Red Hat AMQ 7.4

Using AMQ Streams on OpenShift Container Platform

For Use with AMQ Streams 1.2

Last Updated: 2019-07-15

Red Hat AMQ 7.4 Using AMQ Streams on OpenShift Container Platform

For Use with AMQ Streams 1.2

Legal Notice

Copyright © 2019 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

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 15
1.1. KAFKA KEY FEATURES 15
1.2. DOCUMENT CONVENTIONS 16

CHAPTER 2. GETTING STARTED WITH AMQ STREAMS 17
2.1. INSTALLING AMQ STREAMS AND DEPLOYING COMPONENTS 17
2.2. CUSTOM RESOURCES 17
2.2.1. AMQ Streams custom resource example 18
2.2.2. AMQ Streams custom resource status 20
2.3. CLUSTER OPERATOR 21
2.3.1. Overview of the Cluster Operator component 21
2.3.2. Deploying the Cluster Operator to OpenShift 22
2.3.3. Deploying the Cluster Operator to watch multiple namespaces 23
2.3.4. Deploying the Cluster Operator to watch all namespaces 24
2.4. KAFKA CLUSTER 25
2.4.1. Deploying the Kafka cluster to OpenShift 25
2.5. KAFKA CONNECT 26
2.5.1. Deploying Kafka Connect to your OpenShift cluster 27
2.5.2. Extending Kafka Connect with connector plug-ins 27
2.5.2.1. Creating a Docker image from the Kafka Connect base image 27
2.5.2.2. Creating a container image using OpenShift builds and Source-to-Image 28
2.6. KAFKA MIRROR MAKER 29
2.6.1. Deploying Kafka Mirror Maker to OpenShift 30
2.7. KAFKA BRIDGE 30
2.7.1. Deploying Kafka Bridge to your OpenShift cluster 30
2.8. DEPLOYING EXAMPLE CLIENTS 31
2.9. TOPIC OPERATOR 31
2.9.1. Overview of the Topic Operator component 31
2.9.2. Deploying the Topic Operator using the Cluster Operator 32
2.10. USER OPERATOR 33
2.10.1. Overview of the User Operator component 33
2.10.2. Deploying the User Operator using the Cluster Operator 34
2.11. STRIMZI ADMINISTRATORS 34
2.11.1. Designating Strimzi Administrators 34
2.12. CONTAINER IMAGES 35

CHAPTER 3. DEPLOYMENT CONFIGURATION 37
3.1. KAFKA CLUSTER CONFIGURATION 37
3.1.1. Data storage considerations 37
3.1.1.1. Apache Kafka and Zookeeper storage 37
3.1.1.2. File systems 37
3.1.2. Kafka and Zookeeper storage types 38
3.1.2.1. Ephemeral storage 38
3.1.2.1.1. Log directories 39
3.1.2.2. Persistent storage 39
3.1.2.2.1. Storage class overrides 40
3.1.2.2.2. Persistent Volume Claim naming 41
3.1.2.2.3. Log directories 41
3.1.2.3. Resizing persistent volumes 42
3.1.2.4. JBOD storage overview 43
3.1.2.4.1. JBOD configuration 43

3.1.2.4.2. JBOD and Persistent Volume Claims 43
3.1.2.4.3. Log directories 43
3.1.2.5. Adding volumes to JBOD storage 44
3.1.2.6. Removing volumes from JBOD storage 45
3.1.3. Kafka broker replicas 46
3.1.3.1. Configuring the number of broker nodes 46
3.1.4. Kafka broker configuration 47
3.1.4.1. Kafka broker configuration 47
3.1.4.2. Configuring Kafka brokers 49
3.1.5. Kafka broker listeners 49
3.1.5.1. Mutual TLS authentication for clients 50
3.1.5.1.1. Mutual TLS authentication 50
3.1.5.1.2. When to use mutual TLS authentication for clients 50
3.1.5.2. SCRAM-SHA authentication 50
3.1.5.2.1. Supported SCRAM credentials 50
3.1.5.2.2. When to use SCRAM-SHA authentication for clients 51
3.1.5.3. Kafka listeners 51
3.1.5.3.1. External listener 51
3.1.5.3.1.1. Exposing Kafka using OpenShift Routes 51
3.1.5.3.1.2. Exposing Kafka using loadbalancers 52
3.1.5.3.1.3. Exposing Kafka using node ports 52
3.1.5.3.1.4. Customizing advertised addresses on external listeners 53
3.1.5.3.1.5. Customizing DNS names of external listeners 54
3.1.5.3.2. Listener authentication 55
3.1.5.3.3. Network policies 56
3.1.5.4. Configuring Kafka listeners 56
3.1.5.5. Accessing Kafka using OpenShift routes 57
3.1.5.6. Accessing Kafka using loadbalancers 58
3.1.5.7. Accessing Kafka using node ports 59
3.1.5.8. Restricting access to Kafka listeners using networkPolicyPeers 61
3.1.6. Authentication and Authorization 61
3.1.6.1. Authentication 61
3.1.6.1.1. TLS client authentication 62
3.1.6.2. Configuring authentication in Kafka brokers 62
3.1.6.3. Authorization 63
3.1.6.3.1. Simple authorization 63
3.1.6.4. Configuring authorization in Kafka brokers 63
3.1.7. Zookeeper replicas 64
3.1.7.1. Number of Zookeeper nodes 64
3.1.7.2. Changing the number of Zookeeper replicas 65
3.1.8. Zookeeper configuration 65
3.1.8.1. Zookeeper configuration 66
3.1.8.2. Configuring Zookeeper 67
3.1.9. Zookeeper connection 67
3.1.9.1. Connecting to Zookeeper from a terminal 68
3.1.10. Entity Operator 68
3.1.10.1. Configuration 68
3.1.10.1.1. Topic Operator 69
3.1.10.1.2. User Operator 70
3.1.10.2. Configuring Entity Operator 71
3.1.11. CPU and memory resources 72
3.1.11.1. Resource limits and requests 72
3.1.11.1.1. Resource requests 72

3.1.11.1.2. Resource limits 73
3.1.11.1.3. Supported CPU formats 73
3.1.11.1.4. Supported memory formats 74
3.1.11.2. Configuring resource requests and limits 74
3.1.12. Logging 75
3.1.12.1. Kafka loggers 75
3.1.12.2. Specifying inline logging 76
3.1.12.3. Specifying an external ConfigMap for logging 77
3.1.13. Kafka rack awareness 77
3.1.13.1. Configuring rack awareness in Kafka brokers 77
3.1.14. Healthchecks 78
3.1.14.1. Healthcheck configurations 79
3.1.14.2. Configuring healthchecks 79
3.1.15. Prometheus metrics 80
3.1.15.1. Metrics configuration 80
3.1.15.2. Configuring Prometheus metrics 81
3.1.16. JVM Options 82
3.1.16.1. JVM configuration 82
3.1.16.1.1. Garbage collector logging 84
3.1.16.2. Configuring JVM options 84
3.1.17. Container images 85
3.1.17.1. Container image configurations 85
3.1.17.1.1. Configuring the Kafka.spec.kafka.image property 86
3.1.17.1.2. Configuring the image property in other resources 86
3.1.17.2. Configuring container images 88
3.1.18. TLS sidecar 88
3.1.18.1. TLS sidecar configuration 89
3.1.18.2. Configuring TLS sidecar 90
3.1.19. Configuring pod scheduling 91
3.1.19.1. Scheduling pods based on other applications 91
3.1.19.1.1. Avoid critical applications to share the node 91
3.1.19.1.2. Affinity 91
3.1.19.1.3. Configuring pod anti-affinity in Kafka components 91
3.1.19.2. Scheduling pods to specific nodes 92
3.1.19.2.1. Node scheduling 92
3.1.19.2.2. Affinity 92
3.1.19.2.3. Configuring node affinity in Kafka components 93
3.1.19.3. Using dedicated nodes 94
3.1.19.3.1. Dedicated nodes 94
3.1.19.3.2. Affinity 94
3.1.19.3.3. Tolerations 94
3.1.19.3.4. Setting up dedicated nodes and scheduling pods on them 95
3.1.20. Performing a rolling update of a Kafka cluster 96
3.1.21. Performing a rolling update of a Zookeeper cluster 96
3.1.22. Scaling clusters 97
3.1.22.1. Scaling Kafka clusters 97
3.1.22.1.1. Adding brokers to a cluster 97
3.1.22.1.2. Removing brokers from a cluster 97
3.1.22.2. Partition reassignment 98
3.1.22.2.1. Reassignment JSON file 98
3.1.22.2.2. Reassigning partitions between JBOD volumes 99
3.1.22.3. Generating reassignment JSON files 100
3.1.22.4. Creating reassignment JSON files manually 101

3.1.22.5. Reassignment throttles 101
3.1.22.6. Scaling up a Kafka cluster 101
3.1.22.7. Scaling down a Kafka cluster 103
3.1.23. Deleting Kafka nodes manually 105
3.1.24. Deleting Zookeeper nodes manually 106
3.1.25. Maintenance time windows for rolling updates 106
3.1.25.1. Maintenance time windows overview 107
3.1.25.2. Maintenance time window definition 107
3.1.25.3. Configuring a maintenance time window 107
3.1.26. List of resources created as part of Kafka cluster 108
3.2. KAFKA CONNECT CLUSTER CONFIGURATION 110
3.2.1. Replicas 110
3.2.1.1. Configuring the number of nodes 111
3.2.2. Bootstrap servers 111
3.2.2.1. Configuring bootstrap servers 111
3.2.3. Connecting to Kafka brokers using TLS 112
3.2.3.1. TLS support in Kafka Connect 112
3.2.3.2. Configuring TLS in Kafka Connect 113
3.2.4. Connecting to Kafka brokers with Authentication 114
3.2.4.1. Authentication support in Kafka Connect 114
3.2.4.1.1. TLS Client Authentication 114
3.2.4.1.2. SASL based SCRAM-SHA-512 authentication 114
3.2.4.1.3. SASL based PLAIN authentication 115
3.2.4.2. Configuring TLS client authentication in Kafka Connect 116
3.2.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect 117
3.2.5. Kafka Connect configuration 118
3.2.5.1. Kafka Connect configuration 118
3.2.5.2. Configuring Kafka Connect 119
3.2.6. CPU and memory resources 120
3.2.6.1. Resource limits and requests 120
3.2.6.1.1. Resource requests 121
3.2.6.1.2. Resource limits 121
3.2.6.1.3. Supported CPU formats 122
3.2.6.1.4. Supported memory formats 122
3.2.6.2. Configuring resource requests and limits 123
3.2.7. Logging 123
3.2.7.1. Kafka Connect loggers 124
3.2.7.2. Specifying inline logging 124
3.2.7.3. Specifying an external ConfigMap for logging 124
3.2.8. Healthchecks 125
3.2.8.1. Healthcheck configurations 125
3.2.8.2. Configuring healthchecks 126
3.2.9. Prometheus metrics 127
3.2.9.1. Metrics configuration 127
3.2.9.2. Configuring Prometheus metrics 128
3.2.10. JVM Options 128
3.2.10.1. JVM configuration 128
3.2.10.1.1. Garbage collector logging 131
3.2.10.2. Configuring JVM options 131
3.2.11. Container images 132
3.2.11.1. Container image configurations 132
3.2.11.1.1. Configuring the Kafka.spec.kafka.image property 132
3.2.11.1.2. Configuring the image property in other resources 133

3.2.11.2. Configuring container images 134
3.2.12. Configuring pod scheduling 135
3.2.12.1. Scheduling pods based on other applications 135
3.2.12.1.1. Avoid critical applications to share the node 135
3.2.12.1.2. Affinity 135
3.2.12.1.3. Configuring pod anti-affinity in Kafka components 136
3.2.12.2. Scheduling pods to specific nodes 136
3.2.12.2.1. Node scheduling 137
3.2.12.2.2. Affinity 137
3.2.12.2.3. Configuring node affinity in Kafka components 137
3.2.12.3. Using dedicated nodes 138
3.2.12.3.1. Dedicated nodes 138
3.2.12.3.2. Affinity 138
3.2.12.3.3. Tolerations 139
3.2.12.3.4. Setting up dedicated nodes and scheduling pods on them 139
3.2.13. Using external configuration and secrets 140
3.2.13.1. Storing connector configurations externally 140
3.2.13.1.1. External configuration as environment variables 140
3.2.13.1.2. External configuration as volumes 141
3.2.13.2. Mounting Secrets as environment variables 142
3.2.13.3. Mounting Secrets as volumes 143
3.2.14. List of resources created as part of Kafka Connect cluster 144
3.3. KAFKA CONNECT CLUSTER WITH SOURCE2IMAGE SUPPORT 145
3.3.1. Replicas 145
3.3.1.1. Configuring the number of nodes 145
3.3.2. Bootstrap servers 146
3.3.2.1. Configuring bootstrap servers 146
3.3.3. Connecting to Kafka brokers using TLS 146
3.3.3.1. TLS support in Kafka Connect 146
3.3.3.2. Configuring TLS in Kafka Connect 147
3.3.4. Connecting to Kafka brokers with Authentication 148
3.3.4.1. Authentication support in Kafka Connect 148
3.3.4.1.1. TLS Client Authentication 148
3.3.4.1.2. SASL based SCRAM-SHA-512 authentication 149
3.3.4.1.3. SASL based PLAIN authentication 150
3.3.4.2. Configuring TLS client authentication in Kafka Connect 150
3.3.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Connect 151
3.3.5. Kafka Connect configuration 152
3.3.5.1. Kafka Connect configuration 152
3.3.5.2. Configuring Kafka Connect 154
3.3.6. CPU and memory resources 155
3.3.6.1. Resource limits and requests 155
3.3.6.1.1. Resource requests 155
3.3.6.1.2. Resource limits 156
3.3.6.1.3. Supported CPU formats 156
3.3.6.1.4. Supported memory formats 157
3.3.6.2. Configuring resource requests and limits 157
3.3.7. Logging 158
3.3.7.1. Kafka Connect with Source2Image loggers 158
3.3.7.2. Specifying inline logging 158
3.3.7.3. Specifying an external ConfigMap for logging 159
3.3.8. Healthchecks 159
3.3.8.1. Healthcheck configurations 159

3.3.8.2. Configuring healthchecks 160
3.3.9. Prometheus metrics 161
3.3.9.1. Metrics configuration 161
3.3.9.2. Configuring Prometheus metrics 162
3.3.10. JVM Options 163
3.3.10.1. JVM configuration 163
3.3.10.1.1. Garbage collector logging 165
3.3.10.2. Configuring JVM options 165
3.3.11. Container images 166
3.3.11.1. Container image configurations 166
3.3.11.1.1. Configuring the Kafka.spec.kafka.image property 167
3.3.11.1.2. Configuring the image property in other resources 167
3.3.11.2. Configuring container images 169
3.3.12. Configuring pod scheduling 169
3.3.12.1. Scheduling pods based on other applications 170
3.3.12.1.1. Avoid critical applications to share the node 170
3.3.12.1.2. Affinity 170
3.3.12.1.3. Configuring pod anti-affinity in Kafka components 170
3.3.12.2. Scheduling pods to specific nodes 171
3.3.12.2.1. Node scheduling 171
3.3.12.2.2. Affinity 171
3.3.12.2.3. Configuring node affinity in Kafka components 172
3.3.12.3. Using dedicated nodes 172
3.3.12.3.1. Dedicated nodes 172
3.3.12.3.2. Affinity 173
3.3.12.3.3. Tolerations 173
3.3.12.3.4. Setting up dedicated nodes and scheduling pods on them 173
3.3.13. Using external configuration and secrets 175
3.3.13.1. Storing connector configurations externally 175
3.3.13.1.1. External configuration as environment variables 175
3.3.13.1.2. External configuration as volumes 176
3.3.13.2. Mounting Secrets as environment variables 176
3.3.13.3. Mounting Secrets as volumes 178
3.3.14. List of resources created as part of Kafka Connect cluster with Source2Image support 179
3.3.15. Creating a container image using OpenShift builds and Source-to-Image 179
3.4. KAFKA MIRROR MAKER CONFIGURATION 181
3.4.1. Replicas 181
3.4.1.1. Configuring the number of replicas 181
3.4.2. Bootstrap servers 182
3.4.2.1. Configuring bootstrap servers 182
3.4.3. Whitelist 182
3.4.3.1. Configuring the topics whitelist 183
3.4.4. Consumer group identifier 183
3.4.4.1. Configuring the consumer group identifier 183
3.4.5. Number of consumer streams 184
3.4.5.1. Configuring the number of consumer streams 184
3.4.6. Connecting to Kafka brokers using TLS 185
3.4.6.1. TLS support in Kafka Mirror Maker 185
3.4.6.2. Configuring TLS encryption in Kafka Mirror Maker 186
3.4.7. Connecting to Kafka brokers with Authentication 187
3.4.7.1. Authentication support in Kafka Mirror Maker 187
3.4.7.1.1. TLS Client Authentication 187
3.4.7.1.2. SCRAM-SHA-512 authentication 188

3.4.7.1.3. PLAIN authentication 189
3.4.7.2. Configuring TLS client authentication in Kafka Mirror Maker 189
3.4.7.3. Configuring SCRAM-SHA-512 authentication in Kafka Mirror Maker 190
3.4.8. Kafka Mirror Maker configuration 192
3.4.8.1. Kafka Mirror Maker configuration 192
3.4.8.2. Configuring Kafka Mirror Maker 193
3.4.9. CPU and memory resources 193
3.4.9.1. Resource limits and requests 194
3.4.9.1.1. Resource requests 194
3.4.9.1.2. Resource limits 195
3.4.9.1.3. Supported CPU formats 195
3.4.9.1.4. Supported memory formats 196
3.4.9.2. Configuring resource requests and limits 196
3.4.10. Logging 197
3.4.10.1. Kafka Mirror Maker loggers 197
3.4.10.2. Specifying inline logging 197
3.4.10.3. Specifying an external ConfigMap for logging 198
3.4.11. Prometheus metrics 198
3.4.11.1. Metrics configuration 198
3.4.11.2. Configuring Prometheus metrics 199
3.4.12. JVM Options 200
3.4.12.1. JVM configuration 200
3.4.12.1.1. Garbage collector logging 202
3.4.12.2. Configuring JVM options 202
3.4.13. Container images 203
3.4.13.1. Container image configurations 203
3.4.13.1.1. Configuring the Kafka.spec.kafka.image property 204
3.4.13.1.2. Configuring the image property in other resources 204
3.4.13.2. Configuring container images 206
3.4.14. Configuring pod scheduling 207
3.4.14.1. Scheduling pods based on other applications 207
3.4.14.1.1. Avoid critical applications to share the node 207
3.4.14.1.2. Affinity 207
3.4.14.1.3. Configuring pod anti-affinity in Kafka components 207
3.4.14.2. Scheduling pods to specific nodes 208
3.4.14.2.1. Node scheduling 208
3.4.14.2.2. Affinity 208
3.4.14.2.3. Configuring node affinity in Kafka components 209
3.4.14.3. Using dedicated nodes 210
3.4.14.3.1. Dedicated nodes 210
3.4.14.3.2. Affinity 210
3.4.14.3.3. Tolerations 210
3.4.14.3.4. Setting up dedicated nodes and scheduling pods on them 211
3.4.15. List of resources created as part of Kafka Mirror Maker 212
3.5. KAFKA BRIDGE CLUSTER CONFIGURATION 212
3.5.1. Replicas 212
3.5.1.1. Configuring the number of nodes 213
3.5.2. Bootstrap servers 213
3.5.2.1. Configuring bootstrap servers 213
3.5.3. Connecting to Kafka brokers using TLS 214
3.5.3.1. TLS support for Kafka connection to the Kafka Bridge 214
3.5.3.2. Configuring TLS in Kafka Bridge 215
3.5.4. Connecting to Kafka brokers with Authentication 215

3.5.4.1. Authentication support in Kafka Bridge 216
3.5.4.1.1. TLS Client Authentication 216
3.5.4.1.2. SCRAM-SHA-512 authentication 216
3.5.4.1.3. SASL-based PLAIN authentication 217
3.5.4.2. Configuring TLS client authentication in Kafka Bridge 218
3.5.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Bridge 219
3.5.5. Kafka Bridge configuration 219
3.5.5.1. Kafka Bridge Consumer configuration 220
3.5.5.2. Kafka Bridge Producer configuration 221
3.5.5.3. Kafka Bridge HTTP configuration 221
3.5.5.4. Configuring Kafka Bridge 222
3.5.6. Healthchecks 223
3.5.6.1. Healthcheck configurations 223
3.5.6.2. Configuring healthchecks 224
3.5.7. Container images 224
3.5.7.1. Container image configurations 225
3.5.7.1.1. Configuring the Kafka.spec.kafka.image property 225
3.5.7.1.2. Configuring the image property in other resources 226
3.5.7.2. Configuring container images 227
3.5.8. Configuring pod scheduling 228
3.5.8.1. Scheduling pods based on other applications 228
3.5.8.1.1. Avoid critical applications to share the node 228
3.5.8.1.2. Affinity 228
3.5.8.1.3. Configuring pod anti-affinity in Kafka components 229
3.5.8.2. Scheduling pods to specific nodes 229
3.5.8.2.1. Node scheduling 230
3.5.8.2.2. Affinity 230
3.5.8.2.3. Configuring node affinity in Kafka components 230
3.5.8.3. Using dedicated nodes 231
3.5.8.3.1. Dedicated nodes 231
3.5.8.3.2. Affinity 231
3.5.8.3.3. Tolerations 232
3.5.8.3.4. Setting up dedicated nodes and scheduling pods on them 232
3.5.9. List of resources created as part of Kafka Bridge cluster 233
3.6. CUSTOMIZING DEPLOYMENTS 233
3.6.1. Template properties 234
3.6.2. Labels and Annotations 236
3.6.3. Customizing Pods 236
3.6.4. Customizing the image pull policy 238
3.6.5. Customizing Pod Disruption Budgets 238
3.6.6. Customizing deployments 239

CHAPTER 4. OPERATORS 240
4.1. CLUSTER OPERATOR 240
4.1.1. Overview of the Cluster Operator component 240
4.1.2. Deploying the Cluster Operator to OpenShift 241
4.1.3. Deploying the Cluster Operator to watch multiple namespaces 241
4.1.4. Deploying the Cluster Operator to watch all namespaces 242
4.1.5. Reconciliation 243
4.1.6. Cluster Operator Configuration 243
4.1.7. Role-Based Access Control (RBAC) 245
4.1.7.1. Provisioning Role-Based Access Control (RBAC) for the Cluster Operator 245
4.1.7.2. Delegated privileges 246

4.1.7.3. ServiceAccount 246
4.1.7.4. ClusterRoles 247
4.1.7.5. ClusterRoleBindings 254
4.2. TOPIC OPERATOR 255
4.2.1. Overview of the Topic Operator component 255
4.2.2. Understanding the Topic Operator 256
4.2.3. Deploying the Topic Operator using the Cluster Operator 257
4.2.4. Configuring the Topic Operator with resource requests and limits 258
4.2.5. Deploying the standalone Topic Operator 258
4.2.6. Topic Operator environment 259
4.3. USER OPERATOR 260
4.3.1. Overview of the User Operator component 260
4.3.2. Deploying the User Operator using the Cluster Operator 261
4.3.3. Configuring the User Operator with resource requests and limits 261
4.3.4. Deploying the standalone User Operator 262

CHAPTER 5. USING THE TOPIC OPERATOR 264
5.1. TOPIC OPERATOR USAGE RECOMMENDATIONS 264
5.2. CREATING A TOPIC 264
5.3. CHANGING A TOPIC 265
5.4. DELETING A TOPIC 266

CHAPTER 6. USING THE USER OPERATOR 268
6.1. OVERVIEW OF THE USER OPERATOR COMPONENT 268
6.2. MUTUAL TLS AUTHENTICATION FOR CLIENTS 268
6.2.1. Mutual TLS authentication 268
6.2.2. When to use mutual TLS authentication for clients 268
6.3. CREATING A KAFKA USER WITH MUTUAL TLS AUTHENTICATION 269
6.4. SCRAM-SHA AUTHENTICATION 270
6.4.1. Supported SCRAM credentials 270
6.4.2. When to use SCRAM-SHA authentication for clients 270
6.5. CREATING A KAFKA USER WITH SCRAM SHA AUTHENTICATION 270
6.6. EDITING A KAFKA USER 271
6.7. DELETING A KAFKA USER 272
6.8. KAFKA USER RESOURCE 273
6.8.1. Authentication 273
6.8.1.1. TLS Client Authentication 273
6.8.1.2. SCRAM-SHA-512 Authentication 274
6.8.2. Authorization 275
6.8.2.1. Simple Authorization 275
6.8.3. Additional resources 277

CHAPTER 7. USING THE AMQ STREAMS KAFKA BRIDGE 278
7.1. OVERVIEW OF THE AMQ STREAMS KAFKA BRIDGE 278
7.2. SUPPORTED CLIENTS FOR THE AMQ STREAMS KAFKA BRIDGE 278
7.3. SECURING THE AMQ STREAMS KAFKA BRIDGE 279
7.4. ACCESSING THE AMQ STREAMS KAFKA BRIDGE FROM OUTSIDE OF OPENSHIFT 279
7.5. REQUESTS TO THE AMQ STREAMS KAFKA BRIDGE 280
7.5.1. Data formats and headers 280
7.5.1.1. Content Type headers 280
7.5.1.2. Embedded data format 280
7.5.1.3. Accept headers 281
7.6. AMQ STREAMS KAFKA BRIDGE API RESOURCES 281

CHAPTER 8. SECURITY 282
8.1. CERTIFICATE AUTHORITIES 282
8.1.1. CA certificates 282
8.2. CERTIFICATES AND SECRETS 282
8.2.1. Cluster CA Secrets 282
8.2.2. Client CA Secrets 283
8.2.3. User Secrets 284
8.3. INSTALLING YOUR OWN CA CERTIFICATES 284
8.4. CERTIFICATE RENEWAL 285
8.4.1. Renewal process with generated CAs 286
8.4.2. Client applications 286
8.5. TLS CONNECTIONS 286
8.5.1. Zookeeper communication 286
8.5.2. Kafka interbroker communication 287
8.5.3. Topic and User Operators 287
8.5.4. Kafka Client connections 287
8.6. CONFIGURING INTERNAL CLIENTS TO TRUST THE CLUSTER CA 287
8.7. CONFIGURING EXTERNAL CLIENTS TO TRUST THE CLUSTER CA 288

CHAPTER 9. AMQ STREAMS AND KAFKA UPGRADES 290
9.1. UPGRADE PREREQUISITES 290
9.2. UPGRADE PROCESS 290
9.3. KAFKA VERSIONS 290
9.4. UPGRADING THE CLUSTER OPERATOR 291
9.4.1. Upgrading the Cluster Operator to a later version 291
9.5. UPGRADING KAFKA 292
9.5.1. Kafka version and image mappings 293
9.5.2. Strategies for upgrading clients 293
9.5.3. Upgrading Kafka brokers and client applications 295
9.6. DOWNGRADING KAFKA 298
9.6.1. Target downgrade version 298
9.6.2. Downgrading Kafka brokers and client applications 298

CHAPTER 10. AMQ STREAMS RESOURCE UPGRADES 301
10.1. UPGRADING KAFKA RESOURCES 301
10.2. UPGRADING KAFKA CONNECT RESOURCES 304
10.3. UPGRADING KAFKA CONNECT S2I RESOURCES 305
10.4. UPGRADING KAFKA MIRROR MAKER RESOURCES 306
10.5. UPGRADING KAFKA TOPIC RESOURCES 307
10.6. UPGRADING KAFKA USER RESOURCES 307

CHAPTER 11. UNINSTALLING AMQ STREAMS 309

CHAPTER 12. CHECKING THE STATUS OF A CUSTOM RESOURCE 310

APPENDIX A. CONFIGURABLE LOGGERS 311

APPENDIX B. FREQUENTLY ASKED QUESTIONS 312
B.1. CLUSTER OPERATOR 312
B.1.1. Why do I need cluster admin privileges to install AMQ Streams? 312
B.1.2. Why does the Cluster Operator require the ability to create ClusterRoleBindings? Is that not a security risk? 312
B.1.3. Why can standard OpenShift users not create the custom resource (Kafka, KafkaTopic, and so on)? 313
B.1.4. Log contains warnings about failing to acquire lock 313
B.1.5. Hostname verification fails when connecting to NodePorts using TLS 313

APPENDIX C. CUSTOM RESOURCE API REFERENCE 315
C.1. KAFKA SCHEMA REFERENCE 315
C.2. KAFKASPEC SCHEMA REFERENCE 315
C.3. KAFKACLUSTERSPEC SCHEMA REFERENCE 316
C.4. EPHEMERALSTORAGE SCHEMA REFERENCE 318
C.5. PERSISTENTCLAIMSTORAGE SCHEMA REFERENCE 318
C.6. PERSISTENTCLAIMSTORAGEOVERRIDE SCHEMA REFERENCE 319
C.7. JBODSTORAGE SCHEMA REFERENCE 319
C.8. KAFKALISTENERS SCHEMA REFERENCE 320
C.9. KAFKALISTENERPLAIN SCHEMA REFERENCE 320
C.10. KAFKALISTENERAUTHENTICATIONTLS SCHEMA REFERENCE 321
C.11. KAFKALISTENERAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE 321
C.12. KAFKALISTENERTLS SCHEMA REFERENCE 321
C.13. KAFKALISTENEREXTERNALROUTE SCHEMA REFERENCE 322
C.14. ROUTELISTENEROVERRIDE SCHEMA REFERENCE 323
C.15. ROUTELISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE 323
C.16. ROUTELISTENERBROKEROVERRIDE SCHEMA REFERENCE 323
C.17. KAFKALISTENEREXTERNALLOADBALANCER SCHEMA REFERENCE 324
C.18. LOADBALANCERLISTENEROVERRIDE SCHEMA REFERENCE 324
C.19. LOADBALANCERLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE 325
C.20. LOADBALANCERLISTENERBROKEROVERRIDE SCHEMA REFERENCE 325
C.21. KAFKALISTENEREXTERNALNODEPORT SCHEMA REFERENCE 326
C.22. NODEPORTLISTENEROVERRIDE SCHEMA REFERENCE 326
C.23. NODEPORTLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE 327
C.24. NODEPORTLISTENERBROKEROVERRIDE SCHEMA REFERENCE 327
C.25. KAFKALISTENEREXTERNALINGRESS SCHEMA REFERENCE 327
C.26. INGRESSLISTENERCONFIGURATION SCHEMA REFERENCE 328
C.27. INGRESSLISTENERBOOTSTRAPCONFIGURATION SCHEMA REFERENCE 328
C.28. INGRESSLISTENERBROKERCONFIGURATION SCHEMA REFERENCE 329
C.29. KAFKAAUTHORIZATIONSIMPLE SCHEMA REFERENCE 329
C.30. RACK SCHEMA REFERENCE 330
C.31. PROBE SCHEMA REFERENCE 330
C.32. JVMOPTIONS SCHEMA REFERENCE 331
C.33. RESOURCEREQUIREMENTS SCHEMA REFERENCE 331
C.34. INLINELOGGING SCHEMA REFERENCE 332
C.35. EXTERNALLOGGING SCHEMA REFERENCE 332
C.36. TLSSIDECAR SCHEMA REFERENCE 332
C.37. KAFKACLUSTERTEMPLATE SCHEMA REFERENCE 333
C.38. RESOURCETEMPLATE SCHEMA REFERENCE 334
C.39. METADATATEMPLATE SCHEMA REFERENCE 334
C.40. PODTEMPLATE SCHEMA REFERENCE 335
C.41. PODDISRUPTIONBUDGETTEMPLATE SCHEMA REFERENCE 336
C.42. ZOOKEEPERCLUSTERSPEC SCHEMA REFERENCE 336
C.43. ZOOKEEPERCLUSTERTEMPLATE SCHEMA REFERENCE 337
C.44. TOPICOPERATORSPEC SCHEMA REFERENCE 338
C.45. ENTITYOPERATORJVMOPTIONS SCHEMA REFERENCE 339
C.46. ENTITYOPERATORSPEC SCHEMA REFERENCE 339
C.47. ENTITYTOPICOPERATORSPEC SCHEMA REFERENCE 340
C.48. ENTITYUSEROPERATORSPEC SCHEMA REFERENCE 341
C.49. ENTITYOPERATORTEMPLATE SCHEMA REFERENCE 342
C.50. CERTIFICATEAUTHORITY SCHEMA REFERENCE 342
C.51. KAFKASTATUS SCHEMA REFERENCE 343
C.52. CONDITION SCHEMA REFERENCE 343

C.53. LISTENERSTATUS SCHEMA REFERENCE 344
C.54. LISTENERADDRESS SCHEMA REFERENCE 344
C.55. KAFKACONNECT SCHEMA REFERENCE 345
C.56. KAFKACONNECTSPEC SCHEMA REFERENCE 345
C.57. KAFKACONNECTTEMPLATE SCHEMA REFERENCE 347
C.58. KAFKACONNECTAUTHENTICATIONTLS SCHEMA REFERENCE 347
C.59. CERTANDKEYSECRETSOURCE SCHEMA REFERENCE 348
C.60. KAFKACONNECTAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE 348
C.61. PASSWORDSECRETSOURCE SCHEMA REFERENCE 349
C.62. KAFKACONNECTAUTHENTICATIONPLAIN SCHEMA REFERENCE 349
C.63. EXTERNALCONFIGURATION SCHEMA REFERENCE 350
C.64. EXTERNALCONFIGURATIONENV SCHEMA REFERENCE 350
C.65. EXTERNALCONFIGURATIONENVVARSOURCE SCHEMA REFERENCE 350
C.66. EXTERNALCONFIGURATIONVOLUMESOURCE SCHEMA REFERENCE 350
C.67. KAFKACONNECTTLS SCHEMA REFERENCE 351
C.68. CERTSECRETSOURCE SCHEMA REFERENCE 351
C.69. KAFKACONNECTS2I SCHEMA REFERENCE 351
C.70. KAFKACONNECTS2ISPEC SCHEMA REFERENCE 352
C.71. KAFKATOPIC SCHEMA REFERENCE 354
C.72. KAFKATOPICSPEC SCHEMA REFERENCE 354
C.73. KAFKAUSER SCHEMA REFERENCE 354
C.74. KAFKAUSERSPEC SCHEMA REFERENCE 354
C.75. KAFKAUSERTLSCLIENTAUTHENTICATION SCHEMA REFERENCE 355
C.76. KAFKAUSERSCRAMSHA512CLIENTAUTHENTICATION SCHEMA REFERENCE 355
C.77. KAFKAUSERAUTHORIZATIONSIMPLE SCHEMA REFERENCE 355
C.78. ACLRULE SCHEMA REFERENCE 356
C.79. ACLRULETOPICRESOURCE SCHEMA REFERENCE 356
C.80. ACLRULEGROUPRESOURCE SCHEMA REFERENCE 357
C.81. ACLRULECLUSTERRESOURCE SCHEMA REFERENCE 357
C.82. ACLRULETRANSACTIONALIDRESOURCE SCHEMA REFERENCE 358
C.83. KAFKAMIRRORMAKER SCHEMA REFERENCE 358
C.84. KAFKAMIRRORMAKERSPEC SCHEMA REFERENCE 358
C.85. KAFKAMIRRORMAKERCONSUMERSPEC SCHEMA REFERENCE 360
C.86. KAFKAMIRRORMAKERAUTHENTICATIONTLS SCHEMA REFERENCE 361
C.87. KAFKAMIRRORMAKERAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE 361
C.88. KAFKAMIRRORMAKERAUTHENTICATIONPLAIN SCHEMA REFERENCE 361
C.89. KAFKAMIRRORMAKERTLS SCHEMA REFERENCE 362
C.90. KAFKAMIRRORMAKERPRODUCERSPEC SCHEMA REFERENCE 362
C.91. KAFKAMIRRORMAKERTEMPLATE SCHEMA REFERENCE 363
C.92. KAFKABRIDGE SCHEMA REFERENCE 363
C.93. KAFKABRIDGESPEC SCHEMA REFERENCE 363
C.94. KAFKABRIDGETLS SCHEMA REFERENCE 365
C.95. KAFKABRIDGEAUTHENTICATIONTLS SCHEMA REFERENCE 365
C.96. KAFKABRIDGEAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE 365
C.97. KAFKABRIDGEAUTHENTICATIONPLAIN SCHEMA REFERENCE 366
C.98. KAFKABRIDGEHTTPCONFIG SCHEMA REFERENCE 366
C.99. KAFKABRIDGECONSUMERSPEC SCHEMA REFERENCE 367
C.100. KAFKABRIDGEPRODUCERSPEC SCHEMA REFERENCE 367
C.101. KAFKABRIDGETEMPLATE SCHEMA REFERENCE 367

APPENDIX D. USING YOUR SUBSCRIPTION **369**
Accessing Your Account 369
Activating a Subscription 369

Downloading Zip and Tar Files 369

CHAPTER 1. OVERVIEW OF AMQ STREAMS

AMQ Streams is based on Apache Kafka, a popular platform for streaming data delivery and processing. AMQ Streams makes it easy to run Apache Kafka on OpenShift.

AMQ Streams provides three operators:

Cluster Operator

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

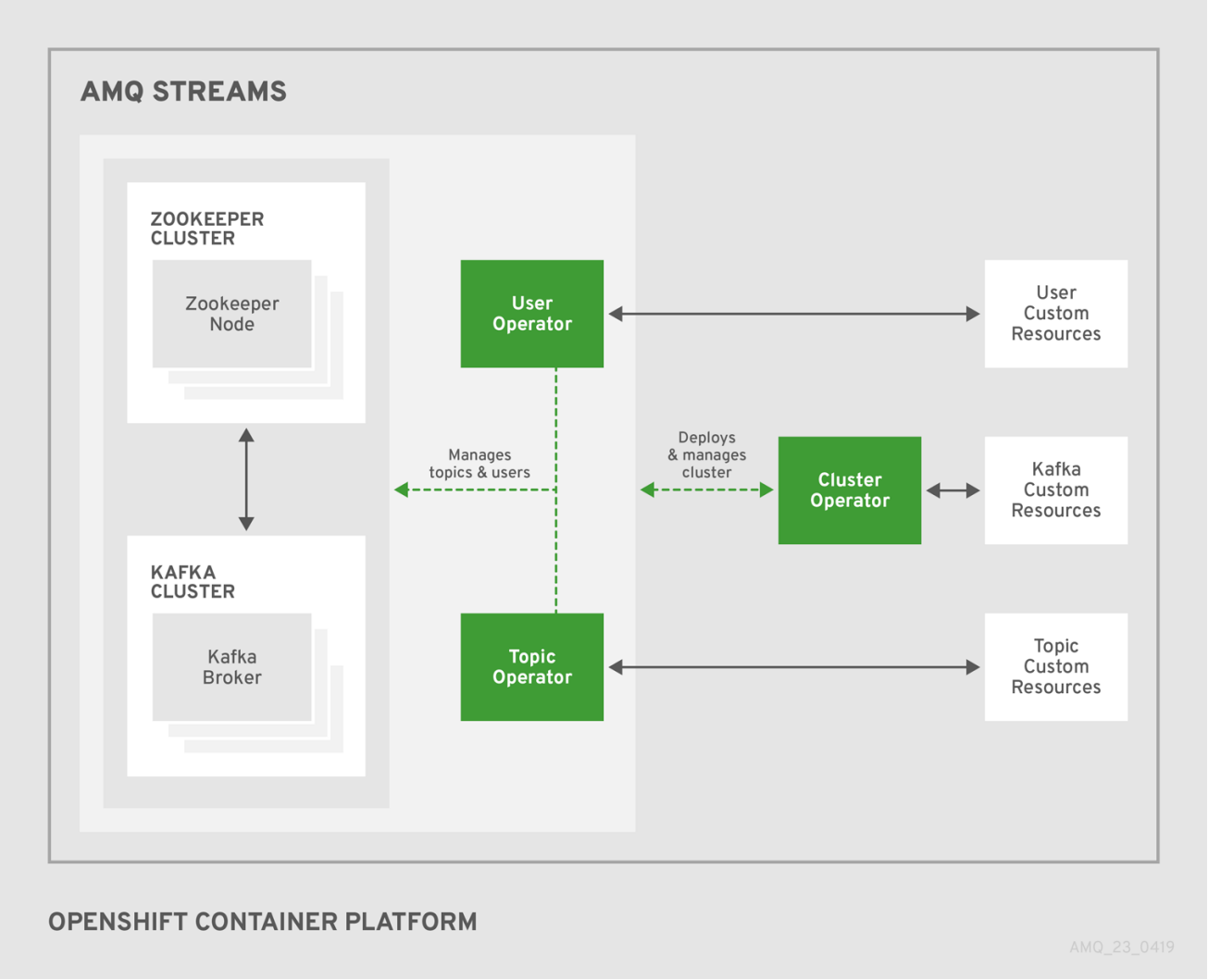
Topic Operator

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

User Operator

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

Operators within the AMQ Streams architecture



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

1.1. KAFKA KEY FEATURES

- Designed for horizontal scalability
- Message ordering guarantee at the partition level

- Message rewind/replay
  - "Long term" storage allows the reconstruction of an application state by replaying the messages
  - Combines with compacted topics to use Kafka as a key-value store

Additional resources

- For more information about Apache Kafka, see the Apache Kafka website.

1.2. DOCUMENT CONVENTIONS

Replaceables

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

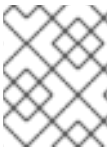
For example, in the following code, you will want to replace ***my-namespace*** with the name of your namespace:

```
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 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 0.12.x. This chapter describes the procedures to deploy AMQ Streams on OpenShift 3.11 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 **amq-streams-x.y.z-ocp-install-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. CUSTOM RESOURCES

Custom resource definitions (CRDs) extend the Kubernetes API, providing definitions to create and modify custom resources to an OpenShift cluster. Custom resources are created as instances of CRDs.

In AMQ Streams, CRDs introduce custom resources specific to AMQ Streams to an OpenShift cluster, such as Kafka, Kafka Connect, Kafka Mirror Maker, and users and topics custom resources. CRDs provide configuration instructions, defining the schemas used to instantiate and manage the AMQ Streams-specific resources. CRDs also allow AMQ Streams resources to benefit from native OpenShift features like CLI accessibility and configuration validation.

CRDs require a one-time installation in a cluster. Depending on the cluster setup, installation typically requires cluster admin privileges.

NOTE

Access to manage custom resources is limited to AMQ Streams administrators .

CRDs and custom resources are defined as YAML files.

A CRD defines a new **kind** of resource, such as **kind:Kafka**, within an OpenShift cluster.

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

WARNING

When CRDs are deleted, custom resources of that type are also deleted. Additionally, the resources created by the custom resource, such as pods and statefulsets are also deleted.

Additional resources

- Extend the Kubernetes API with CustomResourceDefinitions

2.2.1. AMQ Streams custom resource example

Each AMQ Streams-specific custom resource conforms to the schema defined by the CRD for the resource's **kind**.

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

Kafka topic CRD

```
apiVersion: kafka.strimzi.io/v1beta1
kind: CustomResourceDefinition
metadata: 1
  name: kafkatopics.kafka.strimzi.io
  labels:
    app: strimzi
spec: 2
  group: kafka.strimzi.io
  versions:
    v1beta1
  scope: Namespaced
  names:
    # ...
    singular: kafkatopic
    plural: kafkatopics
    shortNames:
    - kt 3
  additionalPrinterColumns: 4
      # ...
  validation: 5
    openAPIV3Schema:
      properties:
        spec:
          type: object
          properties:
```

```
        partitions:
          type: integer
          minimum: 1
        replicas:
          type: integer
          minimum: 1
          maximum: 32767
    # ...
```

1. The metadata for the topic CRD, its name and a label to identify the CRD.
2. The specification for this CRD, including the group (domain) name, the plural name and the supported schema version, which are used in the URL to access the API of the topic. The other names are used to identify instance resources in the CLI. For example, **oc get kafkatopic my-topic** or **oc get kafkatopics**.
3. The shortname can be used in CLI commands. For example, **oc get kt** can be used as an abbreviation instead of **oc get kafkatopic**.
4. The information presented when using a **get** command on the custom resource.
5. openAPIV3Schema validation provides validation for the creation of topic custom resources. For example, a topic requires at least one partition and one replica.

NOTE

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

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

Kafka topic custom resource

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic (1)
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec: (2)
  partitions: 1
  replicas: 1
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
```

1. The **kind** and **apiVersion** identify the CRD of which the custom resource is an instance.
2. The spec shows the number of partitions and replicas for the topic as well as configuration for the retention period for a message to remain in the topic and the segment file size for the log.

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

After a **KafkaTopic** custom resource is created, the Topic Operator is notified and corresponding Kafka topics are created in AMQ Streams.

2.2.2. AMQ Streams custom resource status

The **status** property of a AMQ Streams-specific custom resource publishes the *current* state of the resource to users and tools that need the information.

Status information is useful for tracking progress related to a resource achieving its *desired* state, as defined by the **spec** property. The **status** provides the time and reason the state of the resource changed and details of events preventing or delaying the Operator from realizing the desired state.

AMQ Streams creates and maintains the status of custom resources, periodically evaluating the current state of the custom resource and updating its status accordingly.

When performing an update on a custom resource using **oc edit**, for example, its **status** is not editable. Moreover, changing the **status** would not affect the configuration of the Kafka cluster.

IMPORTANT

The status property feature for AMQ Streams-specific custom resources is still under development and only available for Kafka resources.

Here we see the **status** property specified for a Kafka custom resource.

Kafka custom resource with status

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
spec:
  # ...
status:
  conditions: 1
  - lastTransitionTime: 2019-06-02T23:46:57+0000
    status: "True"
    type: Ready 2
  listeners: 3
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9092
    type: plain
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9093
    type: tls
  - addresses:
    - host: 172.29.49.180
      port: 9094
    type: external
    # ...
```

1 Status **conditions** describe criteria related to the status that cannot be deduced from the existing resource information, or are specific to the instance of a resource.

2 The **Ready** condition indicates whether the Cluster Operator currently considers the Kafka cluster able to handle traffic.

3 The **listeners** describe the current Kafka bootstrap addresses by type.

IMPORTANT

The status for **external** listeners is still under development and does not provide a specific IP address for external listeners of type **nodeport**.

NOTE

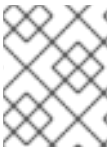
The Kafka bootstrap addresses listed in the status do not signify that those endpoints or the Kafka cluster is in a ready state.

Accessing status information

You can access status information for a resource from the command line. For more information, see Chapter 12, *Checking the status of a custom resource* .

2.3. 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”



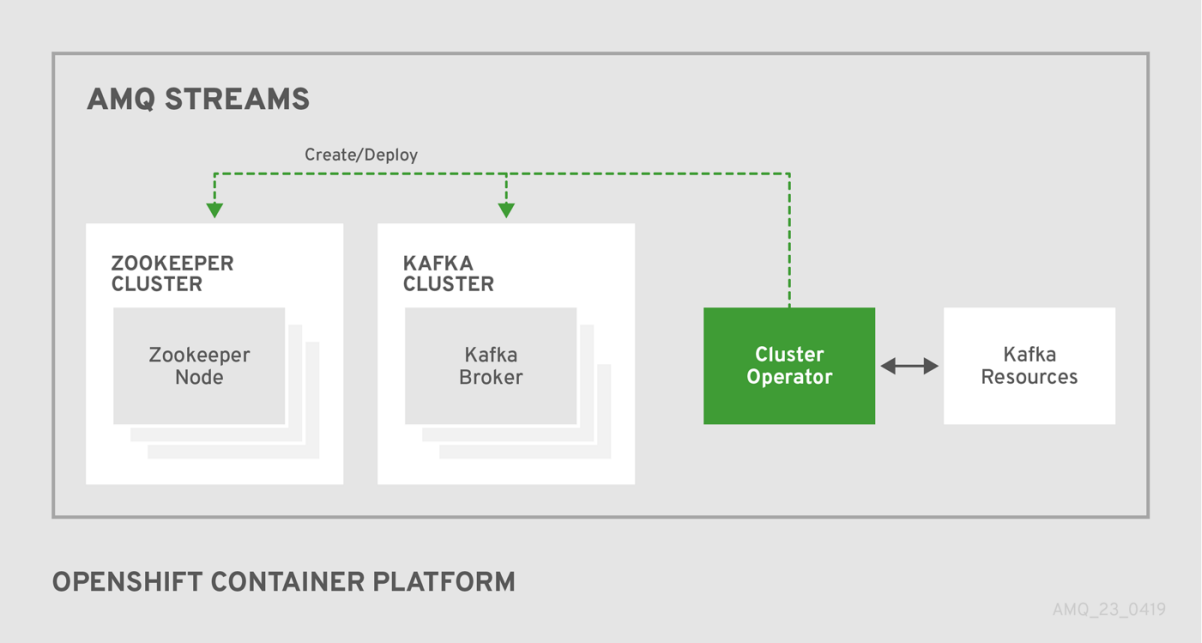
NOTE

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

2.3.1. 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.

Example architecture for the Cluster Operator



When the Cluster Operator is up, it starts to *watch* for certain OpenShift resources containing the desired Kafka, Kafka Connect, or Kafka Mirror Maker 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.
- A **KafkaMirrorMaker** resource for the Kafka Mirror Maker instance.

When a new **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, or **Kafka Mirror Maker** resource is created in the OpenShift cluster, the operator gets the cluster description from the desired resource and starts creating a new Kafka, Kafka Connect, or Kafka Mirror Maker 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, Kafka Connect, or Kafka Mirror Maker cluster. Resources are either patched or deleted and then re-created in order to make the Kafka, Kafka Connect, or Kafka Mirror Maker 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.

2.3.2. 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

- Deploy the Cluster Operator:

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

2.3.3. 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: registry.redhat.io/amq7/amq-streams-operator:1.2.0
        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
```

2.3.4. Deploying the Cluster Operator to watch all namespaces

You can configure the Cluster Operator to watch AMQ Streams resources across all OpenShift projects or Kubernetes namespaces in your OpenShift cluster. When running in this mode, the Cluster Operator automatically manages clusters in any new projects or namespaces that are created.

Prerequisites

- Your OpenShift cluster is running.

Procedure

1. Configure the Cluster Operator to watch all namespaces:
   a. Edit the **050-Deployment-strimzi-cluster-operator.yaml** file.
   b. Set the value of the **STRIMZI_NAMESPACE** environment variable to ***.

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  template:
    spec:
      # ...
      serviceAccountName: strimzi-cluster-operator
      containers:
      - name: strimzi-cluster-operator
        image: registry.redhat.io/amq7/amq-streams-operator:1.2.0
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: "*"
        # ...
```

2. Create **ClusterRoleBindings** that grant cluster-wide access to all OpenShift projects or Kubernetes namespaces to the Cluster Operator.
   On OpenShift, use the **oc adm policy** command:

```
oc adm policy add-cluster-role-to-user strimzi-cluster-operator-namespaced --serviceaccount strimzi-cluster-operator -n my-project
```

```
oc adm policy add-cluster-role-to-user strimzi-entity-operator --serviceaccount strimzi-cluster-operator -n my-project
oc adm policy add-cluster-role-to-user strimzi-topic-operator --serviceaccount strimzi-cluster-operator -n my-project
```

Replace ***my-project*** with the project in which you want to install the Cluster Operator.

3. Deploy the Cluster Operator to your OpenShift cluster.
   On OpenShift, use the **oc apply** command:

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

2.4. 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
# ...
```

2.4.1. 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 2.3, "Cluster Operator". For more information on the different configuration options supported by the **Kafka** resource, see Section 3.1, "Kafka cluster configuration".

2.5. KAFKA CONNECT

Kafka Connect is a tool for streaming data between Apache Kafka and external systems. It provides a framework for moving large amounts of data into and out of your Kafka cluster while maintaining scalability and reliability. Kafka Connect is typically used to integrate Kafka with external databases and storage and messaging systems.

You can use Kafka Connect to:

- Build connector plug-ins (as JAR files) for your Kafka cluster
- Run connectors

Kafka Connect includes the following built-in connectors for moving file-based data into and out of your Kafka cluster.

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

In AMQ Streams, you can use the Cluster Operator to deploy a Kafka Connect or Kafka Connect Source-2-Image (S2I) cluster to your OpenShift cluster.

A Kafka Connect cluster is implemented as a **Deployment** with a configurable number of workers. The Kafka Connect REST API is available on port 8083, as the **<connect-cluster-name>-connect-api** service.

For more information on deploying a Kafka Connect S2I cluster, see Creating a container image using OpenShift builds and Source-to-Image.

2.5.1. Deploying Kafka Connect to your OpenShift cluster

You can deploy a Kafka Connect cluster to your OpenShift cluster by using the Cluster Operator. Kafka Connect is provided as an OpenShift template that you can deploy from the command line or the OpenShift console.

Prerequisites

- Deploying the Cluster Operator to OpenShift

Procedure

- Use the **oc apply** command to create a **KafkaConnect** resource based on the **kafka-connect.yaml** file:

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

Additional resources

- Kafka Connect cluster configuration
- Kafka Connect cluster with Source2Image support

2.5.2. Extending Kafka Connect with connector plug-ins

The AMQ Streams container images for Kafka Connect include the two built-in file connectors: **FileStreamSourceConnector** and **FileStreamSinkConnector**. You can add your own connectors by using one of the following methods:

- Create a Docker image from the Kafka Connect base image.
- Create a container image using OpenShift builds and Source-to-Image (S2I).

2.5.2.1. Creating a Docker image from the Kafka Connect base image

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

The following procedure explains how to create your custom image and add it to the **/opt/kafka/plugins** directory. At startup, the AMQ Streams version of Kafka Connect loads any third-party connector plug-ins contained in the **/opt/kafka/plugins** directory.

Prerequisites

- Deploying the Cluster Operator to OpenShift

Procedure

1. Create a new **Dockerfile** using **registry.redhat.io/amq7/amqstreams-kafka-22** as the base image:

```
FROM registry.redhat.io/amq7/amqstreams-kafka-22
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER kafka:kafka
```

2. Build the container image.

3. Push your custom image to your container registry.

4. Edit the **KafkaConnect.spec.image** property of the **KafkaConnect** custom resource to point to the new container image. If set, this property overrides the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable referred to in the next step.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  image: my-new-container-image
```

5. In the **install/cluster-operator/050-Deployment-strimzi-cluster-operator.yaml** file, edit the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable to point to the new container image.

Additional resources

- For more information on the **KafkaConnect.spec.image property**, see Section 3.2.11, "Container images".
- For more information on the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** variable, see Section 4.1.6, "Cluster Operator Configuration".

2.5.2.2. Creating a container image using OpenShift builds and Source-to-Image

You can use OpenShift builds and 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. Once built, container images are stored in OpenShift's local container image repository and are available for use in deployments.

A Kafka Connect builder image with S2I support is provided on the Red Hat Container Catalog as part of the **registry.redhat.io/amq7/amqstreams-kafka-22** image. This S2I image takes your binaries (with plug-ins and connectors) and stores them in the **/tmp/kafka-plugins/s2i** directory. It creates a new Kafka Connect image from this directory, which can then be used with the Kafka Connect deployment. When started using the enhanced image, Kafka Connect loads any third-party plug-ins from the **/tmp/kafka-plugins/s2i** directory.

Procedure

1. On the command line, use the **oc apply** command to create and deploy a Kafka Connect S2I cluster:

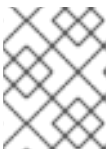
```
oc apply -f examples/kafka-connect/kafka-connect-s2i.yaml
```

2. Create a directory with Kafka Connect plug-ins:

```
$ 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. Use the **oc start-build** command to start a new build of the image using the prepared directory:

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



NOTE

The name of the build is the same as the name of the deployed Kafka Connect cluster.

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

2.6. 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.

2.6.1. 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 2.3, “Cluster Operator”

2.7. KAFKA BRIDGE

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

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

2.7.1. Deploying Kafka Bridge to your OpenShift cluster

You can deploy a Kafka Bridge cluster to your OpenShift cluster by using the Cluster Operator. Kafka Bridge is provided as an OpenShift template that you can deploy from the command line or the OpenShift console.

Prerequisites

- Deploying the Cluster Operator to OpenShift

Procedure

- Use the **oc apply** command to create a **KafkaBridge** resource based on the **kafka-bridge.yaml** file:

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

Additional resources

- Kafka Bridge cluster configuration

2.8. 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.redhat.io/amq7/amq-streams-kafka-22 --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.redhat.io/amq7/amq-streams-kafka-22 --rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server cluster-name-kafka-bootstrap:9092 --topic my-topic --from-beginning
   ```

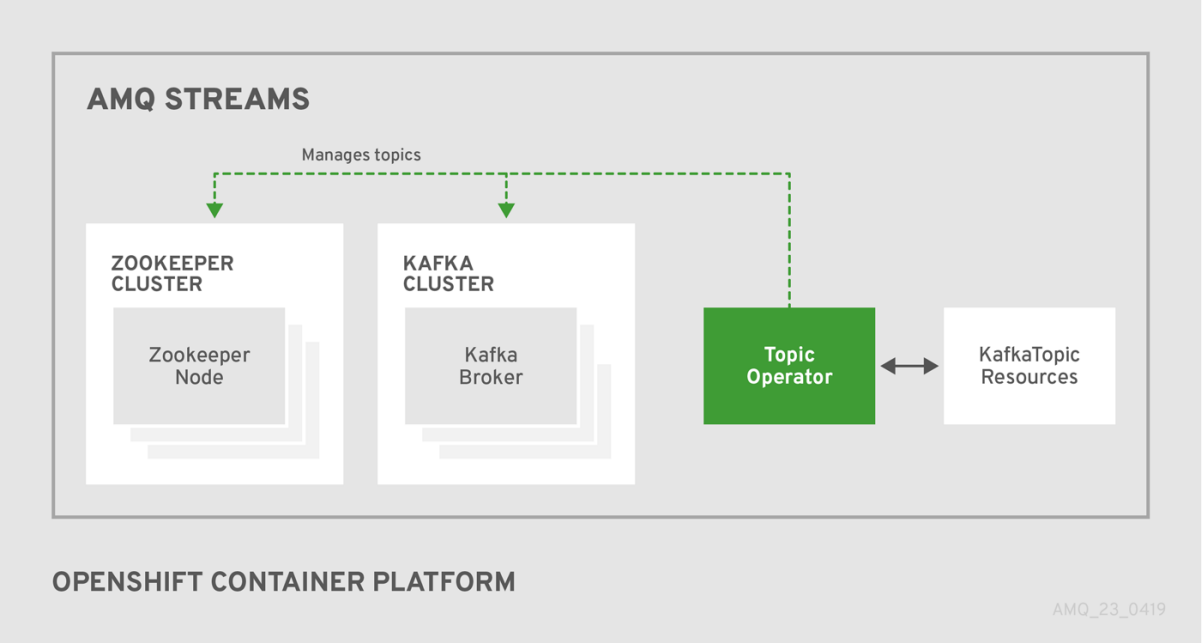
5. Confirm that you see the incoming messages in the consumer console.

2.9. TOPIC OPERATOR

2.9.1. Overview of the Topic Operator component

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

Example architecture for the Topic Operator



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
- Deleted, the operator will delete the topic it describes
- 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
- Deleted from the Kafka cluster, the operator will delete the **KafkaTopic** describing it
- Changed in the Kafka cluster, 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 is 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*.

2.9.2. 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the Topic Operator using the fields described in Section C.47, "**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 2.3, "Cluster Operator".
- For more information about deploying the Entity Operator, see Section 3.1.10, "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 C.46, "**EntityOperatorSpec** schema reference".

2.10. USER OPERATOR

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

2.10.1. 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.

2.10.2. 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 2.3, "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.

2.11. STRIMZI ADMINISTRATORS

AMQ Streams includes several custom resources. By default, permission to create, edit, and delete these resources is limited to OpenShift cluster administrators. If you want to allow non-cluster administators to manage AMQ Streams resources, you must assign them the Strimzi Administrator role.

2.11.1. Designating Strimzi Administrators

Prerequisites

- AMQ Streams **CustomResourceDefinitions** are installed.

Procedure

1. Create the **strimzi-admin** cluster role in OpenShift.
   On OpenShift, use **oc apply**:

```
oc apply -f install/strimzi-admin
```

2. Assign the **strimzi-admin ClusterRole** to one or more existing users in the OpenShift cluster. On OpenShift, use **oc adm**:

```
oc adm policy add-cluster-role-to-user strimzi-admin user1 user2
```

2.12. CONTAINER IMAGES

Container images for AMQ Streams are available in the Red Hat Container Catalog . The installation YAML files provided by AMQ Streams will pull the images directly from the Red Hat Container Catalog .

If you do not have access to the Red Hat Container Catalog or want to use your own container repository:

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

NOTE

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

Container image	Namespace/Repository	Description
Kafka	• registry.redhat.io/amq7/amqstreams-kafka-22 • registry.redhat.io/amq7/amqstreams-kafka-21	AMQ Streams image for running Kafka, including: • Kafka Broker • Kafka Connect / S2I • Kafka Mirror Maker • Zookeeper • TLS Sidecars
Operator	• registry.redhat.io/amq7/amq-streams-operator:1.2.0	AMQ Streams image for running the operators: • Cluster Operator • Topic Operator • User Operator • Kafka Initializer

Container image	Namespace/Repository	Description
Kafka Bridge	• registry.redhat.io/amq7/amq-streams-bridge:1.2.0	AMQ Streams image for running the AMQ Streams Kafka Bridge

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 C.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 resources to be labeled as required.

3.1.1. Data storage considerations

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

AMQ Streams requires block storage and is designed to work optimally with cloud-based block storage solutions, including Amazon Elastic Block Store (EBS). The use of file storage (for example, NFS) is not recommended.

Choose local storage (local persistent volumes) when possible. If local storage is not available, you can use a Storage Area Network (SAN) accessed by a protocol such as Fibre Channel or iSCSI.

3.1.1.1. Apache Kafka and Zookeeper storage

Use separate disks for Apache Kafka and Zookeeper.

Three types of data storage are supported:

- Ephemeral (Recommended for development only)
- Persistent
- JBOD (Just a Bunch of Disks, suitable for Kafka only)

For more information, see Kafka and Zookeeper storage.

Solid-state drives (SSDs), though not essential, can improve the performance of Kafka in large clusters where data is sent to and received from multiple topics asynchronously. SSDs are particularly effective with Zookeeper, which requires fast, low latency data access.

NOTE

You do not need to provision replicated storage because Kafka and Zookeeper both have built-in data replication.

3.1.1.2. File systems

It is recommended that you configure your storage system to use the XFS file system. AMQ Streams is also compatible with the ext4 file system, but this might require additional configuration for best results.

3.1.2. Kafka and Zookeeper storage types

As stateful applications, Kafka and Zookeeper need to store data on disk. AMQ Streams supports three storage types for this data:

- Ephemeral
- Persistent
- JBOD storage

NOTE

JBOD storage is supported only for Kafka, not for Zookeeper.

When configuring a **Kafka** resource, you can specify the type of storage used by the Kafka broker and its corresponding Zookeeper node. You configure the storage type using the **storage** property in the following resources:

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

The storage type is configured in the **type** field.

WARNING

The storage type cannot be changed after a Kafka cluster is deployed.

3.1.2.1. 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/v1beta1
kind: Kafka
metadata:
```

```
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: ephemeral
    # ...
  zookeeper:
    # ...
    storage:
      type: ephemeral
    # ...
```

3.1.2.1.1. Log directories

The ephemeral volume will be used by the Kafka brokers as log directories mounted into the following path:

/var/lib/kafka/data/kafka-log_idx_

Where ***idx*** is the Kafka broker pod index. For example **/var/lib/kafka/data/kafka-log0**.

3.1.2.2. 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:

id (optional)

Storage identification number. This option is mandatory for storage volumes defined in a JBOD storage declaration. Default is **0**.

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 key:value pairs representing labels for selecting such a volume.

deleteClaim (optional)

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

WARNING

Increasing the size of persistent volumes in an existing AMQ Streams cluster is only supported in OpenShift versions that support persistent volume resizing. The persistent volume to be resized must use a storage class that supports volume expansion. For other versions of OpenShift and storage classes which do not support volume expansion, you must decide the necessary storage size before deploying the cluster. Decreasing the size of existing persistent volumes is not possible.

Example 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:
    hdd-type: ssd
  deleteClaim: true
# ...
```

3.1.2.2.1. Storage class overrides

You can specify a different storage class for one or more Kafka brokers, instead of using the default storage class. This is useful if, for example, storage classes are restricted to different availability zones or data centers. You can use the **overrides** field for this purpose.

In this example, the default storage class is named **my-storage-class**:

Example AMQ Streams cluster using storage class overrides

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  # ...
  kafka:
    replicas: 3
    storage:
      deleteClaim: true
      size: 100Gi
      type: persistent-claim
      class: my-storage-class
      overrides:
        - broker: 0
          class: my-storage-class-zone-1a
        - broker: 1
          class: my-storage-class-zone-1b
        - broker: 2
          class: my-storage-class-zone-1c
  # ...
```

As a result of the configured **overrides** property, the broker volumes use the following storage classes:

- The persistent volumes of broker 0 will use **my-storage-class-zone-1a**.
- The persistent volumes of broker 1 will use **my-storage-class-zone-1b**.
- The persistent volumes of broker 2 will use **my-storage-class-zone-1c**.

The **overrides** property is currently used only to override storage class configurations. Overriding other storage configuration fields is not currently supported. Other fields from the storage configuration are currently not supported.

3.1.2.2.2. Persistent Volume Claim naming

When persistent storage is used, it creates 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***.

3.1.2.2.3. Log directories

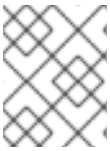
The persistent volume will be used by the Kafka brokers as log directories mounted into the following path:

/var/lib/kafka/data/kafka-log_idx_

Where ***idx*** is the Kafka broker pod index. For example **/var/lib/kafka/data/kafka-log0**.

3.1.2.3. Resizing persistent volumes

You can provision increased storage capacity by increasing the size of the persistent volumes used by an existing AMQ Streams cluster. Resizing persistent volumes is supported in clusters that use either a single persistent volume or multiple persistent volumes in a JBOD storage configuration.



NOTE

You can increase but not decrease the size of persistent volumes. Decreasing the size of persistent volumes is not currently supported in OpenShift.

Prerequisites

- An OpenShift cluster with support for volume resizing.
- The Cluster Operator is running.
- A Kafka cluster using persistent volumes created using a storage class that supports volume expansion.

Procedure

1. In a **Kafka** resource, increase the size of the persistent volume allocated to the Kafka cluster, the Zookeeper cluster, or both.
   - To increase the volume size allocated to the Kafka cluster, edit the **spec.kafka.storage** property.
   - To increase the volume size allocated to the Zookeeper cluster, edit the **spec.zookeeper.storage** property.
     For example, to increase the volume size from **1000Gi** to **2000Gi**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: persistent-claim
      size: 2000Gi
      class: my-storage-class
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   On OpenShift, use **oc apply**:

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

OpenShift increases the capacity of the selected persistent volumes in response to a request from the Cluster Operator. When the resizing is complete, the Cluster Operator restarts all pods that use the resized persistent volumes. This happens automatically.

Additional resources

For more information about resizing persistent volumes in OpenShift, see Resizing Persistent Volumes using Kubernetes.

3.1.2.4. JBOD storage overview

You can configure AMQ Streams to use JBOD, a data storage configuration of multiple disks or volumes. JBOD is one approach to providing increased data storage for Kafka brokers. It can also improve performance.

A JBOD configuration is described by one or more volumes, each of which can be either ephemeral or persistent. The rules and constraints for JBOD volume declarations are the same as those for ephemeral and persistent storage. For example, you cannot change the size of a persistent storage volume after it has been provisioned.

3.1.2.4.1. JBOD configuration

To use JBOD with AMQ Streams, the storage **type** must be set to **jbod**. The **volumes** property allows you to describe the disks that make up your JBOD storage array or configuration. The following fragment shows an example JBOD configuration:

```
# ...
storage:
  type: jbod
  volumes:
  - id: 0
    type: persistent-claim
    size: 100Gi
    deleteClaim: false
  - id: 1
    type: persistent-claim
    size: 100Gi
    deleteClaim: false
# ...
```

The ids cannot be changed once the JBOD volumes are created.

Users can add or remove volumes from the JBOD configuration.

3.1.2.4.2. JBOD and Persistent Volume Claims

When persistent storage is used to declare JBOD volumes, the naming scheme of the resulting Persistent Volume Claims is as follows:

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

Where ***id*** is the ID of the volume used for storing data for Kafka broker pod ***idx***.

3.1.2.4.3. Log directories

The JBOD volumes will be used by the Kafka brokers as log directories mounted into the following path:

/var/lib/kafka/data-*id*/kafka-log_idx_

Where ***id*** is the ID of the volume used for storing data for Kafka broker pod ***idx***. For example **/var/lib/kafka/data-0/kafka-log0**.

3.1.2.5. Adding volumes to JBOD storage

This procedure describes how to add volumes to a Kafka cluster configured to use JBOD storage. It cannot be applied to Kafka clusters configured to use any other storage type.

NOTE

When adding a new volume under an **id** which was already used in the past and removed, you have to make sure that the previously used **PersistentVolumeClaims** have been deleted.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- A Kafka cluster with JBOD storage

Procedure

1. Edit the **spec.kafka.storage.volumes** property in the **Kafka** resource. Add the new volumes to the **volumes** array. For example, add the new volume with id **2**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 2
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
    # ...
  zookeeper:
    # ...
```

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

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

3. Create new topics or reassign existing partitions to the new disks.

Additional resources

For more information about reassigning topics, see Section 3.1.22.2, "Partition reassignment".

3.1.2.6. Removing volumes from JBOD storage

This procedure describes how to remove volumes from Kafka cluster configured to use JBOD storage. It cannot be applied to Kafka clusters configured to use any other storage type. The JBOD storage always has to contain at least one volume.

IMPORTANT

To avoid data loss, you have to move all partitions before removing the volumes.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- A Kafka cluster with JBOD storage with two or more volumes

Procedure

1. Reassign all partitions from the disks which are you going to remove. Any data in partitions still assigned to the disks which are going to be removed might be lost.
2. Edit the **spec.kafka.storage.volumes** property in the **Kafka** resource. Remove one or more volumes from the **volumes** array. For example, remove the volumes with ids **1** and **2**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
    # ...
  zookeeper:
    # ...
```

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

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

Additional resources

For more information about reassigning topics, see Section 3.1.22.2, “Partition reassignment”.

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 JBOD storage, see JBOD schema reference.
- For more information about the schema for **Kafka**, see **Kafka** schema reference.

3.1.3. Kafka broker 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.

3.1.3.1. 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, see Section 3.1.22, “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/v1beta1
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 3.1.22, “Scaling clusters”.

3.1.4. Kafka broker configuration

AMQ Streams allows you to customize the configuration of the Kafka brokers in your Kafka cluster. You can specify and configure most of the options listed in the "Broker Configs" section of the Apache Kafka documentation. You cannot configure options that are 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.

3.1.4.1. Kafka broker configuration

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

This property should contain the Kafka broker configuration options as keys with values in one of the following JSON types:

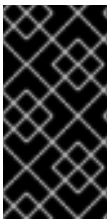
- String
- Number
- Boolean

You can specify and configure all of the options in the "Broker Configs" section of the Apache Kafka documentation apart from those managed directly by AMQ Streams. Specifically, you are prevented from modifying all configuration options with keys equal to or starting with one of the following strings:

- **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**

If the **config** property specifies a restricted option, it is ignored and a warning message is printed to the Cluster Operator log file. All other supported options are passed to Kafka.



IMPORTANT

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

An example Kafka broker configuration

```
apiVersion: kafka.strimzi.io/v1beta1
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
  # ...
```

3.1.4.2. Configuring Kafka brokers

You can configure an existing Kafka broker, or create a new Kafka broker with a specified configuration.

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.kafka.config** property in the **Kafka** resource, enter one or more Kafka configuration settings. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
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:
    # ...
```

3. Apply the new configuration to create or update the resource.
   On OpenShift, use **oc apply**:

```
oc apply -f kafka.yaml
```

   where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

3.1.5. Kafka broker listeners

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

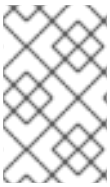
- Plain listener on port 9092 (without encryption)
- TLS listener on port 9093 (with encryption)

- External listener on port 9094 for access from outside of OpenShift

3.1.5.1. Mutual TLS authentication for clients

3.1.5.1.1. Mutual TLS authentication

Mutual TLS authentication is always used for the communication between Kafka brokers and Zookeeper pods.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.



NOTE

TLS authentication is more commonly one-way, with one party authenticating the identity of another. For example, when HTTPS is used between a web browser and a web server, the server obtains proof of the identity of the browser.

3.1.5.1.2. 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.

3.1.5.2. 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 (Simple Authentication and Security Layer) SCRAM-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 for each authentication exchange. This means that the exchange is resilient against replay attacks.

3.1.5.2.1. Supported SCRAM credentials

AMQ Streams supports SCRAM-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.

3.1.5.2.2. 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
- Authentication for unencrypted communication is required

3.1.5.3. 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: {}
  external:
    type: loadbalancer
# ...
```

An example of listeners property with only the plain listener enabled

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

3.1.5.3.1. 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**

3.1.5.3.1.1. 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.

By default, the route hosts are automatically assigned by OpenShift. However, you can override the assigned route hosts by specifying the requested hosts in the **overrides** property. AMQ Streams will not perform any validation that the requested hosts are available; you must ensure that they are free and can be used.

Example of an external listener of type routes configured with overrides for OpenShift route hosts

```
# ...
listeners:
  external:
    type: route
    authentication:
      type: tls
    overrides:
      bootstrap:
        host: bootstrap.myrouter.com
      brokers:
      - broker: 0
        host: broker-0.myrouter.com
      - broker: 1
        host: broker-1.myrouter.com
      - broker: 2
        host: broker-2.myrouter.com
# ...
```

For more information on using **Routes** to access Kafka, see Section 3.1.5.5, "Accessing Kafka using OpenShift routes".

3.1.5.3.1.2. 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 3.1.5.6, "Accessing Kafka using loadbalancers".

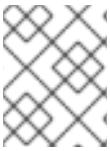
3.1.5.3.1.3. Exposing Kafka using node ports

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.

By default, the port numbers used for the bootstrap and broker services are automatically assigned by OpenShift. However, you can override the assigned node ports by specifying the requested port numbers in the **overrides** property. AMQ Streams does not perform any validation on the requested ports; you must ensure that they are free and available for use.

Example of an external listener configured with overrides for node ports

```
# ...
listeners:
  external:
    type: nodeport
    tls: true
    authentication:
      type: tls
    overrides:
      bootstrap:
        nodePort: 32100
      brokers:
      - broker: 0
        nodePort: 32000
      - broker: 1
        nodePort: 32001
      - broker: 2
        nodePort: 32002
# ...
```

For more information on using node ports to access Kafka, see Section 3.1.5.7, “Accessing Kafka using node ports”.

3.1.5.3.1.4. Customizing advertised addresses on external listeners

By default, AMQ Streams tries to automatically determine the hostnames and ports that your Kafka cluster advertises to its clients. This is not sufficient in all situations, because the infrastructure on which AMQ Streams is running might not provide the right hostname or port through which Kafka can be

accessed. You can customize the advertised hostname and port in the **overrides** property of the external listener. AMQ Streams will then automatically configure the advertised address in the Kafka brokers and add it to the broker certificates so it can be used for TLS hostname verification. Overriding the advertised host and ports is available for all types of external listeners.

Example of an external listener configured with overrides for advertised addresses

```
# ...
listeners:
  external:
    type: route
    authentication:
      type: tls
    overrides:
      brokers:
      - broker: 0
        advertisedHost: example.hostname.0
        advertisedPort: 12340
      - broker: 1
        advertisedHost: example.hostname.1
        advertisedPort: 12341
      - broker: 2
        advertisedHost: example.hostname.2
        advertisedPort: 12342
# ...
```

Additionally, you can specify the name of the bootstrap service. This name will be added to the broker certificates and can be used for TLS hostname verification. Adding the additional bootstrap address is available for all types of external listeners.

Example of an external listener configured with an additional bootstrap address

```
# ...
listeners:
  external:
    type: route
    authentication:
      type: tls
    overrides:
      bootstrap:
        address: example.hostname
# ...
```

3.1.5.3.1.5. Customizing DNS names of external listeners

On **loadbalancer** listeners, you can use the **dnsAnnotations** property to add additional annotations to the load balancer services. You can use these annotations to instrument DNS tooling such as External DNS, which automatically assigns DNS names to the services.

Example of an external listener of type loadbalancer using External DNS annotations

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

```
    type: loadbalancer
    authentication:
      type: tls
    overrides:
      bootstrap:
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-bootstrap.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      brokers:
      - broker: 0
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-0.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 1
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-1.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 2
        dnsAnnotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-2.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
# ...
```

3.1.5.3.2. 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**.

3.1.5.3.3. Network policies

AMQ Streams automatically creates a **NetworkPolicy** resource for every listener that is enabled on a Kafka broker. By default, a **NetworkPolicy** grants access to a listener to all applications and namespaces. If you want to restrict access to a listener to only selected applications or namespaces, use the **networkPolicyPeers** field. Each listener can have a different **networkPolicyPeers** configuration.

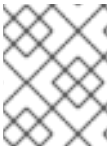
The following example shows a **networkPolicyPeers** configuration for a **plain** and a **tls** listener:

```
# ...
listeners:
    plain:
      authentication:
        type: scram-sha-512
      networkPolicyPeers:
        - podSelector:
            matchLabels:
              app: kafka-sasl-consumer
        - podSelector:
            matchLabels:
              app: kafka-sasl-producer
    tls:
      authentication:
        type: tls
      networkPolicyPeers:
        - namespaceSelector:
            matchLabels:
              project: myproject
        - namespaceSelector:
            matchLabels:
              project: myproject2
# ...
```

In the above example:

- Only application pods matching the labels **app: kafka-sasl-consumer** and **app: kafka-sasl-producer** can connect to the **plain** listener. The application pods must be running in the same namespace as the Kafka broker.
- Only application pods running in namespaces matching the labels **project: myproject** and **project: myproject2** can connect to the **tls** listener.

The syntax of the **networkPolicyPeers** field is the same as the **from** field in the **NetworkPolicy** resource in Kubernetes. For more information about the schema, see NetworkPolicyPeer API reference and the **KafkaListeners** schema reference.



NOTE

Your configuration of OpenShift must support Ingress NetworkPolicies in order to use network policies in AMQ Streams.

3.1.5.4. 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      plain: {}
    # ...
  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 **KafkaListeners** schema reference.

3.1.5.5. 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/v1beta1
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.

3.1.5.6. Accessing Kafka using loadbalancers

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/v1beta1
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.

3.1.5.7. Accessing Kafka using node ports

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/v1beta1
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.

3.1.5.8. Restricting access to Kafka listeners using networkPolicyPeers

You can restrict access to a listener to only selected applications by using the **networkPolicyPeers** field.

Prerequisites

- An OpenShift cluster with support for Ingress NetworkPolicies.
- The Cluster Operator is running.

Procedure

1. Open the **Kafka** resource.
2. In the **networkPolicyPeers** field, define the application pods or namespaces that will be allowed to access the Kafka cluster.
   For example, to configure a **tls** listener to allow connections only from application pods with the label **app** set to **kafka-client**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      tls:
        networkPolicyPeers:
          - podSelector:
              matchLabels:
                app: kafka-client
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.
   On OpenShift use **oc apply**:

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

Additional resources

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

3.1.6. 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.

3.1.6.1. Authentication

Authentication is configured as part of the listener configuration in the **authentication** property. The authentication mechanism is defined by the **type** field.

When the **authentication** property is missing, no authentication is enabled on a given listener. The listener will accept all connections without authentication.

Supported authentication mechanisms:

- TLS client authentication
- SASL SCRAM-SHA-512

3.1.6.1.1. TLS client authentication

TLS Client authentication is 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
# ...
```

3.1.6.2. Configuring authentication in Kafka brokers

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.kafka.listeners** property in the **Kafka** resource, add the **authentication** field to the listeners for which you want to enable authentication. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      tls:
        authentication:
          type: tls
    # ...
  zookeeper:
    # ...
```

3. Apply the new configuration to create or update the resource.
On OpenShift, use **oc apply**:

```
oc apply -f kafka.yaml
```

where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

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.

3.1.6.3. 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.

3.1.6.3.1. 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
# ...
```

3.1.6.4. 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/v1beta1
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.

3.1.7. Zookeeper replicas

Zookeeper clusters or ensembles usually run with an odd number of nodes, typically three, five, or seven.

The majority of nodes must be available in order to maintain an effective quorum. If the Zookeeper cluster loses its quorum, it will stop responding to clients and the Kafka brokers will stop working. Having a stable and highly available Zookeeper cluster is crucial for AMQ Streams.

Three-node cluster

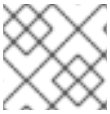
A three-node Zookeeper cluster requires at least two nodes to be up and running in order to maintain the quorum. It can tolerate only one node being unavailable.

Five-node cluster

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

Seven-node cluster

A seven-node Zookeeper cluster 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.

3.1.7.1. 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
```

```
# ...
replicas: 3
# ...
```

3.1.7.2. Changing the number of Zookeeper replicas

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.zookeeper.replicas** property in the **Kafka** resource, enter the number of replicated Zookeeper servers. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    replicas: 3
    # ...
```

3. Apply the new configuration to create or update the resource.
On OpenShift, use **oc apply**:

```
oc apply -f kafka.yaml
```

where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

3.1.8. 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 the Zookeeper documentation.

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.

3.1.8.1. Zookeeper configuration

Zookeeper nodes are configured using the **config** property in **Kafka.spec.zookeeper**. This property contains the Zookeeper configuration options as keys. The values can be described using 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 is ignored and a warning message is printed to the Custer Operator log file. All other options are 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
    # ...
```

3.1.8.2. Configuring Zookeeper

Prerequisites

- An OpenShift cluster is available.
- The Cluster Operator is running.

Procedure

1. Open the YAML configuration file that contains the **Kafka** resource specifying the cluster deployment.
2. In the **spec.zookeeper.config** property in the **Kafka** resource, enter one or more Zookeeper configuration settings. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
    # ...
```

3. Apply the new configuration to create or update the resource.
   On OpenShift, use **oc apply**:

```
oc apply -f kafka.yaml
```

   where ***kafka.yaml*** is the YAML configuration file for the resource that you want to configure; for example, **kafka-persistent.yaml**.

3.1.9. Zookeeper connection

Zookeeper services are secured with encryption and authentication and are not intended to be used by external applications that are not part of AMQ Streams.

However, if you want to use Kafka CLI tools that require a connection to Zookeeper, such as the **kafka-topics** tool, you can use a terminal inside a Kafka container and connect to the local end of the TLS tunnel to Zookeeper by using **localhost:2181** as the Zookeeper address.

3.1.9.1. Connecting to Zookeeper from a terminal

Open a terminal inside a Kafka container to use Kafka CLI tools that require a Zookeeper connection.

Prerequisites

- An OpenShift cluster is available.
- A kafka cluster is running.
- The Cluster Operator is running.

Procedure

1. Open the terminal using the OpenShift console or run the **exec** command from your CLI. For example:

   ```
   oc exec -ti my-cluster-kafka-0 -- bin/kafka-topics.sh --list --zookeeper localhost:2181
   ```

   Be sure to use **localhost:2181**.

   You can now run Kafka commands to Zookeeper.

3.1.10. 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*.

3.1.10.1. Configuration

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

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

- **tlsSidecar**
- **topicOperator**
- **userOperator**
- **template**

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 3.1.18, "TLS sidecar".

The **template** property can be used to configure details of the Entity Operator pod, such as labels, annotations, affinity, tolerations and so on.

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

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

Example of basic configuration enabling both operators

```
apiVersion: kafka.strimzi.io/v1beta1
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.

3.1.10.1.1. Topic Operator

Topic Operator deployment can be configured using additional options inside the **topicOperator** object. The 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 **90**.

zookeeperSessionTimeoutSeconds

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

topicMetadataMaxAttempts

The number of attempts at getting topic metadata from Kafka. The time between each attempt is defined as an exponential back-off. Consider increasing this value when topic creation could take more time due to the number of partitions or replicas. Default **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 3.1.17, "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 3.1.11, "CPU and memory resources".

logging

The **logging** property configures the logging of the Topic Operator.
The Topic Operator has its own configurable logger:

- **rootLogger.level**

Example of Topic Operator configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

3.1.10.1.2. User Operator

User Operator deployment can be configured using additional options inside the **userOperator** object. The 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 **120**.

zookeeperSessionTimeoutSeconds

The Zookeeper session timeout in seconds. Default **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 3.1.17, "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 3.1.11, "CPU and memory resources".

logging

The **logging** property configures the logging of the User Operator.
The User Operator has its own configurable logger:

- **rootLogger.level**

Example of Topic Operator configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-user-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

3.1.10.2. 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/v1beta1
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
```

3.1.11. CPU and memory resources

For every deployed container, AMQ Streams allows you to request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

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

3.1.11.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the 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**
- **KafkaBridge.spec**

Additional resources

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

3.1.11.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.

IMPORTANT

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

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**
- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

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

3.1.11.1.2. Resource limits

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

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

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

3.1.11.1.3. 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.

Example CPU units

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

NOTE

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

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

3.1.11.1.4. 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
# ...
```

Additional resources

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

3.1.11.2. 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/v1beta1
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.

3.1.12. Logging

This section provides information on loggers and how to configure log levels.

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

3.1.12.1. Kafka loggers

Kafka has its own configurable loggers:

- **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**

3.1.12.2. Specifying inline logging

Procedure

1. Edit the YAML file to specify the loggers and logging level for the required components. For example, the logging level here is set to INFO:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    logging:
      type: inline
      loggers:
        logger.name: "INFO"
    # ...
  zookeeper:
    # ...
    logging:
      type: inline
      loggers:
        logger.name: "INFO"
    # ...
  entityOperator:
    # ...
    topicOperator:
      # ...
      logging:
        type: inline
        loggers:
          logger.name: "INFO"
    # ...
    # ...
    userOperator:
      # ...
      logging:
        type: inline
        loggers:
          logger.name: "INFO"
    # ...
```

You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

For more information about the log levels, see the 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
```

3.1.12.3. Specifying an external ConfigMap for logging

Procedure

1. Edit the YAML file to specify the name of the **ConfigMap** to use for the required components.
For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    logging:
      type: external
      name: customConfigMap
    # ...
```

Remember to place your custom ConfigMap under the **log4j.properties** or **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
```

Garbage collector (GC) logging can also be enabled (or disabled). For more information on GC, see Section 3.1.16.1, “JVM configuration”

3.1.13. 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.

3.1.13.1. 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 represents 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/v1beta1
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 3.1.17, “Container images”.

3.1.14. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts 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.

3.1.14.1. Healthcheck configurations

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

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

3.1.14.2. 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/v1beta1
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
```

3.1.15. 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.

3.1.15.1. Metrics configuration

Prometheus metrics are 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/v1beta1
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/v1beta1
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:
    # ...
```

3.1.15.2. 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/v1beta1
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
```

3.1.16. JVM Options

Apache Kafka and Apache Zookeeper run inside a Java Virtual Machine (JVM). JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure some of these options.

3.1.16.1. 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

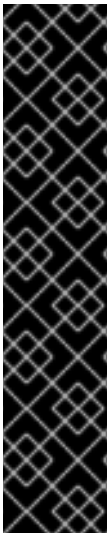
-Xms configures the minimum initial allocation heap size when the JVM starts. **-Xmx** configures the maximum heap size.

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.



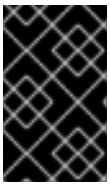
IMPORTANT

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.

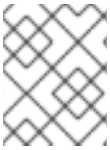
-server

-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

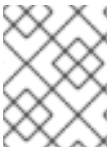
-XX object can be used for configuring advanced runtime options of a JVM. The **-server** and **-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

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.

3.1.16.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is enabled by default. To disable it, set the **gcLoggingEnabled** property as follows:

Example of disabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: false
# ...
```

3.1.16.2. 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/v1beta1
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
```

3.1.17. 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.

3.1.17.1. 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**

- **KafkaBridge.spec**

3.1.17.1.1. Configuring the Kafka.spec.kafka.image property

The **Kafka.spec.kafka.image** property functions differently from the others, because AMQ Streams supports multiple versions of Kafka, each requiring the own image. The **STRIMZI_KAFKA_IMAGES** environment variable of the Cluster Operator configuration is used to provide a mapping between Kafka versions and the corresponding images. This is used in combination with the **Kafka.spec.kafka.image** and **Kafka.spec.kafka.version** properties as follows:

- If neither **Kafka.spec.kafka.image** nor **Kafka.spec.kafka.version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- If **Kafka.spec.kafka.image** is given but **Kafka.spec.kafka.version** is not then the given image will be used and the **version** will be assumed to be the Cluster Operator's default Kafka version.
- If **Kafka.spec.kafka.version** is given but **Kafka.spec.kafka.image** is not then image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- Both **Kafka.spec.kafka.version** and **Kafka.spec.kafka.image** are given the given image will be used, and it will be assumed to contain a Kafka broker with the given version.

WARNING

It is best to provide just **Kafka.spec.kafka.version** and leave the **Kafka.spec.kafka.image** property unspecified. This reduces the chances of making a mistake in configuring the **Kafka** resource. If you need to change the images used for different versions of Kafka, it is better to configure the Cluster Operator's **STRIMZI_KAFKA_IMAGES** environment variable.

3.1.17.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property 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 broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Zookeeper nodes:
  1. Container image specified in the **STRIMZI_DEFAULT_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.

- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Kafka Connect:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.1.17.2. 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/v1beta1
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
```

3.1.18. TLS sidecar

A sidecar is a container that runs in a pod but serves a supporting purpose. In AMQ Streams, the TLS sidecar uses TLS to encrypt and decrypt all communication between the various components and Zookeeper. Zookeeper does not have native TLS support.

The TLS sidecar is used in:

- Kafka brokers
- Zookeeper nodes

- Entity Operator

3.1.18.1. 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 the following additional options:

- **image**
- **resources**
- **logLevel**
- **readinessProbe**
- **livenessProbe**

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 3.1.17, “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*.

For more information about configuring the **readinessProbe** and **livenessProbe** properties for the healthchecks, see Section 3.1.14.1, “Healthcheck configurations”.

Example of TLS sidecar configuration

```
apiVersion: kafka.strimzi.io/v1beta1
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
      readinessProbe:
        initialDelaySeconds: 15
        timeoutSeconds: 5
      livenessProbe:
        initialDelaySeconds: 15
        timeoutSeconds: 5
    # ...
  zookeeper:
    # ...
```

3.1.18.2. 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/v1beta1
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
```

3.1.19. 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.

3.1.19.1. Scheduling pods based on other applications

3.1.19.1.1. 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.

3.1.19.1.2. Affinity

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

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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 .

3.1.19.1.3. 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

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

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

3.1.19.2. Scheduling pods to specific nodes

3.1.19.2.1. 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.

3.1.19.2.2. Affinity

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

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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 .

3.1.19.2.3. 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/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
         pod:
           affinity:
             nodeAffinity:
               requiredDuringSchedulingIgnoredDuringExecution:
                 nodeSelectorTerms:
                   - matchExpressions:
                     - key: node-type
                       operator: In
                       values:
   ```

```
                - fast-network
  # ...
zookeeper:
  # ...
```

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

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

3.1.19.3. Using dedicated nodes

3.1.19.3.1. 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.

3.1.19.3.2. Affinity

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

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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 .

3.1.19.3.3. Tolerations

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

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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

3.1.19.3.4. 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
```

```
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: dedicated
                operator: In
                values:
                - Kafka
  # ...
zookeeper:
  # ...
```

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

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

3.1.20. 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 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. When 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 2.3, “Cluster Operator”.
- For more information about deploying the Kafka cluster on OpenShift, see Section 2.4.1, “Deploying the Kafka cluster to OpenShift”.

3.1.21. 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 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. When 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 2.3, “Cluster Operator”.
- For more information about deploying the Zookeeper cluster, see Section 2.4.1, “Deploying the Kafka cluster to OpenShift”.

3.1.22. Scaling clusters

3.1.22.1. Scaling Kafka clusters

3.1.22.1.1. 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.

3.1.22.1.2. 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.

3.1.22.2. 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.

3.1.22.2.1. 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.

3.1.22.2.2. Reassigning partitions between JBOD volumes

When using JBOD storage in your Kafka cluster, you can choose to reassign the partitions between specific volumes and their log directories (each volume has a single log directory). To reassign a partition to a specific volume, add the **log_dirs** option to *<PartitionObjects>* in the reassignment JSON file.

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

The **log_dirs** object should contain the same number of log directories as the number of replicas specified in the **replicas** object. The value should be either an absolute path to the log directory, or the **any** keyword.

For example:

```
{
      "topic": "topic-a",
```

```
    "partition": 4,
    "replicas": [2,4,7].
    "log_dirs": [ "/var/lib/kafka/data-0/kafka-log2", "/var/lib/kafka/data-0/kafka-log4",
"/var/lib/kafka/data-0/kafka-log7" ]
}
```

3.1.22.3. 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
```

3.1.22.4. Creating reassignment JSON files manually

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

3.1.22.5. 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.

3.1.22.6. 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.

3.1.22.7. 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)**.

3.1.23. 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.
2. Annotate the **Pod** resource in OpenShift.
   On OpenShift use **oc annotate**:

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.3, "Cluster Operator".
- For more information about deploying the Kafka cluster on OpenShift, see Section 2.4.1, "Deploying the Kafka cluster to OpenShift".

3.1.24. 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.
2. Annotate the **Pod** resource in OpenShift.
   On OpenShift use **oc annotate**:

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

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

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.3, "Cluster Operator".
- For more information about deploying the Zookeeper cluster on OpenShift, see Section 2.4.1, "Deploying the Kafka cluster to OpenShift".

3.1.25. Maintenance time windows for rolling updates

Maintenance time windows allow you to schedule certain rolling updates of your Kafka and Zookeeper clusters to start at a convenient time.

3.1.25.1. Maintenance time windows overview

In most cases, the Cluster Operator only updates your Kafka or Zookeeper clusters in response to changes to the corresponding **Kafka** resource. This enables you to plan when to apply changes to a **Kafka** resource to minimize the impact on Kafka client applications.

However, some updates to your Kafka and Zookeeper clusters can happen without any corresponding change to the **Kafka** resource. For example, the Cluster Operator will need to perform a rolling restart if a CA (Certificate Authority) certificate that it manages is close to expiry.

While a rolling restart of the pods should not affect *availability* of the service (assuming correct broker and topic configurations), it could affect *performance* of the Kafka client applications. Maintenance time windows allow you to schedule such spontaneous rolling updates of your Kafka and Zookeeper clusters to start at a convenient time. If maintenance time windows are not configured for a cluster then it is possible that such spontaneous rolling updates will happen at an inconvenient time, such as during a predictable period of high load.

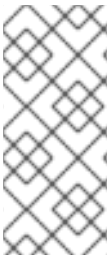
3.1.25.2. Maintenance time window definition

You configure maintenance time windows by entering an array of strings in the **Kafka.spec.maintenanceTimeWindows** property. Each string is a cron expression interpreted as being in UTC (Coordinated Universal Time, which for practical purposes is the same as Greenwich Mean Time).

The following example configures a single maintenance time window that starts at midnight and ends at 01:59am (UTC), on Sundays, Mondays, Tuesdays, Wednesdays, and Thursdays:

```
# ...
maintenanceTimeWindows:
  - "* * 0-1 ? * SUN,MON,TUE,WED,THU *"
# ...
```

In practice, maintenance windows should be set in conjunction with the **Kafka.spec.clusterCa.renewalDays** and **Kafka.spec.clientsCa.renewalDays** properties of the **Kafka** resource, to ensure that the necessary CA certificate renewal can be completed in the configured maintenance time windows.



NOTE

AMQ Streams does not schedule maintenance operations exactly according to the given windows. Instead, for each reconciliation, it checks whether a maintenance window is currently "open". This means that the start of maintenance operations within a given time window can be delayed by up to the Cluster Operator reconciliation interval. Maintenance time windows must therefore be at least this long.

Additional resources

- For more information about the Cluster Operator configuration, see Section 4.1.6, “Cluster Operator Configuration”.

3.1.25.3. Configuring a maintenance time window

You can configure a maintenance time window for rolling updates triggered by supported processes.

Prerequisites

- An OpenShift cluster.
- The Cluster Operator is running.

Procedure

1. Add or edit the **maintenanceTimeWindows** property in the **Kafka** resource. For example to allow maintenance between 0800 and 1059 and between 1400 and 1559 you would set the **maintenanceTimeWindows** as shown below:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  maintenanceTimeWindows:
    - "* * 8-10 * * ?"
    - "* * 14-15 * * ?"
```

2. Create or update the resource.
On OpenShift, use **oc apply**:

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

Additional resources

- Performing a rolling update of a Kafka cluster, see Section 3.1.20, "Performing a rolling update of a Kafka cluster"
- Performing a rolling update of a Zookeeper cluster, see Section 3.1.21, "Performing a rolling update of a Zookeeper cluster"

3.1.26. 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.

***cluster-name*-kafka**

Pod Disruption Budget configured for 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*-zookeeper**

Pod Disruption Budget configured for the Zookeeper nodes.

***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.

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

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

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-*id*-*cluster-name*-kafka-*idx*

Persistent Volume Claim for the volume ***id*** used for storing data for the Kafka broker pod ***idx***. This resource is only created if persistent storage is selected for JBOD volumes when 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 C.55, "**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 resources to be labeled as required.

3.2.1. Replicas

Kafka Connect clusters can run multiple of nodes. The number of nodes is defined in the **KafkaConnect** and **KafkaConnectS2I** resources. Running a 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. If a node where Kafka Connect is deployed to crashes, OpenShift will automatically reschedule 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 up and running already.

3.2.1.1. Configuring the number of nodes

The number of Kafka Connect nodes is 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/v1beta1
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

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

The list of bootstrap servers is configured in the **bootstrapServers** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The servers must be defined as a comma-separated list specifying 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 the 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/v1beta1
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 tries to connect to Kafka brokers using a plain text connection. If you prefer to use TLS, additional configuration is required.

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 must be stored in X509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1beta1
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/v1beta1
kind: KafkaConnectS2I
metadata:
  name: my-cluster
spec:
  # ...
```

```
tls:
  trustedCertificates:
    - secretName: my-secret
      certificate: ca.crt
    - secretName: my-secret
      certificate: ca2.crt
# ...
```

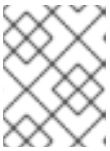
3.2.3.2. Configuring TLS in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** for the certificate used for TLS Server Authentication, and the key under which the certificate is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the TLS certificate used in authentication in a file and create a **Secret**.

   NOTE

   The secrets created by the Cluster Operator for Kafka cluster may be used directly.

   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/v1beta1
   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 authentication. Authentication is enabled through the **KafkaConnect** and **KafkaConnectS2I** resources.

3.2.4.1. Authentication support in Kafka Connect

Authentication is configured through 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 the SCRAM-SHA-512 mechanism
- SASL-based authentication using the PLAIN mechanism

3.2.4.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.

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 TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
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. SASL based SCRAM-SHA-512 authentication

To configure Kafka Connect to use SASL-based SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. This authentication mechanism requires a username and password.

- Specify the username in the **username** property.

- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

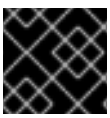
3.2.4.1.3. SASL based PLAIN authentication

To configure Kafka Connect to use SASL-based PLAIN authentication, set the **type** property to **plain**. This authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of such a **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.



IMPORTANT

Do not specify the actual password in the **password** field.

An example showing SASL based PLAIN client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
```

```
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: plain
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

3.2.4.2. Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** with the public and private keys used for TLS Client Authentication, and the keys under which they are stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the keys used for authentication in a file and create the **Secret**.

   NOTE

   Secrets created by the User Operator may be used.

   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/v1beta1
   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
- If they exist, the name of the **Secret** with the password used for authentication and the key under which the password is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare a file with the password used in authentication and create the **Secret**.

NOTE

Secrets created by the User Operator may be used.

   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/v1beta1
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 certain options listed in Apache Kafka documentation.

Configuration options that cannot be configured relate to:

- 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

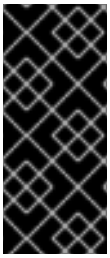
Kafka Connect is configured using the **config** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. This property contains the Kafka Connect configuration options as keys. The values can be one of the following JSON types:

- String
- Number
- Boolean

You can specify and configure the options listed in the Apache Kafka documentation with the exception of those options that are managed directly by AMQ Streams. Specifically, 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 a forbidden option is present in the **config** property, it is ignored and a warning message is printed to the Custer Operator log file. All other options are passed to Kafka Connect.



IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Connect cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaConnect.spec.config** or **KafkaConnectS2I.spec.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Connect nodes.

Certain 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**

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

Example Kafka Connect configuration

```
apiVersion: kafka.strimzi.io/v1beta1
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
    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/v1beta1
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
    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 request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

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

3.2.6.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the following resources:

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

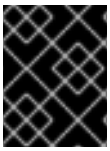
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.spec**

Additional resources

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

3.2.6.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.



IMPORTANT

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

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**
- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

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

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 always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

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

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.

Example CPU units

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

NOTE

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

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

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
# ...
```

Additional resources

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

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/v1beta1
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

This section provides information on loggers and how to configure log levels.

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

3.2.7.1. Kafka Connect loggers

Kafka Connect has its own configurable loggers:

- **connect.root.logger.level**
- **log4j.logger.org.apache.zookeeper**
- **log4j.logger.org.I0Itec.zkclient**
- **log4j.logger.org.reflections**

3.2.7.2. Specifying inline logging

Procedure

1. Edit the YAML file to specify the loggers and logging level for the required components.
   For example, the logging level here is set to INFO:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
spec:
  # ...
  logging:
    type: inline
    loggers:
      logger.name: "INFO"
  # ...
```

   You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

   For more information about the log levels, see the 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
```

3.2.7.3. Specifying an external ConfigMap for logging

Procedure

1. Edit the YAML file to specify the name of the **ConfigMap** to use for the required components.
   For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
spec:
  # ...
  logging:
```

```
      type: external
      name: customConfigMap
    # ...
```

Remember to place your custom ConfigMap under the **log4j.properties** or **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
```

Garbage collector (GC) logging can also be enabled (or disabled). For more information on GC, see Section 3.2.10.1, "JVM configuration"

3.2.8. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts 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.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.zookeeper.tlsSidecar**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.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/v1beta1
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 are 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/v1beta1
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/v1beta1
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/v1beta1
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 run inside a Java Virtual Machine (JVM). JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure 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

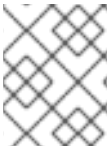
-XX object can be used for configuring advanced runtime options of a JVM. The **-server** and **-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

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.

3.2.10.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is enabled by default. To disable it, set the **gcLoggingEnabled** property as follows:

Example of disabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: false
# ...
```

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/v1beta1
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**
- **KafkaBridge.spec**

3.2.11.1.1. Configuring the **Kafka.spec.kafka.image** property

The **Kafka.spec.kafka.image** property functions differently from the others, because AMQ Streams supports multiple versions of Kafka, each requiring the own image. The **STRIMZI_KAFKA_IMAGES** environment variable of the Cluster Operator configuration is used to provide a mapping between Kafka versions and the corresponding images. This is used in combination with the **Kafka.spec.kafka.image** and **Kafka.spec.kafka.version** properties as follows:

- If neither **Kafka.spec.kafka.image** nor **Kafka.spec.kafka.version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- If **Kafka.spec.kafka.image** is given but **Kafka.spec.kafka.version** is not then the given image will be used and the **version** will be assumed to be the Cluster Operator's default Kafka version.
- If **Kafka.spec.kafka.version** is given but **Kafka.spec.kafka.image** is not then image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- Both **Kafka.spec.kafka.version** and **Kafka.spec.kafka.image** are given the given image will be used, and it will be assumed to contain a Kafka broker with the given version.

WARNING

It is best to provide just **Kafka.spec.kafka.version** and leave the **Kafka.spec.kafka.image** property unspecified. This reduces the chances of making a mistake in configuring the **Kafka** resource. If you need to change the images used for different versions of Kafka, it is better to configure the Cluster Operator's **STRIMZI_KAFKA_IMAGES** environment variable.

3.2.11.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property 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 broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Zookeeper nodes:
  1. Container image specified in the **STRIMZI_DEFAULT_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.

- For Kafka Connect:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** 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/v1beta1
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/v1beta1
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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**

- **KafkaBridge.spec.template.pod**

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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                  - key: node-type
                    operator: In
                    values:
                    - fast-network
    # ...
  zookeeper:
    # ...
```

3. Create or update the resource.
   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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**

- **KafkaBridge.spec.template.pod**

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 can be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
              - matchExpressions:
                - key: dedicated
                  operator: In
                  values:
                  - Kafka
    # ...
  zookeeper:
    # ...
```

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

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

3.2.13. Using external configuration and secrets

Kafka Connect connectors are configured using an HTTP REST interface. The connector configuration is passed to Kafka Connect as part of an HTTP request and stored within Kafka itself.

Some parts of the configuration of a Kafka Connect connector can be externalized using ConfigMaps or Secrets. You can then reference the configuration values in HTTP REST commands (this keeps the configuration separate and more secure, if needed). This method applies especially to confidential data, such as usernames, passwords, or certificates.

ConfigMaps and Secrets are standard OpenShift resources used for storing of configurations and confidential data.

3.2.13.1. Storing connector configurations externally

You can mount ConfigMaps or Secrets into a Kafka Connect pod as volumes or environment variables. Volumes and environment variables are configured in the **externalConfiguration** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**.

3.2.13.1.1. External configuration as environment variables

The **env** property is used to specify one or more environment variables. These variables can contain a value from either a ConfigMap or a Secret.

NOTE

The names of user-defined environment variables cannot start with **KAFKA_** or **STRIMZI_**.

To mount a value from a Secret to an environment variable, use the **valueFrom** property and the **secretKeyRef** as shown in the following example.

Example of an environment variable set to a value from a Secret

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: my-key
```

A common use case for mounting Secrets to environment variables is when your connector needs to communicate with Amazon AWS and needs to read the **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY** environment variables with credentials.

To mount a value from a ConfigMap to an environment variable, use **configMapKeyRef** in the **valueFrom** property as shown in the following example.

Example of an environment variable set to a value from a ConfigMap

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

3.2.13.1.2. External configuration as volumes

You can also mount ConfigMaps or Secrets to a Kafka Connect pod as volumes. Using volumes instead of environment variables is useful in the following scenarios:

- Mounting truststores or keystores with TLS certificates
- Mounting a properties file that is used to configure Kafka Connect connectors

In the **volumes** property of the **externalConfiguration** resource, list the ConfigMaps or Secrets that will be mounted as volumes. Each volume must specify a name in the **name** property and a reference to ConfigMap or Secret.

Example of volumes with external configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    volumes:
      - name: connector1
        configMap:
          name: connector1-configuration
      - name: connector1-certificates
        secret:
          secretName: connector1-certificates
```

The volumes will be mounted inside the Kafka Connect containers in the path **/opt/kafka/external-configuration/*<volume-name>***. For example, the files from a volume named **connector1** would appear in the directory **/opt/kafka/external-configuration/connector1**.

The **FileConfigProvider** has to be used to read the values from the mounted properties files in connector configurations.

3.2.13.2. Mounting Secrets as environment variables

You can create an OpenShift Secret and mount it to Kafka Connect as an environment variable.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing the information that will be mounted as an environment variable. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-creds
type: Opaque
data:
  awsAccessKey: QUtJQVhYWFhYWFhYWFhYWFg=
  awsSecretAccessKey: Ylhsd1lYTnpkMjl5WkE=
```

2. Create or edit the Kafka Connect resource. Configure the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: AWS_ACCESS_KEY_ID
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsAccessKey
      - name: AWS_SECRET_ACCESS_KEY
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsSecretAccessKey
```

3. Apply the changes to your Kafka Connect deployment.
On OpenShift use **oc apply**:

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

The environment variables are now available for use when developing your connectors.

Additional resources

- For more information about external configuration in Kafka Connect, see Section C.63, "**ExternalConfiguration** schema reference".

3.2.13.3. Mounting Secrets as volumes

You can create an OpenShift Secret, mount it as a volume to Kafka Connect, and then use it to configure a Kafka Connect connector.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing a properties file that defines the configuration options for your connector configuration. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
```

```
  connector.properties: |-
    dbUsername: my-user
    dbPassword: my-password
```

2. Create or edit the Kafka Connect resource. Configure the **FileConfigProvider** in the **config** section and the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: file
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
  #...
  externalConfiguration:
    volumes:
      - name: connector-config
        secret:
          secretName: mysecret
```

3. Apply the changes to your Kafka Connect deployment.
On OpenShift use **oc apply**:

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

4. Use the values from the mounted properties file in your JSON payload with connector configuration. For example:

```
{
  "name":"my-connector",
  "config":{
    "connector.class":"MyDbConnector",
    "tasks.max":"3",
    "database": "my-postgresql:5432"
    "username":"${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbUsername}",
    "password":"${file:/opt/kafka/external-configuration/connector-config/connector.properties:dbPassword}",
    # ...
  }
}
```

Additional resources

- For more information about external configuration in Kafka Connect, see Section C.63, "**ExternalConfiguration** schema reference".

3.2.14. 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.

***connect-cluster-name*-connect**

Pod Disruption Budget configured for the Kafka Connect worker nodes.

3.3. KAFKA CONNECT CLUSTER WITH SOURCE2IMAGE SUPPORT

The full schema of the **KafkaConnectS2I** resource is described in the Section C.69, "**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 resources to be labeled as required.

3.3.1. Replicas

Kafka Connect clusters can run multiple of nodes. The number of nodes is defined in the **KafkaConnect** and **KafkaConnectS2I** resources. Running a 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. If a node where Kafka Connect is deployed to crashes, OpenShift will automatically reschedule 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 up and running already.

3.3.1.1. Configuring the number of nodes

The number of Kafka Connect nodes is 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/v1beta1
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

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

The list of bootstrap servers is configured in the **bootstrapServers** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. The servers must be defined as a comma-separated list specifying 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 the 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/v1beta1
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 tries to connect to Kafka brokers using a plain text connection. If you prefer to use TLS, additional configuration is required.

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 must be stored in X509 format.

An example showing TLS configuration with multiple certificates

```
apiVersion: kafka.strimzi.io/v1beta1
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/v1beta1
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
- If they exist, the name of the **Secret** for the certificate used for TLS Server Authentication, and the key under which the certificate is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the TLS certificate used in authentication in a file and create a **Secret**.

NOTE

The secrets created by the Cluster Operator for Kafka cluster may be used directly.

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/v1beta1
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 authentication. Authentication is enabled through the **KafkaConnect** and **KafkaConnectS2I** resources.

3.3.4.1. Authentication support in Kafka Connect

Authentication is configured through 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 the SCRAM-SHA-512 mechanism
- SASL-based authentication using the PLAIN mechanism

3.3.4.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.

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 TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
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. SASL based SCRAM-SHA-512 authentication

To configure Kafka Connect to use SASL-based SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. This authentication mechanism requires a username and password.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: scram-sha-512
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

3.3.4.1.3. SASL based PLAIN authentication

To configure Kafka Connect to use SASL-based PLAIN authentication, set the **type** property to **plain**. This authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of such a **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example showing SASL based PLAIN client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-cluster
spec:
  # ...
  authentication:
    type: plain
    username: my-connect-user
    passwordSecret:
      secretName: my-connect-user
      password: my-connect-password-key
  # ...
```

3.3.4.2. Configuring TLS client authentication in Kafka Connect

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** with the public and private keys used for TLS Client Authentication, and the keys under which they are stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the keys used for authentication in a file and create the **Secret**.

NOTE

Secrets created by the User Operator may be used.

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/v1beta1
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
- If they exist, the name of the **Secret** with the password used for authentication and the key under which the password is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare a file with the password used in authentication and create the **Secret**.

NOTE

Secrets created by the User Operator may be used.

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/v1beta1
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 certain options listed in Apache Kafka documentation.

Configuration options that cannot be configured relate to:

- 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

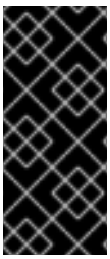
Kafka Connect is configured using the **config** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**. This property contains the Kafka Connect configuration options as keys. The values can be one of the following JSON types:

- String
- Number
- Boolean

You can specify and configure the options listed in the Apache Kafka documentation with the exception of those options that are managed directly by AMQ Streams. Specifically, 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 a forbidden option is present in the **config** property, it is ignored and a warning message is printed to the Custer Operator log file. All other options are passed to Kafka Connect.



IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Connect cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaConnect.spec.config** or **KafkaConnectS2I.spec.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Connect nodes.

Certain 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**

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

Example Kafka Connect configuration

```
apiVersion: kafka.strimzi.io/v1beta1
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
    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/v1beta1
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
    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 request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

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

3.3.6.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the following resources:

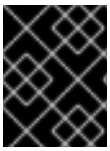
- **Kafka.spec.kafka**
- **Kafka.spec.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.zookeeper.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **Kafka.spec.entityOperator.tlsSidecar**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**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

Additional resources

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

3.3.6.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.



IMPORTANT

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

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**

- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

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

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 always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

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

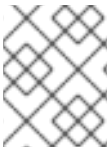
3.3.6.1.3. Supported CPU formats

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.

Example CPU units

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



NOTE

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

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

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
# ...
```

Additional resources

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

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/v1beta1
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

This section provides information on loggers and how to configure log levels.

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

3.3.7.1. Kafka Connect with Source2Image loggers

Kafka Connect with Source2Image support has its own configurable loggers:

- **connect.root.logger.level**
- **log4j.logger.org.apache.zookeeper**
- **log4j.logger.org.I0Itec.zkclient**
- **log4j.logger.org.reflections**

3.3.7.2. Specifying inline logging

Procedure

1. Edit the YAML file to specify the loggers and logging level for the required components.
   For example, the logging level here is set to INFO:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
spec:
  # ...
  logging:
    type: inline
    loggers:
      logger.name: "INFO"
  # ...
```

You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

For more information about the log levels, see the 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
```

3.3.7.3. Specifying an external ConfigMap for logging

Procedure

1. Edit the YAML file to specify the name of the **ConfigMap** to use for the required components.
For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnectS2I
spec:
  # ...
  logging:
    type: external
    name: customConfigMap
  # ...
```

Remember to place your custom ConfigMap under the **log4j.properties** or **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
```

Garbage collector (GC) logging can also be enabled (or disabled). For more information on GC, see Section 3.3.10.1, "JVM configuration"

3.3.8. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts 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.kafka.tlsSidecar**
- **Kafka.spec.zookeeper**
- **Kafka.spec.zookeeper.tlsSidecar**
- **Kafka.spec.entityOperator.tlsSidecar**
- **Kafka.spec.entityOperator.topicOperator**
- **Kafka.spec.entityOperator.userOperator**
- **KafkaConnect.spec**
- **KafkaConnectS2I.spec**
- **KafkaBridge.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/v1beta1
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 are 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/v1beta1
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/v1beta1
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/v1beta1
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 run inside a Java Virtual Machine (JVM). JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure 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

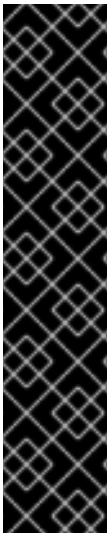
-Xms configures the minimum initial allocation heap size when the JVM starts. **-Xmx** configures the maximum heap size.

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

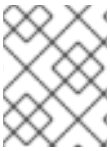
-XX object can be used for configuring advanced runtime options of a JVM. The **-server** and **-XX** options are used to configure the **KAFKA_JVM_PERFORMANCE_OPTS** option of Apache Kafka.

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.

3.3.10.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is enabled by default. To disable it, set the **gcLoggingEnabled** property as follows:

Example of disabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: false
# ...
```

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/v1beta1
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**

- **KafkaBridge.spec**

3.3.11.1.1. Configuring the Kafka.spec.kafka.image property

The **Kafka.spec.kafka.image** property functions differently from the others, because AMQ Streams supports multiple versions of Kafka, each requiring the own image. The **STRIMZI_KAFKA_IMAGES** environment variable of the Cluster Operator configuration is used to provide a mapping between Kafka versions and the corresponding images. This is used in combination with the **Kafka.spec.kafka.image** and **Kafka.spec.kafka.version** properties as follows:

- If neither **Kafka.spec.kafka.image** nor **Kafka.spec.kafka.version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- If **Kafka.spec.kafka.image** is given but **Kafka.spec.kafka.version** is not then the given image will be used and the **version** will be assumed to be the Cluster Operator's default Kafka version.
- If **Kafka.spec.kafka.version** is given but **Kafka.spec.kafka.image** is not then image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- Both **Kafka.spec.kafka.version** and **Kafka.spec.kafka.image** are given the given image will be used, and it will be assumed to contain a Kafka broker with the given version.

WARNING

It is best to provide just **Kafka.spec.kafka.version** and leave the **Kafka.spec.kafka.image** property unspecified. This reduces the chances of making a mistake in configuring the **Kafka** resource. If you need to change the images used for different versions of Kafka, it is better to configure the Cluster Operator's **STRIMZI_KAFKA_IMAGES** environment variable.

3.3.11.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property 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 broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Zookeeper nodes:
  1. Container image specified in the **STRIMZI_DEFAULT_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.

- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Kafka Connect:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** 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/v1beta1
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

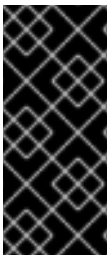
1. Edit the **image** property in the **Kafka**, **KafkaConnect** or **KafkaConnectS2I** resource. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
```

```
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: application
                    operator: In
                    values:
                      - postgresql
                      - mongodb
              topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
         pod:
           affinity:
             nodeAffinity:
               requiredDuringSchedulingIgnoredDuringExecution:
                 nodeSelectorTerms:
                   - matchExpressions:
                     - key: node-type
                       operator: In
                       values:
                       - fast-network
       # ...
     zookeeper:
       # ...
   ```

3. Create or update the resource.
   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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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 can be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
              - matchExpressions:
                - key: dedicated
                  operator: In
                  values:
                  - Kafka
    # ...
  zookeeper:
    # ...
```

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

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

3.3.13. Using external configuration and secrets

Kafka Connect connectors are configured using an HTTP REST interface. The connector configuration is passed to Kafka Connect as part of an HTTP request and stored within Kafka itself.

Some parts of the configuration of a Kafka Connect connector can be externalized using ConfigMaps or Secrets. You can then reference the configuration values in HTTP REST commands (this keeps the configuration separate and more secure, if needed). This method applies especially to confidential data, such as usernames, passwords, or certificates.

ConfigMaps and Secrets are standard OpenShift resources used for storing of configurations and confidential data.

3.3.13.1. Storing connector configurations externally

You can mount ConfigMaps or Secrets into a Kafka Connect pod as volumes or environment variables. Volumes and environment variables are configured in the **externalConfiguration** property in **KafkaConnect.spec** and **KafkaConnectS2I.spec**.

3.3.13.1.1. External configuration as environment variables

The **env** property is used to specify one or more environment variables. These variables can contain a value from either a ConfigMap or a Secret.

NOTE

The names of user-defined environment variables cannot start with **KAFKA_** or **STRIMZI_**.

To mount a value from a Secret to an environment variable, use the **valueFrom** property and the **secretKeyRef** as shown in the following example.

Example of an environment variable set to a value from a Secret

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: my-key
```

A common use case for mounting Secrets to environment variables is when your connector needs to communicate with Amazon AWS and needs to read the **AWS_ACCESS_KEY_ID** and **AWS_SECRET_ACCESS_KEY** environment variables with credentials.

To mount a value from a ConfigMap to an environment variable, use **configMapKeyRef** in the **valueFrom** property as shown in the following example.

Example of an environment variable set to a value from a ConfigMap

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

3.3.13.1.2. External configuration as volumes

You can also mount ConfigMaps or Secrets to a Kafka Connect pod as volumes. Using volumes instead of environment variables is useful in the following scenarios:

- Mounting truststores or keystores with TLS certificates
- Mounting a properties file that is used to configure Kafka Connect connectors

In the **volumes** property of the **externalConfiguration** resource, list the ConfigMaps or Secrets that will be mounted as volumes. Each volume must specify a name in the **name** property and a reference to ConfigMap or Secret.

Example of volumes with external configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    volumes:
      - name: connector1
        configMap:
          name: connector1-configuration
      - name: connector1-certificates
        secret:
          secretName: connector1-certificates
```

The volumes will be mounted inside the Kafka Connect containers in the path **/opt/kafka/external-configuration/*<volume-name>***. For example, the files from a volume named **connector1** would appear in the directory **/opt/kafka/external-configuration/connector1**.

The **FileConfigProvider** has to be used to read the values from the mounted properties files in connector configurations.

3.3.13.2. Mounting Secrets as environment variables

You can create an OpenShift Secret and mount it to Kafka Connect as an environment variable.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing the information that will be mounted as an environment variable. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: aws-creds
type: Opaque
data:
  awsAccessKey: QUtJQVhYWFhYWFhYWFhYWFg=
  awsSecretAccessKey: Ylhsd1IYTnpkMjI5WkE=
```

2. Create or edit the Kafka Connect resource. Configure the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: AWS_ACCESS_KEY_ID
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsAccessKey
      - name: AWS_SECRET_ACCESS_KEY
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsSecretAccessKey
```

3. Apply the changes to your Kafka Connect deployment.
On OpenShift use **oc apply**:

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

The environment variables are now available for use when developing your connectors.

Additional resources

- For more information about external configuration in Kafka Connect, see Section C.63, "**ExternalConfiguration** schema reference".

3.3.13.3. Mounting Secrets as volumes

You can create an OpenShift Secret, mount it as a volume to Kafka Connect, and then use it to configure a Kafka Connect connector.

Prerequisites

- A running Cluster Operator.

Procedure

1. Create a secret containing a properties file that defines the configuration options for your connector configuration. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |-
    dbUsername: my-user
    dbPassword: my-password
```

2. Create or edit the Kafka Connect resource. Configure the **FileConfigProvider** in the **config** section and the **externalConfiguration** section of the **KafkaConnect** or **KafkaConnectS2I** custom resource to reference the secret. For example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: file
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
  #...
  externalConfiguration:
    volumes:
      - name: connector-config
        secret:
          secretName: mysecret
```

3. Apply the changes to your Kafka Connect deployment.
On OpenShift use **oc apply**:

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

4. Use the values from the mounted properties file in your JSON payload with connector configuration. For example:

```
{
  "name":"my-connector",
```

```
  "config":{
    "connector.class":"MyDbConnector",
    "tasks.max":"3",
    "database": "my-postgresql:5432"
    "username":"${file:/opt/kafka/external-configuration/connector-
config/connector.properties:dbUsername}",
    "password":"${file:/opt/kafka/external-configuration/connector-
config/connector.properties:dbPassword}",
    # ...
  }
}
```

Additional resources

- For more information about external configuration in Kafka Connect, see Section C.63, "**ExternalConfiguration** schema reference".

3.3.14. 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.

***connect-cluster-name*-connect**

Pod Disruption Budget configured for the Kafka Connect worker nodes.

3.3.15. Creating a container image using OpenShift builds and Source-to-Image

You can use OpenShift builds and 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. Once built, container images are stored in OpenShift's local container image repository and are available for use in deployments.

A Kafka Connect builder image with S2I support is provided on the Red Hat Container Catalog as part of the **registry.redhat.io/amq7/amqstreams-kafka-22** image. This S2I image takes your binaries (with plug-ins and connectors) and stores them in the **/tmp/kafka-plugins/s2i** directory. It creates a new Kafka

Connect image from this directory, which can then be used with the Kafka Connect deployment. When started using the enhanced image, Kafka Connect loads any third-party plug-ins from the **/tmp/kafka-plugins/s2i** directory.

Procedure

1. On the command line, use the **oc apply** command to create and deploy a Kafka Connect S2I cluster:

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

2. Create a directory with Kafka Connect plug-ins:

```
$ 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. Use the **oc start-build** command to start a new build of the image using the prepared directory:

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

NOTE

The name of the build is the same as the name of the deployed Kafka Connect cluster.

4. Once the build has finished, the new image is 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 C.83, "**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 resources to be labeled as required.

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/v1beta1
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/v1beta1
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/v1beta1
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/v1beta1
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/v1beta1
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/v1beta1
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/v1beta1
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

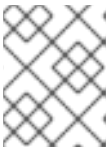
Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** for the certificate used for TLS Server Authentication and the key under which the certificate is stored in the **Secret**

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. (Optional) If they do not already exist, prepare the TLS certificate used for authentication in a file and create a **Secret**.

   NOTE

   The secrets created by the Cluster Operator for Kafka cluster may be used directly.

   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/v1beta1
   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 is enabled through 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 the SCRAM-SHA-512 mechanism
- SASL-based authentication using the PLAIN mechanism

You can use different authentication mechanisms for the Kafka Mirror Maker producer and consumer.

3.4.7.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.

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 TLS client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
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 configure Kafka Mirror Maker to use SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. The broker listener to which clients will connect must also be configured to use SCRAM-SHA-512 SASL authentication. This authentication mechanism requires a username and password.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example SCRAM-SHA-512 client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    authentication:
      type: scram-sha-512
      username: my-source-user
      passwordSecret:
        secretName: my-source-user
        password: my-source-password-key
  # ...
  producer:
    authentication:
      type: scram-sha-512
      username: my-producer-user
      passwordSecret:
        secretName: my-producer-user
        password: my-producer-password-key
  # ...
```

3.4.7.1.3. PLAIN authentication

To configure Kafka Mirror Maker to use PLAIN authentication, set the **type** property to **plain**. The broker listener to which clients will connect must also be configured to use SASL PLAIN authentication. This authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example PLAIN client authentication configuration

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  # ...
  consumer:
    authentication:
      type: plain
      username: my-source-user
      passwordSecret:
        secretName: my-source-user
        password: my-source-password-key
  # ...
  producer:
    authentication:
      type: plain
      username: my-producer-user
      passwordSecret:
        secretName: my-producer-user
        password: my-producer-password-key
  # ...
```

3.4.7.2. Configuring TLS client authentication in Kafka Mirror Maker

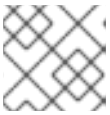
Prerequisites

- An OpenShift cluster
- A running Cluster Operator with a **tls** listener with **tls** authentication enabled
- If they exist, the name of the **Secret** with the public and private keys used for TLS Client Authentication, and the keys under which they are stored in the **Secret**

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. (Optional) If they do not already exist, prepare the keys used for authentication in a file and create the **Secret**.

   NOTE

   Secrets created by the User Operator may be used.

   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/v1beta1
   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
- If they exist, the name of the **Secret** with the password used for authentication, and the key under which it is stored in the **Secret**

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. (Optional) If they do not already exist, prepare a file with the password used for authentication and create the **Secret**.

   NOTE

   Secrets created by the User Operator may be used.

   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/v1beta1
   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/v1beta1
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/v1beta1
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 request specific resources and define the maximum consumption of those resources.

AMQ Streams supports two types of resources:

- CPU
- Memory

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

3.4.9.1. Resource limits and requests

Resource limits and requests are configured using the **resources** property in the 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**
- **KafkaBridge.spec**

Additional resources

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

3.4.9.1.1. Resource requests

Requests specify the resources to reserve for a given container. Reserving the resources ensures that they are always available.

IMPORTANT

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

Resources requests are specified in the **requests** property. Resources requests currently supported by AMQ Streams:

- **cpu**
- **memory**

A request may be configured for one or more supported resources.

Example resource request configuration with all resources

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

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 always be available. A container can use the resources up to the limit only when they are available. Resource limits should be always higher than the resource requests.

Resource limits are specified in the **limits** property. Resource limits currently supported by AMQ Streams:

- **cpu**
- **memory**

A resource may be configured for one or more supported limits.

Example resource limits configuration

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

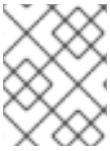
3.4.9.1.3. Supported CPU formats

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.

Example CPU units

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



NOTE

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

Additional resources

- For more information on CPU specification, see the Meaning of CPU.

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
# ...
```

Additional resources

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

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/v1beta1
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

This section provides information on loggers and how to configure log levels.

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

3.4.10.1. Kafka Mirror Maker loggers

Kafka Mirror Maker has its own configurable logger:

- **mirrormaker.root.logger**

3.4.10.2. Specifying inline logging

Procedure

1. Edit the YAML file to specify the loggers and logging level for the required components.
   For example, the logging level here is set to INFO:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: inline
    loggers:
      logger.name: "INFO"
  # ...
```

   You can set the log level to INFO, ERROR, WARN, TRACE, DEBUG, FATAL or OFF.

   For more information about the log levels, see the 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
   ```

3.4.10.3. Specifying an external ConfigMap for logging

Procedure

1. Edit the YAML file to specify the name of the **ConfigMap** to use for the required components.
   For example:

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: KafkaMirrorMaker
   spec:
     # ...
     logging:
       type: external
       name: customConfigMap
     # ...
   ```

   Remember to place your custom ConfigMap under the **log4j.properties** or **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
   ```

Garbage collector (GC) logging can also be enabled (or disabled). For more information on GC, see Section 3.4.12.1, “JVM configuration”

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 are 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/v1beta1
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/v1beta1
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/v1beta1
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 run inside a Java Virtual Machine (JVM). JVM configuration options optimize the performance for different platforms and architectures. AMQ Streams allows you to configure 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.

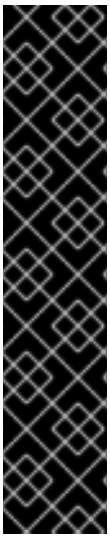
NOTE

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.

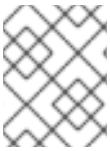
Setting the same value for initial (**-Xms**) and maximum (**-Xmx**) heap sizes avoids the JVM having to allocate memory after startup, at the cost of possibly allocating more heap than is really needed. For Kafka and Zookeeper pods such allocation could cause unwanted latency. For Kafka Connect avoiding 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.

3.4.12.1.1. Garbage collector logging

The **jvmOptions** section also allows you to enable and disable garbage collector (GC) logging. GC logging is enabled by default. To disable it, set the **gcLoggingEnabled** property as follows:

Example of disabling GC logging

```
# ...
jvmOptions:
  gcLoggingEnabled: false
# ...
```

3.4.12.2. 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/v1beta1
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**
- **KafkaBridge.spec**

3.4.13.1.1. Configuring the Kafka.spec.kafka.image property

The **Kafka.spec.kafka.image** property functions differently from the others, because AMQ Streams supports multiple versions of Kafka, each requiring the own image. The **STRIMZI_KAFKA_IMAGES** environment variable of the Cluster Operator configuration is used to provide a mapping between Kafka versions and the corresponding images. This is used in combination with the **Kafka.spec.kafka.image** and **Kafka.spec.kafka.version** properties as follows:

- If neither **Kafka.spec.kafka.image** nor **Kafka.spec.kafka.version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- If **Kafka.spec.kafka.image** is given but **Kafka.spec.kafka.version** is not then the given image will be used and the **version** will be assumed to be the Cluster Operator's default Kafka version.
- If **Kafka.spec.kafka.version** is given but **Kafka.spec.kafka.image** is not then image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- Both **Kafka.spec.kafka.version** and **Kafka.spec.kafka.image** are given the given image will be used, and it will be assumed to contain a Kafka broker with the given version.

WARNING

It is best to provide just **Kafka.spec.kafka.version** and leave the **Kafka.spec.kafka.image** property unspecified. This reduces the chances of making a mistake in configuring the **Kafka** resource. If you need to change the images used for different versions of Kafka, it is better to configure the Cluster Operator's **STRIMZI_KAFKA_IMAGES** environment variable.

3.4.13.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property 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 broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Zookeeper nodes:

1. Container image specified in the **STRIMZI_DEFAULT_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Kafka Connect:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** 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/v1beta1
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/v1beta1
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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

2. Create or update the resource.
   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.template.pod**

- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
         pod:
           affinity:
             nodeAffinity:
               requiredDuringSchedulingIgnoredDuringExecution:
                 nodeSelectorTerms:
                   - matchExpressions:
                     - key: node-type
                       operator: In
                       values:
                       - fast-network
   ```

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

3. Create or update the resource.
   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.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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 can be configured using the **tolerations** property in following resources:

- **Kafka.spec.kafka.template.pod**

- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
```

```
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: dedicated
                operator: In
                values:
                - Kafka
  # ...
zookeeper:
  # ...
```

6. Create or update the resource.
   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.

<*mirror-maker-name*>-mirror-maker

Pod Disruption Budget configured for the Kafka Mirror Maker worker nodes.

3.5. KAFKA BRIDGE CLUSTER CONFIGURATION

The full schema of the **KafkaBridge** resource is described in the Section C.92, "**KafkaBridge** schema reference". All labels that are applied to the desired **KafkaBridge** resource will also be applied to the OpenShift resources making up the Kafka Bridge cluster. This provides a convenient mechanism for resources to be labeled as required.

3.5.1. Replicas

Kafka Bridge can run multiple nodes. The number of nodes is defined in the **KafkaBridge** resource. Running a Kafka Bridge with multiple nodes can provide better availability and scalability. However, when running Kafka Bridge on OpenShift it is not absolutely necessary to run multiple nodes of Kafka Bridge for high availability.

IMPORTANT

If a node where Kafka Bridge is deployed to crashes, OpenShift will automatically reschedule the Kafka Bridge pod to a different node. In order to prevent issues arising when client consumer requests are processed by different Kafka Bridge instances, addressed-based routing must be employed to ensure that requests are routed to the right Kafka Bridge instance. Additionally, each independent Kafka Bridge instance must have a replica. A Kafka Bridge instance has its own state which is not shared with another instances.

3.5.1.1. Configuring the number of nodes

The number of Kafka Bridge nodes is configured using the **replicas** property in **KafkaBridge.spec**.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

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

2. Create or update the resource.
   On OpenShift use:

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

3.5.2. Bootstrap servers

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

The list of bootstrap servers is configured in the **bootstrapServers** property in **KafkaBridge.kafka.spec**. The servers must be defined as a comma-separated list specifying one or more Kafka brokers, or a service pointing to Kafka brokers specified as a ***hostname*:_port_** pairs.

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

3.5.2.1. Configuring bootstrap servers

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **bootstrapServers** property in the **KafkaBridge** resource. For example:

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

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

2. Create or update the resource.
   On OpenShift use:

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

3.5.3. Connecting to Kafka brokers using TLS

By default, Kafka Bridge tries to connect to Kafka brokers using a plain text connection. If you prefer to use TLS, additional configuration is required.

3.5.3.1. TLS support for Kafka connection to the Kafka Bridge

TLS support for Kafka connection is configured in the **tls** property in **KafkaBridge.spec.kafka**. The **tls** property contains a list of secrets with key names under which the certificates are stored. The certificates must be stored in X509 format.

An example showing TLS configuration with multiple certificates

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

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

3.5.3.2. Configuring TLS in Kafka Bridge

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** for the certificate used for TLS Server Authentication, and the key under which the certificate is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the TLS certificate used in authentication in a file and create a **Secret**.

   NOTE

   The secrets created by the Cluster Operator for Kafka cluster may be used directly.

   On OpenShift use:

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

2. Edit the **tls** property in the **KafkaBridge** resource. For example:

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

3. Create or update the resource.
   On OpenShift use:

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

3.5.4. Connecting to Kafka brokers with Authentication

By default, Kafka Bridge will try to connect to Kafka brokers without authentication. Authentication is enabled through the **KafkaBridge** resources.

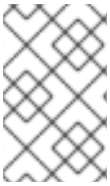
3.5.4.1. Authentication support in Kafka Bridge

Authentication is configured through the **authentication** property in **KafkaBridge.spec.kafka**. 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 the SCRAM-SHA-512 mechanism
- SASL-based authentication using the PLAIN mechanism

3.5.4.1.1. TLS Client Authentication

To use TLS client authentication, set the **type** property to the value **tls**. TLS client authentication uses a TLS certificate to authenticate. The certificate is specified in the **certificateAndKey** property and is always loaded from an OpenShift secret. In the secret, the certificate must be stored in X509 format under two different keys: public and private.



NOTE

TLS client authentication can be used only with TLS connections. For more details about TLS configuration in Kafka Bridge see Section 3.5.3, "Connecting to Kafka brokers using TLS".

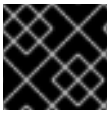
An example TLS client authentication configuration

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

3.5.4.1.2. SCRAM-SHA-512 authentication

To configure Kafka Bridge to use SASL-based SCRAM-SHA-512 authentication, set the **type** property to **scram-sha-512**. This authentication mechanism requires a username and password.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name of the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.



IMPORTANT

Do not specify the actual password in the **password** field.

An example SASL based SCRAM-SHA-512 client authentication configuration

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

3.5.4.1.3. SASL-based PLAIN authentication

To configure Kafka Bridge to use SASL-based PLAIN authentication, set the **type** property to **plain**. This authentication mechanism requires a username and password.

WARNING

The SASL PLAIN mechanism will transfer the username and password across the network in cleartext. Only use SASL PLAIN authentication if TLS encryption is enabled.

- Specify the username in the **username** property.
- In the **passwordSecret** property, specify a link to a **Secret** containing the password. The **secretName** property contains the name the **Secret** and the **password** property contains the name of the key under which the password is stored inside the **Secret**.

IMPORTANT

Do not specify the actual password in the **password** field.

An example showing SASL based PLAIN client authentication configuration

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

```
authentication:
  type: plain
  username: my-bridge-user
  passwordSecret:
    secretName: my-bridge-user
    password: my-bridge-password-key
# ...
```

3.5.4.2. Configuring TLS client authentication in Kafka Bridge

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- If they exist, the name of the **Secret** with the public and private keys used for TLS Client Authentication, and the keys under which they are stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare the keys used for authentication in a file and create the **Secret**.

   NOTE

   Secrets created by the User Operator may be used.

   On OpenShift use:

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

2. Edit the **authentication** property in the **KafkaBridge** resource. For example:

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

3. Create or update the resource.
   On OpenShift use:

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

3.5.4.3. Configuring SCRAM-SHA-512 authentication in Kafka Bridge

Prerequisites

- An OpenShift cluster
- A running Cluster Operator
- Username of the user which should be used for authentication
- If they exist, the name of the **Secret** with the password used for authentication and the key under which the password is stored in the **Secret**

Procedure

1. (Optional) If they do not already exist, prepare a file with the password used in authentication and create the **Secret**.

   NOTE

   Secrets created by the User Operator may be used.

   On OpenShift use:

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

2. Edit the **authentication** property in the **KafkaBridge** resource. For example:

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaBridge
   metadata:
     name: my-bridge
   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 use:

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

3.5.5. Kafka Bridge configuration

AMQ Streams allows you to customize the configuration of Apache Kafka Bridge nodes by editing certain options listed in Apache Kafka documentation and Apache Kafka documentation.

Configuration options that can be configured relate to:

- Kafka cluster bootstrap address
- Security (Encryption, Authentication, and Authorization)
- Consumer configuration
- Producer configuration
- HTTP configuration

3.5.5.1. Kafka Bridge Consumer configuration

Kafka Bridge consumer is configured using the properties in **KafkaBridge.spec.consumer**. This property contains the Kafka Bridge consumer configuration options as keys. The values can be 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.**
- **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

IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Bridge cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaBridge.spec.consumer.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Bridge nodes.

Example Kafka Bridge consumer configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
```

```
# ...
consumer:
  config:
    auto.offset.reset: earliest
    enable.auto.commit: true
# ...
```

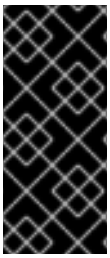
3.5.5.2. Kafka Bridge Producer configuration

Kafka Bridge producer is configured using the properties in **KafkaBridge.spec.producer**. This property contains the Kafka Bridge producer configuration options as keys. The values can be 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.**
- **bootstrap.servers**



IMPORTANT

The Cluster Operator does not validate keys or values in the **config** object provided. When an invalid configuration is provided, the Kafka Bridge cluster might not start or might become unstable. In this circumstance, fix the configuration in the **KafkaBridge.spec.producer.config** object, then the Cluster Operator can roll out the new configuration to all Kafka Bridge nodes.

Example Kafka Bridge producer configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  producer:
    config:
      acks: 1
      delivery.timeout.ms: 300000
  # ...
```

3.5.5.3. Kafka Bridge HTTP configuration

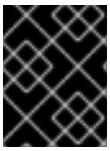
Kafka Bridge HTTP configuration is set using the properties in **KafkaBridge.spec.http**. This property contains the Kafka Bridge HTTP configuration options.

- **port**

When configuring **port** property avoid the value **8081**. This port is used for the health checks.

Example Kafka Bridge HTTP configuration

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  http:
    port: 8080
  # ...
```



IMPORTANT

The port must not be set to 8081 as that will cause a conflict with the healthcheck settings.

3.5.5.4. Configuring Kafka Bridge

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

1. Edit the **kafka**, **http**, **consumer** or **producer** property in the **KafkaBridge** resource. For example:

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    port: 8080
  consumer:
    config:
      auto.offset.reset: earliest
  producer:
    config:
      delivery.timeout.ms: 300000
  # ...
```

2. Create or update the resource.
   On OpenShift use:

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

3.5.6. Healthchecks

Healthchecks are periodical tests which verify the health of an application. When a Healthcheck probe fails, OpenShift assumes that the application is not healthy and attempts 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.

3.5.6.1. Healthcheck configurations

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

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

3.5.6.2. 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/v1beta1
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
```

3.5.7. 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.

3.5.7.1. 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**
- **KafkaBridge.spec**

3.5.7.1.1. Configuring the Kafka.spec.kafka.image property

The **Kafka.spec.kafka.image** property functions differently from the others, because AMQ Streams supports multiple versions of Kafka, each requiring the own image. The **STRIMZI_KAFKA_IMAGES** environment variable of the Cluster Operator configuration is used to provide a mapping between Kafka versions and the corresponding images. This is used in combination with the **Kafka.spec.kafka.image** and **Kafka.spec.kafka.version** properties as follows:

- If neither **Kafka.spec.kafka.image** nor **Kafka.spec.kafka.version** are given in the custom resource then the **version** will default to the Cluster Operator's default Kafka version, and the image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- If **Kafka.spec.kafka.image** is given but **Kafka.spec.kafka.version** is not then the given image will be used and the **version** will be assumed to be the Cluster Operator's default Kafka version.
- If **Kafka.spec.kafka.version** is given but **Kafka.spec.kafka.image** is not then image will be the one corresponding to this version in the **STRIMZI_KAFKA_IMAGES**.
- Both **Kafka.spec.kafka.version** and **Kafka.spec.kafka.image** are given the given image will be used, and it will be assumed to contain a Kafka broker with the given version.

WARNING

It is best to provide just **Kafka.spec.kafka.version** and leave the **Kafka.spec.kafka.image** property unspecified. This reduces the chances of making a mistake in configuring the **Kafka** resource. If you need to change the images used for different versions of Kafka, it is better to configure the Cluster Operator's **STRIMZI_KAFKA_IMAGES** environment variable.

3.5.7.1.2. Configuring the image property in other resources

For the **image** property in the other custom resources, the given value will be used during deployment. If the **image** property 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 broker TLS sidecar:
  1. Container image specified in the **STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Zookeeper nodes:
  1. Container image specified in the **STRIMZI_DEFAULT_ZOOKEEPER_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.
- For Topic Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** container image.
- For User Operator:
  1. Container image specified in the **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amq-streams-operator:1.2.0** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** container image.

- For Kafka Connect:
  1. Container image specified in the **STRIMZI_DEFAULT_KAFKA_CONNECT_IMAGE** environment variable from the Cluster Operator configuration.
  2. **registry.redhat.io/amq7/amqstreams-kafka-22** 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. **registry.redhat.io/amq7/amqstreams-kafka-22** 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

3.5.7.2. 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/v1beta1
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
```

3.5.8. 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.

3.5.8.1. Scheduling pods based on other applications

3.5.8.1.1. 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.

3.5.8.1.2. Affinity

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

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

- **KafkaBridge.spec.template.pod**

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 .

3.5.8.1.3. 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
```

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

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

3.5.8.2. Scheduling pods to specific nodes

3.5.8.2.1. 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.

3.5.8.2.2. Affinity

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

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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 .

3.5.8.2.3. 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                  - key: node-type
                    operator: In
                    values:
                    - fast-network
    # ...
  zookeeper:
    # ...
```

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

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

3.5.8.3. Using dedicated nodes

3.5.8.3.1. 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.

3.5.8.3.2. Affinity

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

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

- **KafkaBridge.spec.template.pod**

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 .

3.5.8.3.3. Tolerations

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

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaConnectS2I.spec.template.pod**
- **KafkaBridge.spec.template.pod**

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

3.5.8.3.4. 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/v1beta1
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        tolerations:
          - key: "dedicated"
            operator: "Equal"
            value: "Kafka"
            effect: "NoSchedule"
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
              - matchExpressions:
                - key: dedicated
                  operator: In
                  values:
                  - Kafka
    # ...
  zookeeper:
    # ...
```

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

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

3.5.9. List of resources created as part of Kafka Bridge cluster

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

***bridge-cluster-name*-bridge**

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

***bridge-cluster-name*-bridge-service**

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

***bridge-cluster-name*-bridge-config**

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

***bridge-cluster-name*-bridge**

Pod Disruption Budget configured for the Kafka Bridge worker nodes.

3.6. CUSTOMIZING DEPLOYMENTS

AMQ Streams creates several OpenShift resources, such as **Deployments**, **StatefulSets**, **Pods**, and **Services**, which are managed by OpenShift operators. Only the operator that is responsible for managing a particular OpenShift resource can change that resource. If you try to manually change an

operator-managed OpenShift resource, the operator will revert your changes back.

However, changing an operator-managed OpenShift resource can be useful if you want to perform certain tasks, such as:

- Adding custom labels or annotations that control how **Pods** are treated by Istio or other services;
- Managing how **Loadbalancer**-type Services are created by the cluster.

You can make these types of changes using the **template** property in the AMQ Streams custom resources.

3.6.1. Template properties

You can use the **template** property to configure aspects of the resource creation process. You can include it in the following resources and properties:

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

In the following example, the **template** property is used to modify the labels in a Kafka broker's **StatefulSet**:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  labels:
    app: my-cluster
spec:
  kafka:
    # ...
    template:
      statefulset:
        metadata:
          labels:
            mylabel: myvalue
    # ...
```

Supported resources in Kafka cluster

When defined in a Kafka cluster, the **template** object can have the following fields:

statefulset

Configures the **StatefulSet** used by the Kafka broker.

pod

Configures the Kafka broker **Pods** created by the **StatefulSet**.

bootstrapService

Configures the bootstrap service used by clients running within OpenShift to connect to the Kafka broker.

brokersService

Configures the headless service.

externalBootstrapService

Configures the bootstrap service used by clients connecting to Kafka brokers from outside of OpenShift.

perPodService

Configures the per-Pod services used by clients connecting to the Kafka broker from outside OpenShift to access individual brokers.

externalBootstrapRoute

Configures the bootstrap route used by clients connecting to the Kafka brokers from outside of OpenShift using OpenShift **Routes**.

perPodRoute

Configures the per-Pod routes used by clients connecting to the Kafka broker from outside OpenShift to access individual brokers using OpenShift **Routes**.

podDisruptionBudget

Configures the Pod Disruption Budget for Kafka broker **StatefulSet**.

Supported resources in Zookeeper cluster

When defined in a Zookeeper cluster, the **template** object can have the following fields:

statefulset

Configures the Zookeeper **StatefulSet**.

pod

Configures the Zookeeper **Pods** created by the **StatefulSet**.

clientsService

Configures the service used by clients to access Zookeeper.

nodesService

Configures the headless service.

podDisruptionBudget

Configures the Pod Disruption Budget for Zookeeper **StatefulSet**.

Supported resources in Entity Operator

When defined in an Entity Operator , the template object can have the following fields:

deployment

Configures the Deployment used by the Entity Operator.

pod

Configures the Entity Operator **Pod** created by the **Deployment**.

Supported resources in Kafka Connect and Kafka Connect with Source2Image support

When used with Kafka Connect and Kafka Connect with Source2Image support , the template object can have the following fields:

deployment

Configures the Kafka Connect **Deployment**.

pod

Configures the Kafka Connect **Pods** created by the **Deployment**.

apiService

Configures the service used by the Kafka Connect REST API.

podDisruptionBudget

Configures the Pod Disruption Budget for Kafka Connect **Deployment**.

Supported resource in Kafka Mirror Maker

When used with Kafka Mirror Maker , the template object can have the following fields:

deployment

Configures the Kafka Mirror Maker **Deployment**.

pod

Configures the Kafka Mirror Maker **Pods** created by the **Deployment**.

podDisruptionBudget

Configures the Pod Disruption Budget for Kafka Mirror Maker **Deployment**.

3.6.2. Labels and Annotations

For every resource, you can configure additional **Labels** and **Annotations**. **Labels** and **Annotations** are configured in the **metadata** property. For example:

```
# ...
template:
    statefulset:
        metadata:
            labels:
                label1: value1
                label2: value2
            annotations:
                annotation1: value1
                annotation2: value2
# ...
```

The **labels** and **annotations** fields can contain any labels or annotations that do not contain the reserved string **strimzi.io**. Labels and annotations containing **strimzi.io** are used internally by AMQ Streams and cannot be configured by the user.

3.6.3. Customizing Pods

In addition to Labels and Annotations, you can customize some other fields on Pods. These fields are described in the following table and affect how the Pod is created.

Field	Description
terminationGracePeriodSeconds	Defines the period of time, in seconds, by which the Pod must have terminated gracefully. After the grace period, the Pod and its containers are forcefully terminated (killed). The default value is **30** seconds. NOTE: You might need to increase the grace period for very large Kafka clusters, so that the Kafka brokers have enough time to transfer their work to another broker before they are terminated.
imagePullSecrets	Defines a list of references to OpenShift Secrets that can be used for pulling container images from private repositories. For more information about how to create a Secret with the credentials, see Pull an Image from a Private Registry. NOTE: When the **STRIMZI_IMAGE_PULL_SECRETS** environment variable in Cluster Operator and the **imagePullSecrets** option are specified, only the **imagePullSecrets** variable is used. The **STRIMZI_IMAGE_PULL_SECRETS** variable is ignored.
securityContext	Configures pod-level security attributes for containers running as part of a given Pod. For more information about configuring SecurityContext, see Configure a Security Context for a Pod or Container

These fields are effective on each type of cluster (Kafka and Zookeeper; Kafka Connect and Kafka Connect with S2I support; and Kafka Mirror Maker).

The following example shows these customized fields on a **template** property:

```
# ...
template:
    pod:
        metadata:
            labels:
                label1: value1
        imagePullSecrets:
            - name: my-docker-credentials
        securityContext:
            runAsUser: 1000001
            fsGroup: 0
        terminationGracePeriodSeconds: 120
# ...
```

Additional resources

- For more information, see Section C.40, "**PodTemplate** schema reference".

3.6.4. Customizing the image pull policy

AMQ Streams allows you to customize the image pull policy for containers in all pods deployed by the Cluster Operator. The image pull policy is configured using the environment variable **STRIMZI_IMAGE_PULL_POLICY** in the Cluster Operator deployment. The **STRIMZI_IMAGE_PULL_POLICY** environment variable can be set to three different values:

Always

Container images are pulled from the registry every time the pod is started or restarted.

IfNotPresent

Container images are pulled from the registry only when they were not pulled before.

Never

Container images are never pulled from the registry.

The image pull policy can be currently customized only for all Kafka, Kafka Connect, and Kafka Mirror Maker clusters at once. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka Mirror Maker clusters.

Additional resources

- For more information about Cluster Operator configuration, see Section 4.1, “Cluster Operator”.
- For more information about Image Pull Policies, see Disruptions.

3.6.5. Customizing Pod Disruption Budgets

AMQ Streams creates a pod disruption budget for every new **StatefulSet** or **Deployment**. By default, these pod disruption budgets only allow a single pod to be unavailable at a given time by setting the **maxUnavailable** value in the`PodDisruptionBudget.spec` resource to 1. You can change the amount of unavailable pods allowed by changing the default value of **maxUnavailable** in the pod disruption budget template. This template applies to each type of cluster (Kafka and Zookeeper; Kafka Connect and Kafka Connect with S2I support; and Kafka Mirror Maker).

The following example shows customized **podDisruptionBudget** fields on a **template** property:

```
# ...
template:
    podDisruptionBudget:
        metadata:
            labels:
                key1: label1
                key2: label2
            annotations:
                key1: label1
                key2: label2
        maxUnavailable: 1
# ...
```

Additional resources

- For more information, see Section C.41, “**PodDisruptionBudgetTemplate** schema reference”.
- The Disruptions chapter of the Kubernetes documentation.

3.6.6. Customizing deployments

This procedure describes how to customize **Labels** of a Kafka cluster.

Prerequisites

- An OpenShift cluster.
- A running Cluster Operator.

Procedure

1. Edit the **template** property in the **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, or **KafkaMirrorMaker** resource. For example, to modify the labels for the Kafka broker **StatefulSet**, use:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  labels:
    app: my-cluster
spec:
  kafka:
    # ...
    template:
      statefulset:
        metadata:
          labels:
            mylabel: myvalue
    # ...
```

2. Create or update the resource.
On OpenShift, use **oc apply**:

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

Alternatively, use **oc edit**:

```
oc edit Resource ClusterName
```

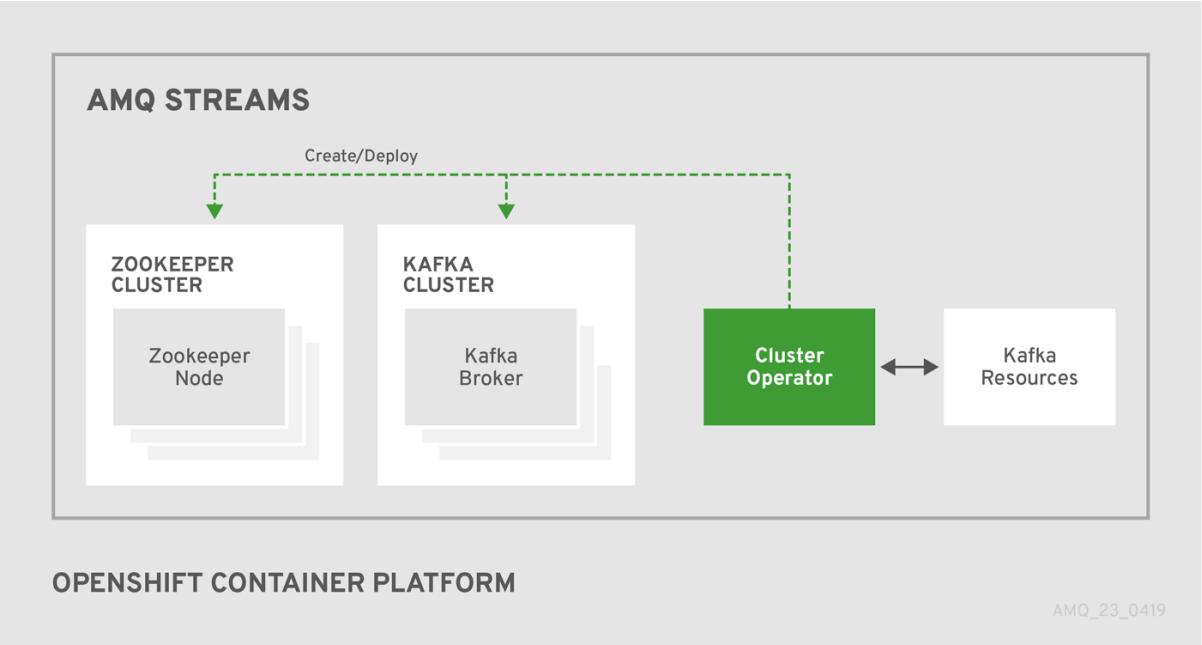
CHAPTER 4. OPERATORS

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.

Example architecture for the Cluster Operator



When the Cluster Operator is up, it starts to *watch* for certain OpenShift resources containing the desired Kafka, Kafka Connect, or Kafka Mirror Maker 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.
- A **KafkaMirrorMaker** resource for the Kafka Mirror Maker instance.

When a new **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, or **Kafka Mirror Maker** resource is created in the OpenShift cluster, the operator gets the cluster description from the desired resource and starts creating a new Kafka, Kafka Connect, or Kafka Mirror Maker 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, Kafka Connect, or Kafka Mirror Maker cluster. Resources are either patched or deleted and then re-created in order to make the Kafka, Kafka Connect,

or Kafka Mirror Maker 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

- 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: registry.redhat.io/amq7/amq-streams-operator:1.2.0
        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. Deploying the Cluster Operator to watch all namespaces

You can configure the Cluster Operator to watch AMQ Streams resources across all OpenShift projects or Kubernetes namespaces in your OpenShift cluster. When running in this mode, the Cluster Operator automatically manages clusters in any new projects or namespaces that are created.

Prerequisites

- Your OpenShift cluster is running.

Procedure

1. Configure the Cluster Operator to watch all namespaces:
   a. Edit the **050-Deployment-strimzi-cluster-operator.yaml** file.
   b. Set the value of the **STRIMZI_NAMESPACE** environment variable to ***.

```
apiVersion: extensions/v1beta1
kind: Deployment
spec:
  template:
    spec:
```

```
    # ...
    serviceAccountName: strimzi-cluster-operator
    containers:
    - name: strimzi-cluster-operator
      image: registry.redhat.io/amq7/amq-streams-operator:1.2.0
      imagePullPolicy: IfNotPresent
      env:
      - name: STRIMZI_NAMESPACE
        value: "*"
      # ...
```

2. Create **ClusterRoleBindings** that grant cluster-wide access to all OpenShift projects or Kubernetes namespaces to the Cluster Operator.
   On OpenShift, use the **oc adm policy** command:

```
oc adm policy add-cluster-role-to-user strimzi-cluster-operator-namespaced --serviceaccount strimzi-cluster-operator -n my-project
oc adm policy add-cluster-role-to-user strimzi-entity-operator --serviceaccount strimzi-cluster-operator -n my-project
oc adm policy add-cluster-role-to-user strimzi-topic-operator --serviceaccount strimzi-cluster-operator -n my-project
```

   Replace ***my-project*** with the project in which you want to install the Cluster Operator.

3. Deploy the Cluster Operator to your OpenShift cluster.
   On OpenShift, use the **oc apply** command:

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

4.1.5. 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.

4.1.6. Cluster Operator Configuration

The Cluster Operator can be configured through the following supported environment variables:

STRIMZI_NAMESPACE

A comma-separated list of OpenShift projects or Kubernetes namespaces that the operator should operate in. When not set, set to empty string, or to * the cluster operator will operate in all OpenShift projects or Kubernetes namespaces. 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 is 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_KAFKA_IMAGES

Required. This provides a mapping from Kafka version to the corresponding Docker image containing a Kafka broker of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.1.1=registry.redhat.io/amq7/amqstreams-kafka-21, 2.2.1=registry.redhat.io/amq7/amqstreams-kafka-22**. This is used when a **Kafka.spec.kafka.version** property is specified but not the **Kafka.spec.kafka.image**, as described in Section 3.1.17, "Container images".

STRIMZI_DEFAULT_KAFKA_INIT_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-operator:1.2.0**. 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 3.1.17, "Container images".

STRIMZI_DEFAULT_TLS_SIDECAR_KAFKA_IMAGE

Optional, default **registry.redhat.io/amq7/amqstreams-kafka-22**. 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 3.1.17, "Container images".

STRIMZI_DEFAULT_ZOOKEEPER_IMAGE

Optional, default **registry.redhat.io/amq7/amqstreams-kafka-22**. 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 3.1.17, "Container images".

STRIMZI_DEFAULT_TLS_SIDECAR_ZOOKEEPER_IMAGE

Optional, default **registry.redhat.io/amq7/amqstreams-kafka-22**. 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 3.1.17, "Container images".

STRIMZI_KAFKA_CONNECT_IMAGES

Required. This provides a mapping from the Kafka version to the corresponding Docker image containing a Kafka connect of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.1.1=registry.redhat.io/amq7/amqstreams-kafka-21, 2.2.1=registry.redhat.io/amq7/amqstreams-kafka-22**. This is used when a **KafkaConnect.spec.version** property is specified but not the **KafkaConnect.spec.image**, as described in Section 3.2.11, "Container images".

STRIMZI_KAFKA_CONNECT_S2I_IMAGES

Required. This provides a mapping from the Kafka version to the corresponding Docker image containing a Kafka connect of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.1.1=registry.redhat.io/amq7/amqstreams-kafka-21,**

2.2.1=registry.redhat.io/amq7/amqstreams-kafka-22. This is used when a **KafkaConnectS2I.spec.version** property is specified but not the **KafkaConnectS2I.spec.image**, as described in Section 3.3.11, "Container images".

STRIMZI_KAFKA_MIRROR_MAKER_IMAGES

Required. This provides a mapping from the Kafka version to the corresponding Docker image containing a Kafka mirror maker of that version. The required syntax is whitespace or comma separated ***<version>=<image>*** pairs. For example **2.1.1=registry.redhat.io/amq7/amqstreams-kafka-21, 2.2.1=registry.redhat.io/amq7/amqstreams-kafka-22**. This is used when a **KafkaMirrorMaker.spec.version** property is specified but not the **KafkaMirrorMaker.spec.image**, as described in Section 3.4.13, "Container images".

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-operator:1.2.0**. The image name to use as the default when deploying the topic operator, if no image is specified as the **Kafka.spec.entityOperator.topicOperator.image** in the Section 3.1.17, "Container images" of the **Kafka** resource.

STRIMZI_DEFAULT_USER_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amq-streams-operator:1.2.0**. The image name to use as the default when deploying the user operator, if no image is specified as the **Kafka.spec.entityOperator.userOperator.image** in the Section 3.1.17, "Container images" of the **Kafka** resource.

STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE

Optional, default **registry.redhat.io/amq7/amqstreams-kafka-22**. 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 3.1.17, "Container images".

STRIMZI_IMAGE_PULL_POLICY

Optional. The **ImagePullPolicy** which will be applied to containers in all pods managed by AMQ Streams Cluster Operator. The valid values are **Always**, **IfNotPresent**, and **Never**. If not specified, the OpenShift defaults will be used. Changing the policy will result in a rolling update of all your Kafka, Kafka Connect, and Kafka Mirror Maker clusters.

STRIMZI_IMAGE_PULL_SECRETS

Optional. A comma-separated list of **Secret** names. The secrets referenced here contain the credentials to the container registries where the container images are pulled from. The secrets are used in the **imagePullSecrets** field for all **Pods** created by the Cluster Operator. Changing this list results in a rolling update of all your Kafka, Kafka Connect, and Kafka Mirror Maker clusters.

4.1.7. Role-Based Access Control (RBAC)

4.1.7.1. 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.

4.1.7.2. 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.

4.1.7.3. 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
        strimzi.io/kind: cluster-operator
      # ...
```

Note line 12, where the the **strimzi-cluster-operator ServiceAccount** is specified as the **serviceAccountName**.

4.1.7.4. 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/v1
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
  - kafkas/status
  - kafkaconnects
  - kafkaconnects2is
  - kafkamirrormakers
  - kafkabridges
  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
  - replicasets
  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:
  - networking.k8s.io
  resources:
  - networkpolicies
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - route.openshift.io
  resources:
  - routes
  - routes/custom-host
  verbs:
  - get
  - list
  - create
  - delete
  - patch
  - update
- apiGroups:
  - ""
  resources:
  - persistentvolumeclaims
  verbs:
  - get
  - list
  - create
  - delete
```

```
  - patch
  - update
- apiGroups:
  - policy
  resources:
  - poddisruptionbudgets
  verbs:
  - get
  - list
  - watch
  - create
  - delete
  - patch
  - update
- apiGroups:
  - extensions
  resources:
  - ingresses
  verbs:
  - get
  - list
  - watch
  - 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/v1
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
- apiGroups:
  - storage.k8s.io
  resources:
  - storageclasses
  verbs:
  - get
```

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/v1
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/v1
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
```

4.1.7.5. 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/v1
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/v1
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/v1
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
```

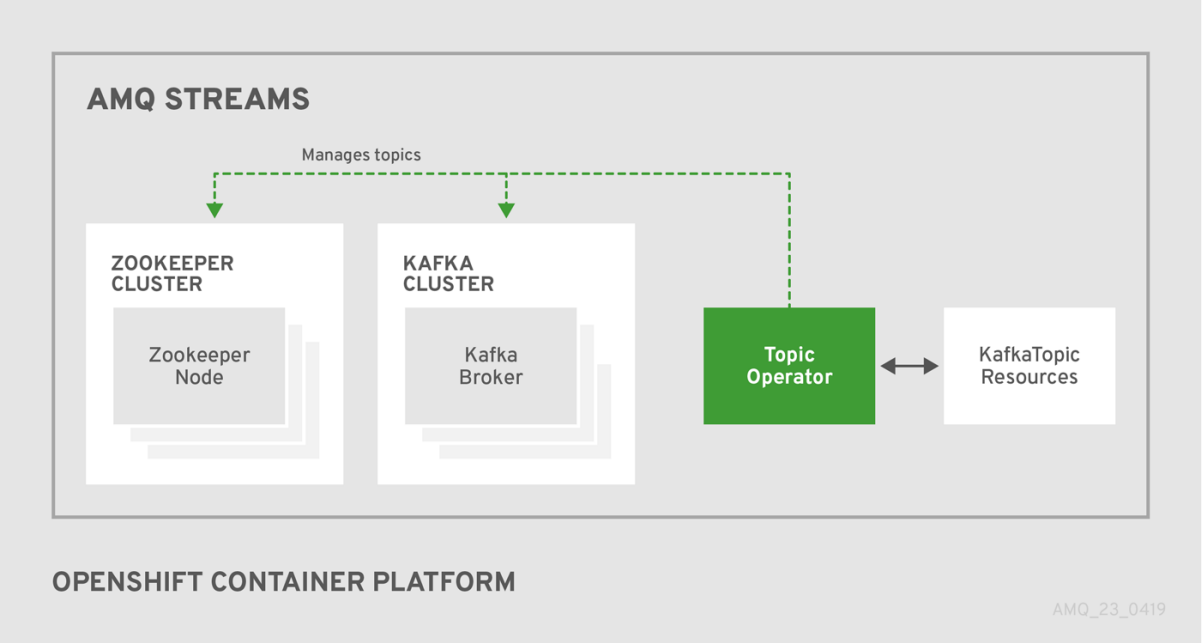
```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-entity-operator-delegation
  labels:
    app: strimzi
subjects:
- kind: ServiceAccount
  name: strimzi-cluster-operator
  namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-entity-operator
  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.

Example architecture for the Topic Operator



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
- Deleted, the operator will delete the topic it describes
- 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
- Deleted from the Kafka cluster, the operator will delete the **KafkaTopic** describing it
- Changed in the Kafka cluster, 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 is 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/v1beta1
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

2. Configure the Topic Operator using the fields described in Section C.47, “**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 2.3, "Cluster Operator".
- For more information about deploying the Entity Operator, see Section 3.1.10, "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 C.46, "**EntityOperatorSpec** schema reference".

4.2.4. Configuring the Topic Operator with resource requests and limits

You can allocate resources, such as CPU and memory, to the Topic Operator and set a limit on the amount of resources it can consume.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Update the Kafka cluster configuration in an editor, as required:
   On OpenShift, use:

```
oc edit kafka my-cluster
```

2. In the **spec.entityOperator.topicOperator.resources** property in the **Kafka** resource, set the resource requests and limits for the Topic Operator.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # kafka and zookeeper sections...
  entityOperator:
    topicOperator:
      resources:
        request:
          cpu: "1"
          memory: 500Mi
        limit:
          cpu: "1"
          memory: 500Mi
```

3. Apply the new configuration to create or update the resource.
   On OpenShift, use **oc apply**:

```
oc apply -f kafka.yaml
```

Additional resources

- For more information about the schema of the **resources** object, see Section C.33, "**ResourceRequirements** 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 it is more flexible. For instance, it 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.

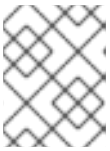
2. Deploy the Topic Operator.
   On OpenShift this can be done using **oc apply**:

   ```
   oc apply -f install/topic-operator
   ```

3. Verify that the Topic Operator has been deployed successfully. On 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

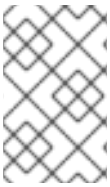
   This could take some time if you have a slow connection to the OpenShift and the images have not been downloaded before.

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 2.9.2, “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 at getting topic metadata from Kafka. The time between each attempt is defined as an exponential back-off. Consider increasing this value when topic creation could take more time due to the number of partitions or 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 2.3, "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. Configuring the User Operator with resource requests and limits

You can allocate resources, such as CPU and memory, to the User Operator and set a limit on the amount of resources it can consume.

Prerequisites

- The Cluster Operator is running.

Procedure

1. Update the Kafka cluster configuration in an editor, as required:
On OpenShift, use:

```
oc edit kafka my-cluster
```

2. In the **spec.entityOperator.userOperator.resources** property in the **Kafka** resource, set the resource requests and limits for the User Operator.

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
spec:
  # kafka and zookeeper sections...
  entityOperator:
    userOperator:
      resources:
        request:
          cpu: "1"
          memory: 500Mi
        limit:
          cpu: "1"
          memory: 500Mi
```

3. Apply the new configuration to create or update the resource.
On OpenShift, use **oc apply**:

```
oc apply -f kafka.yaml
```

Additional resources

- For more information about the schema of the **resources** object, see Section C.33, "**ResourceRequirements** schema reference".

4.3.4. Deploying the standalone User Operator

Deploying the User Operator as a standalone component is more complicated than installing it using the Cluster Operator, but it 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_CERT_NAME** environment variable in **Deployment.spec.template.spec.containers[0].env** should be set to point to an OpenShift **Secret** which should contain the public key of 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 **ca.crt**.

   b. The **STRIMZI_CA_KEY_NAME** environment variable in

Deployment.spec.template.spec.containers[0].env should be set to point to an OpenShift **Secret** which should contain the private key of the Certificate Authority for signing new user certificates for TLS Client Authentication. The **Secret** should contain the private key of the Certificate Authority under the key **ca.key**.

c. 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.

d. 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.

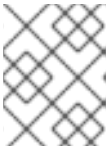
2. Deploy the User Operator.
On OpenShift this can be done using **oc apply**:

```
oc apply -f install/user-operator
```

3. Verify that the User Operator has been deployed successfully. On 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 2.10.2, “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

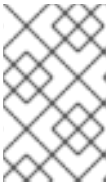
- A running Kafka cluster.
- A running Topic Operator.

Procedure

1. Prepare a file containing the **KafkaTopic** to be created

   An example KafkaTopic

```
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: orders
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
```



NOTE

It is recommended that the topic name given is a valid OpenShift resource name, as it is then not necessary to set the **KafkaTopic.spec.topicName** property. 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 2.3, "Cluster Operator".
- For more information about deploying the Topic Operator using the Cluster Operator, see Section 2.9.2, "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**

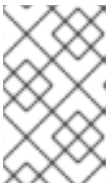
   An example KafkaTopic

```
apiVersion: kafka.strimzi.io/v1beta1
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

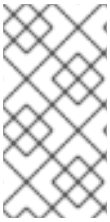
- For more information about the schema for **KafkaTopics**, see **KafkaTopic** schema reference.
- For more information about deploying a Kafka cluster, see Section 2.3, "Cluster Operator".
- For more information about deploying the Topic Operator using the Cluster Operator, see Section 2.9.2, "Deploying 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.
- **delete.topic.enable=true** (default)



NOTE

The **delete.topic.enable** property must be set to **true** in **Kafka.spec.kafka.config**. Otherwise, the steps outlined here will delete the **KafkaTopic** resource, but the Kafka topic and its data will remain. After reconciliation by the Topic Operator, the custom resource is then recreated.

Procedure

- Delete the **KafkaTopic** resource in OpenShift.
  On OpenShift this can be done using **oc**:

```
oc delete kafkatopic your-topic-name
```

Additional resources

- For more information about deploying a Kafka cluster using the Cluster Operator, see Section 2.3, “Cluster Operator”.
- For more information about deploying the Topic Operator using the Cluster Operator, see Section 2.9.2, “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 TLS authentication is always used for the communication between Kafka brokers and Zookeeper pods.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.

NOTE

TLS authentication is more commonly one-way, with one party authenticating the identity of another. For example, when HTTPS is used between a web browser and a web server, the server obtains 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/v1beta1
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 2.3, "Cluster Operator".
- For more information about configuring a listener that authenticates using TLS see Section 3.1.5, "Kafka broker listeners".
- For more information about deploying the Entity Operator, see Section 3.1.10, "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 (Simple Authentication and Security Layer) SCRAM-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 for each authentication exchange. This means that the exchange is resilient against replay attacks.

6.4.1. Supported SCRAM credentials

AMQ Streams supports SCRAM-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
- Authentication for unencrypted communication is required

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/v1beta1
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 2.3, “Cluster Operator”.
- For more information about configuring a listener that authenticates using SCRAM SHA see Section 3.1.5, “Kafka broker listeners”.
- For more information about deploying the Entity Operator, see Section 3.1.10, “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/v1beta1
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 2.3, “Cluster Operator”.
- For more information about deploying the Entity Operator, see Section 3.1.10, “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

- Delete the **KafkaUser** resource in OpenShift.
  On OpenShift this can be done using **oc**:

```
oc delete kafkauser your-user-name
```

Additional resources

- For more information about deploying the Cluster Operator, see Section 2.3, “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/v1beta1
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/v1beta1
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 contains the generated password in the **password** key, which is encoded with base64. In order to use the password it must be decoded.

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: Z2VuZXJhdGVkcGFzc3dvcmQ= # Generated password
```

For decode the generated password:

```
echo "Z2VuZXJhdGVkcGFzc3dvcmQ=" | base64 --decode
```

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:

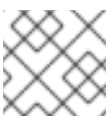
type

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 the rule applies. Simple Authorization supports four different resource types:

- Topics
- Consumer Groups
- Clusters
- Transactional IDs

  The resource type can be specified in the **type** property. Use **topic** for Topics, **group** for Consumer Groups, **cluster** for clusters, and **transactionalId** for Transactional IDs.

  Additionally, Topic, Group, and Transactional ID resources allow you 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/v1beta1
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
```

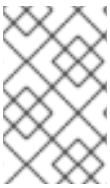
```
    name: my-group
    patternType: prefix
  operation: Read
```

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. USING THE AMQ STREAMS KAFKA BRIDGE

This chapter provides an overview of the AMQ Streams Kafka Bridge and helps you get started using its REST API to interact with AMQ Streams.



NOTE

For the full list of REST API endpoints and descriptions, including example requests and responses, see Kafka Bridge API reference . For information on how to deploy and configure the Kafka Bridge, see Section 2.7, “Kafka Bridge”.

7.1. OVERVIEW OF THE AMQ STREAMS KAFKA BRIDGE

The AMQ Streams Kafka Bridge provides an API for integrating HTTP-based clients with a Kafka cluster running on OpenShift. The API enables such clients to produce and consume messages without the requirement to use the native Kafka protocol.

The API has two main resources — **consumers** and **topics** — that are exposed and made accessible through endpoints to interact with consumers and producers in your Kafka cluster. The resources relate only to the Kafka Bridge, not the consumers and producers connected directly to Kafka.

You can:

- Send messages to a topic.
- Create and delete consumers.
- Subscribe consumers to topics, so that they start receiving messages from those topics.
- Unsubscribe consumers from topics.
- Assign partitions to consumers.
- Retrieve messages from topics.
- Commit a list of consumer offsets.
- Seek on a partition, so that a consumer starts receiving messages from the first or last offset position, or a given offset position.

Similar to a Kafka Connect cluster, you can deploy the Kafka Bridge into your OpenShift cluster using the Cluster Operator. For deployment instructions, see Section 2.7, “Kafka Bridge”.

After the Kafka Bridge is deployed, the Cluster Operator creates a Deployment, Service, and Pod in your OpenShift cluster, each named **strimzi-kafka-bridge** by default.

7.2. SUPPORTED CLIENTS FOR THE AMQ STREAMS KAFKA BRIDGE

You can use the Kafka Bridge to integrate both *internal* and *external* HTTP client applications with your Kafka cluster.

- Internal clients are container-based HTTP clients running in *the same* OpenShift cluster as the Kafka Bridge itself.

- External clients are HTTP clients running *outside* the OpenShift cluster in which the Kafka Bridge is deployed and running.

Internal clients can access the Kafka Bridge on the host and port defined in the **KafkaBridge** custom resource. External clients can access the Kafka Bridge through an OpenShift Route, a LoadBalancer Service, or a Kubernetes Ingress.

Additional resources

- For more information on configuring the host and port for the **KafkaBridge** resource, see Section 3.5.5.3, "Kafka Bridge HTTP configuration" .
- For more information on integrating external clients, see Section 7.4, "Accessing the AMQ Streams Kafka Bridge from outside of OpenShift".

7.3. SECURING THE AMQ STREAMS KAFKA BRIDGE

AMQ Streams does not currently provide any encryption, authentication, or authorization for the Kafka Bridge. This means that requests sent from external clients to the Kafka Bridge are:

- Not encrypted, and must use HTTP rather than HTTPS
- Sent without authentication

However, you can secure the Kafka Bridge using other methods, such as:

- OpenShift Network Policies that define which pods can access the Kafka Bridge.
- Reverse proxies with authentication or authorization, for example, OAuth2 proxies.
- API Gateways.
- Kubernetes Ingress or OpenShift Routes with TLS termination.

The Kafka Bridge supports TLS encryption and TLS and SASL authentication when connecting to the Kafka Brokers. Within your OpenShift cluster, you can configure:

- TLS or SASL-based authentication between the Kafka Bridge and your Kafka cluster
- A TLS-encrypted connection between the Kafka Bridge and your Kafka cluster.

For more information, see Section 3.5.4.1, "Authentication support in Kafka Bridge" .

You can use ACLs in Kafka brokers to restrict the topics that can be consumed and produced using the Kafka Bridge.

7.4. ACCESSING THE AMQ STREAMS KAFKA BRIDGE FROM OUTSIDE OF OPENSHIFT

After deployment, the AMQ Streams Kafka Bridge can only be accessed by applications running in the same OpenShift cluster. These applications use the ***kafka-bridge-name*-bridge-service** Service to access the API.

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

- Kubernetes Services of types LoadBalancer or NodePort
- Kubernetes Ingress resources
- OpenShift Routes

If you decide to create Services, use the following labels in the **selector** to configure the pods to which the service will route the traffic:

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

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

7.5. REQUESTS TO THE AMQ STREAMS KAFKA BRIDGE

7.5.1. Data formats and headers

Specify data formats and HTTP headers to ensure valid requests are submitted to the Kafka Bridge.

7.5.1.1. Content Type headers

API request and response bodies are always encoded as JSON.

- When performing consumer operations, **POST** requests must provide the following **Content-Type** header:

```
Content-Type: application/vnd.kafka.v2+json
```

- When performing producer operations, **POST** requests must provide **Content-Type** headers specifying the desired *embedded data format*, either **json** or **binary**, as shown in the following table.

Embedded data format	Content-Type header
JSON	**Content-Type: application/vnd.kafka.json.v2+json**
Binary	**Content-Type: application/vnd.kafka.binary.v2+json**

You set the embedded data format when creating a consumer using the **consumers/*groupid*** endpoint —for more information, see the next section.

7.5.1.2. Embedded data format

The embedded data format is the format of the Kafka messages that are transmitted, over HTTP, from a producer to a consumer using the Kafka Bridge. Two embedded data formats are supported: JSON and binary.

When creating a consumer using the /**consumers**/***groupid*** endpoint, the **POST** request body must specify an embedded data format of either JSON or binary. This is specified in the **format** field, for example:

```
{
  "name": "my-consumer",
  "format": "binary", (1)
...
}
```

(1) A binary embedded data format.

The embedded data format specified when creating a consumer must match the data format of the Kafka messages it will consume.

If you choose to specify a binary embedded data format, subsequent producer requests must provide the binary data in the request body as Base64-encoded strings. For example, when sending messages using the /**topics**/***topicname*** endpoint, **records.value** must be encoded in Base64:

```
{
  "records": [
    {
      "key": "my-key",
      "value": "ZWR3YXJkdGhldGhyZWVsZWdnZWRjYXQ="
    },
  ]
}
```

Producer requests must also provide a **Content-Type** header that corresponds to the embedded data format, for example, **Content-Type: application/vnd.kafka.binary.v2+json**.

7.5.1.3. Accept headers

After creating a consumer, all subsequent GET requests must provide an **Accept** header in the following format:

```
Accept: application/vnd.kafka.embedded-data-format.v2+json
```

The **embedded-data-format** is either **json** or **binary**.

For example, when retrieving records for a subscribed consumer using an embedded data format of JSON, include this Accept header:

```
Accept: application/vnd.kafka.json.v2+json
```

7.6. AMQ STREAMS KAFKA BRIDGE API RESOURCES

For the full list of REST API endpoints and descriptions, including example requests and responses, see Kafka Bridge API reference .

CHAPTER 8. 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.

8.1. 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.

8.1.1. 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.

8.2. 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**.

8.2.1. Cluster CA Secrets

Table 8.1. 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.
***<cluster>*-cluster-ca-cert**	**ca.crt**	The current certificate for the cluster CA.

Secret name	Field within Secret	Description
***<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.

8.2.2. Client CA Secrets

Table 8.2. 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.
***<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.

8.2.3. User Secrets

Table 8.3. 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

8.3. 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 cluster is not yet deployed.
- Your own X.509 certificates and keys in PEM format for the cluster CA or clients CA.
  - If you want to use a cluster or clients CA which is not a Root CA, you have to include the whole chain in the certificate file. The chain should be in the following order:
    1. The cluster or clients CA
    2. One or more intermediate CAs
    3. The root CA
  - All CAs in the chain should be configured as a CA in the X509v3 Basic Constraints.

Procedure

1. Put your CA certificate in the corresponding **Secret** (***<cluster>*-cluster-ca-cert** for the cluster CA or ***<cluster>*-clients-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>
```

2. Put your CA key in the corresponding **Secret** (***<cluster>*-cluster-ca** for the cluster CA or ***<cluster>*-clients-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>
```

3. Label both **Secrets** with labels **strimzi.io/kind=Kafka** and **strimzi.io/cluster=*<my-cluster>***:
   On OpenShift, run the following commands:

```
oc label secret <ca-cert-secret> strimzi.io/kind=Kafka strimzi.io/cluster=<my-cluster>
oc label secret <ca-key-secret> strimzi.io/kind=Kafka strimzi.io/cluster=<my-cluster>
```

4. Create the **Kafka** resource for your cluster, configuring either the **Kafka.spec.clusterCa** or the **Kafka.spec.clientsCa** object to *not* use generated CAs:

   Example fragment Kafka resource configuring the cluster CA to use certificates you supply for yourself

```
kind: Kafka
version: kafka.strimzi.io/v1beta1
spec:
  # ...
  clusterCa:
    generateCertificateAuthority: false
```

8.4. 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**.

8.4.1. 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.

8.4.2. 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 8.6, “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.

8.5. TLS CONNECTIONS

8.5.1. Zookeeper communication

Zookeeper does not support TLS itself. By deploying a TLS sidecar within every Zookeeper pod, the Cluster Operator is able to provide data encryption and authentication between Zookeeper nodes in a cluster. Zookeeper only communicates with the TLS sidecar over the loopback interface. The TLS 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.

8.5.2. 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 a TLS sidecar, as described above.

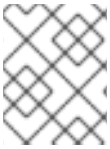
8.5.3. Topic and User Operators

Like the Cluster Operator, the Topic and User Operators each use a TLS sidecar when communicating with Zookeeper. The Topic Operator connects to Kafka brokers on port 9091.

8.5.4. 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.

8.6. 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

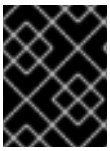
- For the procedure for configuring external clients to trust the cluster CA, see Section 8.7, "Configuring external clients to trust the cluster CA"

8.7. 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 8.6, "Configuring internal clients to trust the cluster CA"

CHAPTER 9. AMQ STREAMS AND KAFKA UPGRADES

AMQ Streams can be upgraded with no cluster downtime. Each version of AMQ Streams supports one or more versions of Apache Kafka: you can upgrade to a higher Kafka version as long as it is supported by your version of AMQ Streams. In some cases, you can also downgrade to a lower supported Kafka version.

Newer versions of AMQ Streams may support newer versions of Kafka, but you need to upgrade AMQ Streams *before* you can upgrade to a higher supported Kafka version.

9.1. UPGRADE PREREQUISITES

Before you begin the upgrade process, make sure that:

- AMQ Streams is installed. For instructions, see Chapter 2, *Getting started with AMQ Streams*.
- You are familiar with any upgrade changes described in the AMQ Streams 1.2 on Red Hat OpenShift Container Platform Release Notes.

9.2. UPGRADE PROCESS

Upgrading AMQ Streams is a two-stage process. To upgrade brokers and clients without downtime, you *must* complete the upgrade procedures in the following order:

1. Update your Cluster Operator to the latest AMQ Streams version.
   - Section 9.4, “Upgrading the Cluster Operator”
2. Upgrade all Kafka brokers and client applications to the latest Kafka version.
   - Section 9.5, “Upgrading Kafka”

9.3. KAFKA VERSIONS

AMQ Streams is based on a specific version of Apache Kafka.

AMQ Streams version	Kafka version
1.2	2.2.1

Kafka’s log message format version and inter-broker protocol version specify the log format version appended to messages and the version of protocol used in a cluster. As a result, the upgrade process involves making configuration changes to existing Kafka brokers and code changes to client applications (consumers and producers) to ensure the correct versions are used.

The following table shows the differences between Kafka versions:

Kafka version	Interbroker protocol version	Log message format version	Zookeeper version
2.1.1	2.1	2.1	3.4.13

2.2.1	2.2	2.2	3.4.14

Although Kafka versions may use the same version of Zookeeper, it is recommended that you update your Zookeeper cluster to use the newest Zookeeper binaries before proceeding with the main AMQ Streams upgrade.

Message format version

When a producer sends a message to a Kafka broker, the message is encoded using a specific format. The format can change between Kafka releases, so messages include a version identifying which version of the format they were encoded with. You can configure a Kafka broker to convert messages from newer format versions to a given older format version before the broker appends the message to the log.

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

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

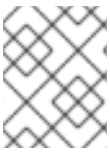
The default value of **message.format.version** for a topic is defined by the **log.message.format.version** that is set on the Kafka broker. You can manually set the **message.format.version** of a topic by modifying its topic configuration.

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

9.4. UPGRADING THE CLUSTER OPERATOR

The steps to upgrade your Cluster Operator deployment to use AMQ Streams 1.2 are outlined in this section.

The availability of Kafka clusters managed by the Cluster Operator is not affected by the upgrade operation.



NOTE

Refer to the documentation supporting a specific version of AMQ Streams for information on how to upgrade to that version.

9.4.1. Upgrading the Cluster Operator to a later version

This procedure describes how to upgrade a Cluster Operator deployment to a later version.

Prerequisites

- An existing Cluster Operator deployment.

Procedure

1. Backup the existing Cluster Operator resources:

```
oc get all -l app=strimzi -o yaml > strimzi-backup.yaml
```

2. Update the Cluster Operator.
   Modify the installation files according to the OpenShift project or Kubernetes namespace the Cluster Operator is running 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
   ```

   If you modified one or more environment variables in your existing Cluster Operator **Deployment**, edit the **install/cluster-operator/050-Deployment-cluster-operator.yaml** file to reflect the changes that you made in the new version of the Cluster Operator.

3. When you have an updated configuration, deploy it along with the rest of the install resources:

   ```
   oc apply -f install/cluster-operator
   ```

   Wait for the rolling updates to complete.

4. Get the image for the Kafka pod to ensure the upgrade was successful:

   ```
   oc get po my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
   ```

   The image tag shows the new AMQ Streams version followed by the Kafka version. For example, ***<New AMQ Streams version>*-kafka-*<Current Kafka version>***.

5. Update existing resources to handle deprecated custom resource properties.

   - AMQ Streams resource upgrades

You now have an updated Cluster Operator, but the version of Kafka running in the cluster it manages is unchanged.

What to do next

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

9.5. UPGRADING KAFKA

After you have upgraded your Cluster Operator, you can upgrade your brokers to a higher supported version of Kafka.

Kafka upgrades are performed using the Cluster Operator. How the Cluster Operator performs an upgrade depends on the differences between versions of:

- Interbroker protocol
- Log message format
- ZooKeeper

When the versions are the same for the current and target Kafka version, as is typically the case for a patch level upgrade, the Cluster Operator can upgrade through a single rolling update of the Kafka brokers.

When one or more of these versions differ, the Cluster Operator requires two or three rolling updates of the Kafka brokers to perform the upgrade.

Additional resources

- Section 9.4, "Upgrading the Cluster Operator"

9.5.1. Kafka version and image mappings

When upgrading Kafka, consider your settings for the **STRIMZI_KAFKA_IMAGES** and **Kafka.spec.kafka.version** properties.

- Each **Kafka** resource can be configured with a **Kafka.spec.kafka.version**.
- The Cluster Operator's **STRIMZI_KAFKA_IMAGES** environment variable provides a mapping between the Kafka version and the image to be used when that version is requested in a given **Kafka** resource.
  - If **Kafka.spec.kafka.image** is not configured, the default image for the given version is used.
  - If **Kafka.spec.kafka.image** is configured, the default image is overridden.

WARNING

The Cluster Operator cannot validate that an image actually contains a Kafka broker of the expected version. Take care to ensure that the given image corresponds to the given Kafka version.

9.5.2. Strategies for upgrading clients

The best approach to upgrading your client applications (including Kafka Connect connectors) depends on your particular circumstances.

Consuming applications need to receive messages in a message format that they understand. You can ensure that this is the case in one of two ways:

- By upgrading all the consumers for a topic *before* upgrading any of the producers.
- By having the brokers down-convert messages to an older format.

Using broker down-conversion puts extra load on the brokers, so it is not ideal to rely on down-conversion for all topics for a prolonged period of time. For brokers to perform optimally they should not be down converting messages at all.

Broker down-conversion is configured in two ways:

- The topic-level **message.format.version** configures it for a single topic.
- The broker-level **log.message.format.version** is the default for topics that do not have the topic-level **message.format.version** configured.

Messages published to a topic in a new-version format will be visible to consumers, because brokers perform down-conversion when they receive messages from producers, not when they are sent to consumers.

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

Consumers first

1. Upgrade all the consuming applications.
2. Change the broker-level **log.message.format.version** to the new version.
3. Upgrade all the producing applications.
   This strategy is straightforward, and avoids any broker down-conversion. However, it assumes that all consumers in your organization can be upgraded in a coordinated way, and it does not work for applications that are both consumers and producers. There is also a risk that, if there is a problem with the upgraded clients, new-format messages might get added to the message log so that you cannot revert to the previous consumer version.

Per-topic consumers first

For each topic:

1. Upgrade all the consuming applications.
2. Change the topic-level **message.format.version** to the new version.
3. Upgrade all the producing applications.
   This strategy avoids any broker down-conversion, and means you can proceed on a topic-by-topic basis. It does not work for applications that are both consumers and producers of the same topic. Again, it has the risk that, if there is a problem with the upgraded clients, new-format messages might get added to the message log.

Per-topic consumers first, with down conversion

For each topic:

1. Change the topic-level **message.format.version** to the old version (or rely on the topic defaulting to the broker-level **log.message.format.version**).
2. Upgrade all the consuming and producing applications.
3. Verify that the upgraded applications function correctly.
4. Change the topic-level **message.format.version** to the new version.
   This strategy requires broker down-conversion, but the load on the brokers is minimized because it is only required for a single topic (or small group of topics) at a time. It also works for applications that are both consumers and producers of the same topic. This approach ensures that the upgraded producers and consumers are working correctly before you commit to using the new message format version.

   The main drawback of this approach is that it can be complicated to manage in a cluster with many topics and applications.

Other strategies for upgrading client applications are also possible.

NOTE

It is also possible to apply multiple strategies. For example, for the first few applications and topics the "per-topic consumers first, with down conversion" strategy can be used. When this has proved successful another, more efficient strategy can be considered acceptable to use instead.

9.5.3. Upgrading Kafka brokers and client applications

This procedure describes how to upgrade a AMQ Streams Kafka cluster to a higher version of Kafka.

Prerequisites

For the **Kafka** resource to be upgraded, check:

- The Cluster Operator, which supports both versions of Kafka, is up and running.
- The **Kafka.spec.kafka.config** does not contain options that are not supported in the version of Kafka that you are upgrading to.
- Whether the **log.message.format.version** for the current Kafka version needs to be updated for the new version.
  Consult the Kafka versions table.

Procedure

1. Update the Kafka cluster configuration in an editor, as required:
   On OpenShift, use:

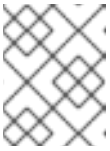
   ```
   oc edit kafka my-cluster
   ```

   a. If the **log.message.format.version** of the current Kafka version is the same as that of the new Kafka version, proceed to the next step.
   Otherwise, ensure that **Kafka.spec.kafka.config** has the **log.message.format.version** configured to the default for the current version.

   For example, if upgrading from Kafka 2.1.1:

   ```
   kind: Kafka
   spec:
     # ...
     kafka:
       version: 2.1.1
       config:
         log.message.format.version: "2.1"
         # ...
   ```

   If the **log.message.format.version** is unset, set it to the current version.



NOTE

The value of **log.message.format.version** must be a string to prevent it from being interpreted as a floating point number.

b. Change the **Kafka.spec.kafka.version** to specify the new version (leaving the **log.message.format.version** as the current version).
For example, if upgrading from Kafka 2.1.1 to 2.2.1:

```
apiVersion: v1alpha1
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.2.1 (1)
    config:
      log.message.format.version: "2.1" (2)
      # ...
```

(1) This is changed to the new version

(2) This remains at the current version

c. If the image for the Kafka version is different from the image defined in **STRIMZI_KAFKA_IMAGES** for the Cluster Operator, update **Kafka.spec.kafka.image**.
See Section 9.5.1, “Kafka version and image mappings”

2. Save and exit the editor, then wait for rolling updates to complete.
Check the update in the logs or by watching the pod state transitions:

On OpenShift, use:

```
oc logs -f <cluster-operator-pod-name> | grep -E "Kafka version upgrade from [0-9.]+ to [0-9.]+, phase ([0-9]+) of \1 completed"
```

```
oc get po -w
```

If the current and new versions of Kafka have different interbroker protocol versions, check the Cluster Operator logs for an **INFO** level message:

```
Reconciliation #<num>(watch) Kafka(<namespace>/<name>): Kafka version upgrade from <from-version> to <to-version>, phase 2 of 2 completed
```

Alternatively, if the current and new versions of Kafka have the same interbroker protocol version, check for:

```
Reconciliation #<num>(watch) Kafka(<namespace>/<name>): Kafka version upgrade from <from-version> to <to-version>, phase 1 of 1 completed
```

The rolling updates:

- Ensure each pod is using the broker binaries for the new version of Kafka

- Configure the brokers to send messages using the interbroker protocol of the new version of Kafka

NOTE

Clients are still using the old version, so brokers will convert messages to the old version before sending them to the clients. To minimize this additional load, updates the clients as quickly as possible.

3. Depending on your chosen strategy for upgrading clients, upgrade all client applications to use the new version of the client binaries.
   See Section 9.5.2, "Strategies for upgrading clients"

WARNING

You cannot downgrade after completing this step. If you need to revert the update at this point, follow the procedure Section 9.6.2, "Downgrading Kafka brokers and client applications".

   If required, set the version property for Kafka Connect and Mirror Maker as the new version of Kafka:

   a. For Kafka Connect, update **KafkaConnect.spec.version**

   b. For MIrror Maker, update **KafkaMirrorMaker.spec.version**

4. If the **log.message.format.version** identified in step 1 is the same as the new version proceed to the next step.
   Otherwise change the **log.message.format.version** in **Kafka.spec.kafka.config** to the default version for the new version of Kafka now being used.

   For example, if upgrading to 2.2.1:

```
apiVersion: v1alpha1
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.2.1
    config:
      log.message.format.version: "2.2"
      # ...
```

5. Wait for the Cluster Operator to update the cluster.
   The Kafka cluster and clients are now using the new Kafka version.

Additional resources

- See Section 9.6.2, "Downgrading Kafka brokers and client applications" for the procedure to downgrade a AMQ Streams Kafka cluster from one version to a lower version.

9.6. DOWNGRADING KAFKA

Kafka version downgrades are performed using the Cluster Operator.

Whether and how the Cluster Operator performs a downgrade depends on the differences between versions of:

- Interbroker protocol
- Log message format
- Zookeeper

9.6.1. Target downgrade version

How the Cluster Operator handles a downgrade operation depends on the **log.message.format.version**.

- If the target downgrade version of Kafka has the same **log.message.format.version** as the current version, the Cluster Operator downgrades by performing a single rolling restart of the brokers.
- If the target downgrade version of Kafka has a different **log.message.format.version**, downgrading is only possible if the running cluster has *always* had **log.message.format.version** set to the version used by the downgraded version.
  This is typically only the case if the upgrade procedure was aborted before the **log.message.format.version** was changed. In this case, the downgrade requires:
  - Two rolling restarts of the brokers if the interbroker protocol of the two versions is different
  - A single rolling restart if they are the same

9.6.2. Downgrading Kafka brokers and client applications

This procedure describes how you can downgrade a AMQ Streams Kafka cluster to a lower (previous) version of Kafka, such as downgrading from 2.2.1 to 2.1.1.

IMPORTANT

Downgrading is *not possible* if the new version has ever used a **log.message.format.version** that is not supported by the previous version, including when the default value for **log.message.format.version** is used. For example, this resource can be downgraded to Kafka version 2.1.1 because the **log.message.format.version** has not been changed:

```
apiVersion: v1alpha1
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.2.1
    config:
      log.message.format.version: "2.1"
      # ...
```

The downgrade would not be possible if the **log.message.format.version** was set at **"2.2"** or a value was absent (so that the parameter took the default value for a 2.2.1 broker of 2.2).

Prerequisites

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

- The Cluster Operator, which supports both versions of Kafka, is up and running.
- The **Kafka.spec.kafka.config** does not contain options that are not supported in the version of Kafka you are downgrading to.
- The **Kafka.spec.kafka.config** has a **log.message.format.version** that is supported by the version being downgraded to.

Procedure

1. Update the Kafka cluster configuration in an editor, as required:
   On OpenShift, use:

   ```
   oc edit kafka my-cluster
   ```

   a. Change the **Kafka.spec.kafka.version** to specify the previous version.
   For example, if downgrading from Kafka 2.2.1 to 2.1.1:

   ```
   apiVersion: v1alpha1
   kind: Kafka
   spec:
     # ...
     kafka:
       version: 2.1.1 (1)
       config:
         log.message.format.version: "2.1" (2)
         # ...
   ```

1. This is changed to the previous version
2. This is unchanged

NOTE

You must format the value of **log.message.format.version** as a string to prevent it from being interpreted as a floating point number.

b. If the image for the Kafka version is different from the image defined in **STRIMZI_KAFKA_IMAGES** for the Cluster Operator, update **Kafka.spec.kafka.image**. See Section 9.5.1, "Kafka version and image mappings"

2. Save and exit the editor, then wait for rolling updates to complete.
Check the update in the logs or by watching the pod state transitions:

On OpenShift use:

```
oc logs -f <cluster-operator-pod-name> | grep -E "Kafka version downgrade from [0-9.]+ to [0-9.]+, phase ([0-9]+) of \1 completed"
```

```
oc get po -w
```

If the previous and current versions of Kafka have different interbroker protocol versions, check the Cluster Operator logs for an **INFO** level message:

```
Reconciliation #<num>(watch) Kafka(<namespace>/<name>): Kafka version downgrade from <from-version> to <to-version>, phase 2 of 2 completed
```

Alternatively, if the previous and current versions of Kafka have the same interbroker protocol version, check for:

```
Reconciliation #<num>(watch) Kafka(<namespace>/<name>): Kafka version downgrade from <from-version> to <to-version>, phase 1 of 1 completed
```

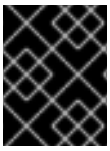
3. Downgrade all client applications (consumers) to use the previous version of the client binaries.
The Kafka cluster and clients are now using the previous Kafka version.

CHAPTER 10. AMQ STREAMS RESOURCE UPGRADES

For this release of AMQ Streams, resources that use the API version **kafka.strimzi.io/v1alpha1** must be updated to use **kafka.strimzi.io/v1beta1**.

The **kafka.strimzi.io/v1alpha1** API version is deprecated in release 1.2.

This section describes the upgrade steps for the resources.



IMPORTANT

The upgrade of resources *must* be performed after upgrading the Cluster Operator, so the Cluster Operator can understand the resources.

What if the resource upgrade does not take effect?

If the upgrade does not take effect, a warning is given in the logs on reconciliation to indicate that the resource cannot be updated until the **apiVersion** is updated.

To trigger the update, make a cosmetic change to the custom resource, such as adding an annotation.

Example annotation:

```
metadata:
  # ...
  annotations:
    upgrade: "Upgraded to kafka.strimzi.io/v1beta1"
```

10.1. UPGRADING KAFKA RESOURCES

Prerequisites

- A Cluster Operator supporting the **v1beta1** API version is up and running.

Procedure

Execute the following steps for each **Kafka** resource in your deployment.

1. Update the **Kafka** resource in an editor.

   ```
   oc edit kafka my-cluster
   ```

2. Replace:

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

   with:

   ```
   apiVersion:kafka.strimzi.io/v1beta1
   ```

3. If the **Kafka** resource has:

   ```
   Kafka.spec.topicOperator
   ```

Replace it with:

```
Kafka.spec.entityOperator.topicOperator
```

For example, replace:

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

with:

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

4. If present, move:

```
Kafka.spec.entityOperator.affinity
```

```
Kafka.spec.entityOperator.tolerations
```

to:

```
Kafka.spec.entityOperator.template.pod.affinity
```

```
Kafka.spec.entityOperator.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  entityOperator:
    affinity {}
    tolerations {}
```

to:

```
spec:
  # ...
  entityOperator:
    template:
      pod:
        affinity {}
        tolerations {}
```

5. If present, move:

```
Kafka.spec.kafka.affinity
```

```
Kafka.spec.kafka.tolerations
```

to:

```
Kafka.spec.kafka.template.pod.affinity
```

```
Kafka.spec.kafka.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  kafka:
    affinity {}
    tolerations {}
```

to:

```
spec:
  # ...
  kafka:
    template:
      pod:
        affinity {}
        tolerations {}
```

6. If present, move:

```
Kafka.spec.zookeeper.affinity
```

```
Kafka.spec.zookeeper.tolerations
```

to:

```
Kafka.spec.zookeeper.template.pod.affinity
```

```
Kafka.spec.zookeeper.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  zookeeper:
    affinity {}
    tolerations {}
```

to:

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

```
    template:
      pod:
        affinity {}
        tolerations {}
```

7. Save the file, exit the editor and wait for the updated resource to be reconciled.

10.2. UPGRADING KAFKA CONNECT RESOURCES

Prerequisites

- A Cluster Operator supporting the **v1beta1** API version is up and running.

Procedure

Execute the following steps for each **KafkaConnect** resource in your deployment.

1. Update the **KafkaConnect** resource in an editor.

```
oc edit kafkaconnect my-connect
```

2. Replace:

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

with:

```
apiVersion:kafka.strimzi.io/v1beta1
```

3. If present, move:

```
KafkaConnect.spec.affinity
```

```
KafkaConnect.spec.tolerations
```

to:

```
KafkaConnect.spec.template.pod.affinity
```

```
KafkaConnect.spec.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  affinity {}
  tolerations {}
```

to:

```
spec:
  # ...
```

```
  template:
    pod:
      affinity {}
      tolerations {}
```

4. Save the file, exit the editor and wait for the updated resource to be reconciled.

10.3. UPGRADING KAFKA CONNECT S2I RESOURCES

Prerequisites

- A Cluster Operator supporting the **v1beta1** API version is up and running.

Procedure

Execute the following steps for each **KafkaConnectS2I** resource in your deployment.

1. Update the **KafkaConnectS2I** resource in an editor.

```
oc edit kafkaconnects2i my-connect
```

2. Replace:

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

with:

```
apiVersion:kafka.strimzi.io/v1beta1
```

3. If present, move:

```
KafkaConnectS2I.spec.affinity
```

```
KafkaConnectS2I.spec.tolerations
```

to:

```
KafkaConnectS2I.spec.template.pod.affinity
```

```
KafkaConnectS2I.spec.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  affinity {}
  tolerations {}
```

to:

```
spec:
  # ...
```

```
  template:
    pod:
      affinity {}
      tolerations {}
```

4. Save the file, exit the editor and wait for the updated resource to be reconciled.

10.4. UPGRADING KAFKA MIRROR MAKER RESOURCES

Prerequisites

- A Cluster Operator supporting the **v1beta1** API version is up and running.

Procedure

Execute the following steps for each **KafkaMirrorMaker** resource in your deployment.

1. Update the **KafkaMirrorMaker** resource in an editor.

```
oc edit kafkamirrormaker my-connect
```

2. Replace:

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

with:

```
apiVersion:kafka.strimzi.io/v1beta1
```

3. If present, move:

```
KafkaConnectMirrorMaker.spec.affinity
```

```
KafkaConnectMirrorMaker.spec.tolerations
```

to:

```
KafkaConnectMirrorMaker.spec.template.pod.affinity
```

```
KafkaConnectMirrorMaker.spec.template.pod.tolerations
```

For example, move:

```
spec:
  # ...
  affinity {}
  tolerations {}
```

to:

```
spec:
  # ...
```

```
template:
  pod:
    affinity {}
    tolerations {}
```

4. Save the file, exit the editor and wait for the updated resource to be reconciled.

10.5. UPGRADING KAFKA TOPIC RESOURCES

Prerequisites

- A Topic Operator supporting the **v1beta1** API version is up and running.

Procedure

Execute the following steps for each **KafkaTopic** resource in your deployment.

1. Update the **KafkaTopic** resource in an editor.

```
oc edit kafkatopic my-topic
```

2. Replace:

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

with:

```
apiVersion:kafka.strimzi.io/v1beta1
```

3. Save the file, exit the editor and wait for the updated resource to be reconciled.

10.6. UPGRADING KAFKA USER RESOURCES

Prerequisites

- A User Operator supporting the **v1beta1** API version is up and running.

Procedure

Execute the following steps for each **KafkaUser** resource in your deployment.

1. Update the **KafkaUser** resource in an editor.

```
oc edit kafkauser my-user
```

2. Replace:

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

with:

```
apiVersion:kafka.strimzi.io/v1beta1
```

3. Save the file, exit the editor and wait for the updated resource to be reconciled.

CHAPTER 11. UNINSTALLING AMQ STREAMS

This procedure describes how to uninstall AMQ Streams and remove resources related to the deployment.

Prerequisites

In order to perform this procedure, identify resources created specifically for a deployment and referenced from the AMQ Streams resource.

Such resources include:

- Secrets (Custom CAs and certificates, Kafka Connect secrets, and other Kafka secrets)
- Logging **ConfigMaps** (of type **external**)

These are resources referenced by **Kafka**, **KafkaConnect**, **KafkaConnectS2I**, or **KafkaMirrorMaker** configuration.

Procedure

1. Delete the cluster operator **Deployment**, related **CustomResourceDefinitions**, and **RBAC** resources:

```
oc delete -f install/cluster-operator
```

WARNING

Deleting **CustomResourceDefinitions** results in the garbage collection of the corresponding custom resources (**Kafka**, **KafkaConnect**, **KafkaConnectS2I**, or **KafkaMirrorMaker**) and the resources dependent on them (Deployments, StatefulSets, and other dependent resources).

2. Delete the resources you identified in the prerequisites.

CHAPTER 12. CHECKING THE STATUS OF A CUSTOM RESOURCE

This procedure describes how to find the status of a custom resource.

Prerequisites

- An OpenShift cluster
- A running Cluster Operator

Procedure

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

  ```
  oc get kafka <kafka_resource_name> -o jsonpath='{.status}'
  ```

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

Additional resources

- Section 2.2.2, "AMQ Streams custom resource status"
- For more information about using JSONPath, see JSONPath support.

APPENDIX A. CONFIGURABLE LOGGERS

Logging allows you to diagnose error and performance issues for AMQ Streams.

The following logger implementations are used in AMQ Streams:

- **log4j** logger for Kafka and Zookeeper
- **log4j2** logger for Topic Operator, User Operator, and other components

AMQ Streams components have their own configurable loggers.

- Kafka loggers
- Kafka Connect loggers
- Kafka Connect with Source2Image loggers
- Kafka Mirror Maker loggers
- Topic Operator loggers
- User Operator loggers

APPENDIX B. FREQUENTLY ASKED QUESTIONS

B.1. CLUSTER OPERATOR

B.1.1. Why do I need cluster admin privileges to install AMQ Streams?

To install AMQ Streams, you must have the ability to create Custom Resource Definitions (CRDs). CRDs instruct OpenShift about resources that are specific to AMQ Streams, such as Kafka, KafkaConnect, and so on. Because CRDs are a cluster-scoped resource rather than being scoped to a particular OpenShift namespace, they typically require cluster admin privileges to install.

In addition, you must also have the ability to create ClusterRoles and ClusterRoleBindings. Like CRDs, these are cluster-scoped resources that typically require cluster admin privileges.

The cluster administrator can inspect all the resources being installed (in the /**install**/ directory) to assure themselves that the **ClusterRoles** do not grant unnecessary privileges. For more information about why the Cluster Operator installation resources grant the ability to create **ClusterRoleBindings** see the following question.

After installation, the Cluster Operator will run as a regular **Deployment**; any non-admin user with privileges to access the **Deployment** can configure it.

By default, normal users will not have the privileges necessary to manipulate the custom resources, such as **Kafka**, **KafkaConnect** and so on, which the Cluster Operator deals with. These privileges can be granted using normal RBAC resources by the cluster administrator. See this procedure for more details of how to do this.

B.1.2. Why does the Cluster Operator require the ability to create ClusterRoleBindings? Is that not a security risk?

OpenShift has built-in privilege escalation prevention. That means that the Cluster Operator cannot grant privileges it does not have itself. Which in turn means that the Cluster Operator needs to have the privileges necessary for *all* the components it orchestrates.

In the context of this question there are two places where the Cluster Operator needs to create bindings to **ClusterRoleBindings** to **ServiceAccounts**:

1. The Topic Operator and User Operator need to be able to manipulate **KafkaTopics** and **KafkaUsers**, respectively. The Cluster Operator therefore needs to be able to grant them this access, which it does by creating a **Role** and **RoleBinding**. For this reason the Cluster Operator itself needs to be able to create **Roles** and **RoleBindings** in the namespace that those operators will run in. However, because of the privilege escalation prevention, the Cluster Operator cannot grant privileges it does not have itself (in particular it cannot grant such privileges in namespace it cannot access).

2. When using rack-aware partition assignment, AMQ Streams needs to be able to discover the failure domain (for example, the Availability Zone in AWS) of the node on which a broker pod is assigned. To do this the broker pod needs to be able to get information about the **Node** it is running on. A **Node** is a cluster-scoped resource, so access to it can only be granted via a **ClusterRoleBinding** (not a namespace-scoped **RoleBinding**). Therefore the Cluster Operator needs to be able to create **ClusterRoleBindings**. But again, because of privilege escalation prevention, the Cluster Operator cannot grant privileges it does not have itself (so it cannot, for example, create a **ClusterRoleBinding** to a **ClusterRole** to grant privileges that the Cluster Operator does not not already have).

B.1.3. Why can standard OpenShift users not create the custom resource (Kafka, KafkaTopic, and so on)?

Because, when they installed AMQ Streams, the OpenShift cluster administrator did not grant the necessary privileges to standard users.

See this FAQ answer for more details.

B.1.4. 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.

B.1.5. Hostname verification fails when connecting to NodePorts using TLS

Currently, off-cluster access using NodePorts with TLS encryption enabled does not support TLS hostname verification. As a result, the clients that verify the hostname will fail to connect. For example, the Java client will fail with the following exception:

```
Caused by: java.security.cert.CertificateException: No subject alternative names matching IP address 168.72.15.231 found
    at sun.security.util.HostnameChecker.matchIP(HostnameChecker.java:168)
    at sun.security.util.HostnameChecker.match(HostnameChecker.java:94)
    at sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:455)
    at sun.security.ssl.X509TrustManagerImpl.checkIdentity(X509TrustManagerImpl.java:436)
    at sun.security.ssl.X509TrustManagerImpl.checkTrusted(X509TrustManagerImpl.java:252)
    at sun.security.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:136)
    at sun.security.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:1501)
    ... 17 more
```

To connect, you must disable hostname verification. In the Java client, you can do this by setting the configuration option **ssl.endpoint.identification.algorithm** to an empty string.

When configuring the client using a properties file, you can do it this way:

```
ssl.endpoint.identification.algorithm=
```

When configuring the client directly in Java, set the configuration option to an empty string:

```
props.put("ssl.endpoint.identification.algorithm", "");
```

APPENDIX C. CUSTOM RESOURCE API REFERENCE

C.1. KAFKA SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka and Zookeeper clusters, and Topic Operator.
KafkaSpec	
status	The status of the Kafka and Zookeeper clusters, and Topic Operator.
KafkaStatus	

C.2. KAFKASPEC SCHEMA REFERENCE

Used in: **Kafka**

Property	Description
kafka	Configuration of the Kafka cluster.
KafkaClusterSpec	
zookeeper	Configuration of the Zookeeper cluster.
ZookeeperClusterSpec	
topicOperator	**The property topicOperator has been deprecated. This feature should now be configured at path spec.entityOerator.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	

Property	Description
maintenanceTimeWindows string array	A list of time windows for the maintenance tasks (that is, certificates renewal). Each time window is defined by a cron expression.

C.3. KAFKACLUSTERSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
replicas integer	The number of pods in the cluster.
image string	The docker image for the pods. The default value depends on the configured **Kafka.spec.kafka.version**.
storage **EphemeralStorage**, **PersistentClaimStorage**, **JbodStorage**	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, jbod].
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	The image of the init container used for initializing the **broker.rack**.

Property	Description
string	
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.kafka.template.pod.affinity.** The pod's affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.kafka.template.pod.tolerations.** The pod's tolerations.See external documentation of core/v1 toleration.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
jvmOptions **JvmOptions**	JVM Options for pods.
resources **ResourceRequirements**	Resource constraints (limits and requests).
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
logging **InlineLogging**, **ExternalLogging**	Logging configuration for Kafka. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
tlsSidecar **TlsSidecar**	TLS sidecar configuration.
template **KafkaClusterTemplate**	Template for Kafka cluster resources. The template allows users to specify how are the **StatefulSet**, **Pods** and **Services** generated.

Property	Description
version	The kafka broker version. Defaults to 2.2.1. Consult the user documentation to understand the process required to upgrade or downgrade the version.
string	

C.4. EPHEMERALSTORAGE SCHEMA REFERENCE

Used in: **JbodStorage**, **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**.

Property	Description
id	Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'.
integer	
type	Must be **ephemeral**.
string	

C.5. PERSISTENTCLAIMSTORAGE SCHEMA REFERENCE

Used in: **JbodStorage**, **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**.

Property	Description
type	Must be **persistent-claim**.
string	
size	When type=persistent-claim, defines the size of the persistent volume claim (i.e 1Gi). Mandatory when type=persistent-claim.
string	
selector	Specifies a specific persistent volume to use. It contains key:value pairs representing labels for selecting such a volume.
map	

Property	Description
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.
id integer	Storage identification number. It is mandatory only for storage volumes defined in a storage of type 'jbod'.
overrides **PersistentClaimStorageOverride** array	Overrides for individual brokers. The **overrides** field allows to specify a different configuration for different brokers.

C.6. PERSISTENTCLAIMSTORAGEOVERRIDE SCHEMA REFERENCE

Used in: **PersistentClaimStorage**

Property	Description
class string	The storage class to use for dynamic volume allocation for this broker.
broker integer	Id of the kafka broker (broker identifier).

C.7. JBODSTORAGE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

The **type** property is a discriminator that distinguishes the use of the type **JbodStorage** from **EphemeralStorage**, **PersistentClaimStorage**. It must have the value **jbod** for the type **JbodStorage**.

Property	Description
type string	Must be **jbod**.

Property	Description
volumes	List of volumes as Storage objects representing the JBOD disks array.
EphemeralStorage, **PersistentClaimStorage** array	

C.8. KAFKALISTENERS SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
plain	Configures plain listener on port 9092.
KafkaListenerPlain	
tls	Configures TLS listener on port 9093.
KafkaListenerTls	
external	Configures external listener on port 9094. The type depends on the value of the **external.type** property within the given object, which must be one of [route, loadbalancer, nodeport, ingress].
KafkaListenerExternalRoute, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalIngress**	

C.9. KAFKALISTENERPLAIN SCHEMA REFERENCE

Used in: **KafkaListeners**

Property	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**	

Property	Description
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list.See external documentation of networking.k8s.io/v1 networkpolicypeer.

C.10. KAFKALISTENERAUTHENTICATIONTLS SCHEMA REFERENCE

Used in: **KafkaListenerExternalIngress**, **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**.

Property	Description
type string	Must be **tls**.

C.11. KAFKALISTENERAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE

Used in: **KafkaListenerExternalIngress**, **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**.

Property	Description
type string	Must be **scram-sha-512**.

C.12. KAFKALISTENERTLS SCHEMA REFERENCE

Used in: **KafkaListeners**

Property	Description
authentication **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**	Authentication configuration for this listener. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list.See external documentation of networking.k8s.io/v1 networkpolicypeer.

C.13. KAFKALISTENEREXTERNALROUTE SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalRoute** from **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalIngress**. It must have the value **route** for the type **KafkaListenerExternalRoute**.

Property	Description
type string	Must be **route**.
authentication **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
overrides **RouteListenerOverride**	Overrides for external bootstrap and broker services and externally advertised addresses.
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list.See external documentation of networking.k8s.io/v1 networkpolicypeer.

C.14. ROUTELISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalRoute**

Property	Description
bootstrap	External bootstrap service configuration.
RouteListenerBootstrapOverride	
brokers	External broker services configuration.
RouteListenerBrokerOverride array	

C.15. ROUTELISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **RouteListenerOverride**

Property	Description
address	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
string	
host	Host for the bootstrap route. This field will be used in the **spec.host** field of the OpenShift Route.
string	

C.16. ROUTELISTENERBROKEROVERRIDE SCHEMA REFERENCE

Used in: **RouteListenerOverride**

Property	Description
broker	Id of the kafka broker (broker identifier).
integer	
advertisedHost	The host name which will be used in the brokers' **advertised.brokers**.
string	
advertisedPort	The port number which will be used in the brokers' **advertised.brokers**.
integer	

Property	Description
host	Host for the broker route. This field will be used in the **spec.host** field of the OpenShift Route.
string	

C.17. KAFKALISTENEREXTERNALLOADBALANCER SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalLoadBalancer** from **KafkaListenerExternalRoute**, **KafkaListenerExternalNodePort**, **KafkaListenerExternalIngress**. It must have the value **loadbalancer** for the type **KafkaListenerExternalLoadBalancer**.

Property	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**	
overrides	Overrides for external bootstrap and broker services and externally advertised addresses.
LoadBalancerListenerOverride	
networkPolicyPeers	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list.See external documentation of networking.k8s.io/v1 networkpolicypeer.
NetworkPolicyPeer array	
tls	Enables TLS encryption on the listener. By default set to **true** for enabled TLS encryption.
boolean	

C.18. LOADBALANCERLISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalLoadBalancer**

Property	Description
bootstrap **LoadBalancerListenerBootstrapOverride**	External bootstrap service configuration.
brokers **LoadBalancerListenerBrokerOverride** array	External broker services configuration.

C.19. LOADBALANCERLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **LoadBalancerListenerOverride**

Property	Description
address string	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
dnsAnnotations map	Annotations which will be added to the Service resource. You can use this field to instrument DNS providers such as External DNS.

C.20. LOADBALANCERLISTENERBROKEROVERRIDE SCHEMA REFERENCE

Used in: **LoadBalancerListenerOverride**

Property	Description
broker integer	Id of the kafka broker (broker identifier).
advertisedHost string	The host name which will be used in the brokers' **advertised.brokers**.
advertisedPort integer	The port number which will be used in the brokers' **advertised.brokers**.
dnsAnnotations map	Annotations which will be added to the Service resources for individual brokers. You can use this field to instrument DNS providers such as External DNS.

C.21. KAFKALISTENEREXTERNALNODEPORT SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type **KafkaListenerExternalNodePort** from **KafkaListenerExternalRoute**, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalIngress**. It must have the value **nodeport** for the type **KafkaListenerExternalNodePort**.

Property	Description
type string	Must be **nodeport**.
authentication **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
overrides **NodePortListenerOverride**	Overrides for external bootstrap and broker services and externally advertised addresses.
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list.See external documentation of networking.k8s.io/v1 networkpolicypeer.
tls boolean	Enables TLS encryption on the listener. By default set to **true** for enabled TLS encryption.

C.22. NODEPORTLISTENEROVERRIDE SCHEMA REFERENCE

Used in: **KafkaListenerExternalNodePort**

Property	Description
bootstrap **NodePortListenerBootstrapOverride**	External bootstrap service configuration.
brokers	External broker services configuration.

Property	Description
NodePortListenerBrokerOverride array	

C.23. NODEPORTLISTENERBOOTSTRAPOVERRIDE SCHEMA REFERENCE

Used in: **NodePortListenerOverride**

Property	Description
address	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
string	
nodePort	Node port for the bootstrap service.
integer	

C.24. NODEPORTLISTENERBROKEROVERRIDE SCHEMA REFERENCE

Used in: **NodePortListenerOverride**

Property	Description
broker	Id of the kafka broker (broker identifier).
integer	
advertisedHost	The host name which will be used in the brokers' **advertised.brokers**.
string	
advertisedPort	The port number which will be used in the brokers' **advertised.brokers**.
integer	
nodePort	Node port for the broker service.
integer	

C.25. KAFKALISTENEREXTERNALINGRESS SCHEMA REFERENCE

Used in: **KafkaListeners**

The **type** property is a discriminator that distinguishes the use of the type

KafkaListenerExternalIngress from **KafkaListenerExternalRoute**, **KafkaListenerExternalLoadBalancer**, **KafkaListenerExternalNodePort**. It must have the value **ingress** for the type **KafkaListenerExternalIngress**.

Property	Description
type string	Must be **ingress**.
authentication **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**	Authentication configuration for Kafka brokers. The type depends on the value of the **authentication.type** property within the given object, which must be one of [tls, scram-sha-512].
configuration **IngressListenerConfiguration**	Overrides for external bootstrap and broker services and externally advertised addresses.
networkPolicyPeers NetworkPolicyPeer array	List of peers which should be able to connect to this listener. Peers in this list are combined using a logical OR operation. If this field is empty or missing, all connections will be allowed for this listener. If this field is present and contains at least one item, the listener only allows the traffic which matches at least one item in this list.See external documentation of networking.k8s.io/v1 networkpolicypeer.

C.26. INGRESSLISTENERCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaListenerExternalIngress**

Property	Description
bootstrap **IngressListenerBootstrapConfiguration**	External bootstrap ingress configuration.
brokers **IngressListenerBrokerConfiguration** array	External broker ingress configuration.

C.27. INGRESSLISTENERBOOTSTRAPCONFIGURATION SCHEMA REFERENCE

Used in: **IngressListenerConfiguration**

Property	Description
address string	Additional address name for the bootstrap service. The address will be added to the list of subject alternative names of the TLS certificates.
dnsAnnotations map	Annotations which will be added to the Ingress resource. You can use this field to instrument DNS providers such as External DNS.
host string	Host for the bootstrap route. This field will be used in the Ingress resource.

C.28. INGRESSLISTENERBROKERCONFIGURATION SCHEMA REFERENCE

Used in: **IngressListenerConfiguration**

Property	Description
broker integer	Id of the kafka broker (broker identifier).
advertisedHost string	The host name which will be used in the brokers' **advertised.brokers**.
advertisedPort integer	The port number which will be used in the brokers' **advertised.brokers**.
host string	Host for the broker ingress. This field will be used in the Ingress resource.
dnsAnnotations map	Annotations which will be added to the Ingress resources for individual brokers. You can use this field to instrument DNS providers such as External DNS.

C.29. 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**.

Property	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.

C.30. RACK SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
topologyKey string	A key that matches labels assigned to the OpenShift or Kubernetes cluster nodes. The value of the label is used to set the broker's **broker.rack** config.

C.31. PROBE SCHEMA REFERENCE

Used in: **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **TlsSidecar**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Property	Description
failureThreshold integer	Minimum consecutive failures for the probe to be considered failed after having succeeded. Defaults to 3. Minimum value is 1.
initialDelaySeconds integer	The initial delay before first the health is first checked.
periodSeconds integer	How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.

Property	Description
successThreshold integer	Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.
timeoutSeconds integer	The timeout for each attempted health check.

C.32. JVMOPTIONS SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMakerSpec**, **ZookeeperClusterSpec**

Property	Description
-XX map	A map of -XX options to the JVM.
-Xms string	-Xms option to to the JVM.
-Xmx string	-Xmx option to to the JVM.
gcLoggingEnabled boolean	Specifies whether the Garbage Collection logging is enabled. The default is true.

C.33. RESOURCEREQUIREMENTS SCHEMA REFERENCE

Used in: **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **KafkaClusterSpec**, **KafkaConnectS2ISpec**, **KafkaConnectSpec**, **KafkaMirrorMakerSpec**, **TlsSidecar**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Property	Description
limits map	

Property	Description
requests	
map	

C.34. INLINELOGGING SCHEMA REFERENCE

Used in: **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **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**.

Property	Description
type	Must be **inline**.
string	
loggers	A Map from logger name to logger level.
map	

C.35. EXTERNALLOGGING SCHEMA REFERENCE

Used in: **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **KafkaBridgeSpec**, **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**.

Property	Description
type	Must be **external**.
string	
name	The name of the **ConfigMap** from which to get the logging configuration.
string	

C.36. TLSSIDECAR SCHEMA REFERENCE

Used in: **EntityOperatorSpec**, **KafkaClusterSpec**, **TopicOperatorSpec**, **ZookeeperClusterSpec**

Property	Description
image	The docker image for the container.
string	
livenessProbe	Pod liveness checking.
Probe	
logLevel	The log level for the TLS sidecar. Default value is **notice**.
string (one of [emerg, debug, crit, err, alert, warning, notice, info])	
readinessProbe	Pod readiness checking.
Probe	
resources	Resource constraints (limits and requests).
ResourceRequirements	

C.37. KAFKACLUSTERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaClusterSpec**

Property	Description
statefulset	Template for Kafka **StatefulSet**.
ResourceTemplate	
pod	Template for Kafka **Pods**.
PodTemplate	
bootstrapService	Template for Kafka bootstrap **Service**.
ResourceTemplate	
brokersService	Template for Kafka broker **Service**.
ResourceTemplate	

Property	Description
externalBootstrapIngress	Template for Kafka external bootstrap **Ingress**.
ResourceTemplate	
externalBootstrapRoute	Template for Kafka external bootstrap **Route**.
ResourceTemplate	
externalBootstrapService	Template for Kafka external bootstrap **Service**.
ResourceTemplate	
perPodIngress	Template for Kafka per-pod **Ingress** used for access from outside of Kubernetes.
ResourceTemplate	
perPodRoute	Template for Kafka per-pod **Routes** used for access from outside of OpenShift.
ResourceTemplate	
perPodService	Template for Kafka per-pod **Services** used for access from outside of Kubernetes.
ResourceTemplate	
podDisruptionBudget	Template for Kafka **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

C.38. RESOURCETEMPLATE SCHEMA REFERENCE

Used in: **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata	Metadata which should be applied to the resource.
MetadataTemplate	

C.39. METADATATEMPLATE SCHEMA REFERENCE

Used in: **PodDisruptionBudgetTemplate**, **PodTemplate**, **ResourceTemplate**

Property	Description
labels map	Labels which should be added to the resource template. Can be applied to different resources such as **StatefulSets**, **Deployments**, **Pods**, and **Services**.
annotations map	Annotations which should be added to the resource template. Can be applied to different resources such as **StatefulSets**, **Deployments**, **Pods**, and **Services**.

C.40. PODTEMPLATE SCHEMA REFERENCE

Used in: **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata **MetadataTemplate**	Metadata which should be applied to the resource.
imagePullSecrets LocalObjectReference array	List of references to secrets in the same namespace to use for pulling any of the images used by this Pod.See external documentation of core/v1 localobjectreference.
securityContext PodSecurityContext	Configures pod-level security attributes and common container settings.See external documentation of core/v1 podsecuritycontext.
terminationGracePeriodSeconds integer	The grace period is the duration in seconds after the processes running in the pod are sent a termination signal and the time when the processes are forcibly halted with a kill signal. Set this value longer than the expected cleanup time for your process.Value must be non-negative integer. The value zero indicates delete immediately. Defaults to 30 seconds.
affinity Affinity	The pod's affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	The pod's tolerations.See external documentation of core/v1 toleration.

C.41. PODDISRUPTIONBUDGETTEMPLATE SCHEMA REFERENCE

Used in: **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaMirrorMakerTemplate**, **ZookeeperClusterTemplate**

Property	Description
metadata **MetadataTemplate**	Metadata which should be applied to the **PodDistruptionBugetTemplate** resource.
maxUnavailable integer	Maximum number of unavailable pods to allow voluntary Pod eviction. A Pod eviction will only be allowed when "maxUnavailable" or fewer pods are unavailable after the eviction. Setting this value to 0 will prevent all voluntary evictions and the pods will need to be evicted manually. Defaults to 1.

C.42. ZOOKEEPERCLUSTERSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	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	**The property affinity has been deprecated. This feature should now be configured at path spec.zookeeper.template.pod.affinity.** The pod's affinity rules.See external documentation of core/v1 affinity.

Property	Description
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.zookeeper.template.pod.tolerations.** The pod's tolerations.See external documentation of core/v1 toleration.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
jvmOptions **JvmOptions**	JVM Options for pods.
resources **ResourceRequirements**	Resource constraints (limits and requests).
metrics map	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
logging **InlineLogging**, **ExternalLogging**	Logging configuration for Zookeeper. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
tlsSidecar **TlsSidecar**	TLS sidecar configuration.
template **ZookeeperClusterTemplate**	Template for Zookeeper cluster resources. The template allows users to specify how are the **StatefulSet**, **Pods** and **Services** generated.

C.43. ZOOKEEPERCLUSTERTEMPLATE SCHEMA REFERENCE

Used in: **ZookeeperClusterSpec**

Property	Description
statefulset	Template for Zookeeper **StatefulSet**.
ResourceTemplate	
pod	Template for Zookeeper **Pods**.
PodTemplate	
clientService	Template for Zookeeper client **Service**.
ResourceTemplate	
nodesService	Template for Zookeeper nodes **Service**.
ResourceTemplate	
podDisruptionBudget	Template for Zookeeper **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

C.44. TOPICOPERATORSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	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	
affinity	Pod affinity rules.See external documentation of core/v1 affinity.

Property	Description
Affinity	
resources **ResourceRequirements**	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].
jvmOptions **EntityOperatorJvmOptions**	JVM Options for pods.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.

C.45. ENTITYOPERATORJVMOPTIONS SCHEMA REFERENCE

Used in: **EntityTopicOperatorSpec**, **EntityUserOperatorSpec**, **TopicOperatorSpec**

Property	Description
gcLoggingEnabled boolean	Specifies whether the Garbage Collection logging is enabled. The default is true.

C.46. ENTITYOPERATORSPEC SCHEMA REFERENCE

Used in: **KafkaSpec**

Property	Description
topicOperator **EntityTopicOperatorSpec**	Configuration of the Topic Operator.
userOperator **EntityUserOperatorSpec**	Configuration of the User Operator.
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations.See external documentation of core/v1 toleration.
tlsSidecar **TlsSidecar**	TLS sidecar configuration.
template **EntityOperatorTemplate**	Template for Entity Operator resources. The template allows users to specify how is the **Deployment** and **Pods** generated.

C.47. ENTITYTOPICOPERATORSPEC SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
watchedNamespace string	The namespace the Topic Operator should watch.
image string	The image to use for the Topic Operator.
reconciliationIntervalSeconds integer	Interval between periodic reconciliations.

Property	Description
zookeeperSessionTimeoutSeconds integer	Timeout for the Zookeeper session.
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
resources **ResourceRequirements**	Resource constraints (limits and requests).
topicMetadataMaxAttempts integer	The number of attempts at getting topic metadata.
logging **InlineLogging**, **ExternalLogging**	Logging configuration. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
jvmOptions **EntityOperatorJvmOptions**	JVM Options for pods.

C.48. ENTITYUSEROPERATORSPEC SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
watchedNamespace string	The namespace the User Operator should watch.
image string	The image to use for the User Operator.
reconciliationIntervalSeconds	Interval between periodic reconciliations.

Property	Description
integer	
zookeeperSessionTimeoutSeconds	Timeout for the Zookeeper session.
integer	
livenessProbe	Pod liveness checking.
Probe	
readinessProbe	Pod readiness checking.
Probe	
resources	Resource constraints (limits and requests).
ResourceRequirements	
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**	
jvmOptions	JVM Options for pods.
EntityOperatorJvmOptions	

C.49. ENTITYOPERATORTEMPLATE SCHEMA REFERENCE

Used in: **EntityOperatorSpec**

Property	Description
deployment	Template for Entity Operator **Deployment**.
ResourceTemplate	
pod	Template for Entity Operator **Pods**.
PodTemplate	

C.50. 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**.

Property	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. The default is 365.
renewalDays integer	The number of days in the certificate renewal period. This is the number of days before the a certificate expires during which renewal actions may be performed. When **generateCertificateAuthority** is true, this will cause the generation of a new certificate. When **generateCertificateAuthority** is true, this will cause extra logging at WARN level about the pending certificate expiry. Default is 30.
certificateExpirationPolicy string (one of [replace-key, renew-certificate])	How should CA certificate expiration be handled when **generateCertificateAuthority=true**. The default is for a new CA certificate to be generated reusing the existing private key.

C.51. KAFKASTATUS SCHEMA REFERENCE

Used in: **Kafka**

Property	Description
conditions **Condition** array	List of status conditions.
observedGeneration integer	The generation of the CRD which was last reconciled by the operator.
listeners **ListenerStatus** array	Addresses of the internal and external listeners.

C.52. CONDITION SCHEMA REFERENCE

Used in: **KafkaStatus**

Property	Description
type string	The unique identifier of a condition, used to distinguish between other conditions in the resource.
status string	The status of the condition, one of True, False, Unknown.
lastTransitionTime string	Last time the condition of a type changes from one status to another.The required format is 'yyyy-MM-ddTHH:mm:ssZ', in the UTC time zone.
reason string	One-word CamelCase reason for the condition's last transition.
message string	Human-readable message indicating details about last transition.

C.53. LISTENERSTATUS SCHEMA REFERENCE

Used in: **KafkaStatus**

Property	Description
type string	The type of the listener. Can be one of the following three types: **plain**, **tls**, and **external**.
addresses **ListenerAddress** array	A list of the addresses for this listener.

C.54. LISTENERADDRESS SCHEMA REFERENCE

Used in: **ListenerStatus**

Property	Description

Property	Description
host string	The DNS name or IP address of Kafka bootstrap service.
port integer	The port of the Kafka bootstrap service.

C.55. KAFKACONNECT SCHEMA REFERENCE

Property	Description
spec **KafkaConnectSpec**	The specification of the Kafka Connect deployment.

C.56. KAFKACONNECTSPEC SCHEMA REFERENCE

Used in: **KafkaConnect**

Property	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.

Property	Description
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations.See external documentation of core/v1 toleration.
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.
template **KafkaConnectTemplate**	Template for Kafka Connect and Kafka Connect S2I resources. The template allows users to specify how is the **Deployment**, **Pods** and **Service** generated.
authentication **KafkaConnectAuthenticationTls**, **KafkaConnectAuthenticationScramSha512**, **KafkaConnectAuthenticationPlain**	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, plain].
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.
externalConfiguration **ExternalConfiguration**	Pass data from Secrets or ConfigMaps to the Kafka Connect pods and use them to configure connectors.
resources **ResourceRequirements**	Resource constraints (limits and requests).

Property	Description
tls **KafkaConnectTls**	TLS configuration.
version string	The Kafka Connect version. Defaults to 2.2.1. Consult the user documentation to understand the process required to upgrade or downgrade the version.

C.57. KAFKACONNECTTEMPLATE SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

Property	Description
deployment **ResourceTemplate**	Template for Kafka Connect **Deployment**.
pod **PodTemplate**	Template for Kafka Connect **Pods**.
apiService **ResourceTemplate**	Template for Kafka Connect API **Service**.
podDisruptionBudget **PodDisruptionBudgetTemplate**	Template for Kafka Connect **PodDisruptionBudget**.

C.58. KAFKACONNECTAUTHENTICATIONTLS SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaConnectAuthenticationTls** from **KafkaConnectAuthenticationScramSha512**, **KafkaConnectAuthenticationPlain**. It must have the value **tls** for the type **KafkaConnectAuthenticationTls**.

Property	Description
certificateAndKey	Certificate and private key pair for TLS authentication.

Property	Description
CertAndKeySecretSource	
type	Must be **tls**.
string	

C.59. CERTANDKEYSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaBridgeAuthenticationTls**, **KafkaConnectAuthenticationTls**, **KafkaMirrorMakerAuthenticationTls**

Property	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	

C.60. KAFKACONNECTAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaConnectAuthenticationScramSha512** from **KafkaConnectAuthenticationTls**, **KafkaConnectAuthenticationPlain**. It must have the value **scram-sha-512** for the type **KafkaConnectAuthenticationScramSha512**.

Property	Description
passwordSecret	Password used for the authentication.
PasswordSecretSource	
type	Must be **scram-sha-512**.
string	

Property	Description
username	Username used for the authentication.
string	

C.61. PASSWORDSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaBridgeAuthenticationPlain**, **KafkaBridgeAuthenticationScramSha512**, **KafkaConnectAuthenticationPlain**, **KafkaConnectAuthenticationScramSha512**, **KafkaMirrorMakerAuthenticationPlain**, **KafkaMirrorMakerAuthenticationScramSha512**

Property	Description
password	The name of the key in the Secret under which the password is stored.
string	
secretName	The name of the Secret containing the password.
string	

C.62. KAFKACONNECTAUTHENTICATIONPLAIN SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaConnectAuthenticationPlain** from **KafkaConnectAuthenticationTls**, **KafkaConnectAuthenticationScramSha512**. It must have the value **plain** for the type **KafkaConnectAuthenticationPlain**.

Property	Description
passwordSecret	Password used for the authentication.
PasswordSecretSource	
type	Must be **plain**.
string	
username	Username used for the authentication.
string	

C.63. EXTERNALCONFIGURATION SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

Property	Description
env	Allows to pass data from Secret or ConfigMap to the Kafka Connect pods as environment variables.
ExternalConfigurationEnv array	
volumes	Allows to pass data from Secret or ConfigMap to the Kafka Connect pods as volumes.
ExternalConfigurationVolumeSource array	

C.64. EXTERNALCONFIGURATIONENV SCHEMA REFERENCE

Used in: **ExternalConfiguration**

Property	Description
name	Name of the environment variable which will be passed to the Kafka Connect pods. The name of the environment variable cannot start with **KAFKA_** or **STRIMZI_**.
string	
valueFrom	Value of the environment variable which will be passed to the Kafka Connect pods. It can be passed either as a reference to Secret or ConfigMap field. The field has to specify exactly one Secret or ConfigMap.
ExternalConfigurationEnvVarSource	

C.65. EXTERNALCONFIGURATIONENVVARSOURCE SCHEMA REFERENCE

Used in: **ExternalConfigurationEnv**

Property	Description
configMapKeyRef	Refernce to a key in a ConfigMap.See external documentation of core/v1 configmapkeyselector.
ConfigMapKeySelector	
secretKeyRef	Reference to a key in a Secret.See external documentation of core/v1 secretkeyselector.
SecretKeySelector	

C.66. EXTERNALCONFIGURATIONVOLUMESOURCE SCHEMA REFERENCE

Used in: **ExternalConfiguration**

Property	Description
configMap	Reference to a key in a ConfigMap. Exactly one Secret or ConfigMap has to be specified.See external documentation of core/v1 configmapvolumesource.
ConfigMapVolumeSource	
name	Name of the volume which will be added to the Kafka Connect pods.
string	
secret	Reference to a key in a Secret. Exactly one Secret or ConfigMap has to be specified.See external documentation of core/v1 secretvolumesource.
SecretVolumeSource	

C.67. KAFKACONNECTTLS SCHEMA REFERENCE

Used in: **KafkaConnectS2ISpec**, **KafkaConnectSpec**

Property	Description
trustedCertificates	Trusted certificates for TLS connection.
CertSecretSource array	

C.68. CERTSECRETSOURCE SCHEMA REFERENCE

Used in: **KafkaBridgeTls**, **KafkaConnectTls**, **KafkaMirrorMakerTls**

Property	Description
certificate	The name of the file certificate in the Secret.
string	
secretName	The name of the Secret containing the certificate.
string	

C.69. KAFKACONNECTS2I SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Connect deployment.
KafkaConnectS2ISpec	

C.70. KAFKACONNECTS2ISPEC SCHEMA REFERENCE

Used in: **KafkaConnectS2I**

Property	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	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules.See external documentation of core/v1 affinity.
Affinity	
logging	Logging configuration for Kafka Connect. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].
InlineLogging, **ExternalLogging**	
metrics	The Prometheus JMX Exporter configuration. See https://github.com/prometheus/jmx_exporter for details of the structure of this configuration.
map	

Property	Description
template **KafkaConnectTemplate**	Template for Kafka Connect and Kafka Connect S2I resources. The template allows users to specify how is the **Deployment**, **Pods** and **Service** generated.
authentication **KafkaConnectAuthenticationTls**, **KafkaConnectAuthenticationScramSha512**, **KafkaConnectAuthenticationPlain**	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, plain].
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.
externalConfiguration **ExternalConfiguration**	Pass data from Secrets or ConfigMaps to the Kafka Connect pods and use them to configure connectors.
insecureSourceRepository boolean	When true this configures the source repository with the 'Local' reference policy and an import policy that accepts insecure source tags.
resources **ResourceRequirements**	Resource constraints (limits and requests).
tls **KafkaConnectTls**	TLS configuration.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations.See external documentation of core/v1 toleration.
version string	The Kafka Connect version. Defaults to 2.2.1. Consult the user documentation to understand the process required to upgrade or downgrade the version.

C.71. KAFKATOPIC SCHEMA REFERENCE

Property	Description
spec	The specification of the topic.
KafkaTopicSpec	

C.72. KAFKATOPICSPEC SCHEMA REFERENCE

Used in: **KafkaTopic**

Property	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.
integer	
replicas	The number of replicas the topic should have.
integer	
config	The topic configuration.
map	
topicName	The name of the topic. When absent this will default to the metadata.name of the topic. It is recommended to not set this unless the topic name is not a valid Kubernetes resource name.
string	

C.73. KAFKAUSER SCHEMA REFERENCE

Property	Description
spec	The specification of the user.
KafkaUserSpec	

C.74. KAFKAUSERSPEC SCHEMA REFERENCE

Used in: **KafkaUser**

Property	Description
authentication **KafkaUserTlsClientAuthentication**, **KafkaUserScramSha512ClientAuthentication**	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].
authorization **KafkaUserAuthorizationSimple**	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].

C.75. 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**.

Property	Description
type	Must be **tls**.
string	

C.76. 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**.

Property	Description
type	Must be **scram-sha-512**.
string	

C.77. 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**.

Property	Description
type string	Must be **simple**.
acls **AclRule** array	List of ACL rules which should be applied to this user.

C.78. ACLRULE SCHEMA REFERENCE

Used in: **KafkaUserAuthorizationSimple**

Property	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**, **AclRuleTransactionalIdResource**	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, transactionalId].
type string (one of [allow, deny])	The type of the rule. Currently the only supported type is **allow**. ACL rules with type**allow** are used to allow user to execute the specified operations. Default value is **allow**.

C.79. ACLRULETOPICRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleTopicResource** from **AclRuleGroupResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource**. It must have the value **topic** for the type **AclRuleTopicResource**.

Property	Description
type	Must be **topic**.
string	
name	Name of resource for which given ACL rule applies. Can be combined with **patternType** field to use prefix pattern.
string	
patternType	Describes the pattern used in the resource field. The supported types are **literal** and **prefix**. With **literal** pattern type, the resource field will be used as a definition of a full topic name. With **prefix** pattern type, the resource name will be used only as a prefix. Default value is **literal**.
string (one of [prefix, literal])	

C.80. ACLRULEGROUPRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleGroupResource** from **AclRuleTopicResource**, **AclRuleClusterResource**, **AclRuleTransactionalIdResource**. It must have the value **group** for the type **AclRuleGroupResource**.

Property	Description
type	Must be **group**.
string	
name	Name of resource for which given ACL rule applies. Can be combined with **patternType** field to use prefix pattern.
string	
patternType	Describes the pattern used in the resource field. The supported types are **literal** and **prefix**. With **literal** pattern type, the resource field will be used as a definition of a full topic name. With **prefix** pattern type, the resource name will be used only as a prefix. Default value is **literal**.
string (one of [prefix, literal])	

C.81. ACLRULECLUSTERRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleClusterResource** from **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleTransactionalIdResource**. It must have the value **cluster** for the type **AclRuleClusterResource**.

Property	Description
type	Must be **cluster**.
string	

C.82. ACLRULETRANSACTIONALIDRESOURCE SCHEMA REFERENCE

Used in: **AclRule**

The **type** property is a discriminator that distinguishes the use of the type **AclRuleTransactionalIdResource** from **AclRuleTopicResource**, **AclRuleGroupResource**, **AclRuleClusterResource**. It must have the value **transactionalId** for the type **AclRuleTransactionalIdResource**.

Property	Description
type	Must be **transactionalId**.
string	
name	Name of resource for which given ACL rule applies. Can be combined with **patternType** field to use prefix pattern.
string	
patternType	Describes the pattern used in the resource field. The supported types are **literal** and **prefix**. With **literal** pattern type, the resource field will be used as a definition of a full name. With **prefix** pattern type, the resource name will be used only as a prefix. Default value is **literal**.
string (one of [prefix, literal])	

C.83. KAFKAMIRRORMAKER SCHEMA REFERENCE

Property	Description
spec	The specification of the mirror maker.
KafkaMirrorMakerSpec	

C.84. KAFKAMIRRORMAKERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMaker**

Property	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 **KafkaMirrorMakerProducerSpec**	Configuration of target cluster.
resources **ResourceRequirements**	Resource constraints (limits and requests).
affinity Affinity	**The property affinity has been deprecated. This feature should now be configured at path spec.template.pod.affinity.** The pod's affinity rules.See external documentation of core/v1 affinity.
tolerations Toleration array	**The property tolerations has been deprecated. This feature should now be configured at path spec.template.pod.tolerations.** The pod's tolerations.See external documentation of core/v1 toleration.
jvmOptions **JvmOptions**	JVM Options for pods.
logging **InlineLogging**, **ExternalLogging**	Logging configuration for Mirror Maker. The type depends on the value of the **logging.type** property within the given object, which must be one of [inline, external].

Property	Description
metrics	The Prometheus JMX Exporter configuration. See JMX Exporter documentation for details of the structure of this configuration.
map	
template	Template for Kafka Mirror Maker resources. The template allows users to specify how is the **Deployment** and **Pods** generated.
KafkaMirrorMakerTemplate	
version	The Kafka Mirror Maker version. Defaults to 2.2.1. Consult the user documentation to understand the process required to upgrade or downgrade the version.
string	

C.85. KAFKAMIRRORMAKERCONSUMERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Property	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	
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, plain].
KafkaMirrorMakerAuthenticationTls, **KafkaMirrorMakerAuthenticationScramSha512**, **KafkaMirrorMakerAuthenticationPlain**	
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	

C.86. KAFKAMIRRORMAKERAUTHENTICATIONTLS SCHEMA REFERENCE

Used in: KafkaMirrorMakerConsumerSpec, KafkaMirrorMakerProducerSpec

The **type** property is a discriminator that distinguishes the use of the type **KafkaMirrorMakerAuthenticationTls** from KafkaMirrorMakerAuthenticationScramSha512, KafkaMirrorMakerAuthenticationPlain. It must have the value **tls** for the type **KafkaMirrorMakerAuthenticationTls**.

Property	Description
certificateAndKey CertAndKeySecretSource	Reference to the **Secret** which holds the certificate and private key pair.
type string	Must be **tls**.

C.87. KAFKAMIRRORMAKERAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE

Used in: KafkaMirrorMakerConsumerSpec, KafkaMirrorMakerProducerSpec

The **type** property is a discriminator that distinguishes the use of the type **KafkaMirrorMakerAuthenticationScramSha512** from KafkaMirrorMakerAuthenticationTls, KafkaMirrorMakerAuthenticationPlain. It must have the value **scram-sha-512** for the type **KafkaMirrorMakerAuthenticationScramSha512**.

Property	Description
passwordSecret PasswordSecretSource	Reference to the **Secret** which holds the password.
type string	Must be **scram-sha-512**.
username string	Username used for the authentication.

C.88. KAFKAMIRRORMAKERAUTHENTICATIONPLAIN SCHEMA REFERENCE

Used in: KafkaMirrorMakerConsumerSpec, KafkaMirrorMakerProducerSpec

The **type** property is a discriminator that distinguishes the use of the type **KafkaMirrorMakerAuthenticationPlain** from KafkaMirrorMakerAuthenticationTls,

KafkaMirrorMakerAuthenticationScramSha512. It must have the value **plain** for the type **KafkaMirrorMakerAuthenticationPlain**.

Property	Description
passwordSecret **PasswordSecretSource**	Reference to the **Secret** which holds the password.
type string	Must be **plain**.
username string	Username used for the authentication.

C.89. KAFKAMIRRORMAKERTLS SCHEMA REFERENCE

Used in: **KafkaMirrorMakerConsumerSpec**, **KafkaMirrorMakerProducerSpec**

Property	Description
trustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection.

C.90. KAFKAMIRRORMAKERPRODUCERSPEC SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Property	Description
bootstrapServers string	A list of host:port pairs to use for establishing the initial connection to the Kafka cluster.
authentication **KafkaMirrorMakerAuthenticationTls**, **KafkaMirrorMakerAuthenticationScramSha512**, **KafkaMirrorMakerAuthenticationPlain**	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, plain].
config map	The mirror maker producer config. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, sasl., security.

Property	Description
tls	TLS configuration for connecting to the cluster.
KafkaMirrorMakerTls	

C.91. KAFKAMIRRORMAKERTEMPLATE SCHEMA REFERENCE

Used in: **KafkaMirrorMakerSpec**

Property	Description
deployment	Template for Kafka Mirror Maker **Deployment**.
ResourceTemplate	
pod	Template for Kafka Mirror Maker **Pods**.
PodTemplate	
podDisruptionBudget	Template for Kafka Mirror Maker **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

C.92. KAFKABRIDGE SCHEMA REFERENCE

Property	Description
spec	The specification of the Kafka Bridge.
KafkaBridgeSpec	

C.93. KAFKABRIDGESPEC SCHEMA REFERENCE

Used in: **KafkaBridge**

Property	Description
replicas	The number of pods in the **Deployment**.
integer	
image	The docker image for the pods.

Property	Description
string	
bootstrapServers	A list of host:port pairs to use for establishing the initial connection to the Kafka cluster.
string	
tls	TLS configuration for connecting to the cluster.
KafkaBridgeTls	
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, plain].
KafkaBridgeAuthenticationTls, **KafkaBridgeAuthenticationScramSha512**, **KafkaBridgeAuthenticationPlain**	
http	The HTTP related configuration.
KafkaBridgeHttpConfig	
consumer	Kafka consumer related configuration.
KafkaBridgeConsumerSpec	
producer	Kafka producer related configuration.
KafkaBridgeProducerSpec	
resources	Resource constraints (limits and requests).
ResourceRequirements	
jvmOptions	**Currently not supported** JVM Options for pods.
JvmOptions	
logging	Logging configuration for Kafka Bridge. 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	**Currently not supported** The Prometheus JMX Exporter configuration. See JMX Exporter documentation for details of the structure of this configuration.
map	

Property	Description
livenessProbe **Probe**	Pod liveness checking.
readinessProbe **Probe**	Pod readiness checking.
template **KafkaBridgeTemplate**	Template for Kafka Bridge resources. The template allows users to specify how is the **Deployment** and **Pods** generated.

C.94. KAFKABRIDGETLS SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
trustedCertificates **CertSecretSource** array	Trusted certificates for TLS connection.

C.95. KAFKABRIDGEAUTHENTICATIONTLS SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaBridgeAuthenticationTls** from **KafkaBridgeAuthenticationScramSha512**, **KafkaBridgeAuthenticationPlain**. It must have the value **tls** for the type **KafkaBridgeAuthenticationTls**.

Property	Description
certificateAndKey **CertAndKeySecretSource**	Reference to the **Secret** which holds the certificate and private key pair.
type string	Must be **tls**.

C.96. KAFKABRIDGEAUTHENTICATIONSCRAMSHA512 SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaBridgeAuthenticationScramSha512** from **KafkaBridgeAuthenticationTls**, **KafkaBridgeAuthenticationPlain**. It must have the value **scram-sha-512** for the type **KafkaBridgeAuthenticationScramSha512**.

Property	Description
passwordSecret **PasswordSecretSource**	Reference to the **Secret** which holds the password.
type string	Must be **scram-sha-512**.
username string	Username used for the authentication.

C.97. KAFKABRIDGEAUTHENTICATIONPLAIN SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

The **type** property is a discriminator that distinguishes the use of the type **KafkaBridgeAuthenticationPlain** from **KafkaBridgeAuthenticationTls**, **KafkaBridgeAuthenticationScramSha512**. It must have the value **plain** for the type **KafkaBridgeAuthenticationPlain**.

Property	Description
passwordSecret **PasswordSecretSource**	Reference to the **Secret** which holds the password.
type string	Must be **plain**.
username string	Username used for the authentication.

C.98. KAFKABRIDGEHTTPCONFIG SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
port	The port which is the server listening on. Avoid using port 8081 which is used for readiness checking.
integer	

C.99. KAFKABRIDGECONSUMERSPEC SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
config	The Kafka consumer configuration used for consumer instances created by the bridge. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, group.id, sasl., security.
map	

C.100. KAFKABRIDGEPRODUCERSPEC SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
config	The Kafka producer configuration used for producer instances created by the bridge. Properties with the following prefixes cannot be set: ssl., bootstrap.servers, sasl., security.
map	

C.101. KAFKABRIDGETEMPLATE SCHEMA REFERENCE

Used in: **KafkaBridgeSpec**

Property	Description
deployment	Template for Kafka Bridge **Deployment**.
ResourceTemplate	
pod	Template for Kafka Bridge **Pods**.
PodTemplate	
apiService	Template for Kafka Bridge API **Service**.
ResourceTemplate	

Property	Description
podDisruptionBudget	Template for Kafka Bridge **PodDisruptionBudget**.
PodDisruptionBudgetTemplate	

APPENDIX D. 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.

Revised on 2019-07-15 15:33:47 UTC