be used for TLS Server Authentication and the key under which the certificate is stored in the secret. If such secret does not exist yet, prepare the certificate in a file and create the secret.
   On OpenShift this can be done using **`oc create`**:

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

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

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

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.4. Connecting to Kafka brokers with Authentication

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

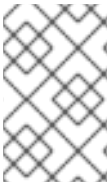
3.2.4.1. Authentication support in Kafka Connect

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

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

3.2.4.1.1. TLS Client Authentication

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



NOTE

TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Connect see Section 3.2.3, “Connecting to Kafka brokers using TLS”.

An example showing TLS client authentication configuration

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

3.2.4.1.2. SCRAM-SHA-512 authentication

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

An example showing SCRAM-SHA-512 client authentication configuration

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

```
      secretName: my-connect-user
      password: password
  # ...
```

3.2.4.2. Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Find out the name of the **Secret** with the public and private keys which should be used for TLS Client Authentication and the keys under which they are stored in the **Secret**. If such a **Secret** does not exist yet, prepare the keys in a file and create the **Secret**.
   On OpenShift this can be done using **oc create**:

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

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

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

3. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.2.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

- Username of the user which should be used for authentication

Procedure

1. Find out the name of the **Secret** with the password which should be used for authentication and the key under which the password is stored in the **Secret**. If such a **Secret** does not exist yet, prepare a file with the password and create the **Secret**.
   On OpenShift this can be done using **oc create**:

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

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

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

3. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.2.5. Kafka Connect configuration

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

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

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

These options are automatically configured by AMQ Streams.

3.2.5.1. Kafka Connect configuration

Kafka Connect can be configured using the **config** property in **KafkaConnect.spec** and

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

- String
- Number
- Boolean

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

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

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

IMPORTANT

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

Selected options have default values:

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

- **`internal.key.converter`** with default value **`org.apache.kafka.connect.json.JsonConverter`**
- **`internal.value.converter`** with default value **`org.apache.kafka.connect.json.JsonConverter`**
- **`internal.key.converter.schemas.enable`** with default value **`false`**
- **`internal.value.converter.schemas.enable`** with default value **`false`**

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

An example showing Kafka Connect configuration

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

3.2.5.2. Configuring Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaConnect
metadata:
```

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.6. CPU and memory resources

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

- Memory
- CPU

AMQ Streams is using the OpenShift syntax for specifying CPU and memory resources.

3.2.6.1. Resource limits and requests

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

- **`Kafka.spec.kafka`**
- **`Kafka.spec.kafka.tlsSidecar`**
- **`Kafka.spec.zookeeper`**
- **`Kafka.spec.zookeeper.tlsSidecar`**
- **`Kafka.spec.entityOperator.topicOperator`**
- **`Kafka.spec.entityOperator.userOperator`**

- **`Kafka.spec.entityOperator.tlsSidecar`**
- **`KafkaConnect.spec`**
- **`KafkaConnectS2I.spec`**

3.2.6.1.1. Resource requests

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

IMPORTANT

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

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

An example showing resource request configuration

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

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

An example showing resource request configuration with memory request only

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

Or:

An example showing resource request configuration with CPU request only

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

3.2.6.1.2. Resource limits

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

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

An example showing resource limits configuration

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

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

An example showing resource limit configuration with memory request only

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

Or:

An example showing resource limits configuration with CPU request only

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

3.2.6.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

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

An example of using different CPU units

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

–

NOTE

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

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

3.2.6.1.4. Supported memory formats

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

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

An example of using different memory units

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

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

3.2.6.1.5. Additional resources

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

3.2.6.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

```
kafka:
  # ...
  resources:
    requests:
      cpu: "8"
      memory: 64Gi
    limits:
      cpu: "12"
      memory: 128Gi
  # ...
zookeeper:
  # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

Additional resources

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

3.2.7. Logging

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

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

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

3.2.7.1. Using inline logging setting

Procedure

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

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

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

2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.7.2. Using external ConfigMap for logging setting

Procedure

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

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

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

2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.7.3. Loggers

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

Components and their loggers are listed below.

- Kafka
  - **`kafka.root.logger.level`**
  - **`log4j.logger.org.I0Itec.zkclient.ZkClient`**
  - **`log4j.logger.org.apache.zookeeper`**
  - **`log4j.logger.kafka`**
  - **`log4j.logger.org.apache.kafka`**
  - **`log4j.logger.kafka.request.logger`**
  - **`log4j.logger.kafka.network.Processor`**
  - **`log4j.logger.kafka.server.KafkaApis`**

  - **`log4j.logger.kafka.network.RequestChannel$`**
  - **`log4j.logger.kafka.controller`**
  - **`log4j.logger.kafka.log.LogCleaner`**
  - **`log4j.logger.state.change.logger`**
  - **`log4j.logger.kafka.authorizer.logger`**
- Zookeeper
  - **`zookeeper.root.logger`**
- Kafka Connect and Kafka Connect with Source2Image support
  - **`connect.root.logger.level`**
  - **`log4j.logger.org.apache.zookeeper`**
  - **`log4j.logger.org.I0Itec.zkclient`**
  - **`log4j.logger.org.reflections`**
- Kafka Mirror Maker
  - **`mirrormaker.root.logger`**
- Topic Operator
  - **`rootLogger.level`**
- User Operator
  - **`rootLogger.level`**

3.2.8. Healthchecks

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

- Liveness probes
- Readiness probes

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

Users can configure selected options for liveness and readiness probes

3.2.8.1. Healthcheck configurations

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

- **`Kafka.spec.kafka`**

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

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

- **`initialDelaySeconds`**
- **`timeoutSeconds`**

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

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

An example of liveness and readiness probe configuration

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

3.2.8.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.2.9. Prometheus metrics

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

3.2.9.1. Metrics configuration

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

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

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

Example of enabling metrics without any further configuration

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

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

Example of enabling metrics with additional Prometheus JMX Exporter configuration

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

```
  kafka:
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=
(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

3.2.9.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.10. JVM Options

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

3.2.10.1. JVM configuration

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

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

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

-Xms and -Xmx

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

NOTE

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

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

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

IMPORTANT

Setting **`-Xmx`** explicitly requires some care:

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

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

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

IMPORTANT

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

Example fragment configuring -Xmx and -Xms

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

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

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

-server

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

Example fragment configuring -server

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

NOTE

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

-XX

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

Example showing the use of the -XX object

```
jvmOptions:
```

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

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

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```

NOTE

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

3.2.10.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.11. Container images

AMQ Streams allows you to configure container images which will be used for its components.

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

3.2.11.1. Container image configurations

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

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

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

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

1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
2. **strimzi/zookeeper-stunnel:latest** container image.

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

WARNING

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

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
```

```
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.2.11.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.12. Configuring pod scheduling

IMPORTANT

When two application are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.2.12.1. Scheduling pods based on other applications

3.2.12.1.1. Avoid critical applications to share the node

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

3.2.12.1.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.2.12.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
```

```
                    operator: In
                    values:
                      - postgresql
                      - mongodb
              topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.12.2. Scheduling pods to specific nodes

3.2.12.2.1. Node scheduling

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

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

3.2.12.2.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.2.12.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   On OpenShift this can be done using **`oc label`**:

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

   Alternatively, some of the existing labels might be reused.

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

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

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.12.3. Using dedicated nodes

3.2.12.3.1. Dedicated nodes

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

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

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

3.2.12.3.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.2.12.3.3. Tolerations

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

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

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

3.2.12.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

3. Set the taints on the selected nodes
   On OpenShift this can be done using **`oc adm taint`**:

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

4. Additionally, add a label to the selected nodes as well.
   On OpenShift this can be done using **`oc label`**:

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

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

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

6. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.2.13. List of resources created as part of Kafka Connect cluster

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

***connect-cluster-name*-connect**

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

***connect-cluster-name*-connect-api**

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

***connect-cluster-name*-config**

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

3.3. KAFKA CONNECT CLUSTER WITH SOURCE2IMAGE SUPPORT

The full schema of the **KafkaConnectS2I** resource is described in the Section B.37, "**KafkaConnectS2I** schema reference". All labels that are applied to the desired **KafkaConnectS2I** resource will also be applied to the OpenShift resources making up the Kafka Connect cluster with Source2Image support. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.3.1. Replicas

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

3.3.1.1. Configuring the number of nodes

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

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.3.2. Bootstrap servers

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

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

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

3.3.2.1. Configuring bootstrap servers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.3.3. Connecting to Kafka brokers using TLS

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

3.3.3.1. TLS support in Kafka Connect

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

An example showing TLS configuration with multiple certificates

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

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

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

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

3.3.3.2. Configuring TLS in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Find out the name of the secret with the certificate which should be used for TLS Server Authentication and the key under which the certificate is stored in the secret. If such secret does not exist yet, prepare the certificate in a file and create the secret.
   On OpenShift this can be done using **`oc create`**:

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

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnect
   ```

```
metadata:
  name: my-connect
spec:
  # ...
  tls:
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        certificate: ca.crt
  # ...
```

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.3.4. Connecting to Kafka brokers with Authentication

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

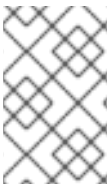
3.3.4.1. Authentication support in Kafka Connect

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

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

3.3.4.1.1. TLS Client Authentication

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



NOTE

TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Connect see Section 3.3.3, “Connecting to Kafka brokers using TLS”.

An example showing TLS client authentication configuration

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

```
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  # ...
```

3.3.4.1.2. SCRAM-SHA-512 authentication

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

An example showing SCRAM-SHA-512 client authentication configuration

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

3.3.4.2. Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Find out the name of the **Secret** with the public and private keys which should be used for TLS Client Authentication and the keys under which they are stored in the **Secret**. If such a **Secret** does not exist yet, prepare the keys in a file and create the **Secret**.
   On OpenShift this can be done using **oc create**:

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

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   ```

```
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: my-public.crt
      key: my-private.key
  # ...
```

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.3.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect

Prerequisites

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

Procedure

1. Find out the name of the **`Secret`** with the password which should be used for authentication and the key under which the password is stored in the **`Secret`**. If such a **`Secret`** does not exist yet, prepare a file with the password and create the **`Secret`**.
   On OpenShift this can be done using **`oc create`**:

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

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

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

```
        secretName: _my-secret_
        password: _my-password.txt_
    # ...
```

3. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.3.5. Kafka Connect configuration

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

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

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

These options are automatically configured by AMQ Streams.

3.3.5.1. Kafka Connect configuration

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

- String
- Number
- Boolean

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

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

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

IMPORTANT

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

Selected options have default values:

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

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

An example showing Kafka Connect configuration

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

```
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    internal.key.converter: org.apache.kafka.connect.json.JsonConverter
    internal.value.converter: org.apache.kafka.connect.json.JsonConverter
    internal.key.converter.schemas.enable: false
    internal.value.converter.schemas.enable: false
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  # ...
```

3.3.5.2. Configuring Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.3.6. CPU and memory resources

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

- Memory
- CPU

AMQ Streams is using the OpenShift syntax for specifying CPU and memory resources.

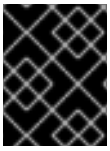
3.3.6.1. Resource limits and requests

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

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

3.3.6.1.1. Resource requests

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



IMPORTANT

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

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

An example showing resource request configuration

```
# ...
resources:
  requests:
```

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

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

An example showing resource request configuration with memory request only

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

Or:

An example showing resource request configuration with CPU request only

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

3.3.6.1.2. Resource limits

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

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

An example showing resource limits configuration

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

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

An example showing resource limit configuration with memory request only

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

Or:

An example showing resource limits configuration with CPU request only

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

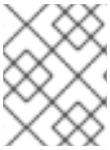
3.3.6.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

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

An example of using different CPU units

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



NOTE

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

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

3.3.6.1.4. Supported memory formats

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

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

An example of using different memory units

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

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

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

3.3.6.1.5. Additional resources

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

3.3.6.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

Additional resources

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

3.3.7. Logging

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

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

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

3.3.7.1. Using inline logging setting

Procedure

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

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

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

2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

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

3.3.7.2. Using external ConfigMap for logging setting

Procedure

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

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

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

2. Create or update the Kafka resource in OpenShift.

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

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

3.3.7.3. Loggers

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

Components and their loggers are listed below.

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

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

3.3.8. Healthchecks

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

- Liveness probes
- Readiness probes

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

Users can configure selected options for liveness and readiness probes

3.3.8.1. Healthcheck configurations

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

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

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

- **`initialDelaySeconds`**
- **`timeoutSeconds`**

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

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

An example of liveness and readiness probe configuration

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

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

3.3.8.2. Configuring healthchecks

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.3.9. Prometheus metrics

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

3.3.9.1. Metrics configuration

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

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

- **`KafkaConnect.spec`**
- **`KafkaConnectS2I.spec`**

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

Example of enabling metrics without any further configuration

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

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

Example of enabling metrics with additional Prometheus JMX Exporter configuration

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

3.3.9.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.3.10. JVM Options

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

3.3.10.1. JVM configuration

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

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

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

-Xms and -Xmx

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

NOTE

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

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

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

IMPORTANT

Setting **-Xmx** explicitly requires some care:

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

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

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

IMPORTANT

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

Example fragment configuring -Xmx and -Xms

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

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

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

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

-server

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

Example fragment configuring -server

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

NOTE

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

-XX

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

Example showing the use of the -XX object

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

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

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```

NOTE

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

3.3.10.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.3.11. Container images

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

3.3.11.1. Container image configurations

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

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.zookeeper.tlsSidecar**

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

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

- For Kafka brokers:
  1. Container image specified in the **`STRIMZI_DEFAULT_KAFKA_IMAGE`** environment variable from the Cluster Operator configuration.
  2. **`strimzi/kafka:latest`** container image.
- For Kafka broker TLS sidecar:
  1. Container image specified in the **`STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE`** environment variable from the Cluster Operator configuration.
  2. **`strimzi/kafka-stunnel:latest`** container image.
- For Zookeeper nodes:
  1. Container image specified in the **`STRIMZI_DEFAULT_ZOOKEEPER_IMAGE`** environment variable from the Cluster Operator configuration.
  2. **`strimzi/zookeeper:latest`** container image.
- For Zookeeper node TLS sidecar:
  1. Container image specified in the **`STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE`** environment variable from the Cluster Operator configuration.
  2. **`strimzi/zookeeper-stunnel:latest`** container image.
- For Topic Operator:
  1. Container image specified in the **`STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE`** environment variable from the Cluster Operator configuration.
- For User Operator:
  1. Container image specified in the **`STRIMZI_DEFAULT_USER_OPERATOR_IMAGE`** environment variable from the Cluster Operator configuration.
  2. **`strimzi/user-operator:latest`** container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the **`STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE`** environment variable

from the Cluster Operator configuration.

2. **`strimzi/entity-operator-stunnel:latest`** container image.

- For Kafka Connect:
  1. Container image specified in the **`STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE`** environment variable from the Cluster Operator configuration.
  2. **`strimzi/kafka-connect:latest`** container image.
- For Kafka Connect with Source2image support:
  1. Container image specified in the **`STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE`** environment variable from the Cluster Operator configuration.
  2. **`strimzi/kafka-connect-s2i:latest`** container image.

WARNING

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

Example of container image configuration

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

3.3.11.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.3.12. Configuring pod scheduling

IMPORTANT

When two application are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.3.12.1. Scheduling pods based on other applications

3.3.12.1.1. Avoid critical applications to share the node

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

3.3.12.1.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.3.12.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

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

3.3.12.2. Scheduling pods to specific nodes

3.3.12.2.1. Node scheduling

The OpenShift cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while other might be optimized for storage (fast local SSDs) or

network. Using different nodes helps to optimize both costs and performance. To achieve the best possible performance, it is important to allow scheduling of AMQ Streams components to use the right nodes.

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

3.3.12.2.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.3.12.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   On OpenShift this can be done using **`oc label`**:

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

   Alternatively, some of the existing labels might be reused.

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

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

```
    # ...
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
              - key: node-type
                operator: In
                values:
                - fast-network
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.3.12.3. Using dedicated nodes

3.3.12.3.1. Dedicated nodes

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

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

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

3.3.12.3.2. Affinity

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

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

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

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

3.3.12.3.3. Tolerations

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

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

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

3.3.12.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated
2. Make sure there are no workloads scheduled on these nodes
3. Set the taints on the selected nodes
   On OpenShift this can be done using **oc adm taint**:

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

4. Additionally, add a label to the selected nodes as well.
   On OpenShift this can be done using **oc label**:

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

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

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       tolerations:
         - key: "dedicated"
   ```

```
        operator: "Equal"
        value: "Kafka"
        effect: "NoSchedule"
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: dedicated
              operator: In
              values:
              - Kafka
    # ...
  zookeeper:
    # ...
```

6. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.3.13. List of resources created as part of Kafka Connect cluster with Source2Image support

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

***connect-cluster-name*-connect-source**

ImageStream which is used as the base image for the newly-built Docker images.

***connect-cluster-name*-connect**

BuildConfig which is responsible for building the new Kafka Connect Docker images.

***connect-cluster-name*-connect**

ImageStream where the newly built Docker images will be pushed.

***connect-cluster-name*-connect**

DeploymentConfig which is in charge of creating the Kafka Connect worker node pods.

***connect-cluster-name*-connect-api**

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

***connect-cluster-name*-config**

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

3.3.14. Using OpenShift builds and S2I to create new images

OpenShift supports builds, which can be used together with the Source-to-Image (S2I) framework to create new container images. An OpenShift build takes a builder image with S2I support together with source code and binaries provided by the user and uses them to build a new container image. The newly created container image is stored in OpenShift's local container image repository and can be used in deployments. AMQ Streams provides a Kafka Connect builder image, which can be found on Red Hat Container Catalog as **`registry.access.redhat.com/amqstreams-1/amqstreams10-`**

kafkaconnects2i-openshift:1.0.0 with this S2I support. It takes user-provided binaries (with plugins and connectors) and creates a new Kafka Connect image. This enhanced Kafka Connect image can be used with the Kafka Connect deployment.

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

Procedure

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

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

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

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

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

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

NOTE

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

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

3.4. KAFKA MIRROR MAKER CONFIGURATION

The full schema of the **`KafkaMirrorMaker`** resource is described in the Section B.50, "**`KafkaMirrorMaker`** schema reference". All labels that apply to the desired **`KafkaMirrorMaker`** resource will also be applied to the OpenShift resources making up Mirror Maker. This provides a convenient mechanism for those resources to be labelled in whatever way the user requires.

3.4.1. Replicas

It is possible to run multiple Mirror Maker replicas. The number of replicas is defined in the **`KafkaMirrorMaker`** resource. You can run multiple Mirror Maker replicas to provide better availability and scalability. However, when running Kafka Mirror Maker on OpenShift it is not absolutely necessary to run multiple replicas of the Kafka Mirror Maker for high availability. When the node where the Kafka Mirror Maker has deployed crashes, OpenShift will automatically reschedule the Kafka Mirror Maker pod to a different node. However, running Kafka Mirror Maker with multiple replicas can provide faster failover times as the other nodes will be up and running.

3.4.1.1. Configuring the number of replicas

The number of Kafka Mirror Maker replicas can be configured using the **`replicas`** property in **`KafkaMirrorMaker.spec`**.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  replicas: 3
  # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.4.2. Bootstrap servers

Kafka Mirror Maker always works together with two Kafka clusters (source and target). The source and the target Kafka clusters are specified in the form of two lists of comma-separated list of **`<hostname>:<port>`** pairs. The bootstrap server lists can refer to Kafka clusters which do not need to be deployed in the same OpenShift cluster. They can even refer to any Kafka cluster not deployed by AMQ Streams or even deployed by AMQ Streams but on a different OpenShift cluster and accessible from outside.

If on the same OpenShift cluster, each list must ideally contain the Kafka cluster bootstrap service which is named **`<cluster-name>-kafka-bootstrap`** and a port of 9092 for plain traffic or 9093 for encrypted traffic. If deployed by AMQ Streams but on different OpenShift clusters, the list content depends on the way used for exposing the clusters (routes, nodeports or loadbalancers).

The list of bootstrap servers can be configured in the **`KafkaMirrorMaker.spec.consumer.bootstrapServers`** and **`KafkaMirrorMaker.spec.producer.bootstrapServers`** properties. The servers should be a comma-separated list containing one or more Kafka brokers or a **`Service`** pointing to Kafka brokers specified as a **`<hostname>:<port>`** pairs.

When using Kafka Mirror Maker with a Kafka cluster not managed by AMQ Streams, you can specify the bootstrap servers list according to the configuration of the given cluster.

3.4.2.1. Configuring bootstrap servers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **`KafkaMirrorMaker.spec.consumer.bootstrapServers`** and **`KafkaMirrorMaker.spec.producer.bootstrapServers`** properties. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    bootstrapServers: my-source-cluster-kafka-bootstrap:9092
  # ...
  producer:
    bootstrapServers: my-target-cluster-kafka-bootstrap:9092
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.4.3. Whitelist

You specify the list topics that the Kafka Mirror Maker has to mirror from the source to the target Kafka cluster in the KafkaMirrorMaker resource using the *whitelist* option. It allows any regular expression from the simplest case with a single topic name to complex patterns. For example, you can mirror topics A and B using "A|B" or all topics using "*". You can also pass multiple regular expressions separated by commas to the Kafka Mirror Maker.

3.4.3.1. Configuring the topics whitelist

Specify the list topics that have to be mirrored by the Kafka Mirror Maker from source to target Kafka cluster using the **`whitelist`** property in **`KafkaMirrorMaker.spec`**.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **`whitelist`** property in the **`KafkaMirrorMaker`** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  whitelist: "my-topic|other-topic"
  # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.4.4. Consumer group identifier

The Kafka Mirror Maker uses Kafka consumer to consume messages and it behaves like any other Kafka consumer client. It is in charge to consume the messages from the source Kafka cluster which will be mirrored to the target Kafka cluster. The consumer needs to be part of a *consumer group* for being assigned partitions.

3.4.4.1. Configuring the consumer group identifier

The consumer group identifier can be configured in the **`KafkaMirrorMaker.spec.consumer.groupId`** property.

Prerequisites

- An OpenShift cluster

- A running Cluster Operator

Procedure

1. Edit the **`KafkaMirrorMaker.spec.consumer.groupId`** property. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    groupId: "my-group"
  # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

3.4.5. Number of consumer streams

You can increase the throughput in mirroring topics by increase the number of consumer threads. More consumer threads will belong to the same configured *consumer group*. The topic partitions will be assigned across these consumer threads which will consume messages in parallel.

3.4.5.1. Configuring the number of consumer streams

The number of consumer streams can be configured using the **`KafkaMirrorMaker.spec.consumer.numStreams`** property.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **`KafkaMirrorMaker.spec.consumer.numStreams`** property. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    numStreams: 2
  # ...
```

2. Create or update the resource.

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

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

3.4.6. Connecting to Kafka brokers using TLS

By default, Kafka Mirror Maker will try to connect to Kafka brokers, in the source and target clusters, using a plain text connection. You must make additional configurations to use TLS.

3.4.6.1. TLS support in Kafka Mirror Maker

TLS support is configured in the **`tls`** sub-property of **`consumer`** and **`producer`** properties in **`KafkaMirrorMaker.spec`**. The **`tls`** property contains a list of secrets with key names under which the certificates are stored. The certificates should be stored in X.509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    tls:
      trustedCertificates:
        - secretName: my-source-secret
          certificate: ca.crt
        - secretName: my-other-source-secret
          certificate: certificate.crt
  # ...
  producer:
    tls:
      trustedCertificates:
        - secretName: my-target-secret
          certificate: ca.crt
        - secretName: my-other-target-secret
          certificate: certificate.crt
  # ...
```

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

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

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    tls:
      trustedCertificates:
        - secretName: my-source-secret
```

```
          certificate: ca.crt
        - secretName: my-source-secret
          certificate: ca2.crt
# ...
producer:
  tls:
    trustedCertificates:
      - secretName: my-target-secret
        certificate: ca.crt
      - secretName: my-target-secret
        certificate: ca2.crt
# ...
```

3.4.6.2. Configuring TLS encryption in Kafka Mirror Maker

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

As the Kafka Mirror Maker connects to two Kafka clusters (source and target), you can choose to configure TLS for one or both the clusters. The following steps describe how to configure TLS on the consumer side for connecting to the source Kafka cluster:

1. Find out the name of the secret with the certificate which should be used for TLS Server Authentication and the key under which the certificate is stored in the secret. If such secret does not exist yet, prepare the certificate in a file and create the secret.
   On OpenShift this can be done using **`oc create`**:

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

2. Edit the **`KafkaMirrorMaker.spec.consumer.tls`** property. For example:

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

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

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

Repeat the above steps for configuring TLS on the target Kafka cluster. In this case, the secret containing the certificate has to be configured in the **`KafkaMirrorMaker.spec.producer.tls`** property.

3.4.7. Connecting to Kafka brokers with Authentication

By default, Kafka Mirror Maker will try to connect to Kafka brokers without any authentication. Authentication can be enabled in the **`KafkaMirrorMaker`** resource.

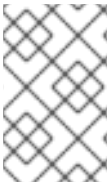
3.4.7.1. Authentication support in Kafka Mirror Maker

Authentication can be configured in the **`KafkaMirrorMaker.spec.consumer.authentication`** and **`KafkaMirrorMaker.spec.producer.authentication`** properties. The **`authentication`** property specifies the type of the authentication method which should be used and additional configuration details depending on the mechanism. The currently supported authentication types are:

- TLS client authentication
- SASL based authentication using SCRAM-SHA-512 mechanism

3.4.7.1.1. TLS Client Authentication

To use the TLS client authentication, set the **`type`** property to the value **`tls`**. The TLS client authentication uses TLS certificate to authenticate. The certificate has to be specified in the certificateAndKey property. It is always loaded from an OpenShift secret. Inside the secret, it has to be stored in the X.509 format separately as public and private keys.



NOTE

TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Mirror Maker see Section 3.4.6, “Connecting to Kafka brokers using TLS”.

An example showing TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    authentication:
      type: tls
      certificateAndKey:
        secretName: my-source-secret
        certificate: public.crt
        key: private.key
  # ...
  producer:
    authentication:
      type: tls
      certificateAndKey:
        secretName: my-target-secret
```

```
        certificate: public.crt
        key: private.key
  # ...
```

3.4.7.1.2. SCRAM-SHA-512 authentication

To use the authentication using the SCRAM-SHA-512 SASL mechanism, set the **`type`** property to the value **`scram-sha-512`**. It is possible to use it only if the broker listener, clients are connecting to, is configured to use it. SCRAM-SHA-512 uses a username and password to authenticate. Specify the username in the **`username`** property. Specify the password as a link to a **`Secret`** containing the password in the **`passwordSecret`** property. It has to specify the name of the **`Secret`** containing the password and the name of the key under which the password is stored inside the **`Secret`**.

An example showing SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    authentication:
      type: scram-sha-512
      username: my-source-user
      passwordSecret:
        secretName: my-source-user
        password: password
  # ...
  producer:
    authentication:
      type: scram-sha-512
      username: my-producer-user
      passwordSecret:
        secretName: my-producer-user
        password: password
  # ...
```

3.4.7.2. Configuring TLS client authentication in Kafka Mirror Maker

Prerequisites

- An OpenShift cluster
- A running Cluster Operator with a **`tls`** listener with **`tls`** authentication enabled

Procedure

As the Kafka Mirror Maker connects to two Kafka clusters (source and target), you can choose to configure TLS client authentication for one or both the clusters. The following steps describe how to configure TLS client authentication on the consumer side for connecting to the source Kafka cluster:

1. Find out the name of the **Secret** with the public and private keys which should be used for TLS Client Authentication and the keys under which they are stored in the **Secret**. If such a **Secret** does not exist yet, prepare the keys in a file and create the **Secret**.
   On OpenShift this can be done using **oc create**:

   ```
   oc create secret generic <my-secret> --from-file=<my-public.crt> --
   from-file=<my-private.key>
   ```

2. Edit the **KafkaMirrorMaker.spec.consumer.authentication** property. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaMirrorMaker
   metadata:
     name: my-mirror-maker
   spec:
     # ...
     consumer:
       authentication:
         type: tls
         certificateAndKey:
           secretName: my-secret
           certificate: my-public.crt
           key: my-private.key
     # ...
   ```

3. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

   ```
   oc apply -f <your-file>
   ```

Repeat the above steps for configuring TLS client authentication on the target Kafka cluster. In this case, the secret containing the certificate has to be configured in the **KafkaMirrorMaker.spec.producer.authentication** property.

3.4.7.3. Configuring SCRAM-SHA-512 authentication in Kafka Mirror Maker

Prerequisites

- An OpenShift cluster
- A running Cluster Operator with a **listener** configured for SCRAM-SHA-512 authentication
- Username to be used for authentication

Procedure

As the Kafka Mirror Maker connects to two Kafka clusters (source and target), you can choose to configure SCRAM-SHA-512 authentication for one or both the clusters. The following steps describe how to configure SCRAM-SHA-512 authentication on the consumer side for connecting to the source Kafka cluster:

1. Find out the name of the **Secret** with the password which should be used for authentication and the key under which the password is stored in the **Secret**. If such a **Secret** does not exist yet, prepare a file with the password and create the **Secret**.

On OpenShift this can be done using **`oc create`**:

```
echo -n '1f2d1e2e67df' > <my-password.txt>
oc create secret generic <my-secret> --from-file=<my-password.txt>
```

2. Edit the **`KafkaMirrorMaker.spec.consumer.authentication`** property. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    authentication:
      type: scram-sha-512
      username: _<my-username>_
      passwordSecret:
        secretName: _<my-secret>_
        password: _<my-password.txt>_
  # ...
```

3. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f <your-file>
```

Repeat the above steps for configuring SCRAM-SHA-512 authentication on the target Kafka cluster. In this case, the secret containing the certificate has to be configured in the **`KafkaMirrorMaker.spec.producer.authentication`** property.

3.4.8. Kafka Mirror Maker configuration

AMQ Streams allows you to customize the configuration of the Kafka Mirror Maker by editing most of the options for the related consumer and producer. Producer options are listed in Apache Kafka documentation. Consumer options are listed in Apache Kafka documentation.

The only options which cannot be configured are those related to the following areas:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Consumer group identifier

These options are automatically configured by AMQ Streams.

3.4.8.1. Kafka Mirror Maker configuration

Kafka Mirror Maker can be configured using the **`config`** sub-property in **`KafkaMirrorMaker.spec.consumer`** and **`KafkaMirrorMaker.spec.producer`**. This property should contain the Kafka Mirror Maker consumer and producer configuration options as keys. The values could be in one of the following JSON types:

- String
- Number
- Boolean

Users can specify and configure the options listed in the Apache Kafka documentation and Apache Kafka documentation with the exception of those options which are managed directly by AMQ Streams. Specifically, all configuration options with keys equal to or starting with one of the following strings are forbidden:

- **`ssl.`**
- **`sasl.`**
- **`security.`**
- **`bootstrap.servers`**
- **`group.id`**

When one of the forbidden options is present in the **`config`** property, it will be ignored and a warning message will be printed to the Custer Operator log file. All other options will be passed to Kafka Mirror Maker.

IMPORTANT

The Cluster Operator does not validate keys or values in the provided **`config`** object. When an invalid configuration is provided, the Kafka Mirror Maker might not start or might become unstable. In such cases, the configuration in the **`KafkaMirrorMaker.spec.consumer.config`** or **`KafkaMirrorMaker.spec.producer.config`** object should be fixed and the cluster operator will roll out the new configuration for Kafka Mirror Maker.

An example showing Kafka Mirror Maker configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirroMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    config:
      max.poll.records: 100
      receive.buffer.bytes: 32768
  producer:
    config:
      compression.type: gzip
      batch.size: 8192
  # ...
```

3.4.8.2. Configuring Kafka Mirror Maker

Prerequisites

- Two running Kafka clusters (source and target)
- A running Cluster Operator

Procedure

1. Edit the **`KafkaMirrorMaker.spec.consumer.config`** and **`KafkaMirrorMaker.spec.producer.config`** properties. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaMirroMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    config:
      max.poll.records: 100
      receive.buffer.bytes: 32768
  producer:
    config:
      compression.type: gzip
      batch.size: 8192
  # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f <your-file>
```

3.4.9. CPU and memory resources

For every deployed container, AMQ Streams allows you to specify the resources which should be reserved for it and the maximum resources that can be consumed by it. AMQ Streams supports two types of resources:

- Memory
- CPU

AMQ Streams is using the OpenShift syntax for specifying CPU and memory resources.

3.4.9.1. Resource limits and requests

Resource limits and requests can be configured using the **`resources`** property in following resources:

- **`Kafka.spec.kafka`**
- **`Kafka.spec.kafka.tlsSidecar`**
- **`Kafka.spec.zookeeper`**
- **`Kafka.spec.zookeeper.tlsSidecar`**

- **`Kafka.spec.entityOperator.topicOperator`**
- **`Kafka.spec.entityOperator.userOperator`**
- **`Kafka.spec.entityOperator.tlsSidecar`**
- **`KafkaConnect.spec`**
- **`KafkaConnectS2I.spec`**

3.4.9.1.1. Resource requests

Requests specify the resources that will be reserved for a given container. Reserving the resources will ensure that they are always available.

IMPORTANT

If the resource request is for more than the available free resources in the OpenShift cluster, the pod will not be scheduled.

Resource requests can be specified in the **`request`** property. The resource requests currently supported by AMQ Streams are memory and CPU. Memory is specified under the property **`memory`**. CPU is specified under the property **`cpu`**.

An example showing resource request configuration

```
# ...
resources:
  requests:
    cpu: 12
    memory: 64Gi
# ...
```

It is also possible to specify a resource request just for one of the resources:

An example showing resource request configuration with memory request only

```
# ...
resources:
  requests:
    memory: 64Gi
# ...
```

Or:

An example showing resource request configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

3.4.9.1.2. Resource limits

Limits specify the maximum resources that can be consumed by a given container. The limit is not reserved and might not be always available. The container can use the resources up to the limit only when they are available. The resource limits should be always higher than the resource requests.

Resource limits can be specified in the **`limits`** property. The resource limits currently supported by AMQ Streams are memory and CPU. Memory is specified under the property **`memory`**. CPU is specified under the property **`cpu`**.

An example showing resource limits configuration

```
# ...
resources:
  limits:
    cpu: 12
    memory: 64Gi
# ...
```

It is also possible to specify the resource limit just for one of the resources:

An example showing resource limit configuration with memory request only

```
# ...
resources:
  limits:
    memory: 64Gi
# ...
```

Or:

An example showing resource limits configuration with CPU request only

```
# ...
resources:
  requests:
    cpu: 12
# ...
```

3.4.9.1.3. Supported CPU formats

CPU requests and limits are supported in the following formats:

- Number of CPU cores as integer (**`5`** CPU core) or decimal (**`2.5`** CPU core).
- Number or *millicpus* / *millicores* (**`100m`**) where 1000 *millicores* is the same **`1`** CPU core.

An example of using different CPU units

```
# ...
resources:
  requests:
    cpu: 500m
```

```
  limits:
    cpu: 2.5
# ...
```

NOTE

The amount of computing power of 1 CPU core might differ depending on the platform where the OpenShift is deployed.

For more details about the CPU specification, see the Meaning of CPU website.

3.4.9.1.4. Supported memory formats

Memory requests and limits are specified in megabytes, gigabytes, mebibytes, and gibibytes.

- To specify memory in megabytes, use the **M** suffix. For example **1000M**.
- To specify memory in gigabytes, use the **G** suffix. For example **1G**.
- To specify memory in mebibytes, use the **Mi** suffix. For example **1000Mi**.
- To specify memory in gibibytes, use the **Gi** suffix. For example **1Gi**.

An example of using different memory units

```
# ...
resources:
  requests:
    memory: 512Mi
  limits:
    memory: 2Gi
# ...
```

For more details about the memory specification and additional supported units, see the Meaning of memory website.

3.4.9.1.5. Additional resources

- For more information about managing computing resources on OpenShift, see Managing Compute Resources for Containers.

3.4.9.2. Configuring resource requests and limits

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **resources** property in the resource specifying the cluster deployment. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    resources:
      requests:
        cpu: "8"
        memory: 64Gi
      limits:
        cpu: "12"
        memory: 128Gi
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema, see **`Resources`** schema reference.

3.4.10. Logging

Logging enables you to diagnose error and performance issues of AMQ Streams. For the logging, various logger implementations are used. Kafka and Zookeeper use **`log4j`** logger and Topic Operator, User Operator, and other components use **`log4j2`** logger.

This section provides information about different loggers and describes how to configure log levels.

You can set the log levels by specifying the loggers and their levels directly (inline) or by using a custom (external) config map.

3.4.10.1. Using inline logging setting

Procedure

1. Edit the YAML file to specify the loggers and their level for the required components. For example:

```
apiVersion: {KafkaApiVersion}
kind: Kafka
spec:
  kafka:
    # ...
    logging:
      type: inline
      loggers:
       logger.name: "INFO"
    # ...
```

In the above example, the log level is set to INFO. You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF. For more information about the log levels, see log4j manual.

2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

3.4.10.2. Using external ConfigMap for logging setting

Procedure

1. Edit the YAML file to specify the name of the **`ConfigMap`** which should be used for the required components. For example:

```
apiVersion: {KafkaApiVersion}
kind: Kafka
spec:
  kafka:
    # ...
    logging:
      type: external
      name: customConfigMap
    # ...
```

   Remember to place your custom ConfigMap under **`log4j.properties`** eventually **`log4j2.properties`** key.

2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

3.4.10.3. Loggers

AMQ Streams consists of several components. Each component has its own loggers and is configurable. This section provides information about loggers of various components.

Components and their loggers are listed below.

- Kafka
  - **`kafka.root.logger.level`**
  - **`log4j.logger.org.I0Itec.zkclient.ZkClient`**
  - **`log4j.logger.org.apache.zookeeper`**
  - **`log4j.logger.kafka`**
  - **`log4j.logger.org.apache.kafka`**
  - **`log4j.logger.kafka.request.logger`**

  - `log4j.logger.kafka.network.Processor`
  - `log4j.logger.kafka.server.KafkaApis`
  - `log4j.logger.kafka.network.RequestChannel$`
  - `log4j.logger.kafka.controller`
  - `log4j.logger.kafka.log.LogCleaner`
  - `log4j.logger.state.change.logger`
  - `log4j.logger.kafka.authorizer.logger`
- Zookeeper
  - `zookeeper.root.logger`
- Kafka Connect and Kafka Connect with Source2Image support
  - `connect.root.logger.level`
  - `log4j.logger.org.apache.zookeeper`
  - `log4j.logger.org.I0Itec.zkclient`
  - `log4j.logger.org.reflections`
- Kafka Mirror Maker
  - `mirrormaker.root.logger`
- Topic Operator
  - `rootLogger.level`
- User Operator
  - `rootLogger.level`

3.4.11. Prometheus metrics

AMQ Streams supports Prometheus metrics using Prometheus JMX exporter to convert the JMX metrics supported by Apache Kafka and Zookeeper to Prometheus metrics. When metrics are enabled, they are exposed on port 9404.

3.4.11.1. Metrics configuration

Prometheus metrics can be enabled by configuring the `metrics` property in following resources:

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `KafkaConnect.spec`
- `KafkaConnectS2I.spec`

When the **metrics** property is not defined in the resource, the Prometheus metrics will be disabled. To enable Prometheus metrics export without any further configuration, you can set it to an empty object (**{}**).

Example of enabling metrics without any further configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics: {}
    # ...
  zookeeper:
    # ...
```

The **metrics** property might contain additional configuration for the Prometheus JMX exporter.

Example of enabling metrics with additional Prometheus JMX Exporter configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metrics:
      lowercaseOutputName: true
      rules:
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*><>Count"
          name: "kafka_server_$1_$2_total"
        - pattern: "kafka.server<type=(.+), name=(.+)PerSec\\w*, topic=
(.+)><>Count"
          name: "kafka_server_$1_$2_total"
          labels:
            topic: "$3"
    # ...
  zookeeper:
    # ...
```

3.4.11.2. Configuring Prometheus metrics

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **metrics** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    metrics:
      lowercaseOutputName: true
    # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

```
oc apply -f your-file
```

3.4.12. JVM Options

Apache Kafka and Apache Zookeeper are running inside of a Java Virtual Machine (JVM). JVM has many configuration options to optimize the performance for different platforms and architectures. AMQ Streams allows configuring some of these options.

3.4.12.1. JVM configuration

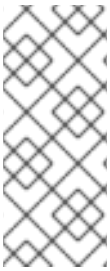
JVM options can be configured using the **jvmOptions** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**

Only a selected subset of available JVM options can be configured. The following options are supported:

-Xms and -Xmx

-Xms configures the minimum initial allocation heap size when the JVM starts. **-Xmx** configures the maximum heap size.



NOTE

The units accepted by JVM settings such as **-Xmx** and **-Xms** are those accepted by the JDK **java** binary in the corresponding image. Accordingly, **1g** or **1G** means 1,073,741,824 bytes, and **Gi** is not a valid unit suffix. This is in contrast to the units used for memory requests and limits, which follow the OpenShift convention where **1G** means 1,000,000,000 bytes, and **1Gi** means 1,073,741,824 bytes

The default values used for **-Xms** and **-Xmx** depends on whether there is a memory request limit configured for the container:

- If there is a memory limit then the JVM's minimum and maximum memory will be set to a value corresponding to the limit.
- If there is no memory limit then the JVM's minimum memory will be set to **128M** and the JVM's maximum memory will not be defined. This allows for the JVM's memory to grow as-needed, which is ideal for single node environments in test and development.

IMPORTANT

Setting **-Xmx** explicitly requires some care:

- The JVM's overall memory usage will be approximately 4 × the maximum heap, as configured by **-Xmx**.
- If **-Xmx** is set without also setting an appropriate OpenShift memory limit, it is possible that the container will be killed should the OpenShift node experience memory pressure (from other Pods running on it).
- If **-Xmx** is set without also setting an appropriate OpenShift memory request, it is possible that the container will be scheduled to a node with insufficient memory. In this case, the container will not start but crash (immediately if **-Xms** is set to **-Xmx**, or some later time if not).

When setting **-Xmx** explicitly, it is recommended to:

- set the memory request and the memory limit to the same value,
- use a memory request that is at least 4.5 × the **-Xmx**,
- consider setting **-Xms** to the same value as **-Xms**.

IMPORTANT

Containers doing lots of disk I/O (such as Kafka broker containers) will need to leave some memory available for use as operating system page cache. On such containers, the requested memory should be significantly higher than the memory used by the JVM.

Example fragment configuring -Xmx and -Xms

```
# ...
jvmOptions:
  "-Xmx": "2g"
  "-Xms": "2g"
# ...
```

In the above example, the JVM will use 2 GiB (=2,147,483,648 bytes) for its heap. Its total memory usage will be approximately 8GiB.

Setting the same value for initial (**-Xms**) and maximum (**-Xmx**) heap sizes avoids the JVM having to allocate memory after startup, at the cost of possibly allocating more heap than is really needed. For Kafka and Zookeeper pods such allocation could cause unwanted latency. For Kafka Connect avoiding

over allocation may be the most important concern, especially in distributed mode where the effects of over-allocation will be multiplied by the number of consumers.

-server

`-server` enables the server JVM. This option can be set to true or false.

Example fragment configuring `-server`

```
# ...
jvmOptions:
  "-server": true
# ...
```

NOTE

When neither of the two options (**`-server`** and **`-XX`**) is specified, the default Apache Kafka configuration of **`KAFKA_JVM_PERFORMANCE_OPTS`** will be used.

-XX

`-XX` object can be used for configuring advanced runtime options of a JVM. The **`-server`** and **`-XX`** options are used to configure the **`KAFKA_JVM_PERFORMANCE_OPTS`** option of Apache Kafka.

Example showing the use of the `-XX` object

```
jvmOptions:
  "-XX":
    "UseG1GC": true,
    "MaxGCPauseMillis": 20,
    "InitiatingHeapOccupancyPercent": 35,
    "ExplicitGCInvokesConcurrent": true,
    "UseParNewGC": false
```

The example configuration above will result in the following JVM options:

```
-XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+ExplicitGCInvokesConcurrent -XX:-UseParNewGC
```

NOTE

When neither of the two options (**`-server`** and **`-XX`**) is specified, the default Apache Kafka configuration of **`KAFKA_JVM_PERFORMANCE_OPTS`** will be used.

3.4.12.2. Configuring JVM options

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

Procedure

1. Edit the **jvmOptions** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jvmOptions:
      "-Xmx": "8g"
      "-Xms": "8g"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

```
oc apply -f your-file
```

3.4.13. Container images

AMQ Streams allows you to configure container images which will be used for its components. Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such a case, you should either copy the AMQ Streams images or build them from the source. If the configured image is not compatible with AMQ Streams images, it might not work properly.

3.4.13.1. Container image configurations

Container image which should be used for given components can be specified using the **image** property in:

- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.zookeeper.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**

The **image** specified in the component-specific custom resource will be used during deployment. If the **image** field is missing, the **image** specified in the Cluster Operator configuration will be used. If the **image** name is not defined in the Cluster Operator configuration, then the default value will be used.

- For Kafka brokers:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/kafka:latest** container image.
- For Kafka broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/kafka-stunnel:latest** container image.
- For Zookeeper nodes:
  1. Container image specified in the **STRIMZI_DEFAULT_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/zookeeper:latest** container image.
- For Zookeeper node TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/zookeeper-stunnel:latest** container image.
- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/user-operator:latest** container image.
- For Entity Operator TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/entity-operator-stunnel:latest** container image.
- For Kafka Connect:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/kafka-connect:latest** container image.

- For Kafka Connect with Source2image support:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE** environment variable from the Cluster Operator configuration.
  2. **strimzi/kafka-connect-s2i:latest** container image.

> **WARNING**
>
> Overriding container images is recommended only in special situations, where you need to use a different container registry. For example, because your network does not allow access to the container repository used by AMQ Streams. In such case, you should either copy the AMQ Streams images or build them from source. In case the configured image is not compatible with AMQ Streams images, it might not work properly.

Example of container image configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.4.13.2. Configuring container images

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **image** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
```

```
      # ...
      image: my-org/my-image:latest
      # ...
    zookeeper:
      # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

3.4.14. Configuring pod scheduling

IMPORTANT

When two application are scheduled to the same OpenShift node, both applications might use the same resources like disk I/O and impact performance. That can lead to performance degradation. Scheduling Kafka pods in a way that avoids sharing nodes with other critical workloads, using the right nodes or dedicated a set of nodes only for Kafka are the best ways how to avoid such problems.

3.4.14.1. Scheduling pods based on other applications

3.4.14.1.1. Avoid critical applications to share the node

Pod anti-affinity can be used to ensure that critical applications are never scheduled on the same disk. When running Kafka cluster, it is recommended to use pod anti-affinity to ensure that the Kafka brokers do not share the nodes with other workloads like databases.

3.4.14.1.2. Affinity

Affinity can be configured using the **`affinity`** property in following resources:

- **`Kafka.spec.kafka`**
- **`Kafka.spec.zookeeper`**
- **`Kafka.spec.entityOperator`**
- **`KafkaConnect.spec`**
- **`KafkaConnectS2I.spec`**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **`affinity`** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation.

3.4.14.1.3. Configuring pod anti-affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **affinity** property in the resource specifying the cluster deployment. Use labels to specify the pods which should not be scheduled on the same nodes. The **topologyKey** should be set to **kubernetes.io/hostname** to specify that the selected pods should not be scheduled on nodes with the same hostname. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  kafka:
    # ...
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

```
oc apply -f your-file
```

3.4.14.2. Scheduling pods to specific nodes

3.4.14.2.1. Node scheduling

The OpenShift cluster usually consists of many different types of worker nodes. Some are optimized for CPU heavy workloads, some for memory, while other might be optimized for storage (fast local SSDs) or network. Using different nodes helps to optimize both costs and performance. To achieve the best possible performance, it is important to allow scheduling of AMQ Streams components to use the right nodes.

OpenShift uses node affinity to schedule workloads onto specific nodes. Node affinity allows you to create a scheduling constraint for the node on which the pod will be scheduled. The constraint is specified as a label selector. You can specify the label using either the built-in node label like **beta.kubernetes.io/instance-type** or custom labels to select the right node.

3.4.14.2.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation.

3.4.14.2.3. Configuring node affinity in Kafka components

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Label the nodes where AMQ Streams components should be scheduled.
   On OpenShift this can be done using **oc label**:

   ```
   oc label node your-node node-type=fast-network
   ```

   Alternatively, some of the existing labels might be reused.

2. Edit the **affinity** property in the resource specifying the cluster deployment. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
               - matchExpressions:
                 - key: node-type
                   operator: In
                   values:
                   - fast-network
   ```

```
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.
   On OpenShift this can be done using **oc apply**:

```
oc apply -f your-file
```

3.4.14.3. Using dedicated nodes

3.4.14.3.1. Dedicated nodes

Cluster administrators can mark selected OpenShift nodes as tainted. Nodes with taints are excluded from regular scheduling and normal pods will not be scheduled to run on them. Only services which can tolerate the taint set on the node can be scheduled on it. The only other services running on such nodes will be system services such as log collectors or software defined networks.

Taints can be used to create dedicated nodes. Running Kafka and its components on dedicated nodes can have many advantages. There will be no other applications running on the same nodes which could cause disturbance or consume the resources needed for Kafka. That can lead to improved performance and stability.

To schedule Kafka pods on the dedicated nodes, configure node affinity and tolerations.

3.4.14.3.2. Affinity

Affinity can be configured using the **affinity** property in following resources:

- **Kafka.spec.kafka**
- **Kafka.spec.zookeeper**
- **Kafka.spec.entityOperator**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**

The affinity configuration can include different types of affinity:

- Pod affinity and anti-affinity
- Node affinity

The format of the **affinity** property follows the OpenShift specification. For more details, see the Kubernetes node and pod affinity documentation.

3.4.14.3.3. Tolerations

Tolerations ca be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka**

- **`Kafka.spec.zookeeper`**
- **`Kafka.spec.entityOperator`**
- **`KafkaConnect.spec`**
- **`KafkaConnectS2I.spec`**

The format of the **`tolerations`** property follows the OpenShift specification. For more details, see the Kubernetes taints and tolerations.

3.4.14.3.4. Setting up dedicated nodes and scheduling pods on them

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Select the nodes which should be used as dedicated
2. Make sure there are no workloads scheduled on these nodes
3. Set the taints on the selected nodes
   On OpenShift this can be done using **`oc adm taint`**:

   ```
   oc adm taint node your-node dedicated=Kafka:NoSchedule
   ```

4. Additionally, add a label to the selected nodes as well.
   On OpenShift this can be done using **`oc label`**:

   ```
   oc label node your-node dedicated=Kafka
   ```

5. Edit the **`affinity`** and **`tolerations`** properties in the resource specifying the cluster deployment. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: Kafka
   spec:
     kafka:
       # ...
       tolerations:
         - key: "dedicated"
           operator: "Equal"
           value: "Kafka"
           effect: "NoSchedule"
       affinity:
         nodeAffinity:
           requiredDuringSchedulingIgnoredDuringExecution:
             nodeSelectorTerms:
             - matchExpressions:
               - key: dedicated
   ```

```
                operator: In
                values:
                - Kafka
    # ...
  zookeeper:
    # ...
```

6. Create or update the resource.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

3.4.15. List of resources created as part of Kafka Mirror Maker

The following resources will created by the Cluster Operator in the OpenShift cluster:

***<mirror-maker-name>*-mirror-maker**

Deployment which is in charge to create the Kafka Mirror Maker pods.

***<mirror-maker-name>*-config**

ConfigMap which contains the Kafka Mirror Maker ancillary configuration and is mounted as a volume by the Kafka broker pods.

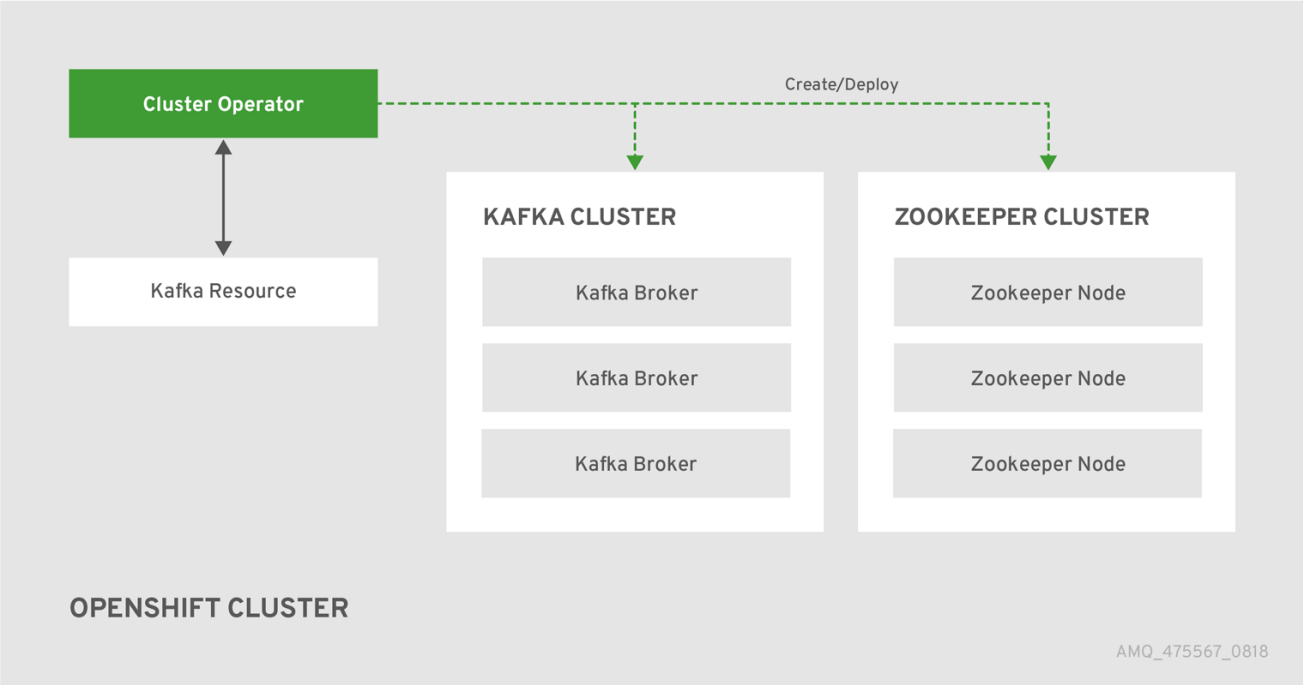
CHAPTER 4. OPERATORS

4.1. CLUSTER OPERATOR

4.1.1. Overview of the Cluster Operator component

The Cluster Operator is in charge of deploying a Kafka cluster alongside a Zookeeper ensemble. As part of the Kafka cluster, it can also deploy the topic operator which provides operator-style topic management via **`KafkaTopic`** custom resources. The Cluster Operator is also able to deploy a Kafka Connect cluster which connects to an existing Kafka cluster. On OpenShift such a cluster can be deployed using the Source2Image feature, providing an easy way of including more connectors.

Figure 4.1. Example Architecture diagram of the Cluster Operator.



When the Cluster Operator is up, it starts to *watch* for certain OpenShift resources containing the desired Kafka or Kafka Connect cluster configuration. By default, it watches only in the same namespace or project where it is installed. The Cluster Operator can be configured to watch for more OpenShift projects or Kubernetes namespaces. Cluster Operator watches the following resources:

- A **`Kafka`** resource for the Kafka cluster.
- A **`KafkaConnect`** resource for the Kafka Connect cluster.
- A **`KafkaConnectS2I`** resource for the Kafka Connect cluster with Source2Image support.

When a new **`Kafka`**, **`KafkaConnect`**, or **`KafkaConnectS2I`** resource is created in the OpenShift cluster, the operator gets the cluster description from the desired resource and starts creating a new Kafka or Kafka Connect cluster by creating the necessary other OpenShift resources, such as StatefulSets, Services, ConfigMaps, and so on.

Every time the desired resource is updated by the user, the operator performs corresponding updates on the OpenShift resources which make up the Kafka or Kafka Connect cluster. Resources are either patched or deleted and then re-created in order to make the Kafka or Kafka Connect cluster reflect the

state of the desired cluster resource. This might cause a rolling update which might lead to service disruption.

Finally, when the desired resource is deleted, the operator starts to undeploy the cluster and delete all the related OpenShift resources.

4.1.2. Deploying the Cluster Operator to OpenShift

Prerequisites

- A user with **`cluster-admin`** role needs to be used, for example, **`system:admin`**.
- Modify the installation files according to the namespace the Cluster Operator is going to be installed in.
  On Linux, use:

  ```
  sed -i 's/namespace: .*/namespace: my-project/' install/cluster-operator/*RoleBinding*.yaml
  ```

  On MacOS, use:

  ```
  sed -i '' 's/namespace: .*/namespace: my-project/' install/cluster-operator/*RoleBinding*.yaml
  ```

Procedure

1. Deploy the Cluster Operator

   ```
   oc apply -f install/cluster-operator -n _my-project_
   oc apply -f examples/templates/cluster-operator -n _my-project_
   ```

4.1.3. Deploying the Cluster Operator to watch multiple namespaces

Prerequisites

- Edit the installation files according to the OpenShift project or Kubernetes namespace the Cluster Operator is going to be installed in.
  On Linux, use:

  ```
  sed -i 's/namespace: .*/namespace: my-namespace/' install/cluster-operator/*RoleBinding*.yaml
  ```

  On MacOS, use:

  ```
  sed -i '' 's/namespace: .*/namespace: my-namespace/' install/cluster-operator/*RoleBinding*.yaml
  ```

Procedure

1. Edit the file **`install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml`** and in the environment variable **`STRIMZI_NAMESPACE`** list all the OpenShift

projects or Kubernetes namespaces where Cluster Operator should watch for resources. For example:

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
      - name: strimzi-cluster-operator
        image: strimzi/cluster-operator:latest
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: myproject,myproject2,myproject3
```

2. For all namespaces or projects which should be watched by the Cluster Operator, install the **RoleBindings**. Replace the ***my-namespace*** or ***my-project*** with the OpenShift project or Kubernetes namespace used in the previous step.
   On OpenShift this can be done using **oc apply**:

```
oc apply -f install/cluster-operator/020-RoleBinding-strimzi-
cluster-operator.yaml -n my-project
oc apply -f install/cluster-operator/031-RoleBinding-strimzi-
cluster-operator-entity-operator-delegation.yaml -n my-project
oc apply -f install/cluster-operator/032-RoleBinding-strimzi-
cluster-operator-topic-operator-delegation.yaml -n my-project
```

3. Deploy the Cluster Operator
   On OpenShift this can be done using **oc apply**:

```
oc apply -f install/cluster-operator -n my-project
```

4.1.4. Reconciliation

Although the operator reacts to all notifications about the desired cluster resources received from the OpenShift cluster, if the operator is not running, or if a notification is not received for any reason, the desired resources will get out of sync with the state of the running OpenShift cluster.

In order to handle failovers properly, a periodic reconciliation process is executed by the Cluster Operator so that it can compare the state of the desired resources with the current cluster deployments in order to have a consistent state across all of them. You can set the time interval for the periodic reconciliations using the [STRIMZI_FULL_RECONCILIATION_INTERVAL_MS] variable.

4.1.5. Cluster Operator Configuration

The Cluster Operator can be configured through the following supported environment variables:

STRIMZI_NAMESPACE

Required. A comma-separated list of namespaces that the operator should operate in. The Cluster Operator deployment might use the Kubernetes Downward API to set this automatically to the namespace the Cluster Operator is deployed in. See the example below:

```
env:
  - name: STRIMZI_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_FULL_RECONCILIATION_INTERVAL_MS

Optional, default: 120000 ms. The interval between periodic reconciliations, in milliseconds.

STRIMZI_LOG_LEVEL

Optional, default **INFO**. The level for printing logging messages. The value can be set to: **ERROR**, **WARNING**, **INFO**, **DEBUG**, and **TRACE**.

STRIMZI_OPERATION_TIMEOUT_MS

Optional, default: 300000 ms. The timeout for internal operations, in milliseconds. This value should be increased when using AMQ Streams on clusters where regular OpenShift operations take longer than usual (because of slow downloading of Docker images, for example).

STRIMZI_DEFAULT_KAFKA_IMAGE

Optional, default **strimzi/kafka:latest**. The image name to use as the default when deploying Kafka, if no image is specified as the **Kafka.spec.kafka.image** in the Section 3.1.15, “Container images”.

STRIMZI_DEFAULT_KAFKA_INIT_IMAGE

Optional, default **strimzi/kafka-init:latest**. The image name to use as default for the init container started before the broker for initial configuration work (that is, rack support), if no image is specified as the **kafka-init-image** in the Section 3.1.15, “Container images”.

STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE

Optional, default **strimzi/kafka-stunnel:latest**. The image name to use as the default when deploying the sidecar container which provides TLS support for Kafka, if no image is specified as the **Kafka.spec.kafka.tlsSidecar.image** in the Section 3.1.15, “Container images”.

STRIMZI_DEFAULT_ZOOKEEPER_IMAGE

Optional, default **strimzi/zookeeper:latest**. The image name to use as the default when deploying Zookeeper, if no image is specified as the **Kafka.spec.zookeeper.image** in the Section 3.1.15, “Container images”.

STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE

Optional, default **strimzi/zookeeper-stunnel:latest**. The image name to use as the default when deploying the sidecar container which provides TLS support for Zookeeper, if no image is specified as the **Kafka.spec.zookeeper.tlsSidecar.image** in the Section 3.1.15, “Container images”.

STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE

Optional, default **strimzi/kafka-connect:latest**. The image name to use as the default when deploying Kafka Connect, if no image is specified as the **image** in the Kafka Connect cluster ConfigMap

STRIMZI_DEFAULT_KAFKA_CONNECT_S2I_IMAGE

Optional, default **strimzi/kafka-connect-s2i:latest**. The image name to use as the default when deploying Kafka Connect S2I, if no image is specified as the **image** in the cluster ConfigMap.

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

Optional, default **strimzi/topic-operator:latest**. The image name to use as the default when deploying the topic operator, if no image is specified as the

`Kafka.spec.entityOperator.topicOperator.image` in the Section 3.1.15, “Container images” of the **`Kafka`** resource.

`STRIMZI_DEFAULT_USER_OPERATOR_IMAGE`

Optional, default **`strimzi/user-operator:latest`**. The image name to use as the default when deploying the user operator, if no image is specified as the **`Kafka.spec.entityOperator.userOperator.image`** in the Section 3.1.15, “Container images” of the **`Kafka`** resource.

`STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE`

Optional, default **`strimzi/entity-operator-stunnel:latest`**. The image name to use as the default when deploying the sidecar container which provides TLS support for the Entity Operator, if no image is specified as the **`Kafka.spec.entityOperator.tlsSidecar.image`** in the Section 3.1.15, “Container images”.

4.1.6. Role-Based Access Control (RBAC)

4.1.6.1. Provisioning Role-Based Access Control (RBAC) for the Cluster Operator

For the Cluster Operator to function it needs permission within the OpenShift cluster to interact with resources such as **`Kafka`**, **`KafkaConnect`**, and so on, as well as the managed resources, such as **`ConfigMaps`**, **`Pods`**, **`Deployments`**, **`StatefulSets`**, **`Services`**, and so on. Such permission is described in terms of OpenShift role-based access control (RBAC) resources:

- **`ServiceAccount`**,
- **`Role`** and **`ClusterRole`**,
- **`RoleBinding`** and **`ClusterRoleBinding`**.

In addition to running under its own **`ServiceAccount`** with a **`ClusterRoleBinding`**, the Cluster Operator manages some RBAC resources for the components that need access to OpenShift resources.

OpenShift also includes privilege escalation protections that prevent components operating under one **`ServiceAccount`** from granting other **`ServiceAccounts`** privileges that the granting **`ServiceAccount`** does not have. Because the Cluster Operator must be able to create the **`ClusterRoleBindings`**, and **`RoleBindings`** needed by resources it manages, the Cluster Operator must also have those same privileges.

4.1.6.2. Delegated privileges

When the Cluster Operator deploys resources for a desired **`Kafka`** resource it also creates **`ServiceAccounts`**, **`RoleBindings`**, and **`ClusterRoleBindings`**, as follows:

- The Kafka broker pods use a **`ServiceAccount`** called ***`cluster-name`*-`kafka`**
  - When the rack feature is used, the **`strimzi-`*`cluster-name`*`-kafka-init`** **`ClusterRoleBinding`** is used to grant this **`ServiceAccount`** access to the nodes within the cluster via a **`ClusterRole`** called **`strimzi-kafka-broker`**
  - When the rack feature is not used no binding is created.
- The Zookeeper pods use the default **`ServiceAccount`**, as they do not need access to the OpenShift resources.

- The Topic Operator pod uses a **`ServiceAccount`** called ***`cluster-name`*-`topic-operator`**
  - The Topic Operator produces OpenShift events with status information, so the **`ServiceAccount`** is bound to a **`ClusterRole`** called **`strimzi-topic-operator`** which grants this access via the **`strimzi-topic-operator-role-binding`** **`RoleBinding`**.

The pods for **`KafkaConnect`** and **`KafkaConnectS2I`** resources use the default **`ServiceAccount`**, as they do not require access to the OpenShift resources.

4.1.6.3. `ServiceAccount`

The Cluster Operator is best run using a **`ServiceAccount`**:

Example `ServiceAccount` for the Cluster Operator

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
```

The **`Deployment`** of the operator then needs to specify this in its **`spec.template.spec.serviceAccountName`**:

Partial example of `Deployment` for the Cluster Operator

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: strimzi-cluster-operator
    spec:
      # ...
```

Note line 12, where the the **`strimzi-cluster-operator`** **`ServiceAccount`** is specified as the **`serviceAccountName`**.

4.1.6.4. `ClusterRoles`

The Cluster Operator needs to operate using **`ClusterRoles`** that gives access to the necessary resources. Depending on the OpenShift cluster setup, a cluster administrator might be needed to create the **`ClusterRoles`**.

NOTE

Cluster administrator rights are only needed for the creation of the **ClusterRoles**. The Cluster Operator will not run under the cluster admin account.

The **ClusterRoles** follow the *principle of least privilege* and contain only those privileges needed by the Cluster Operator to operate Kafka, Kafka Connect, and Zookeeper clusters. The first set of assigned privileges allow the Cluster Operator to manage OpenShift resources such as **StatefulSets**, **Deployments**, **Pods**, and **ConfigMaps**.

Cluster Operator uses ClusterRoles to grant permission at the namespace-scoped resources level and cluster-scoped resources level:

ClusterRole with namespaced resources for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-namespaced
  labels:
    app: strimzi
rules:
- apiGroups:
  - ""
  resources:
  - serviceaccounts
  verbs:
  - get
  - create
  - delete
  - patch
  - update
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - rolebindings
  verbs:
  - get
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
```

```
  - kafka.strimzi.io
  resources:
  - kafkas
  - kafkaconnects
  - kafkaconnects2is
  - kafkamirrormakers
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - get
  - list
  - watch
  - delete
- apiGroups:
  - ""
  resources:
  - services
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - endpoints
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - extensions
  resources:
  - deployments
  - deployments/scale
  - replicasets
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
```

```
    - update
  - apiGroups:
    - apps
    resources:
    - deployments
    - deployments/scale
    - deployments/status
    - statefulsets
    verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - ""
    resources:
    - events
    verbs:
    - create
  - apiGroups:
    - extensions
    resources:
    - replicationcontrollers
    verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - apps.openshift.io
    resources:
    - deploymentconfigs
    - deploymentconfigs/scale
    - deploymentconfigs/status
    - deploymentconfigs/finalizers
    verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - build.openshift.io
    resources:
    - buildconfigs
    - builds
    verbs:
    - create
```

```
    - delete
    - get
    - list
    - patch
    - watch
    - update
  - apiGroups:
    - image.openshift.io
    resources:
    - imagestreams
    - imagestreams/status
    verbs:
    - create
    - delete
    - get
    - list
    - watch
    - patch
    - update
  - apiGroups:
    - ""
    resources:
    - replicationcontrollers
    verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - ""
    resources:
    - secrets
    verbs:
    - get
    - list
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - extensions
    resources:
    - networkpolicies
    verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
  - apiGroups:
    - route.openshift.io
```

```
  resources:
  - routes
  verbs:
  - get
  - list
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - persistentvolumeclaims
  verbs:
  - get
  - list
  - create
  - delete
  - patch
  - update
```

The second includes the permissions needed for cluster-scoped resources.

`ClusterRole` with cluster-scoped resources for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-global
  labels:
    app: strimzi
rules:
- apiGroups:
  - rbac.authorization.k8s.io
  resources:
  - clusterrolebindings
  verbs:
  - get
  - create
  - delete
  - patch
  - update
```

The **`strimzi-kafka-broker`** **`ClusterRole`** represents the access needed by the init container in Kafka pods that is used for the rack feature. As described in the Delegated privileges section, this role is also needed by the Cluster Operator in order to be able to delegate this access.

`ClusterRole` for the Cluster Operator allowing it to delegate access to OpenShift nodes to the Kafka broker pods

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-kafka-broker
  labels:
```

```
    app: strimzi
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  verbs:
  - get
```

The **`strimzi-topic-operator`** **`ClusterRole`** represents the access needed by the Topic Operator. As described in the Delegated privileges section, this role is also needed by the Cluster Operator in order to be able to delegate this access.

`ClusterRole` for the Cluster Operator allowing it to delegate access to events to the Topic Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: strimzi-entity-operator
  labels:
    app: strimzi
rules:
- apiGroups:
  - kafka.strimzi.io
  resources:
  - kafkatopics
  verbs:
  - get
  - list
  - watch
  - create
  - patch
  - update
  - delete
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - create
- apiGroups:
  - kafka.strimzi.io
  resources:
  - kafkausers
  verbs:
  - get
  - list
  - watch
  - create
  - patch
  - update
  - delete
- apiGroups:
  - ""
  resources:
```

```
  - secrets
  verbs:
  - get
  - list
  - create
  - patch
  - update
  - delete
```

4.1.6.5. `ClusterRoleBindings`

The operator needs **`ClusterRoleBindings`** and **`RoleBindings`** which associates its **`ClusterRole`** with its **`ServiceAccount`**: **`ClusterRoleBindings`** are needed for **`ClusterRoles`** containing cluster-scoped resources.

Example `ClusterRoleBinding` for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-cluster-operator-global
  apiGroup: rbac.authorization.k8s.io
```

`ClusterRoleBindings` are also needed for the **`ClusterRoles`** needed for delegation:

Examples `RoleBinding` for the Cluster Operator

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator-kafka-broker-delegation
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-broker
  apiGroup: rbac.authorization.k8s.io
```

`ClusterRoles` containing only namespaced resources are bound using **`RoleBindings`** only.

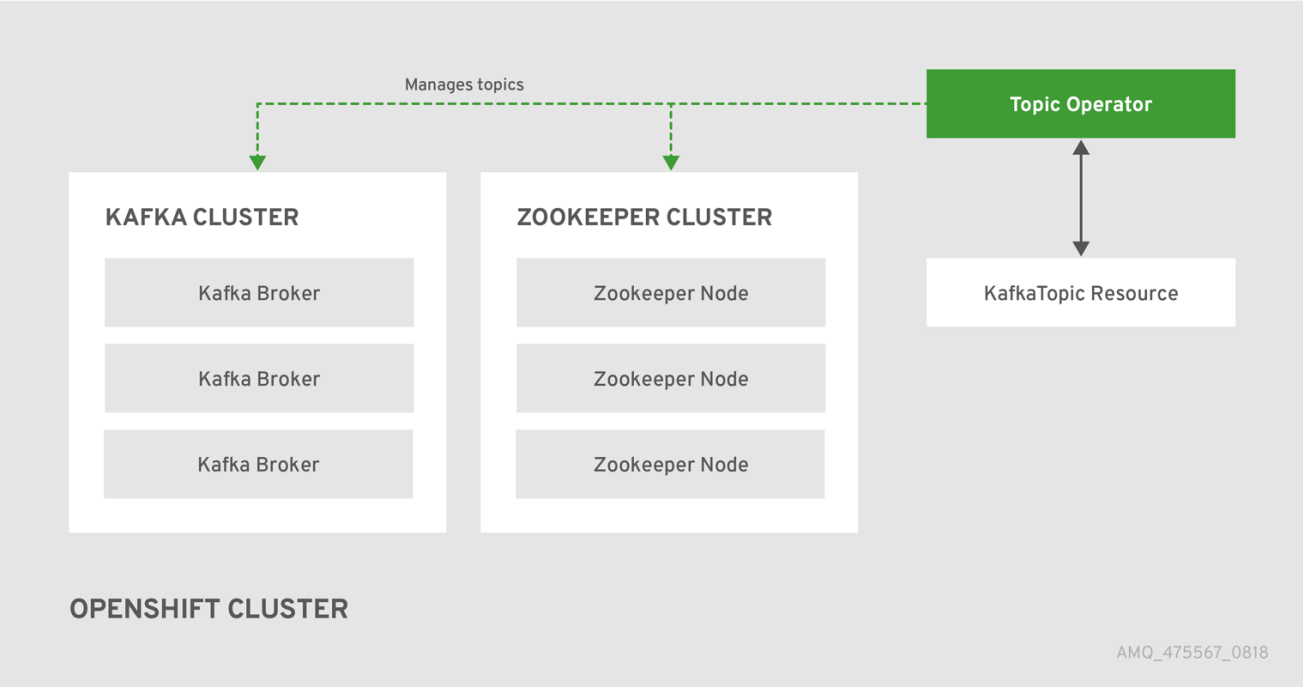
```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-cluster-operator-namespaced
  apiGroup: rbac.authorization.k8s.io
```

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-entity-operator-delegation
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-entity-operator
  apiGroup: rbac.authorization.k8s.io
```

4.2. TOPIC OPERATOR

4.2.1. Overview of the Topic Operator component

The Topic Operator provides a way of managing topics in a Kafka cluster via OpenShift resources.



The role of the Topic Operator is to keep a set of **`KafkaTopic`** OpenShift resources describing Kafka topics in-sync with corresponding Kafka topics.

Specifically:

- if a **`KafkaTopic`** is created, the operator will create the topic it describes
- if a **`KafkaTopic`** is deleted, the operator will delete the topic it describes
- if a **`KafkaTopic`** is changed, the operator will update the topic it describes

And also, in the other direction:

- if a topic is created within the Kafka cluster, the operator will create a **`KafkaTopic`** describing it
- if a topic is deleted from the Kafka cluster, the operator will create the **`KafkaTopic`** describing it
- if a topic in the Kafka cluster is changed, the operator will update the **`KafkaTopic`** describing it

This allows you to declare a **`KafkaTopic`** as part of your application's deployment and the Topic Operator will take care of creating the topic for you. Your application just needs to deal with producing or consuming from the necessary topics.

If the topic be reconfigured or reassigned to different Kafka nodes, the **`KafkaTopic`** will always be up to date.

For more details about creating, modifying and deleting topics, see Chapter 5, *Using the Topic Operator*.

4.2.2. Understanding the Topic Operator

A fundamental problem that the operator has to solve is that there is no single source of truth: Both the **`KafkaTopic`** resource and the topic within Kafka can be modified independently of the operator. Complicating this, the Topic Operator might not always be able to observe changes at each end in real time (for example, the operator might be down).

To resolve this, the operator maintains its own private copy of the information about each topic. When a change happens either in the Kafka cluster, or in OpenShift, it looks at both the state of the other system and at its private copy in order to determine what needs to change to keep everything in sync. The same thing happens whenever the operator starts, and periodically while it is running.

For example, suppose the Topic Operator is not running, and a **`KafkaTopic`** **`my-topic`** gets created. When the operator starts it will lack a private copy of "my-topic", so it can infer that the **`KafkaTopic`** has been created since it was last running. The operator will create the topic corresponding to "my-topic" and also store a private copy of the metadata for "my-topic".

The private copy allows the operator to cope with scenarios where the topic configuration gets changed both in Kafka and in OpenShift, so long as the changes are not incompatible (for example, both changing the same topic config key, but to different values). In the case of incompatible changes, the Kafka configuration wins, and the **`KafkaTopic`** will be updated to reflect that.

The private copy is held in the same ZooKeeper ensemble used by Kafka itself. This mitigates availability concerns, because if ZooKeeper is not running then Kafka itself cannot run, so the operator will be no less available than it would even if it was stateless.

4.2.3. Deploying the Topic Operator using the Cluster Operator

This procedure describes how to deploy the Topic Operator using the Cluster Operator. If you want to use the Topic Operator with a Kafka cluster that is not managed by AMQ Streams, you must deploy the Topic Operator as a standalone component. For more information, see Section 4.2.5, “Deploying the standalone Topic Operator”.

Prerequisites

- A running Cluster Operator
- A **`Kafka`** resource to be created or updated

Procedure

1. Ensure that the **`Kafka.spec.entityOperator`** object exists in the **`Kafka`** resource. This configures the Entity Operator.

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the Topic Operator using the fields described in Section B.26, “**`EntityTopicOperatorSpec`** schema reference”.

3. Create or update the Kafka resource in OpenShift.
   On OpenShift, use **`oc apply`**:

```
oc apply -f your-file
```

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Entity Operator, see Section 3.1.8, “Entity Operator”.
- For more information about the **`Kafka.spec.entityOperator`** object used to configure the Topic Operator when deployed by the Cluster Operator, see Section B.25, “**`EntityOperatorSpec`** schema reference”.

4.2.4. Configuring the Topic Operator with resource requests and limits

Prerequisites

- A running Cluster Operator

Procedure

1. Edit the **`Kafka`** resource specifying in the **`Kafka.spec.entityOperator.topicOperator.resources`** property the resource requests and limits you want the Topic Operator to have.

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: Kafka
spec:
  # kafka and zookeeper sections...
  topicOperator:
    resources:
      request:
        cpu: "1"
        memory: 500Mi
      limit:
        cpu: "1"
        memory: 500Mi
```

2. Create or update the **`Kafka`** resource.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema of the resources object, see Section B.18, “**`Resources`** schema reference”.

4.2.5. Deploying the standalone Topic Operator

Deploying the Topic Operator as a standalone component is more complicated than installing it using the Cluster Operator, but is more flexible. For instance is can operate *with* any Kafka cluster, not necessarily one deployed by the Cluster Operator.

Prerequisites

- An existing Kafka cluster for the Topic Operator to connect to.

Procedure

1. Edit the **install/topic-operator/05-Deployment-strimzi-topic-operator.yaml** resource. You will need to change the following

   a. The **STRIMZI_KAFKA_BOOTSTRAP_SERVERS** environment variable in **Deployment.spec.template.spec.containers[0].env** should be set to a list of bootstrap brokers in your Kafka cluster, given as a comma-separated list of ***hostname*:*port*** pairs.

   b. The **STRIMZI_ZOOKEEPER_CONNECT** environment variable in **Deployment.spec.template.spec.containers[0].env** should be set to a list of the Zookeeper nodes, given as a comma-separated list of ***hostname*:*port*** pairs. This should be the same Zookeeper cluster that your Kafka cluster is using.

   c. The **STRIMZI_NAMESPACE** environment variable in **Deployment.spec.template.spec.containers[0].env** should be set to the OpenShift namespace in which you want the operator to watch for **KafkaTopic** resources.

2. Deploy the Cluster Operator.
   On OpenShift this can be done using **oc apply**:

   ```
   oc apply -f install/topic-operator
   ```

3. Verify that the Topic Operator has been deployed successfully. On OpenShift this can be done using **oc describe**:

   ```
   oc describe deployment strimzi-topic-operator
   ```

   The Topic Operator is deployed once the **Replicas:** entry shows **1 available**.

   NOTE

   This could take some time if you have a slow connection to the OpenShift and the images have not been downloaded before.

Additional resources

- For more information about the environment variables used to configure the Topic Operator, see Section 4.2.6, “Topic Operator environment”.
- For more information about getting the Cluster Operator to deploy the Topic Operator for you, see Section 2.7.2, “Deploying the Topic Operator using the Cluster Operator”.

4.2.6. Topic Operator environment

When deployed standalone the Topic Operator can be configured using environment variables.

NOTE

The Topic Operator should be configured using the **`Kafka.spec.entityOperator.topicOperator`** property when deployed by the Cluster Operator.

`STRIMZI_RESOURCE_LABELS`

The label selector used to identify **`KafkaTopics`** to be managed by the operator.

`STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS`

The Zookeeper session timeout, in milliseconds. For example, **`10000`**. Default: **`20000`** (20 seconds).

`STRIMZI_KAFKA_BOOTSTRAP_SERVERS`

The list of Kafka bootstrap servers. This variable is mandatory.

`STRIMZI_ZOOKEEPER_CONNECT`

The Zookeeper connection information. This variable is mandatory.

`STRIMZI_FULL_RECONCILIATION_INTERVAL_MS`

The interval between periodic reconciliations, in milliseconds.

`STRIMZI_TOPIC_METADATA_MAX_ATTEMPTS`

The number of attempts for getting topics metadata from Kafka. The time between each attempt is defined as an exponential back-off. You might want to increase this value when topic creation could take more time due to its larger size (that is, many partitions/replicas). Default **`6`**.

`STRIMZI_LOG_LEVEL`

The level for printing logging messages. The value can be set to: **`ERROR`**, **`WARNING`**, **`INFO`**, **`DEBUG`**, and **`TRACE`**. Default **`INFO`**.

`STRIMZI_TLS_ENABLED`

For enabling the TLS support so encrypting the communication with Kafka brokers. Default **`true`**.

`STRIMZI_TRUSTSTORE_LOCATION`

The path to the truststore containing certificates for enabling TLS based communication. This variable is mandatory only if TLS is enabled through **`STRIMZI_TLS_ENABLED`**.

`STRIMZI_TRUSTSTORE_PASSWORD`

The password for accessing the truststore defined by **`STRIMZI_TRUSTSTORE_LOCATION`**. This variable is mandatory only if TLS is enabled through **`STRIMZI_TLS_ENABLED`**.

`STRIMZI_KEYSTORE_LOCATION`

The path to the keystore containing private keys for enabling TLS based communication. This variable is mandatory only if TLS is enabled through **`STRIMZI_TLS_ENABLED`**.

`STRIMZI_KEYSTORE_PASSWORD`

The password for accessing the keystore defined by **`STRIMZI_KEYSTORE_LOCATION`**. This variable is mandatory only if TLS is enabled through **`STRIMZI_TLS_ENABLED`**.

4.3. USER OPERATOR

The User Operator provides a way of managing Kafka users via OpenShift resources.

4.3.1. Overview of the User Operator component

The User Operator manages Kafka users for a Kafka cluster by watching for **KafkaUser** OpenShift resources that describe Kafka users and ensuring that they are configured properly in the Kafka cluster. For example:

- if a **KafkaUser** is created, the User Operator will create the user it describes
- if a **KafkaUser** is deleted, the User Operator will delete the user it describes
- if a **KafkaUser** is changed, the User Operator will update the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the OpenShift resources. Unlike the Kafka topics which might be created by applications directly in Kafka, it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator, so this should not be needed.

The User Operator allows you to declare a **KafkaUser** as part of your application's deployment. When the user is created, the credentials will be created in a **Secret**. Your application needs to use the user and its credentials for authentication and to produce or consume messages.

In addition to managing credentials for authentication, the User Operator also manages authorization rules by including a description of the user's rights in the **KafkaUser** declaration.

4.3.2. Deploying the User Operator using the Cluster Operator

Prerequisites

- A running Cluster Operator
- A **Kafka** resource to be created or updated.

Procedure

1. Edit the **Kafka** resource ensuring it has a **Kafka.spec.entityOperator.userOperator** object that configures the User Operator how you want.
2. Create or update the Kafka resource in OpenShift.
   On OpenShift this can be done using **oc apply**:

   ```
   oc apply -f your-file
   ```

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, "Cluster Operator".
- For more information about the **Kafka.spec.entityOperator** object used to configure the User Operator when deployed by the Cluster Operator, see **EntityOperatorSpec** schema reference.

4.3.3. Deploying the standalone User Operator

Deploying the User Operator as a standalone component is more complicated than installing it using the Cluster Operator, but is more flexible. For instance it can operate *with* any Kafka cluster, not only the one deployed by the Cluster Operator.

Prerequisites

- An existing Kafka cluster for the User Operator to connect to.

Procedure

1. Edit the **`install/user-operator/05-Deployment-strimzi-user-operator.yaml`** resource. You will need to change the following

   a. The **`STRIMZI_CA_NAME`** environment variable in **`Deployment.spec.template.spec.containers[0].env`** should be set to point to an OpenShift **`Secret`** which should contain the Certificate Authority for signing new user certificates for TLS Client Authentication. The **`Secret`** should contain the public key of the Certificate Authority under the key **`clients-ca.crt`** and the private key under **`clients-ca.key`**.

   b. The **`STRIMZI_ZOOKEEPER_CONNECT`** environment variable in **`Deployment.spec.template.spec.containers[0].env`** should be set to a list of the Zookeeper nodes, given as a comma-separated list of ***`hostname:port`*** pairs. This should be the same Zookeeper cluster that your Kafka cluster is using.

   c. The **`STRIMZI_NAMESPACE`** environment variable in **`Deployment.spec.template.spec.containers[0].env`** should be set to the OpenShift namespace in which you want the operator to watch for **`KafkaUser`** resources.

2. Deploy the Cluster Operator.
   On OpenShift this can be done using **`oc apply`**:

   ```
   oc apply -f install/user-operator
   ```

3. Verify that the User Operator has been deployed successfully. On OpenShift this can be done using **`oc describe`**:

   ```
   oc describe deployment strimzi-user-operator
   ```

   The User Operator is deployed once the **`Replicas:`** entry shows **`1 available`**.

   NOTE

   This could take some time if you have a slow connection to the OpenShift and the images have not been downloaded before.

Additional resources

- For more information about getting the Cluster Operator to deploy the User Operator for you, see Section 2.8.2, “Deploying the User Operator using the Cluster Operator”.

CHAPTER 5. USING THE TOPIC OPERATOR

5.1. TOPIC OPERATOR USAGE RECOMMENDATIONS

- Be consistent and always operate on **`KafkaTopic`** resources or always operate on topics directly. Avoid routinely using both methods for a given topic.
- When creating a **`KafkaTopic`** resource:
  - Remember that the name cannot be changed later.
  - Choose a name for the **`KafkaTopic`** resource that reflects the name of the topic it describes.
  - Ideally the **`KafkaTopic.metadata.name`** should be the same as its **`spec.topicName`**. To do this, the topic name will have to be a valid Kubernetes resource name.
- When creating a topic:
  - Remember that the name cannot be changed later.
  - It is best to use a name that is a valid Kubernetes resource name, otherwise the operator will have to modify the name when creating the corresponding **`KafkaTopic`**.

5.2. CREATING A TOPIC

This procedure describes how to create a Kafka topic using a **`KafkaTopic`** OpenShift resource.

Prerequisites

- A running Kafka cluster.
- A running Topic Operator.

Procedure

1. Prepare a file containing the **`KafkaTopic`** to be created

   An example `KafkaTopic`

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaTopic
   metadata:
     name: orders
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     partitions: 10
     replicas: 2
   ```

NOTE

It is recommended to use a topic name that is a valid OpenShift resource name. Doing this means that it is not necessary to set the **`KafkaTopic.spec.topicName`** property. In any case the **`KafkaTopic.spec.topicName`** cannot be changed after creation.

NOTE

The **`KafkaTopic.spec.partitions`** cannot be decreased.

2. Create the **`KafkaTopic`** resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema for **`KafkaTopics`**, see **`KafkaTopic`** schema reference.
- For more information about deploying a Kafka cluster using the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Topic Operator using the Cluster Operator, see Section 2.7.2, “Deploying the Topic Operator using the Cluster Operator”.
- For more information about deploying the standalone Topic Operator, see Section 4.2.5, “Deploying the standalone Topic Operator”.

5.3. CHANGING A TOPIC

This procedure describes how to change the configuration of an existing Kafka topic by using a **`KafkaTopic`** OpenShift resource.

Prerequisites

- A running Kafka cluster.
- A running Topic Operator.
- An existing **`KafkaTopic`** to be changed.

Procedure

1. Prepare a file containing the desired **`KafkaTopic`**

   An example `KafkaTopic`

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaTopic
metadata:
  name: orders
```

```
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 16
  replicas: 2
```

TIP

You can get the current version of the resource using **`oc get kafkatopic orders -o yaml`**.

NOTE

Changing topic names using the **`KafkaTopic.spec.topicName`** variable and decreasing partition size using the **`KafkaTopic.spec.partitions`** variable is not supported by Kafka.

CAUTION

Increasing **`spec.partitions`** for topics with keys will change how records are partitioned, which can be particularly problematic when the topic uses *semantic partitioning*.

2. Update the **`KafkaTopic`** resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

Additional resources

- For more information about the schema for **`KafkaTopics`**, see **`KafkaTopic`** schema reference.
- For more information about deploying a Kafka cluster, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Topic Operator using the Cluster Operator, see Section 2.7.2, “Deploying the Topic Operator using the Cluster Operator”.
- For more information about creating a topic using the Topic Operator, see Section 5.2, “Creating a topic”.

5.4. DELETING A TOPIC

This procedure describes how to delete a Kafka topic using a **`KafkaTopic`** OpenShift resource.

Prerequisites

- A running Kafka cluster.
- A running Topic Operator.
- An existing **`KafkaTopic`** to be deleted.

Procedure

Procedure

1. Delete the **`KafkaTopic`** resource in OpenShift.
   On OpenShift this can be done using **`oc`**:

   ```
   oc delete kafkatopic your-topic-name
   ```

   NOTE

   Whether the topic can actually be deleted depends on the value of the **`delete.topic.enable`** Kafka broker configuration, specified in the **`Kafka.spec.kafka.config`** property.

Additional resources

- For more information about deploying a Kafka cluster using the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about deploying the Topic Operator using the Cluster Operator, see Section 2.7.2, “Deploying the Topic Operator using the Cluster Operator”.
- For more information about creating a topic using the Topic Operator, see Section 5.2, “Creating a topic”.

CHAPTER 6. USING THE USER OPERATOR

The User Operator provides a way of managing Kafka users via OpenShift resources.

6.1. OVERVIEW OF THE USER OPERATOR COMPONENT

The User Operator manages Kafka users for a Kafka cluster by watching for **`KafkaUser`** OpenShift resources that describe Kafka users and ensuring that they are configured properly in the Kafka cluster. For example:

- if a **`KafkaUser`** is created, the User Operator will create the user it describes
- if a **`KafkaUser`** is deleted, the User Operator will delete the user it describes
- if a **`KafkaUser`** is changed, the User Operator will update the user it describes

Unlike the Topic Operator, the User Operator does not sync any changes from the Kafka cluster with the OpenShift resources. Unlike the Kafka topics which might be created by applications directly in Kafka, it is not expected that the users will be managed directly in the Kafka cluster in parallel with the User Operator, so this should not be needed.

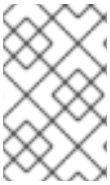
The User Operator allows you to declare a **`KafkaUser`** as part of your application's deployment. When the user is created, the credentials will be created in a **`Secret`**. Your application needs to use the user and its credentials for authentication and to produce or consume messages.

In addition to managing credentials for authentication, the User Operator also manages authorization rules by including a description of the user's rights in the **`KafkaUser`** declaration.

6.2. MUTUAL TLS AUTHENTICATION FOR CLIENTS

6.2.1. Mutual TLS authentication

Mutual authentication or two-way authentication is when both the server and the client present certificates. AMQ Streams can configure Kafka to use TLS (Transport Layer Security) to provide encrypted communication between Kafka brokers and clients either with or without mutual authentication. When you configure mutual authentication, the broker authenticates the client and the client authenticates the broker. Mutual TLS authentication is always used for the communication between Kafka brokers and Zookeeper pods.



NOTE

In many common uses of TLS (such as the HTTPS protocol used between a web browser and a web server) the authentication is not mutual: Only one party to the communication gets proof of the identity of the other party.

TLS authentication is more commonly one-way, where only one party authenticates to another. For example, when the HTTPS protocol is used between a web browser and a web server, the authentication is not usually mutual and only the server gets proof of the identity of the browser.

6.2.2. When to use mutual TLS authentication for clients

Mutual TLS authentication is recommended for authenticating Kafka clients when:

- The client supports authentication using mutual TLS authentication
- It is necessary to use the TLS certificates rather than passwords
- You can reconfigure and restart client applications periodically so that they do not use expired certificates.

6.3. CREATING A KAFKA USER WITH MUTUAL TLS AUTHENTICATION

Prerequisites

- A running Kafka cluster configured with a listener using TLS authentication.
- A running User Operator.

Procedure

1. Prepare a YAML file containing the **`KafkaUser`** to be created.

   An example `KafkaUser`

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaUser
   metadata:
     name: my-user
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     authentication:
       type: tls
     authorization:
       type: simple
       acls:
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Read
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Describe
         - resource:
             type: group
             name: my-group
             patternType: literal
           operation: Read
   ```

2. Create the **`KafkaUser`** resource in OpenShift.
   On OpenShift this can be done using **`oc apply`**:

   ```
   oc apply -f your-file
   ```

3. Use the credentials from the secret **`my-user`** in your application

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about configuring a listener that authenticates using TLS see Section 3.1.4, “Kafka broker listeners”.
- For more information about deploying the Entity Operator, see Section 3.1.8, “Entity Operator”.
- For more information about the **`KafkaUser`** object, see **`KafkaUser`** schema reference.

6.4. SCRAM-SHA AUTHENTICATION

SCRAM (Salted Challenge Response Authentication Mechanism) is an authentication protocol that can establish mutual authentication using passwords. AMQ Streams can configure Kafka to use SASL SCRMA-SHA-512 to provide authentication on both unencrypted and TLS-encrypted client connections. TLS authentication is always used internally between Kafka brokers and Zookeeper nodes. When used with a TLS client connection, the TLS protocol provides encryption, but is not used for authentication.

The following properties of SCRAM make it safe to use SCRAM-SHA even on unencrypted connections:

- The passwords are not sent in the clear over the communication channel. Instead the client and the server are each challenged by the other to offer proof that they know the password of the authenticating user.
- The server and client each generate a new challenge one each authentication exchange. This means that the exchange is resilient against replay attacks.

6.4.1. Supported SCRAM credentials

AMQ Streams supports SCRMA-SHA-512 only. When a **`KafkaUser.spec.authentication.type`** is configured with **`scram-sha-512`** the User Operator will generate a random 12 character password consisting of upper and lowercase ASCII letters and numbers.

6.4.2. When to use SCRAM-SHA authentication for clients

SCRAM-SHA is recommended for authenticating Kafka clients when:

- The client supports authentication using SCRAM-SHA-512
- It is necessary to use passwords rather than the TLS certificates
- When you want to have authentication for unencrypted communication

6.5. CREATING A KAFKA USER WITH SCRAM SHA AUTHENTICATION

Prerequisites

- A running Kafka cluster configured with a listener using SCRAM SHA authentication.
- A running User Operator.

Procedure

1. Prepare a YAML file containing the **KafkaUser** to be created.

   An example KafkaUser

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaUser
   metadata:
     name: my-user
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     authentication:
       type: scram-sha-512
     authorization:
       type: simple
       acls:
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Read
         - resource:
             type: topic
             name: my-topic
             patternType: literal
           operation: Describe
         - resource:
             type: group
             name: my-group
             patternType: literal
           operation: Read
   ```

2. Create the **KafkaUser** resource in OpenShift.
   On OpenShift this can be done using **oc apply**:

   ```
   oc apply -f your-file
   ```

3. Use the credentials from the secret **my-user** in your application

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
- For more information about configuring a listener that authenticates using SCRAM SHA see Section 3.1.4, “Kafka broker listeners”.
- For more information about deploying the Entity Operator, see Section 3.1.8, “Entity Operator”.
- For more information about the **KafkaUser** object, see **KafkaUser** schema reference.

6.6. EDITING A KAFKA USER

This procedure describes how to change the configuration of an existing Kafka user by using a **`KafkaUser`** OpenShift resource.

Prerequisites

- A running Kafka cluster.
- A running User Operator.
- An existing **`KafkaUser`** to be changed

Procedure

1. Prepare a YAML file containing the desired **`KafkaUser`**.

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Read
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Describe
      - resource:
          type: group
          name: my-group
          patternType: literal
        operation: Read
```

2. Update the **`KafkaUser`** resource in OpenShift.
   + On OpenShift this can be done using **`oc apply`**:

```
oc apply -f your-file
```

3. Use the updated credentials from the **`my-user`** secret in your application.

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.

- For more information about deploying the Entity Operator, see Section 3.1.8, “Entity Operator”.
- For more information about the **KafkaUser** object, see **KafkaUser** schema reference.

6.7. DELETING A KAFKA USER

This procedure describes how to delete a Kafka user created with **KafkaUser** OpenShift resource.

Prerequisites

- A running Kafka cluster.
- A running User Operator.
- An existing **KafkaUser** to be deleted.

Procedure

1. Delete the **KafkaUser** resource in OpenShift.
   On OpenShift this can be done using **oc**:

   ```
   oc delete kafkauser your-user-name
   ```

   - For more information about deploying the Cluster Operator, see Section 2.2, “Cluster Operator”.
   - For more information about the **KafkaUser** object, see **KafkaUser** schema reference.

6.8. KAFKA USER RESOURCE

The **KafkaUser** resource is used to declare a user with its authentication mechanism, authorization mechanism, and access rights.

6.8.1. Authentication

Authentication is configured using the **authentication** property in **KafkaUser.spec**. The authentication mechanism enabled for this user will be specified using the **type** field. Currently, the only supported authentication mechanisms are the TLS Client Authentication mechanism and the SCRAM-SHA-512 mechanism.

When no authentication mechanism is specified, User Operator will not create the user or its credentials.

6.8.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** field to **tls**.

An example of KafkaUser with enabled TLS Client Authentication

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
```

```
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  # ...
```

When the user is created by the User Operator, it will create a new secret with the same name as the **`KafkaUser`** resource. The secret will contain a public and private key which should be used for the TLS Client Authentication. Bundled with them will be the public key of the client certification authority which was used to sign the user certificate. All keys will be in X509 format.

An example of the `Secret` with user credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: # Public key of the Clients CA
  user.crt: # Public key of the user
  user.key: # Private key of the user
```

6.8.1.2. SCRAM-SHA-512 Authentication

To use SCRAM-SHA-512 authentication mechanism, set the **`type`** field to **`scram-sha-512`**.

An example of `KafkaUser` with enabled SCRAM-SHA-512 authentication

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  # ...
```

When the user is created by the User Operator, the User Operator will create a new secret with the same name as the **`KafkaUser`** resource. The secret will contain the generated password.

An example of the `Secret` with user credentials

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
```

```
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  password: # Generated password
```

6.8.2. Authorization

Authorization is configured using the **authorization** property in **KafkaUser.spec**. The authorization type enabled for this user will be specified using the **type** field. Currently, the only supported authorization type is the Simple authorization.

When no authorization is specified, the User Operator will not provision any access rights for the user.

6.8.2.1. Simple Authorization

To use Simple Authorization, set the **type** property to **simple**. Simple authorization is using the **SimpleAclAuthorizer** plugin. **SimpleAclAuthorizer** is the default authorization plugin which is part of Apache Kafka. Simple Authorization allows you to specify list of ACL rules in the **acls** property.

The **acls** property should contain a list of **AclRule** objects. **AclRule** specifies the access rights whcih will be granted to the user. The **AclRule** object contains following properties:

type

Specifies the type of the ACL rule. The type can be either **allow** or **deny**. The **type** field is optional and when not specified, the ACL rule will be treated as **allow** rule.

operation

Specifies the operation which will be allowed or denied. Following operations are supported:

- Read
- Write
- Delete
- Alter
- Describe
- All
- IdempotentWrite
- ClusterAction
- Create
- AlterConfigs
- DescribeConfigs

NOTE

Not every operation can be combined with every resource.

host

Specifies a remote host from which is the rule allowed or denied. Use `*` to allow or deny the operation from all hosts. The **host** field is optional and when not specified, the value `*` will be used as default.

resource

Specifies the resource for which does the rule apply. Simple Authorization supports 3 different resource types:

- Topics
- Consumer Groups
- Clusters
  The resource type can be specified in the **type** property. Use **topic** for Topics, **group** for Consumer Groups and **cluster** for clusters.

  Topic and Group resources additionally allow to specify the name of the resource for which the rule applies. The name can be specified in the **name** property. The name can be either specified as literal or as a prefix. To specify the name as literal, set the **patternType** property to the value **literal**. Literal names will be taken exactly as they are specified in the **name** field. To specify the name as a prefix, set the **patternType** property to the value **prefix**. Prefix type names will use the value from the **name** only a prefix and will apply the rule to all resources with names starting with the value. The cluster type resources have no name.

For more details about **SimpleAclAuthorizer**, its ACL rules and the allowed combinations of resources and operations, see Authorization and ACLs.

For more information about the **AclRule** object, see **AclRule** schema reference.

An example KafkaUser

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Read
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Describe
      - resource:
```

```
          type: group
          name: my-group
          patternType: prefix
        operation: Read
```

6.8.3. Additional resources

- For more information about the **KafkaUser** object, see **KafkaUser** schema reference.
- For more information about the TLS Client Authentication, see Section 6.2, “Mutual TLS authentication for clients”.
- For more information about the SASL SCRAM-SHA-512 authentication, see Section 6.4, “SCRAM-SHA authentication”.

CHAPTER 7. SECURITY

AMQ Streams supports encrypted communication between the Kafka and AMQ Streams components using the TLS protocol. Communication between Kafka brokers (interbroker communication), between Zookeeper nodes (internodal communication), and between these and the AMQ Streams operators is always encrypted. Communication between Kafka clients and Kafka brokers is encrypted according to how the cluster is configured. For the Kafka and AMQ Streams components, TLS certificates are also used for authentication.

The Cluster Operator automatically sets up TLS certificates to enable encryption and authentication within your cluster. It also sets up other TLS certificates if you want to enable encryption or TLS authentication between Kafka brokers and clients.

7.1. CERTIFICATE AUTHORITIES

To support encryption, each AMQ Streams component needs its own private keys and public key certificates. All component certificates are signed by a Certificate Authority (CA) called the *cluster CA*.

Similarly, each Kafka client application connecting using TLS client authentication needs private keys and certificates. The *clients CA* is used to sign the certificates for the Kafka clients.

7.1.1. CA certificates

Each CA has a self-signed public key certificate.

Kafka brokers are configured to trust certificates signed by either the clients CA or the cluster CA. Components to which clients do not need to connect, such as Zookeeper, only trust certificates signed by the cluster CA. Client applications that perform mutual TLS authentication have to trust the certificates signed by the cluster CA.

By default, AMQ Streams generates and renews CA certificates automatically. You can configure the management of CA certificates in the **`Kafka.spec.clusterCa`** and **`Kafka.spec.clientsCa`** objects.

7.2. CERTIFICATES AND SECRETS

AMQ Streams stores CA, component and Kafka client private keys and certificates in **`Secrets`**. All keys are 2048 bits in size.

CA certificate validity periods, expressed as a number of days after certificate generation, can be configured in **`Kafka.spec.clusterCa.validityDays`** and **`Kafka.spec.clusterCa.validityDays`**.

7.2.1. Cluster CA `Secrets`

Table 7.1. Cluster CA `Secrets` managed by the Cluster Operator in *<cluster>*

`Secret` name	Field within `Secret`	Description
***<cluster>*-cluster-ca**	**`ca.key`**	The current private key for the cluster CA.

Secret name	Field within Secret	Description
<*cluster*>-cluster-ca-cert	**ca.crt**	The current certificate for the cluster CA.
<*cluster*>-kafka-brokers	**<*cluster*>-kafka-<*num*>.crt**	Certificate for Kafka broker pod *<num>*. Signed by a current or former cluster CA private key in **<*cluster*>-cluster-ca**.
	<*cluster*>-kafka-<*num*>.key	Private key for Kafka broker pod *<num>*.
<*cluster*>-zookeeper-nodes	**<*cluster*>-zookeeper-<*num*>.crt**	Certificate for Zookeeper node *<num>*. Signed by a current or former cluster CA private key in **<*cluster*>-cluster-ca**.
	<*cluster*>-zookeeper-<*num*>.key	Private key for Zookeeper pod *<num>*.
<*cluster*>-entity-operator-certs	**entity-operator_.crt**	Certificate for TLS communication between the Entity Operator and Kafka or Zookeeper. Signed by a current or former cluster CA private key in **<*cluster*>-cluster-ca**.
	entity-operator.key	Private key for TLS communication between the Entity Operator and Kafka or Zookeeper

The CA certificates in **<*cluster*>-cluster-ca-cert** must be trusted by Kafka client applications so that they validate the Kafka broker certificates when connecting to Kafka brokers over TLS.

NOTE

Only **<*cluster*>-cluster-ca-cert** needs to be used by clients. All other **Secrets** in the table above only need to be accessed by the AMQ Streams components. You can enforce this using OpenShift role-based access controls if necessary.

7.2.2. Client CA Secrets

Table 7.2. Clients CA Secrets managed by the Cluster Operator in *<cluster>*

Secret name	Field within Secret	Description
<*cluster*>-clients-ca	**ca.key**	The current private key for the clients CA.

Secret name	Field within Secret	Description
<*cluster*>-clients-ca-cert	**ca.crt**	The current certificate for the clients CA.

The certificates in **<*cluster*>-clients-ca-cert** are those which the Kafka brokers trust.

NOTE

<*cluster*>-cluster-ca is used to sign certificates of client applications. It needs to be accessible to the AMQ Streams components and for administrative access if you are intending to issue application certificates without using the User Operator. You can enforce this using OpenShift role-based access controls if necessary.

7.2.3. User Secrets

Table 7.3. Secrets managed by the User Operator

Secret name	Field within Secret	Description
<*user*>	**user.crt**	Certificate for the user, signed by the clients CA
	user.key	Private key for the user

7.3. INSTALLING YOUR OWN CA CERTIFICATES

This procedure describes how to install your own CA certificates and private keys instead of using CA certificates and private keys generated by the Cluster Operator.

Prerequisites

- The Cluster Operator is running.
- A **Kafka** resource within OpenShift
- Your own X.509 certificates and keys in PEM format for the cluster CA or clients CA. For example, these could be generated by **openssl**, using a command such as:

```
openssl req -x509 -new -days <validity> --nodes -out ca.crt -keyout
ca.key
```

Procedure

1. Put your CA certificate in the corresponding **Secret** (**<*cluster*>-cluster-ca-cert** for the cluster CA or **<*cluster*>-client-ca-cert** for the clients CA):
   On OpenShift, run the following commands:

```
# Delete any existing secret (ignore "Not Exists" errors)
```

```
oc delete secret <ca-cert-secret>
# Create the new one
oc create secret generic <ca-cert-secret> \
  --from-file=ca.crt=<ca-cert-file> \
  && oc label secret <ca-cert-secret> \
  strimzi.io/kind=Kafka \
  strimzi.io/cluster=<my-cluster>
```

2. Put your CA key in the corresponding **`Secret`** (**`<cluster>-cluster-ca`** for the cluster CA or **`<cluster>-client-ca`** for the clients CA)
   On OpenShift, run the following commands:

```
# Delete the existing secret
oc delete secret <ca-key-secret>
# Create the new one
oc create secret generic <ca-key-secret> \
  --from-file=ca.key=<ca-key-file> \
  && kubectl label secret <ca-key-secret> \
  strimzi.io/kind=Kafka \
  strimzi.io/cluster=<my-cluster>
```

3. Create the **`Kafka`** resource for your cluster, configuring either the **`Kafka.spec.clusterCa`** or the **`Kafka.spec.clientsCa`** object to *not* use generated CAs:

   Example fragment `Kafka` resource configuring the cluster CA to use certificates you supply for yourself

```
kind: Kafka
version: v1alpha1
spec:
  # ...
  clusterCa:
    generateCertificateAuthority: false
```

7.4. CERTIFICATE RENEWAL

The cluster CA and clients CA certificates are only valid for a limited time period, known as the validity period. This is usually defined as a number of days since the certificate was generated. For auto-generated CA certificates, you can configure the validity period in **`Kafka.spec.clusterCa.validityDays`** and **`Kafka.spec.clientsCa.validityDays`**. The default validity period for both certificates is 365 days. Manually-installed CA certificates should have their own validity period defined.

When a CA certificate expires, components and clients which still trust that certificate will not accept TLS connections from peers whose certificate were signed by the CA private key. The components and clients need to trust the *new* CA certificate instead.

To allow the renewal of CA certificates without a loss of service, the Cluster Operator will initiate certificate renewal before the old CA certificates expire. You can configure the renewal period in **`Kafka.spec.clusterCa.renewalDays`** and **`Kafka.spec.clientsCa.renewalDays`** (both default to 30 days). The renewal period is measured backwards, from the expiry date of the current certificate.

```
Not Before                                        Not After
    |                                                 |
    |<--------------- validityDays --------------->|
                              <--- renewalDays --->|
```

The behavior of the Cluster Operator during the renewal period depends on whether the relevant setting is enabled, in either **`Kafka.spec.clusterCa.generateCertificateAuthority`** or **`Kafka.spec.clientsCa.generateCertificateAuthority`**.

7.4.1. Renewal process with generated CAs

The Cluster Operator performs the following process to renew CA certificates:

1. Generate a new CA certificate, but retaining the existing key. The new certificate replaces the old one with the name **`ca.crt`** within the corresponding **`Secret`**.
2. Generate new client certificates (for Zookeeper nodes, Kafka brokers, and the Entity Operator). This is not strictly necessary because the signing key has not changed, but it keeps the validity period of the client certificate in sync with the CA certificate.
3. Restart Zookeeper nodes so that they will trust the new CA certificate and use the new client certificates.
4. Restart Kafka brokers so that they will trust the new CA certificate and use the new client certificates.
5. Restart the Topic and User Operators so that they will trust the new CA certificate and use the new client certificates.

7.4.2. Client applications

The Cluster Operator is not aware of all the client applications using the Kafka cluster.

IMPORTANT

Depending on how your applications are configured, you might need take action to ensure they continue working after certificate renewal.

Consider the following important points to ensure that client applications continue working.

- When they connect to the cluster, client applications must trust the cluster CA certificate published in *<cluster>*-cluster-ca-cert.
- When using the User Operator to provision client certificates, client applications must use the current **`user.crt`** and **`user.key`** published in their **`<user> Secret`** when they connect to the cluster. For workloads running inside the same OpenShift cluster this can be achieved by mounting the secrets as a volume and having the client Pods construct their key- and truststores from the current state of the **`Secrets`**. For more details on this procedure, see Section 7.6, “Configuring internal clients to trust the cluster CA”.
- When renewing client certificates, if you are provisioning client certificates and keys manually, you must generate new client certificates and ensure the new certificates are used by clients within the renewal period. Failure to do this by the end of the renewal period could result in client applications being unable to connect.

7.5. TLS CONNECTIONS

7.5.1. Zookeeper communication

Zookeeper does not support TLS itself. By deploying an **stunnel** sidecar within every Zookeeper pod, the Cluster Operator is able to provide data encryption and authentication between Zookeeper nodes in a cluster. Zookeeper communicates only with the **stunnel** sidecar over the loopback interface. The **stunnel** sidecar then proxies all Zookeeper traffic, TLS decrypting data upon entry into a Zookeeper pod and TLS encrypting data upon departure from a Zookeeper pod.

This TLS encrypting **stunnel** proxy is instantiated from the **spec.zookeeper.stunnelImage** specified in the Kafka resource.

7.5.2. Kafka interbroker communication

Communication between Kafka brokers is done through the **REPLICATION** listener on port 9091, which is encrypted by default.

Communication between Kafka brokers and Zookeeper nodes uses an **stunnel** sidecar, as described above.

7.5.3. Topic and User Operators

```
Like the Cluster Operator, the Topic and User Operators each use an
`stunnel` sidecar when communicating with Zookeeper.
The Topic Operator connects to Kafka brokers on port 9091.
```

7.5.4. Kafka Client connections

Encrypted communication between Kafka brokers and clients running within the same OpenShift cluster is provided through the **CLIENTTLS** listener on port 9093.

Encrypted communication between Kafka brokers and clients running outside the same OpenShift cluster is provided through the **EXTERNAL** listener on port 9094.

NOTE

You can use the **CLIENT** listener on port 9092 for unencrypted communication with brokers.

7.6. CONFIGURING INTERNAL CLIENTS TO TRUST THE CLUSTER CA

This procedure describes how to configure a Kafka client that resides inside the OpenShift cluster — connecting to the **tls** listener on port 9093 — to trust the cluster CA certificate.

The easiest way to achieve this for an internal client is to use a volume mount to access the **Secrets** containing the necessary certificates and keys.

Prerequisites

- The Cluster Operator is running.

- A **`Kafka`** resource within the OpenShift cluster.
- A Kafka client application inside the OpenShift cluster which will connect using TLS and needs to trust the cluster CA certificate.

Procedure

1. When defining the client **`Pod`**
2. The Kafka client has to be configured to trust certificates signed by this CA. For the Java-based Kafka Producer, Consumer, and Streams APIs, you can do this by importing the CA certificate into the JVM's truststore using the following **`keytool`** command:

```
keytool -keystore client.truststore.jks -alias CARoot -import -file
ca.crt
```

3. To configure the Kafka client, specify the following properties:
   - **`security.protocol: SSL`** when using TLS for encryption (with or without TLS authentication), or **`security.protocol: SASL_SSL`** when using SCRAM-SHA authentication over TLS.
   - **`ssl.truststore.location`**: the truststore location where the certificates were imported.
   - **`ssl.truststore.password`**: the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.

Additional resources

- For the procedure for configuring external clients to trust the cluster CA, see Section 7.7, "Configuring external clients to trust the cluster CA"

7.7. CONFIGURING EXTERNAL CLIENTS TO TRUST THE CLUSTER CA

This procedure describes how to configure a Kafka client that resides outside the OpenShift cluster – connecting to the **`external`** listener on port 9094 – to trust the cluster CA certificate.

You can use the same procedure to configure clients inside OpenShift, which connect to the **`tls`** listener on port 9093, but it is usually more convenient to access the **`Secrets`** using a volume mount in the client **`Pod`**.

Follow this procedure when setting up the client and during the renewal period, when the old clients CA certificate is replaced.

IMPORTANT

The **`<cluster-name>-cluster-ca-cert Secret`** will contain more than one CA certificate during CA certificate renewal. Clients must add *all* of them to their truststores.

Prerequisites

- The Cluster Operator is running.
- A **`Kafka`** resource within the OpenShift cluster.

- A Kafka client application outside the OpenShift cluster which will connect using TLS and needs to trust the cluster CA certificate.

Procedure

1. Extract the cluster CA certificate from the generated **`<cluster-name>-cluster-ca-cert Secret`**.
   On OpenShift, run the following command to extract the certificates:

   ```
   oc extract secret/<cluster-name>-cluster-ca-cert --keys ca.crt
   ```

2. The Kafka client has to be configured to trust certificates signed by this CA. For the Java-based Kafka Producer, Consumer, and Streams APIs, you can do this by importing the CA certificates into the JVM's truststore using the following **`keytool`** command:

   ```
   keytool -keystore client.truststore.jks -alias CARoot -import -file
   ca.crt
   ```

3. To configure the Kafka client, specify the following properties:

   - **`security.protocol: SSL`** when using TLS for encryption (with or without TLS authentication), or **`security.protocol: SASL_SSL`** when using SCRAM-SHA authentication over TLS.
   - **`ssl.truststore.location`**: the truststore location where the certificates were imported.
   - **`ssl.truststore.password`**: the password for accessing the truststore. This property can be omitted if it is not needed by the truststore.

Additional resources

- For the procedure for configuring internal clients to trust the cluster CA, see Section 7.6, "Configuring internal clients to trust the cluster CA"

APPENDIX A. FREQUENTLY ASKED QUESTIONS

A.1. CLUSTER OPERATOR

A.1.1. Log contains warnings about failing to acquire lock

For each cluster, the Cluster Operator always executes only one operation at a time. The Cluster Operator uses locks to make sure that there are never two parallel operations running for the same cluster. In case an operation requires more time to complete, other operations will wait until it is completed and the lock is released.

INFO

Examples of cluster operations are *cluster creation*, *rolling update*, *scale down* or *scale up* and so on.

If the wait for the lock takes too long, the operation times out and the following warning message will be printed to the log:

```
2018-03-04 17:09:24 WARNING AbstractClusterOperations:290 - Failed to
acquire lock for kafka cluster lock::kafka::myproject::my-cluster
```

Depending on the exact configuration of **`STRIMZI_FULL_RECONCILIATION_INTERVAL_MS`** and **`STRIMZI_OPERATION_TIMEOUT_MS`**, this warning message may appear regularly without indicating any problems. The operations which time out will be picked up by the next periodic reconciliation. It will try to acquire the lock again and execute.

Should this message appear periodically even in situations when there should be no other operations running for a given cluster, it might indicate that due to some error the lock was not properly released. In such cases it is recommended to restart the cluster operator.

APPENDIX B. CUSTOM RESOURCE API REFERENCE

B.1. KAFKA SCHEMA REFERENCE

Field	Description
spec	The specification of the Kafka and Zookeeper clusters, and Topic Operator.
`KafkaSpec`	

B.2. KAFKASPEC SCHEMA REFERENCE

Used in: `Kafka`

Field	Description
kafka	Configuration of the Kafka cluster.
`KafkaClusterSpec`	
zookeeper	Configuration of the Zookeeper cluster.
`ZookeeperClusterSpec`	
topicOperator	Configuration of the Topic Operator.
`TopicOperatorSpec`	
entityOperator	Configuration of the Entity Operator.
`EntityOperatorSpec`	
clusterCa	Configuration of the cluster certificate authority.
`CertificateAuthority`	
clientsCa	Configuration of the clients certificate authority.
`CertificateAuthority`	
maintenanceTimeWindows	A list of time windows for the maintenance tasks (that is, certificates renewal). Each time window is defined by a cron expression.
string array	

B.3. KAFKACLUSTERSPEC SCHEMA REFERENCE

Used in: **`KafkaSpec`**

Field	Description
replicas integer	The number of pods in the cluster.
image string	The docker image for the pods.
storage **`EphemeralStorage`**, **`PersistentClaimStorage`**	Storage configuration (disk). Cannot be updated. The type depends on the value of the **`storage.type`** property within the given object, which must be one of [ephemeral, persistent-claim].
listeners **`KafkaListeners`**	Configures listeners of Kafka brokers.
authorization **`KafkaAuthorizationSimple`**	Authorization configuration for Kafka brokers. The type depends on the value of the **`authorization.type`** property within the given object, which must be one of [simple].
config map	The kafka broker config. Properties with the following prefixes cannot be set: listeners, advertised., broker., listener., host.name, port, inter.broker.listener.name, sasl., ssl., security., password., principal.builder.class, log.dir, zookeeper.connect, zookeeper.set.acl, authorizer., super.user.
rack **`Rack`**	Configuration of the **`broker.rack`** broker config.
brokerRackInitImage string	The image of the init container used for initializing the **`broker.rack`**.
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.

Field	Description
livenessProbe	Pod liveness checking.
`Probe`	
readinessProbe	Pod readiness checking.
`Probe`	
jvmOptions	JVM Options for pods.
`JvmOptions`	
resources	Resource constraints (limits and requests).
`Resources`	
metrics	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
map	
logging	Logging configuration for Kafka. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].
`InlineLogging`, **`ExternalLogging`**	
tlsSidecar	TLS sidecar configuration.
`TlsSidecar`	

B.4. `EPHEMERALSTORAGE` SCHEMA REFERENCE

Used in: **`KafkaClusterSpec`**, **`ZookeeperClusterSpec`**

The **`type`** property is a discriminator that distinguishes the use of the type **`EphemeralStorage`** from **`PersistentClaimStorage`**. It must have the value **`ephemeral`** for the type **`EphemeralStorage`**.

Field	Description
type	Must be **`ephemeral`**.
string	

B.5. `PERSISTENTCLAIMSTORAGE` SCHEMA REFERENCE

Used in: `KafkaClusterSpec`, `ZookeeperClusterSpec`

The `type` property is a discriminator that distinguishes the use of the type `PersistentClaimStorage` from `EphemeralStorage`. It must have the value `persistent-claim` for the type `PersistentClaimStorage`.

Field	Description
type string	Must be `persistent-claim`.
size string	When type=persistent-claim, defines the size of the persistent volume claim (i.e 1Gi). Mandatory when type=persistent-claim.
selector map	Specifies a specific persistent volume to use. It contains a matchLabels field which defines an inner JSON object with key:value representing labels for selecting such a volume.
deleteClaim boolean	Specifies if the persistent volume claim has to be deleted when the cluster is un-deployed.
class string	The storage class to use for dynamic volume allocation.

B.6. KAFKALISTENERS SCHEMA REFERENCE

Used in: `KafkaClusterSpec`

Field	Description
plain `KafkaListenerPlain`	Configures plain listener on port 9092.
tls `KafkaListenerTls`	Configures TLS listener on port 9093.
external	Configures external listener on port 9094. The type depends on the value of the `external.type` property within the given object, which must be one of [route, loadbalancer, nodeport].

Field	Description
`KafkaListenerExternalRoute`, `KafkaListenerExternalLoadBalancer`, `KafkaListenerExternalNodePort`	

B.7. `KAFKALISTENERPLAIN` SCHEMA REFERENCE

Used in: `KafkaListeners`

Field	Description
authentication	Authentication configuration for this listener. Since this listener does not use TLS transport you cannot configure an authentication with **`type: tls`**. The type depends on the value of the **`authentication.type`** property within the given object, which must be one of [tls, scram-sha-512].
`KafkaListenerAuthenticationTls`, `KafkaListenerAuthenticationScramSha512`	

B.8. `KAFKALISTENERAUTHENTICATIONTLS` SCHEMA REFERENCE

Used in: `KafkaListenerExternalLoadBalancer`, `KafkaListenerExternalNodePort`, `KafkaListenerExternalRoute`, `KafkaListenerPlain`, `KafkaListenerTls`

The **`type`** property is a discriminator that distinguishes the use of the type **`KafkaListenerAuthenticationTls`** from `KafkaListenerAuthenticationScramSha512`. It must have the value **`tls`** for the type **`KafkaListenerAuthenticationTls`**.

Field	Description
type	Must be **`tls`**.
string	

B.9. `KAFKALISTENERAUTHENTICATIONSCRAMSHA512` SCHEMA REFERENCE

Used in: `KafkaListenerExternalLoadBalancer`, `KafkaListenerExternalNodePort`, `KafkaListenerExternalRoute`, `KafkaListenerPlain`, `KafkaListenerTls`

The **`type`** property is a discriminator that distinguishes the use of the type **`KafkaListenerAuthenticationScramSha512`** from `KafkaListenerAuthenticationTls`. It must have the value **`scram-sha-512`** for the type **`KafkaListenerAuthenticationScramSha512`**.

Field	Description
type	Must be **`scram-sha-512`**.

Field	Description
string	

B.10. KAFKALISTENERTLS SCHEMA REFERENCE

Used in: **KafkaListeners**

Field	Description
authentication	Authentication configuration for this listener. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
KafkaListenerAuthenticationTls, **KafkaListenerAuthenticationScramSha512**	

B.11. KAFKALISTENEREXTERNALROUTE SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalRoute** from **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**. It must have the value **route** for the type **KafkaListenerExternalRoute**.

Field	Description
type	Must be **route**.
string	
authentication	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
KafkaListenerAuthenticationTls, **KafkaListenerAuthenticationScramSha512**	

B.12. KAFKALISTENEREXTERNALLOADBALANCER SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalLoadBalancer** from **KafkaListenerExternalRoute**, **KafkaListenerExternalNodePort**. It must have the value **loadbalancer** for the type **KafkaListenerExternalLoadBalancer**.

Field	Description
type	Must be **`loadbalancer`**.
string	
authentication	Authentication configuration for Kafka brokers. The type depends on the value of the **`authentication.type`** property within the given object, which must be one of [tls, scram-sha-512].
`KafkaListenerAuthenticationTls`, **`KafkaListenerAuthenticationScramSha512`**	
tls	Enables TLS encryption on the listener. By default set to **`true`** for enabled TLS encryption.
boolean	

B.13. `KAFKALISTENEREXTERNALNODEPORT` SCHEMA REFERENCE

Used in: **`KafkaListeners`**

The **`type`** property is a discriminator that distinguishes the use of the type **`KafkaListenerExternalNodePort`** from **`KafkaListenerExternalRoute`**, **`KafkaListenerExternalLoadBalancer`**. It must have the value **`nodeport`** for the type **`KafkaListenerExternalNodePort`**.

Field	Description
type	Must be **`nodeport`**.
string	
authentication	Authentication configuration for Kafka brokers. The type depends on the value of the **`authentication.type`** property within the given object, which must be one of [tls, scram-sha-512].
`KafkaListenerAuthenticationTls`, **`KafkaListenerAuthenticationScramSha512`**	
tls	Enables TLS encryption on the listener. By default set to **`true`** for enabled TLS encryption.
boolean	

B.14. `KAFKAAUTHORIZATIONSIMPLE` SCHEMA REFERENCE

Used in: **`KafkaClusterSpec`**

The **type** property is a discriminator that distinguishes the use of the type **KafkaAuthorizationSimple** from other subtypes which may be added in the future. It must have the value **simple** for the type **KafkaAuthorizationSimple**.

Field	Description
type string	Must be **simple**.
superUsers string array	List of super users. Should contain list of user principals which should get unlimited access rights.

B.15. RACK SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Field	Description
topologyKey string	A key that matches labels assigned to the OpenShift or Kubernetes cluster nodes. The value of the label is used to set the broker's **broker.rack** config.

B.16. PROBE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **ZookeeperClusterSpec**

Field	Description
initialDelaySeconds integer	The initial delay before first the health is first checked.
timeoutSeconds integer	The timeout for each attempted health check.

B.17. JVMOPTIONS SCHEMA REFERENCE

Used in: **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMakerSpec**, **ZookeeperClusterSpec**

Field	Description
-XX	A map of -XX options to the JVM.
map	
-Xms	-Xms option to to the JVM.
string	
-Xmx	-Xmx option to to the JVM.
string	

B.18. RESOURCES SCHEMA REFERENCE

Used in: `EntityTopicOperatorSpec`, `EntityUserOperatorSpec`, `KafkaClusterSpec`, `KafkaConnectS2ISpec`, `KafkaConnectSpec`, `KafkaMirrorMakerSpec`, `TlsSidecar`, `TopicOperatorSpec`, `ZookeeperClusterSpec`

Field	Description
limits	Resource limits applied at runtime.
`CpuMemory`	
requests	Resource requests applied during pod scheduling.
`CpuMemory`	

B.19. CPUMEMORY SCHEMA REFERENCE

Used in: `Resources`

Field	Description
cpu	CPU.
string	
memory	Memory.
string	

B.20. INLINELOGGING SCHEMA REFERENCE

Used in: **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMakerSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **InlineLogging** from **ExternalLogging**. It must have the value **inline** for the type **InlineLogging**.

Field	Description
type string	Must be **inline**.
loggers map	A Map from logger name to logger level.

B.21. EXTERNALLOGGING SCHEMA REFERENCE

Used in: **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMakerSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **ExternalLogging** from **InlineLogging**. It must have the value **external** for the type **ExternalLogging**.

Field	Description
type string	Must be **external**.
name string	The name of the **ConfigMap** from which to get the logging configuration.

B.22. TLSSIDECAR SCHEMA REFERENCE

Used in: **EntityOperatorSpec**, **KafkaClusterSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Field	Description
image	The docker image for the container.

Field	Description
string	
logLevel string (one of [emerg, debug, crit, err, alert, warning, notice, info])	The log level for the TLS sidecar.Default value is **`notice`**.
resources **`Resources`**	Resource constraints (limits and requests).

B.23. ZOOKEEPERCLUSTERSPEC SCHEMA REFERENCE

Used in: **`KafkaSpec`**

Field	Description
replicas integer	The number of pods in the cluster.
image string	The docker image for the pods.
storage **`EphemeralStorage`**, **`PersistentClaimStorage`**	Storage configuration (disk). Cannot be updated. The type depends on the value of the **`storage.type`** property within the given object, which must be one of [ephemeral, persistent-claim].
config map	The zookeeper broker config. Properties with the following prefixes cannot be set: server., dataDir, dataLogDir, clientPort, authProvider, quorum.auth, requireClientAuthScheme.
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.
livenessProbe	Pod liveness checking.

Field	Description
`Probe`	
readinessProbe	Pod readiness checking.
`Probe`	
jvmOptions	JVM Options for pods.
`JvmOptions`	
resources	Resource constraints (limits and requests).
`Resources`	
metrics	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
map	
logging	Logging configuration for Zookeeper. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].
`InlineLogging`, **`ExternalLogging`**	
tlsSidecar	TLS sidecar configuration.
`TlsSidecar`	

B.24. `TOPICOPERATORSPEC` SCHEMA REFERENCE

Used in: **`KafkaSpec`**

Field	Description
watchedNamespace	The namespace the Topic Operator should watch.
string	
image	The image to use for the Topic Operator.
string	
reconciliationIntervalSeconds	Interval between periodic reconciliations.
integer	

Field	Description
zookeeperSessionTimeoutSeconds integer	Timeout for the Zookeeper session.
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
resources **`Resources`**	Resource constraints (limits and requests).
topicMetadataMaxAttempts integer	The number of attempts at getting topic metadata.
tlsSidecar **`TlsSidecar`**	TLS sidecar configuration.
logging **`InlineLogging`**, **`ExternalLogging`**	Logging configuration. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].

B.25. `ENTITYOPERATORSPEC` SCHEMA REFERENCE

Used in: **`KafkaSpec`**

Field	Description
topicOperator **`EntityTopicOperatorSpec`**	Configuration of the Topic Operator.
userOperator **`EntityUserOperatorSpec`**	Configuration of the User Operator.
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations	Pod's tolerations.See external documentation of core/v1 tolerations.

Field	Description
Toleration array	
tlsSidecar	TLS sidecar configuration.
`TlsSidecar`	

B.26. `ENTITYTOPICOPERATORSPEC` SCHEMA REFERENCE

Used in: **`EntityOperatorSpec`**

Field	Description
watchedNamespace	The namespace the Topic Operator should watch.
string	
image	The image to use for the Topic Operator.
string	
reconciliationIntervalSeconds	Interval between periodic reconciliations.
integer	
zookeeperSessionTimeoutSeconds	Timeout for the Zookeeper session.
integer	
resources	Resource constraints (limits and requests).
`Resources`	
topicMetadataMaxAttempts	The number of attempts at getting topic metadata.
integer	
logging	Logging configuration. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].
`InlineLogging`, **`ExternalLogging`**	

B.27. `ENTITYUSEROPERATORSPEC` SCHEMA REFERENCE

Used in: **`EntityOperatorSpec`**

Field	Description
watchedNamespace string	The namespace the User Operator should watch.
image string	The image to use for the User Operator.
reconciliationIntervalSeconds integer	Interval between periodic reconciliations.
zookeeperSessionTimeoutSeconds integer	Timeout for the Zookeeper session.
resources **`Resources`**	Resource constraints (limits and requests).
logging **`InlineLogging`**, **`ExternalLogging`**	Logging configuration. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].

B.28. `CERTIFICATEAUTHORITY` SCHEMA REFERENCE

Used in: **`KafkaSpec`**

Configuration of how TLS certificates are used within the cluster.This applies to certificates used for both internal communication within the cluster and to certificates used for client access via **`Kafka.spec.kafka.listeners.tls`**.

Field	Description
generateCertificateAuthority boolean	If true then Certificate Authority certificates will be generated automatically. Otherwise the user will need to provide a Secret with the CA certificate. Default is true.
validityDays integer	The number of days generated certificates should be valid for. Default is 365.

Field	Description
renewalDays integer	The number of days in the certificate renewal period. This is the number of days before the a certificate expires during which renewal actions may be performed.When **`generateCertificateAuthority`** is true, this will cause the generation of a new certificate. When **`generateCertificateAuthority`** is true, this will cause extra logging at WARN level about the pending certificate expiry. Default is 30.

B.29. KAFKACONNECT SCHEMA REFERENCE

Field	Description
spec **`KafkaConnectSpec`**	The specification of the Kafka Connect deployment.

B.30. KAFKACONNECTSPEC SCHEMA REFERENCE

Used in: **`KafkaConnect`**

Field	Description
replicas integer	The number of pods in the Kafka Connect group.
image string	The docker image for the pods.
livenessProbe **`Probe`**	Pod liveness checking.
readinessProbe **`Probe`**	Pod readiness checking.
jvmOptions **`JvmOptions`**	JVM Options for pods.

Field	Description
affinity Affinity	Pod affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.
logging **`InlineLogging`**, **`ExternalLogging`**	Logging configuration for Kafka Connect. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
authentication **`KafkaConnectAuthenticationTls`**, **`KafkaConnectAuthenticationScramSha512`**	Authentication configuration for Kafka Connect. The type depends on the value of the **`authentication.type`** property within the given object, which must be one of [tls, scram-sha-512].
bootstrapServers string	Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>:<port>* pairs.
config map	The Kafka Connect configuration. Properties with the following prefixes cannot be set: ssl., sasl., security., listeners, plugin.path, rest., bootstrap.servers.
resources **`Resources`**	Resource constraints (limits and requests).
tls **`KafkaConnectTls`**	TLS configuration.

B.31. `KAFKACONNECTAUTHENTICATIONTLS` SCHEMA REFERENCE

Used in: **`KafkaConnectS2ISpec`**, **`KafkaConnectSpec`**

The **type** property is a discriminator that distinguishes the use of the type **KafkaConnectAuthenticationTls** from **KafkaConnectAuthenticationScramSha512**. It must have the value **tls** for the type **KafkaConnectAuthenticationTls**.

Field	Description
certificateAndKey	Certificate and private key pair for TLS authentication.
CertAndKeySecretSource	
type	Must be **tls**.
string	

B.32. CERTANDKEYSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaConnectAuthenticationTls**, **KafkaMirrorMakerAuthenticationTls**

Field	Description
certificate	The name of the file certificate in the Secret.
string	
key	The name of the private key in the Secret.
string	
secretName	The name of the Secret containing the certificate.
string	

B.33. KAFKACONNECTAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaConnectAuthenticationScramSha512** from **KafkaConnectAuthenticationTls**. It must have the value **scram-sha-512** for the type **KafkaConnectAuthenticationScramSha512**.

Field	Description
passwordSecret	Password used for the authentication.
PasswordSecretSource	

Field	Description
type string	Must be `scram-sha-512`.
username string	Username used for the authentication.

B.34. PASSWORDSECRETSOURCE SCHEMA REFERENCE

Used in: `KafkaConnectAuthenticationScramSha512`, `KafkaMirrorMakerAuthenticationScramSha512`

Field	Description
password string	The name of the key in the Secret under which the password is stored.
secretName string	The name of the Secret containing the password.

B.35. KAFKACONNECTTLS SCHEMA REFERENCE

Used in: `KafkaConnectS2ISpec`, `KafkaConnectSpec`

Field	Description
trustedCertificates `CertSecretSource` array	Trusted certificates for TLS connection.

B.36. CERTSECRETSOURCE SCHEMA REFERENCE

Used in: `KafkaConnectTls`, `KafkaMirrorMakerTls`

Field	Description
certificate	The name of the file certificate in the Secret.

Field	Description
string	
secretName	The name of the Secret containing the certificate.
string	

B.37. KAFKACONNECTS2I SCHEMA REFERENCE

Field	Description
spec	The specification of the Kafka Connect deployment.
KafkaConnectS2ISpec	

B.38. KAFKACONNECTS2ISPEC SCHEMA REFERENCE

Used in: **KafkaConnectS2I**

Field	Description
replicas	The number of pods in the Kafka Connect group.
integer	
image	The docker image for the pods.
string	
livenessProbe	Pod liveness checking.
Probe	
readinessProbe	Pod readiness checking.
Probe	
jvmOptions	JVM Options for pods.
JvmOptions	
affinity	Pod affinity rules.See external documentation of core/v1 affinity.

Field	Description
Affinity	
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
authentication **`KafkaConnectAuthenticationTls`**, **`KafkaConnectAuthenticationScramSha512`**	Authentication configuration for Kafka Connect. The type depends on the value of the **`authentication.type`** property within the given object, which must be one of [tls, scram-sha-512].
bootstrapServers string	Bootstrap servers to connect to. This should be given as a comma separated list of *<hostname>:<port>* pairs.
config map	The Kafka Connect configuration. Properties with the following prefixes cannot be set: ssl., sasl., security., listeners, plugin.path, rest., bootstrap.servers.
insecureSourceRepository boolean	When true this configures the source repository with the 'Local' reference policy and an import policy that accepts insecure source tags.
logging **`InlineLogging`**, **`ExternalLogging`**	Logging configuration for Kafka Connect. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].
resources **`Resources`**	Resource constraints (limits and requests).
tls **`KafkaConnectTls`**	TLS configuration.
tolerations Toleration array	Pod's tolerations.See external documentation of core/v1 tolerations.

B.39. `KAFKATOPIC` SCHEMA REFERENCE

Field	Description
spec	The specification of the topic.
`KafkaTopicSpec`	

B.40. KAFKATOPICSPEC SCHEMA REFERENCE

Used in: `KafkaTopic`

Field	Description
partitions	The number of partitions the topic should have. This cannot be decreased after topic creation. It can be increased after topic creation, but it is important to understand the consequences that has, especially for topics with semantic partitioning. If unspecified this will default to the broker's **`num.partitions`** config.
integer	
replicas	The number of replicas the topic should have. If unspecified this will default to the broker's **`default.replication.factor`** config.
integer	
config	The topic configuration.
map	
topicName	The name of the topic. When absent this will default to the metadata.name of the topic. It is recommended to not set this unless the topic name is not a valid Kubernetes resource name.
string	

B.41. KAFKAUSER SCHEMA REFERENCE

Field	Description
spec	The specification of the user.
`KafkaUserSpec`	

B.42. KAFKAUSERSPEC SCHEMA REFERENCE

Used in: `KafkaUser`

Field	Description
authentication	Authentication mechanism enabled for this Kafka user. The type depends on the value of the **`authentication.type`** property within the given object, which must be one of [tls, scram-sha-512].
`KafkaUserTlsClientAuthentication`, **`KafkaUserScramSha512ClientAuthentication`**	
authorization	Authorization rules for this Kafka user. The type depends on the value of the **`authorization.type`** property within the given object, which must be one of [simple].
`KafkaUserAuthorizationSimple`	

B.43. `KAFKAUSERTLSCLIENTAUTHENTICATION` SCHEMA REFERENCE

Used in: **`KafkaUserSpec`**

The **`type`** property is a discriminator that distinguishes the use of the type **`KafkaUserTlsClientAuthentication`** from **`KafkaUserScramSha512ClientAuthentication`**. It must have the value **`tls`** for the type **`KafkaUserTlsClientAuthentication`**.

Field	Description
type	Must be **`tls`**.
string	

B.44. `KAFKAUSERSCRAMSHA512CLIENTAUTHENTICATION` SCHEMA REFERENCE

Used in: **`KafkaUserSpec`**

The **`type`** property is a discriminator that distinguishes the use of the type **`KafkaUserScramSha512ClientAuthentication`** from **`KafkaUserTlsClientAuthentication`**. It must have the value **`scram-sha-512`** for the type **`KafkaUserScramSha512ClientAuthentication`**.

Field	Description
type	Must be **`scram-sha-512`**.
string	

B.45. `KAFKAUSERAUTHORIZATIONSIMPLE` SCHEMA REFERENCE

Used in: **`KafkaUserSpec`**

The **type** property is a discriminator that distinguishes the use of the type **KafkaUserAuthorizationSimple** from other subtypes which may be added in the future. It must have the value **simple** for the type **KafkaUserAuthorizationSimple**.

Field	Description
type string	Must be **simple**.
acls **AclRule** array	List of ACL rules which should be applied to this user.

B.46. ACLRULE SCHEMA REFERENCE

Used in: **KafkaUserAuthorizationSimple**

Field	Description
host string	The host from which the action described in the ACL rule is allowed or denied.
operation string (one of [Read, Write, Delete, Alter, Describe, All, IdempotentWrite, ClusterAction, Create, AlterConfigs, DescribeConfigs])	Operation which will be allowed or denied. Supported operations are: Read, Write, Create, Delete, Alter, Describe, ClusterAction, AlterConfigs, DescribeConfigs, IdempotentWrite and All.
resource **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleClusterResource**	Indicates the resource for which given ACL rule applies. The type depends on the value of the **resource.type** property within the given object, which must be one of [topic, group, cluster].
type string (one of [allow, deny])	The type of the rule.Currently the only supported type is **allow**.ACL rules with type **allow** are used to allow user to execute the specified operations. Default value is **allow**.

B.47. ACLRULETOPICRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleTopicResource** from **AclRuleGroupResource**, **AclRuleClusterResource**. It must have the value **topic** for the type **AclRuleTopicResource**.

Field	Description
type string	Must be **`topic`**.
name string	Name of resource for which given ACL rule applies. Can be combined with **`patternType`** field to use prefix pattern.
patternType string (one of [prefix, literal])	Describes the pattern used in the resource field. The supported types are **`literal`** and **`prefix`**. With **`literal`** pattern type, the resource field will be used as a definition of a full topic name. With **`prefix`** pattern type, the resource name will be used only as a prefix. Default value is **`literal`**.

B.48. ACLRULEGROUPRESOURCE SCHEMA REFERENCE

Used in: **`AclRule`**

The **`type`** property is a discriminator that distinguishes the use of the type **`AclRuleGroupResource`** from **`AclRuleTopicResource`**, **`AclRuleClusterResource`**. It must have the value **`group`** for the type **`AclRuleGroupResource`**.

Field	Description
type string	Must be **`group`**.
name string	Name of resource for which given ACL rule applies. Can be combined with **`patternType`** field to use prefix pattern.
patternType string (one of [prefix, literal])	Describes the pattern used in the resource field. The supported types are **`literal`** and **`prefix`**. With **`literal`** pattern type, the resource field will be used as a definition of a full topic name. With **`prefix`** pattern type, the resource name will be used only as a prefix. Default value is **`literal`**.

B.49. ACLRULECLUSTERRESOURCE SCHEMA REFERENCE

Used in: **`AclRule`**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleClusterResource** from **AclRuleTopicResource**, **AclRuleGroupResource**. It must have the value **cluster** for the type **AclRuleClusterResource**.

Field	Description
type string	Must be **cluster**.

B.50. KAFKAMIRRORMAKER SCHEMA REFERENCE

Field	Description
spec **KafkaMirrorMakerSpec**	The specification of the mirror maker.

B.51. KAFKAMIRRORMAKERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker**

Field	Description
replicas integer	The number of pods in the **Deployment**.
image string	The docker image for the pods.
whitelist string	List of topics which are included for mirroring. This option allows any regular expression using Java-style regular expressions.Mirroring two topics named A and B can be achieved by using the whitelist **'A|B'**. Or, as a special case, you can mirror all topics using the whitelist '*'. Multiple regular expressions separated by commas can be specified as well.
consumer **KafkaMirrorMakerConsumerSpec**	Configuration of source cluster.
producer	Configuration of target cluster.

Field	Description
`KafkaMirrorMakerProducerSpec`	
resources	Resource constraints (limits and requests).
`Resources`	
affinity	Pod affinity rules.See external documentation of core/v1 affinity.
Affinity	
tolerations	Pod's tolerations.See external documentation of core/v1 tolerations.
Toleration array	
jvmOptions	JVM Options for pods.
`JvmOptions`	
logging	Logging configuration for Mirror Maker. The type depends on the value of the **`logging.type`** property within the given object, which must be one of [inline, external].
`InlineLogging`, **`ExternalLogging`**	
metrics	The Prometheus JMX Exporter configuration. See JMX Exporter documentation for details of the structure of this configuration.
map	

B.52. `KAFKAMIRRORMAKERCONSUMERSPEC` SCHEMA REFERENCE

Used in: **`KafkaMirrorMakerSpec`**

Field	Description
numStreams	Specifies the number of consumer stream threads to create.
integer	
groupId	A unique string that identifies the consumer group this consumer belongs to.
string	
bootstrapServers	A list of host:port pairs to use for establishing the initial connection to the Kafka cluster.
string	

Field	Description
authentication	Authentication configuration for connecting to the cluster. The type depends on the value of the **`authentication.type`** property within the given object, which must be one of [tls, scram-sha-512].
`KafkaMirrorMakerAuthenticationTls`, **`KafkaMirrorMakerAuthenticationScramSha512`**	
config	The mirror maker consumer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, group.id, sasl., security.
map	
tls	TLS configuration for connecting to the cluster.
`KafkaMirrorMakerTls`	

B.53. `KAFKAMIRRORMAKERAUTHENTICATIONTLS` SCHEMA REFERENCE

Used in: **`KafkaMirrorMakerConsumerSpec`**, **`KafkaMirrorMakerProducerSpec`**

The **`type`** property is a discriminator that distinguishes the use of the type **`KafkaMirrorMakerAuthenticationTls`** from **`KafkaMirrorMakerAuthenticationScramSha512`**. It must have the value **`tls`** for the type **`KafkaMirrorMakerAuthenticationTls`**.

Field	Description
certificateAndKey	Reference to the **`Secret`** which holds the certificate and private key pair.
`CertAndKeySecretSource`	
type	Must be **`tls`**.
string	

B.54. `KAFKAMIRRORMAKERAUTHENTICATIONSCRAMSHA512` SCHEMA REFERENCE

Used in: **`KafkaMirrorMakerConsumerSpec`**, **`KafkaMirrorMakerProducerSpec`**

The **`type`** property is a discriminator that distinguishes the use of the type **`KafkaMirrorMakerAuthenticationScramSha512`** from **`KafkaMirrorMakerAuthenticationTls`**. It must have the value **`scram-sha-512`** for the type **`KafkaMirrorMakerAuthenticationScramSha512`**.

Field	Description
passwordSecret **PasswordSecretSource**	Reference to the **Secret** which holds the password.
type string	Must be **scram-sha-512**.
username string	Username used for the authentication.

B.55. KAFKAMIRRORMAKERTLS SCHEMA REFERENCE

Used in: **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Field	Description
trustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection.

B.56. KAFKAMIRRORMAKERPRODUCERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Field	Description
bootstrapServers string	A list of host:port pairs to use for establishing the initial connection to the Kafka cluster.
authentication **KafkaMirrorMakerAuthenticationTls**, **KafkaMirrorMakerAuthenticationScramSha512**	Authentication configuration for connecting to the cluster. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
config map	The mirror maker producer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, sasl., security.
tls	TLS configuration for connecting to the cluster.

Field	Description
`KafkaMirrorMakerTls`	

APPENDIX C. 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 **Red Hat AMQ Streams** entries in the **JBOSS INTEGRATION AND AUTOMATION** category.
3. Select the desired AMQ Streams product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

Registering Your System for Packages

To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using zip or tar files, this step is not required.

1. Go to access.redhat.com.
2. Navigate to **Registration Assistant**.
3. Select your OS version and continue to the next page.
4. Use the listed command in your system terminal to complete the registration.

To learn more see How to Register and Subscribe a System to the Red Hat Customer Portal.

Revised on 2018-10-25 17:45:54 UTC