Red Hat AMQ Streams 2.0

AMQ Streams on OpenShift の使用

OpenShift Container Platform での AMQ Streams 2.0 のデプロイメントの設定および管理

Last Updated: 2022-07-03

Red Hat AMQ Streams 2.0 AMQ Streams on OpenShift の使用

OpenShift Container Platform での AMQ Streams 2.0 のデプロイメントの設定および管理

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Using_AMQ_Streams_on_OpenShift.ent file |.

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.

概要

AMQ Streams でデプロイされた operator および Kafka コンポーネントを設定し、大規模なメッセージングネットワークをビルドします。

目次

多様性を受け入れるオープンソースの強化 14

第1章 AMQ STREAMS の概要 15
1.1. KAFKA の機能 15
1.2. KAFKA のユースケース 15
1.3. AMQ STREAMS による KAFKA のサポート 16
1.4. AMQ STREAMS の OPERATOR 16
Operator 16
1.4.1. Cluster Operator 17
1.4.2. Topic Operator 18
1.4.3. User Operator 19
1.4.4. AMQ Streams operator のフィーチャーゲート 20
1.5. AMQ STREAMS のカスタムリソース 20
1.5.1. AMQ Streams カスタムリソースの例 20
1.6. リスナーの設定 23
1.7. 本書の表記慣例 24

第2章 デプロイメント設定 25
2.1. KAFKA クラスターの設定 25
2.1.1. Kafka の設定 26
2.1.2. Entity Operator の設定 31
2.1.2.1. Entity Operator の設定プロパティー 32
2.1.2.2. Topic Operator 設定プロパティー 32
2.1.2.3. User Operator 設定プロパティー 33
2.1.3. Kafka および ZooKeeper のストレージタイプ 34
2.1.3.1. データストレージに関する留意事項 35
2.1.3.1.1. ファイルシステム 35
2.1.3.1.2. Apache Kafka および ZooKeeper ストレージ 35
2.1.3.2. 一時ストレージ 36
2.1.3.2.1. ログディレクトリー 36
2.1.3.3. 永続ストレージ 37
2.1.3.3.1. ストレージクラスのオーバーライド 38
2.1.3.3.2. Persistent Volume Claim (永続ボリューム要求、PVC) の命名 39
2.1.3.3.3. ログディレクトリー 39
2.1.3.4. 永続ボリュームのサイズ変更 40
2.1.3.5. JBOD ストレージの概要 41
2.1.3.5.1. JBOD の設定 41
2.1.3.5.2. JBOD および 永続ボリューム要求 (PVC) 41
2.1.3.5.3. ログディレクトリー 42
2.1.3.6. JBOD ストレージへのボリュームの追加 42
2.1.3.7. JBOD ストレージからのボリュームの削除 43
2.1.4. クラスターのスケーリング 44
2.1.4.1. ブローカーのスケーリング設定 44
ブローカーの追加 44
ブローカーの削除 44
2.1.4.2. パーティション再割り当てツール 45
パーティション再割り当ての JSON ファイル 45
JBOD ボリューム間のパーティション再割り当て 46
パーティション再割り当てスロットル 47
2.1.4.3. 再割り当て JSON ファイルの生成 47
2.1.4.4. Kafka クラスターのスケールアップ 51

2.1.4.5. Kafka クラスターのスケールダウン 53
2.1.5. ローリングアップデートのメンテナンス時間枠 55
2.1.5.1. メンテナンス時間枠の概要 55
2.1.5.2. メンテナンス時間枠の定義 55
2.1.5.3. メンテナンス時間枠の設定 56
2.1.6. ターミナルからの ZooKeeper への接続 57
2.1.7. Kafka ノードの手動による削除 57
2.1.8. ZooKeeper ノードの手動による削除 58
2.1.9. Kafka クラスターリソースのリスト 59
2.2. KAFKA CONNECT クラスターの設定 63
2.2.1. Kafka Connect の設定 63
2.2.2. 複数インスタンスの Kafka Connect 設定 67
2.2.3. Kafka Connect のユーザー承認の設定 68
2.2.4. Kafka コネクターの再起動の実行 71
2.2.5. Kafka コネクタータスクの再起動の実行 71
2.2.6. Kafka Connect API の公開 72
2.2.7. Kafka Connect クラスターリソースの一覧 74
2.2.8. 変更データキャプチャーのための Debezium との統合 74
2.3. KAFKA MIRRORMAKER クラスターの設定 74
2.3.1. Kafka MirrorMaker の設定 75
2.3.2. Kafka MirrorMaker クラスターリソースの一覧 79
2.4. KAFKA MIRRORMAKER 2.0 クラスターの設定 79
2.4.1. MirrorMaker 2.0 のデータレプリケーション 80
2.4.2. クラスターの設定 80
2.4.2.1. 双方向レプリケーション (active/active) 81
2.4.2.2. 一方向レプリケーション (active/passive) 81
2.4.2.3. トピック設定の同期 82
2.4.2.4. データの整合性 82
2.4.2.5. オフセットの追跡 82
2.4.2.6. コンシューマーグループオフセットの同期 82
2.4.2.7. 接続性チェック 83
2.4.3. ACL ルールの同期 83
2.4.4. MirrorMaker 2.0 を使用した Kafka クラスター間でのデータの同期 83
2.4.5. Kafka MirrorMaker 2.0 コネクターの再起動の実行 89
2.4.6. Kafka MirrorMaker 2.0 コネクタータスクの再起動の実行 90
2.5. KAFKA BRIDGE クラスターの設定 90
2.5.1. Kafka Bridge の設定 91
2.5.2. Kafka Bridge クラスターリソースのリスト 93
2.6. OPENSHIFT リソースのカスタマイズ 94
2.6.1. イメージプルポリシーのカスタマイズ 95
2.7. POD スケジューリングの設定 95
2.7.1. アフィニティー、容認 (Toleration)、およびトポロジー分散制約の指定 96
2.7.1.1. Pod の非アフィニティーを使用して重要なアプリケーションがノードを共有しないようにする 96
2.7.1.2. ノードのアフィニティーを使用したワークロードの特定ノードへのスケジュール 96
2.7.1.3. 専用ノードへのノードのアフィニティーと容認 (Toleration) の使用 97
2.7.2. それぞれの Kafka ブローカーを別のワーカーノードでスケジュールするための Pod の非アフィニティーの設定 97
2.7.3. Kafka コンポーネントでの Pod の非アフィニティーの設定 99
2.7.4. Kafka コンポーネントでのノードのアフィニティーの設定 100
2.7.5. 専用ノードの設定と Pod のスケジューリング 101
2.8. ロギングの設定 102
2.8.1. Kafka コンポーネントおよび Operator のロギングオプション 102
2.8.2. ロギングの ConfigMap の作成 103

2.8.3. ロギングフィルターの Operator への追加 104

第3章 外部ソースからの設定値の読み込み **108**
3.1. 設定マップからの設定値の読み込み 108
3.2. 環境変数から設定値の読み込み 112

第4章 OPENSHIFT クラスター外の KAFKA へのアクセス **115**
4.1. ノードポートを使用した KAFKA へのアクセス 115
4.2. ロードバランサーを使用した KAFKA へのアクセス 117
4.3. INGRESS を使用した KAFKA へのアクセス 118
4.4. OPENSHIFT ルートを使用した KAFKA へのアクセス 121

第5章 KAFKA へのセキュアなアクセスの管理 **124**
5.1. KAFKA のセキュリティーオプション 124
5.1.1. リスナー認証 124
5.1.1.1. 相互 TLS 認証 127
5.1.1.2. SCRAM-SHA-512 認証 128
5.1.1.3. ネットワークポリシー 128
5.1.1.4. 追加のリスナー設定オプション 129
5.1.2. Kafka の承認 129
5.1.2.1. スーパーユーザー 130
5.2. KAFKA クライアントのセキュリティーオプション 131
5.2.1. ユーザー処理用の Kafka クラスターの特定 131
5.2.2. ユーザー認証 132
5.2.2.1. TLS クライアント認証 133
5.2.2.2. User Operator の外部で発行された証明書を使用した TLS クライアント認証 134
5.2.2.3. SCRAM-SHA-512 認証 134
5.2.2.3.1. カスタムパスワード設定 136
5.2.3. ユーザーの承認 137
5.2.3.1. ACL ルール 138
5.2.3.2. Kafka ブローカーへのスーパーユーザーアクセス 138
5.2.3.3. ユーザークォータ 138
5.3. KAFKA ブローカーへのアクセスのセキュア化 139
5.3.1. Kafka ブローカーのセキュア化 141
5.3.2. Kafka へのユーザーアクセスのセキュア化 143
5.3.3. ネットワークポリシーを使用した Kafka リスナーへのアクセス制限 145
5.4. OAUTH 2.0 トークンベース認証の使用 147
5.4.1. OAuth 2.0 認証メカニズム 148
5.4.2. OAuth 2.0 Kafka ブローカーの設定 150
5.4.2.1. 承認サーバーの OAuth 2.0 クライアント設定 151
5.4.2.2. Kafka クラスターでの OAuth 2.0 認証設定 151
5.4.2.3. 高速なローカル JWT トークン検証の設定 153
5.4.2.4. OAuth 2.0 イントロスペクションエンドポイントの設定 154
5.4.3. Kafka ブローカーの再認証の設定 155
5.4.4. OAuth 2.0 Kafka クライアントの設定 157
5.4.5. OAuth 2.0 のクライアント認証フロー 158
5.4.5.1. クライアント認証フローの例 159
5.4.6. OAuth 2.0 認証の設定 162
5.4.6.1. OAuth 2.0 承認サーバーとしての Red Hat Single Sign-On の設定 163
5.4.6.2. Kafka ブローカーの OAuth 2.0 サポートの設定 165
5.4.6.3. OAuth 2.0 を使用するよう Kafka Java クライアントを設定 172
5.4.6.4. Kafka コンポーネントの OAuth 2.0 の設定 174
5.5. OAUTH 2.0 トークンベース承認の使用 178
5.5.1. OAuth 2.0 の承認メカニズム 179

5.5.1.1. Kafka ブローカーのカスタムオーソライザー 179
5.5.2. OAuth 2.0 承認サポートの設定 179
5.5.3. Red Hat Single Sign-On の Authorization Services でのポリシーおよびパーミッションの管理 182
5.5.3.1. Kafka および Red Hat Single Sign-On 承認モデルの概要 183
Kafka 承認モデル 183
Red Hat Single Sign-On の Authorization Services モデル 185
5.5.3.2. Red Hat Single Sign-On Authorization Services の Kafka 承認モデルへのマッピング 186
5.5.3.3. Kafka 操作に必要なパーミッションの例 190
5.5.4. Red Hat Single Sign-On の Authorization Services の試行 193
5.5.4.1. Red Hat Single Sign-On 管理コンソールへのアクセス 195
5.5.4.2. Red Hat Single Sign-On 承認をでの Kafka クラスターのデプロイメント 198
5.5.4.3. CLI Kafka クライアントセッションの TLS 接続の準備 200
5.5.4.4. CLI Kafka クライアントセッションを使用した Kafka への承認されたアクセスの確認 201

第6章 AMQ STREAMS OPERATOR の使用 211
6.1. CLUSTER OPERATOR の使用 211
6.1.1. Cluster Operator の設定 211
6.1.1.1. フィーチャーゲート 216
フィーチャーゲートの設定 218
6.1.1.1.1. コントロールプレーンリスナーフィーチャーゲート 218
6.1.1.1.2. サービスアカウントパッチ適用のフィーチャーゲート 219
6.1.1.2. ConfigMap による設定のロギング 220
6.1.1.3. ネットワークポリシーによる Cluster Operator アクセスの制限 221
6.1.1.4. 定期的な調整 221
6.1.2. ロールベースアクセス制御 (RBAC) のプロビジョニング 221
6.1.2.1. 委譲された権限 222
6.1.2.2. ServiceAccount 223
6.1.2.3. ClusterRoles 224
6.1.2.4. ClusterRoleBindings 232
6.1.3. デフォルトのプロキシー設定を使用した Cluster Operator の設定 234
6.2. TOPIC OPERATOR の使用 236
6.2.1. Kafka トピックリソース 237
6.2.1.1. トピック処理用の Kafka クラスターの特定 237
6.2.1.2. Kafka トピックの使用に関する推奨事項 237
6.2.1.3. Kafka トピックの命名規則 238
6.2.2. Topic Operator のトピックストア 239
6.2.2.1. 内部トピックストアトピック 240
6.2.2.2. ZooKeeper からのトピックメタデータの移行 240
6.2.2.3. ZooKeeper を使用してトピックメタデータを保存する AMQ Streams バージョンへのダウングレード 240
6.2.2.4. Topic Operator トピックのレプリケーションおよびスケーリング 241
6.2.2.5. トピック変更の処理 242
6.2.3. Kafka トピックの設定 243
6.2.4. リソース要求および制限のある Topic Operator の設定 245
6.3. USER OPERATOR の使用 246
6.3.1. リソース要求および制限のある User Operator の設定 247
6.4. PROMETHEUS メトリクスを使用した OPERATOR の監視 247

第7章 KAFKA BRIDGE 249
7.1. KAFKA BRIDGE API ドキュメント 249
7.2. KAFKA BRIDGE の概要 249
7.2.1. Kafka Bridge インターフェース 249
7.2.1.1. HTTP リクエスト 249

7.2.2. Kafka Bridge でサポートされるクライアント 250
7.2.3. Kafka Bridge のセキュリティー保護 251
7.2.4. OpenShift 外部の Kafka Bridge へのアクセス 252
7.2.5. Kafka Bridge へのリクエスト 253
7.2.5.1. コンテンツタイプヘッダー 253
7.2.5.2. 埋め込みデータ形式 254
7.2.5.3. メッセージの形式 255
7.2.5.4. Accept ヘッダー 256
7.2.6. CORS 256
7.2.6.1. シンプルなリクエスト 257
7.2.6.2. プリフライトリクエスト 258
7.2.7. Kafka Bridge デプロイメント 259
7.3. KAFKA BRIDGE クイックスタート 260
7.3.1. OpenShift クラスターへの Kafka Bridge のデプロイメント 261
7.3.2. Kafka Bridge サービスのローカルマシンへの公開 262
7.3.3. トピックおよびパーティションへのメッセージの作成 263
7.3.4. Kafka Bridge コンシューマーの作成 271
7.3.5. Kafka Bridge コンシューマーのトピックへのサブスクライブ 272
7.3.6. Kafka Bridge コンシューマーからの最新メッセージの取得 273
7.3.7. ログへのオフセットのコミット 274
7.3.8. パーティションのオフセットのシーク 275
7.3.9. Kafka Bridge コンシューマーの削除 277

第8章 3SCALE での KAFKA BRIDGE の使用 278
8.1. 3SCALE での KAFKA BRIDGE の使用 278
8.1.1. Kafka Bridge のサービス検出 278
8.1.2. 3scale APIcast ゲートウェイポリシー 279
8.1.3. TLS の検証 282
8.1.4. 3scale ドキュメント 282
8.2. KAFKA BRIDGE を使用するための 3SCALE のデプロイメント 282

第9章 CRUISE CONTROL によるクラスターのリバランス 290
9.1. CRUISE CONTROL とは 290
9.2. 最適化ゴールの概要 291
AMQ Streams カスタムリソースでのゴールの設定 292
ハードゴールおよびソフトゴール 292
メイン最適化ゴール 294
デフォルトの最適化ゴール 295
ユーザー提供の最適化ゴール 296
9.3. 最適化プロポーザルの概要 297
キャッシュされた最適化プロポーザル 297
最適化プロポーザルの内容 298
サマリー 298
ブローカーの負荷 299
9.4. リバランスパフォーマンスチューニングの概要 301
パーティション再割り当てコマンド 301
レプリカの移動ストラテジー 301
リバランスチューニングオプション 302
9.5. CRUISE CONTROL の設定 304
CORS (Corss-Origin Resource Sharing) の設定 305
容量の設定 305
ロギングの設定 307
Cruise Control REST API のセキュリティー 308

9.6. CRUISE CONTROL のデプロイ 309
自動作成されたトピック 311
9.7. 最適化プロポーザルの生成 312
9.8. 最適化プロポーザルの承認 315
9.9. クラスターリバランスの停止 317
9.10. KAFKAREBALANCE リソースの問題の修正 318

第10章 SERVICE REGISTRY を使用したスキーマの検証 321

第11章 分散トレーシング 322
AMQ Streams によるトレーシングのサポート方法 322
手順の概要 323
11.1. OPENTRACING および JAEGER の概要 323
11.2. KAFKA クライアントのトレーシング設定 325
11.2.1. Kafka クライアント用の Jaeger トレーサーの初期化 325
11.2.2. トレーシングの環境変数 326
11.3. トレーサーでの KAFKA クライアントのインストルメント化 328
11.3.1. トレーシングのための Kafka プロデューサーおよびコンシューマーのインストルメント化 328
11.3.1.1. Decorator パターンのカスタムスパン名 330
11.3.1.2. ビルトインスパン名 331
11.3.2. Kafka Streams アプリケーションをトレース用にインストルメント化 332
11.4. MIRRORMAKER、KAFKA CONNECT、および KAFKA BRIDGE のトレーシング設定 332
11.4.1. MirrorMaker、Kafka Connect、および Kafka Bridge リソースでのトレーシングの有効化 333

第12章 TLS 証明書の管理 337
12.1. 認証局 337
12.1.1. CA 証明書 338
12.1.2. 独自の CA 証明書のインストール 338
12.2. SECRET 343
12.2.1. PKCS #12 ストレージ 344
12.2.2. クラスター CA Secret 344
12.2.3. クライアント CA Secret 346
12.2.4. ラベルおよびアノテーションの Secret への追加 346
12.2.5. CA Secret での ownerReference の無効化 347
12.2.6. User Secret 348
12.3. 証明書の更新および有効期間 348
12.3.1. 自動生成された CA 証明書での更新プロセス 350
12.3.2. クライアント証明書の更新 351
12.3.3. Cluster Operator によって生成される CA 証明書の手動更新 352
12.3.4. Cluster Operator によって生成された CA 証明書によって使用される秘密鍵の置き換え 354
12.3.5. 独自の CA 証明書の更新 355
12.4. TLS 接続 358
12.4.1. ZooKeeper の通信 358
12.4.2. Kafka のブローカー間の通信 359
12.4.3. Topic Operator および User Operator 359
12.4.4. Cruise Control 359
12.4.5. Kafka クライアント接続 359
12.5. クラスター CA を信頼する内部クライアントの設定 359
12.6. クラスター CA を信頼する外部クライアントの設定 362
12.7. KAFKA リスナー証明書 364
12.7.1. 独自の Kafka リスナー証明書の指定 364
12.7.2. Kafka リスナーのサーバー証明書の SAN 367
12.7.2.1. TLS リスナー SAN の例 367
12.7.2.2. 外部リスナー SAN の例 368

第13章 AMQ STREAMS の管理 370
13.1. カスタムリソースの使用 370
13.1.1. カスタムリソースでの oc 操作の実施 370
13.1.1.1. リソースカテゴリー 371
13.1.1.2. サブリソースのステータスのクエリー 372
13.1.2. AMQ Streams カスタムリソースのステータス情報 373
13.1.3. カスタムリソースのステータスの検出 376
13.2. カスタムリソースの調整の一時停止 377
13.3. AMQ STREAMS DRAIN CLEANER での POD のエビクト 379
13.3.1. 前提条件 380
13.3.2. AMQ Streams の Drain クリーニングのデプロイ 380
13.3.3. AMQ Streams の Drain クリーニングの使用 383
13.4. KAFKA および ZOOKEEPER クラスターの手動によるローリングアップデートの開始 384
13.4.1. 前提条件 385
13.4.2. StatefulSet アノテーションを使用したローリングアップデートの実行 385
13.4.3. Pod アノテーションを使用したローリングアップデートの実行 386
13.5. ラベルおよびアノテーションを使用したサービスの検出 387
内部 Kafka ブートストラップサービスの例 388
HTTP Bridge サービスの例 388
13.5.1. サービスの接続詳細の返信 388
13.6. 永続ボリュームからのクラスターの復元 389
13.6.1. namespace が削除された場合の復元 389
13.6.2. OpenShift クラスター喪失からの復旧 390
13.6.3. 削除したクラスターの永続ボリュームからの復元 390
13.7. KAFKA STATIC QUOTA プラグインを使用したブローカーへの制限の設定 396
13.8. KAFKA 設定のチューニング 398
13.8.1. Kafka ブローカー設定のチューニング 399
13.8.1.1. 基本的なブローカー設定 399
13.8.1.2. 高可用性のためのトピックの複製 400
13.8.1.3. トランザクションおよびコミットの内部トピック設定 401
13.8.1.4. I/O スレッドの増加によるリクエスト処理スループットの向上 402
13.8.1.5. レイテンシーの高い接続に対する帯域幅の引き上げ 403
13.8.1.6. データ保持ポリシーでのログの管理 404
13.8.1.7. クリーンアップポリシーによるログデータの削除 405
13.8.1.8. ディスク使用率の管理 408
13.8.1.9. 大きなメッセージサイズの処理 409
13.8.1.10. メッセージデータのログフラッシュの制御 411
13.8.1.11. 可用性のためのパーティションリバランス 412
13.8.1.12. クリーンでないリーダーの選出 (unclean leader election) 414
13.8.1.13. 不要なコンシューマーグループリバランスの回避 414
13.8.2. Kafka プロデューサー設定のチューニング 415
13.8.2.1. 基本のプロデューサー設定 415
13.8.2.2. データの持続性 416
13.8.2.3. 順序付き配信 417
13.8.2.4. 信頼性の保証 419
13.8.2.5. スループットおよびレイテンシーの最適化 419
13.8.3. Kafka コンシューマー設定の調整 423
13.8.3.1. 基本的なコンシューマー設定 423
13.8.3.2. コンシューマーグループを使用したデータ消費のスケーリング 424
13.8.3.3. メッセージの順序の保証 425
13.8.3.4. スループットおよびレイテンシーの最適化 425
13.8.3.5. オフセットをコミットする際のデータ損失または重複の回避 427
13.8.3.5.1. トランザクションメッセージの制御 428

13.8.3.6. データ損失を回避するための障害からの復旧 428
13.8.3.7. オフセットポリシーの管理 429
13.8.3.8. リバランスの影響を最小限にする 430
13.9. AMQ STREAMS のアンインストール 431
13.10. よくある質問 432
13.10.1. Cluster Operator に関する質問 432
13.10.1.1. AMQ Streams のインストールに、クラスター管理者の権限が必要なのはなぜですか? 432
13.10.1.2. Cluster Operator が ClusterRoleBindings を作成する必要があるのはなぜですか? 433
13.10.1.3. OpenShift の標準ユーザーは Kafka カスタムリソースを作成できますか? 434
13.10.1.4. ログの Failed to acquire lock 警告の意味 434
13.10.1.5. TLS を使用して NodePort に接続するとホスト名の検証に失敗するのはなぜですか? 435

第14章 カスタムリソース API のリファレンス 436
14.1. 共通の設定プロパティー 436
14.1.1. replicas 436
14.1.2. bootstrapServers 436
14.1.3. ssl 437
14.1.4. trustedCertificates 438
14.1.5. resources 439
14.1.6. image 442
14.1.7. livenessProbe および readinessProbe healthcheck 447
14.1.8. metricsConfig 448
14.1.9. jvmOptions 450
14.1.10. ガベッジコレクターのロギング 454
14.2. スキーマプロパティー 455
14.2.1. Kafka スキーマ参照 455
14.2.2. KafkaSpec スキーマ参照 455
14.2.3. KafkaClusterSpec スキーマ参照 456
14.2.3.1. listeners 456
14.2.3.2. 設定 457
14.2.3.3. brokerRackInitImage 460
14.2.3.4. ログ 461
14.2.3.5. KafkaClusterSpec スキーマプロパティー 463
14.2.4. Generic KafkaListener スキーマ参照 465
14.2.4.1. listeners 466
14.2.4.2. type 467
14.2.4.3. port 471
14.2.4.4. tls 472
14.2.4.5. 認証 472
14.2.4.6. networkPolicyPeers 472
14.2.4.7. GenericKafkaListenerスキーマプロパティ 473
14.2.5. KafkaListenerAuthenticationTls スキーマ参照 475
14.2.6. KafkaListenerAuthenticationScramSha512 スキーマ参照 475
14.2.7. KafkaListenerAuthenticationOAuth スキーマ参照 476
14.2.8. GenericSecretSource スキーマ参照 479
14.2.9. CertSecretSource スキーマ参照 480
14.2.10. GenericKafkaListenerConfiguration スキーマ参照 480
14.2.10.1. brokerCertChainAndKey 480
14.2.10.2. externalTrafficPolicy 481
14.2.10.3. loadBalancerSourceRanges 481
14.2.10.4. class 482
14.2.10.5. preferredNodePortAddressType 482
14.2.10.6. useServiceDnsDomain 483

14.2.10.7. GenericKafkaListenerConfigurationスキーマプロパティ 484
14.2.11. CertAndKeySecretSource スキーマ参照 487
14.2.12. GenericKafkaListenerConfigurationBootstrap schema reference 488
14.2.12.1. alternativeNames 488
14.2.12.2. host 489
14.2.12.3. nodePort 490
14.2.12.4. loadBalancerIP 491
14.2.12.5. annotations 491
14.2.12.6. GenericKafkaListenerConfigurationBootstrapスキーマのプロパティ 492
14.2.13. GenericKafkaListenerConfigurationBroker schema reference 493
14.2.13.1. GenericKafkaListenerConfigurationBrokerスキーマプロパティ 494
14.2.14. EphemeralStorage スキーマ参照 495
14.2.15. PersistentClaimStorage スキーマ参照 496
14.2.16. PersistentClaimStorageOverride スキーマ参照 497
14.2.17. JbodStorage スキーマ参照 497
14.2.18. KafkaAuthorizationSimple スキーマ参照 498
14.2.18.1. superUsers 498
14.2.18.2. KafkaAuthorizationSimpleスキーマのプロパティ 499
14.2.19. KafkaAuthorizationOpa スキーマ参照 499
14.2.19.1. url 500
14.2.19.2. allowOnError 500
14.2.19.3. initialCacheCapacity 500
14.2.19.4. maximumCacheSize 500
14.2.19.5. expireAfterMs 500
14.2.19.6. superUsers 500
14.2.19.7. KafkaAuthorizationOpaスキーマのプロパティ 501
14.2.20. KafkaAuthorizationKeycloak スキーマ参照 502
14.2.21. KafkaAuthorizationCustomスキーマリファレンス 503
14.2.21.1. authorizerClass 504
14.2.21.2. superUsers 504
14.2.21.3. KafkaAuthorizationCustomスキーマのプロパティ 505
14.2.22. Rack スキーマ参照 506
14.2.22.1. ラック間でのパーティションレプリカの分散 506
14.2.22.2. 最も近いレプリカからのメッセージの消費 507
14.2.22.3. Rackスキーマのプロパティ 509
14.2.23. Probe スキーマ参照 509
14.2.24. JvmOptions スキーマ参照 510
14.2.25. SystemProperty スキーマ参照 511
14.2.26. KafkaJmxOptions スキーマ参照 511
14.2.26.1. KafkaJmxOptionsスキーマプロパティ 513
14.2.27. KafkaJmxAuthenticationPassword スキーマ参照 513
14.2.28. JmxPrometheusExporterMetrics schema reference 514
14.2.29. ExternalConfigurationReferenceのスキーマ参照 514
14.2.30. InlineLogging スキーマ参照 514
14.2.31. ExternalLogging スキーマ参照 515
14.2.32. KafkaClusterTemplate スキーマ参照 515
14.2.33. StatefulSetTemplate スキーマ参照 517
14.2.34. MetadataTemplate スキーマ参照 517
14.2.34.1. MetadataTemplateスキーマのプロパティ 518
14.2.35. PodTemplate スキーマ参照 518
14.2.35.1. hostAliases 519
14.2.35.2. PodTemplateスキーマのプロパティ 520
14.2.36. InternalServiceTemplateのスキーマ参照 521

14.2.37. ResourceTemplate スキーマ参照 522
14.2.38. PodDisruptionBudgetTemplate スキーマ参照 522
14.2.38.1. PodDisruptionBudgetTemplate schema properties 523
14.2.39. ContainerTemplate スキーマ参照 523
14.2.39.1. ContainerTemplateスキーマのプロパティ 524
14.2.40. ContainerEnvVar スキーマ参照 524
14.2.41. ZookeeperClusterSpec スキーマ参照 525
14.2.41.1. 設定 525
14.2.41.2. ログ 527
14.2.41.3. ZookeeperClusterSpec schema properties 529
14.2.42. ZookeeperClusterTemplate スキーマ参照 530
14.2.43. EntityOperatorSpec スキーマ参照 531
14.2.44. EntityTopicOperatorSpec スキーマ参照 532
14.2.44.1. ログ 532
14.2.44.2. EntityTopicOperatorSpec schema properties 534
14.2.45. EntityUserOperatorSpec スキーマ参照 535
14.2.45.1. ログ 535
14.2.45.2. EntityUserOperatorSpec schema properties 537
14.2.46. TlsSidecar スキーマ参照 538
14.2.46.1. TlsSidecarスキーマのプロパティ 541
14.2.47. EntityOperatorTemplate スキーマ参照 541
14.2.48. CertificateAuthority スキーマ参照 542
14.2.49. CruiseControlSpec スキーマ参照 543
14.2.50. CruiseControlTemplate スキーマ参照 545
14.2.51. BrokerCapacity スキーマー参照 546
14.2.52. KafkaExporterSpec スキーマ参照 546
14.2.53. KafkaExporterTemplate スキーマ参照 547
14.2.54. KafkaStatus スキーマ参照 548
14.2.55. Condition スキーマ参照 549
14.2.56. ListenerStatus スキーマ参照 549
14.2.57. ListenerAddress スキーマ参照 550
14.2.58. KafkaConnect スキーマ参照 550
14.2.59. KafkaConnectSpec スキーマ参照 550
14.2.59.1. 設定 551
14.2.59.2. ログ 554
14.2.59.3. KafkaConnectSpecスキーマプロパティ 556
14.2.60. ClientTls スキーマ参照 558
14.2.60.1. trustedCertificates 558
14.2.60.2. ClientTls スキーマプロパティー 558
14.2.61. KafkaClientAuthenticationTlsスキーマ参照 558
14.2.61.1. certificateAndKey 559
14.2.61.2. KafkaClientAuthenticationTlsスキーマプロパティ 559
14.2.62. KafkaClientAuthenticationScramSha512 スキーマ参照 560
14.2.62.1. username 560
14.2.62.2. passwordSecret 560
14.2.62.3. KafkaClientAuthenticationScramSha512 schema properties 562
14.2.63. PasswordSecretSource スキーマ参照 562
14.2.64. KafkaClientAuthenticationPlain スキーマ参照 562
14.2.64.1. username 563
14.2.64.2. passwordSecret 563
14.2.64.3. KafkaClientAuthenticationPlainスキーマプロパティ 564
14.2.65. KafkaClientAuthenticationOAuth スキーマ参照 565
14.2.65.1. KafkaClientAuthenticationOAuthスキーマプロパティ 568

14.2.66. JaegerTracing スキーマ参照 569
14.2.67. KafkaConnectTemplate スキーマ参照 570
14.2.68. DeploymentTemplateスキーマ参照 571
14.2.69. BuildConfigTemplate schema reference 571
14.2.70. ExternalConfiguration スキーマ参照 572
14.2.70.1. env 572
14.2.70.2. volumes 574
14.2.70.3. ExternalConfigurationスキーマのプロパティ 579
14.2.71. ExternalConfigurationEnv スキーマ参照 579
14.2.72. ExternalConfigurationEnvVarSource スキーマ参照 580
14.2.73. ExternalConfigurationVolumeSource スキーマ参照 580
14.2.74. スキーマ・リファレンスの構築 581
14.2.74.1. 出力 581
14.2.74.2. plugins 583
14.2.74.3. スキーマ・プロパティの構築 590
14.2.75. DockerOutputスキーマリファレンス 590
14.2.76. ImageStreamOutputスキーマリファレンス 591
14.2.77. プラグインのスキーマ参照 591
14.2.78. JarArtifactスキーマリファレンス 592
14.2.79. TgzArtifactスキーマリファレンス 593
14.2.80. ZipArtifactスキーマリファレンス 593
14.2.81. MavenArtifactスキーマリファレンス 594
14.2.82. OtherArtifactスキーマ参照 595
14.2.83. KafkaConnectStatus スキーマ参照 596
14.2.84. ConnectorPlugin スキーマ参照 597
14.2.85. KafkaTopic スキーマ参照 597
14.2.86. KafkaTopicSpec スキーマ参照 597
14.2.87. KafkaTopicStatus スキーマ参照 598
14.2.88. KafkaUser スキーマ参照 598
14.2.89. KafkaUserSpec スキーマ参照 599
14.2.90. KafkaUserTlsClientAuthentication スキーマ参照 599
14.2.91. KafkaUserTlsExternalClientAuthentication schema reference 600
14.2.92. KafkaUserScramSha512ClientAuthentication スキーマ参照 600
14.2.93. パスワードスキーマの参照 601
14.2.94. PasswordSourceスキーマリファレンス 601
14.2.95. KafkaUserAuthorizationSimple スキーマ参照 601
14.2.96. AclRule スキーマ参照 602
14.2.96.1. resource 603
14.2.96.2. type 604
14.2.96.3. operation 604
14.2.96.4. host 605
14.2.96.5. AclRuleスキーマのプロパティ 605
14.2.97. AclRuleTopicResource スキーマ参照 606
14.2.98. AclRuleGroupResource スキーマ参照 607
14.2.99. AclRuleClusterResource スキーマ参照 607
14.2.100. AclRuleTransactionalIdResource スキーマ参照 608
14.2.101. KafkaUserQuotas スキーマ参照 608
14.2.101.1. quotas 609
14.2.101.2. KafkaUserQuotasスキーマのプロパティ 610
14.2.102. KafkaUserTemplateスキーマリファレンス 611
14.2.102.1. KafkaUserTemplateスキーマのプロパティ 611
14.2.103. KafkaUserStatus スキーマ参照 611
14.2.104. KafkaMirrorMaker スキーマ参照 612

14.2.105. KafkaMirrorMakerSpec スキーマ参照 612
14.2.105.1. include 613
14.2.105.2. KafkaMirrorMakerConsumerSpec and KafkaMirrorMakerProducerSpec 613
14.2.105.3. ログ 613
14.2.105.4. KafkaMirrorMakerSpecスキーマのプロパティ 614
14.2.106. KafkaMirrorMakerConsumerSpec スキーマ参照 616
14.2.106.1. numStreams 616
14.2.106.2. offsetCommitInterval 617
14.2.106.3. 設定 617
14.2.106.4. groupId 619
14.2.106.5. KafkaMirrorMakerConsumerSpec schema properties 619
14.2.107. KafkaMirrorMakerProducerSpec スキーマ参照 620
14.2.107.1. abortOnSendFailure 620
14.2.107.2. 設定 621
14.2.107.3. KafkaMirrorMakerProducerSpec schema properties 622
14.2.108. KafkaMirrorMakerTemplate スキーマ参照 623
14.2.109. KafkaMirrorMakerStatus スキーマ参照 624
14.2.110. KafkaBridge スキーマ参照 624
14.2.111. KafkaBridgeSpec スキーマ参照 624
14.2.111.1. ログ 625
14.2.111.2. KafkaBridgeSpecのスキーマプロパティ 628
14.2.112. KafkaBridgeHttpConfig スキーマ参照 630
14.2.112.1. cors 630
14.2.112.2. KafkaBridgeHttpConfig schema properties 631
14.2.113. KafkaBridgeHttpCors スキーマ参照 631
14.2.114. KafkaBridgeAdminClientSpec schema reference 632
14.2.115. KafkaBridgeConsumerSpec スキーマ参照 632
14.2.115.1. KafkaBridgeConsumerSpec schema properties 634
14.2.116. KafkaBridgeProducerSpec スキーマ参照 634
14.2.116.1. KafkaBridgeProducerSpec schema properties 636
14.2.117. KafkaBridgeTemplate スキーマ参照 636
14.2.118. KafkaBridgeStatus スキーマ参照 637
14.2.119. KafkaConnector スキーマ参照 637
14.2.120. KafkaConnectorSpec スキーマ参照 638
14.2.121. KafkaConnectorStatus スキーマ参照 638
14.2.122. KafkaMirrorMaker2 スキーマ参照 639
14.2.123. KafkaMirrorMaker2Spec スキーマ参照 639
14.2.124. KafkaMirrorMaker2ClusterSpec スキーマ参照 641
14.2.124.1. 設定 641
14.2.124.2. KafkaMirrorMaker2ClusterSpec schema properties 641
14.2.125. KafkaMirrorMaker2MirrorSpec スキーマ参照 642
14.2.126. KafkaMirrorMaker2ConnectorSpec スキーマ参照 643
14.2.127. KafkaMirrorMaker2Status スキーマ参照 644
14.2.128. KafkaRebalance スキーマ参照 645
14.2.129. KafkaRebalanceSpec スキーマ参照 645
14.2.130. KafkaRebalanceStatus スキーマ参照 646

付録A サブスクリプションの使用 648
アカウントへのアクセス 648
サブスクリプションのアクティベート 648
Zip および Tar ファイルのダウンロード 648

多様性を受け入れるオープンソースの強化

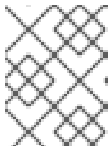
Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージをご覧ください。

第1章 AMQ STREAMS の概要

AMQ Streams は、OpenShift クラスターで Apache Kafka を実行するプロセスを簡素化します。

本ガイドでは、Kafka コンポーネントの設定方法と、AMQ Streams Operator の使用方法を説明します。手順は、デプロイメントの変更方法や、Cruise Control や分散トレーシングなどの追加機能を導入する方法に関連しています。

AMQ Streams カスタムリソース を使用して、デプロイメントを設定できます。カスタムリソース API リファレンス は、設定で使用できるプロパティーを説明します。



注記

AMQ Streams を使用する方法ステップごとのデプロイメント手順は、『OpenShift での AMQ Streams のデプロイおよびアップグレード』を参照してください 。

1.1. KAFKA の機能

Kafka の基盤のデータストリーム処理機能とコンポーネントアーキテクチャーによって以下が提供されます。

- スループットが非常に高く、レイテンシーが低い状態でデータを共有するマイクロサービスおよびその他のアプリケーション。
- メッセージの順序の保証。
- アプリケーションの状態を再構築するためにデータストレージからメッセージを巻き戻し/再生。
- キーバリューログの使用時に古いレコードを削除するメッセージ圧縮。
- クラスター設定での水平スケーラビリティー。
- 耐障害性を制御するデータのレプリケーション。
- 即時アクセス用の大量データの保持

1.2. KAFKA のユースケース

Kafka の機能は、以下に適しています。

- イベント駆動型のアーキテクチャー。
- アプリケーションの状態に加えられた変更をイベントのログとしてキャプチャーするイベントソーシング。
- メッセージのブローカー
- Web サイトアクティビティーの追跡
- メトリクスによる運用上のモニタリング
- ログの収集および集計
- 分散システムのコミットログ

- アプリケーションがリアルタイムでデータに対応できるようにするストリーム処理。

1.3. AMQ STREAMS による KAFKA のサポート

AMQ Streams は、Kafka を OpenShift で実行するためのコンテナーイメージおよび Operator を提供します。AMQ Streams Operator は、AMQ Streams の実行に必要です。AMQ Streams で提供される Operator は、Kafka を効果的に管理するために、専門的なオペレーション情報で目的に合うよう構築されています。

Operator は以下のプロセスを単純化します。

- Kafka クラスターのデプロイおよび実行。
- Kafka コンポーネントのデプロイおよび実行。
- Kafka へアクセスするための設定。
- Kafka へのアクセスをセキュア化。
- Kafka のアップグレード。
- ブローカーの管理。
- トピックの作成および管理。
- ユーザーの作成および管理。

1.4. AMQ STREAMS の OPERATOR

AMQ Streams では **Operator** を使用して Kafka をサポートし、Kafka のコンポーネントおよび依存関係を OpenShift にデプロイして管理します。

Operator は、OpenShift アプリケーションのパッケージ化、デプロイメント、および管理を行う方法です。AMQ Streams Operator は OpenShift の機能を拡張し、Kafka デプロイメントに関連する共通タスクや複雑なタスクを自動化します。Kafka 操作の情報をコードに実装することで、Kafka の管理タスクは簡素化され、必要な手動の作業が少なくなります。

Operator

AMQ Streams は、OpenShift クラスター内で実行中の Kafka クラスターを管理するための Operator を提供します。

Cluster Operator

Apache Kafka クラスター、Kafka Connect、Kafka MirrorMaker、Kafka Bridge、Kafka Exporter、および Entity Operator をデプロイおよび管理します。

Entitiy Operator

Topic Operator および User Operator を構成します。

Topic Operator

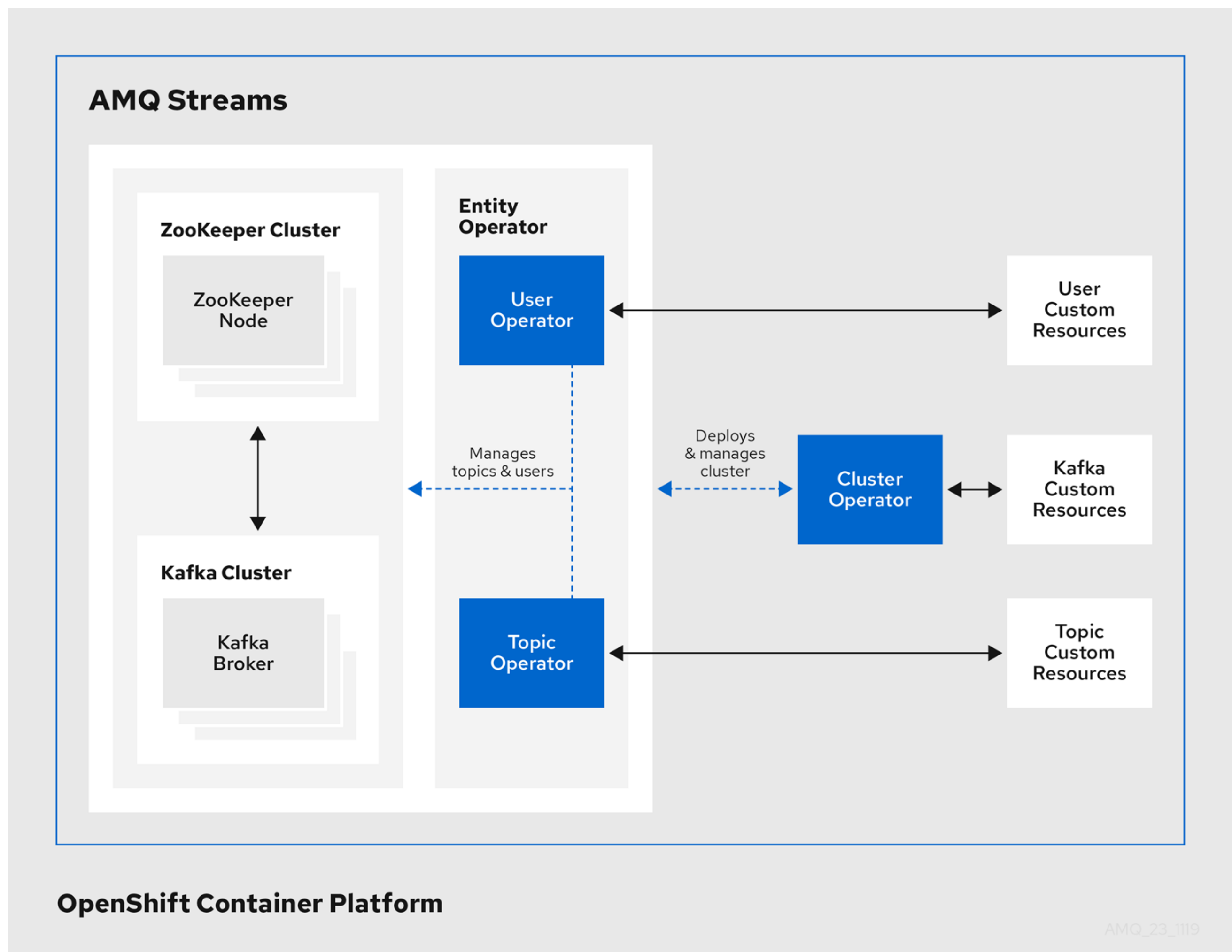
Kafka トピックを管理します。

User Operator

Kafka ユーザーを管理します。

Cluster Operator は、Kafka クラスターと同時に、Topic Operator および User Operator を **Entity Operator** 設定の一部としてデプロイできます。

AMQ Streams アーキテクチャー内の Operator



1.4.1. Cluster Operator

AMQ Streams では、Cluster Operator を使用して以下のクラスターをデプロイおよび管理します。

- Kafka (ZooKeeper、Entity Operator、Kafka Exporter、Cruise Control を含む)
- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

クラスターのデプロイメントにはカスタムリソースが使用されます。

たとえば、以下のように Kafka クラスターをデプロイします。

- クラスター設定のある **Kafka** リソースが OpenShift クラスター内で作成されます。
- **Kafka** リソースに宣言された内容を基にして、該当する Kafka クラスターが Cluster Operator によってデプロイされます。

Cluster Operator で以下もデプロイできます (**Kafka** リソースの設定より)。

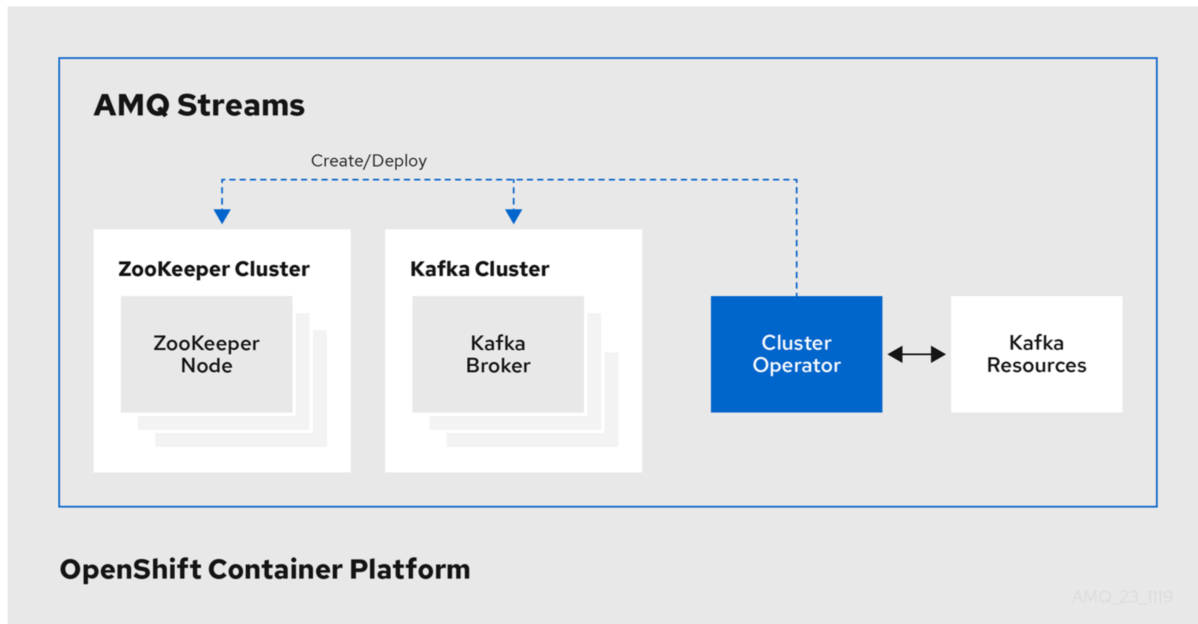
- **KafkaTopic** カスタムリソースより Operator スタイルのトピック管理を提供する Topic Operator

- **KafkaUser** カスタムリソースより Operator スタイルのユーザー管理を提供する User Operator

デプロイメントの Entity Operator 内の Topic Operator および User Operator 関数。

Cluster Operator を AMQ Streams の Drain Cleaner のデプロイメントと共に使用して、Pod のエビクションに役立ちます。AMQ Streams Drain Cleaner をデプロイすると、Cluster Operator を使用して OpenShift ではなく Kafka Pod を移動できます。AMQ Streams の Drain クリーニングでは、Pod にローリングアップデートアノテーションが付けられます。アノテーションは、ローリングアップデートを実行するために Cluster Operator に通知します。

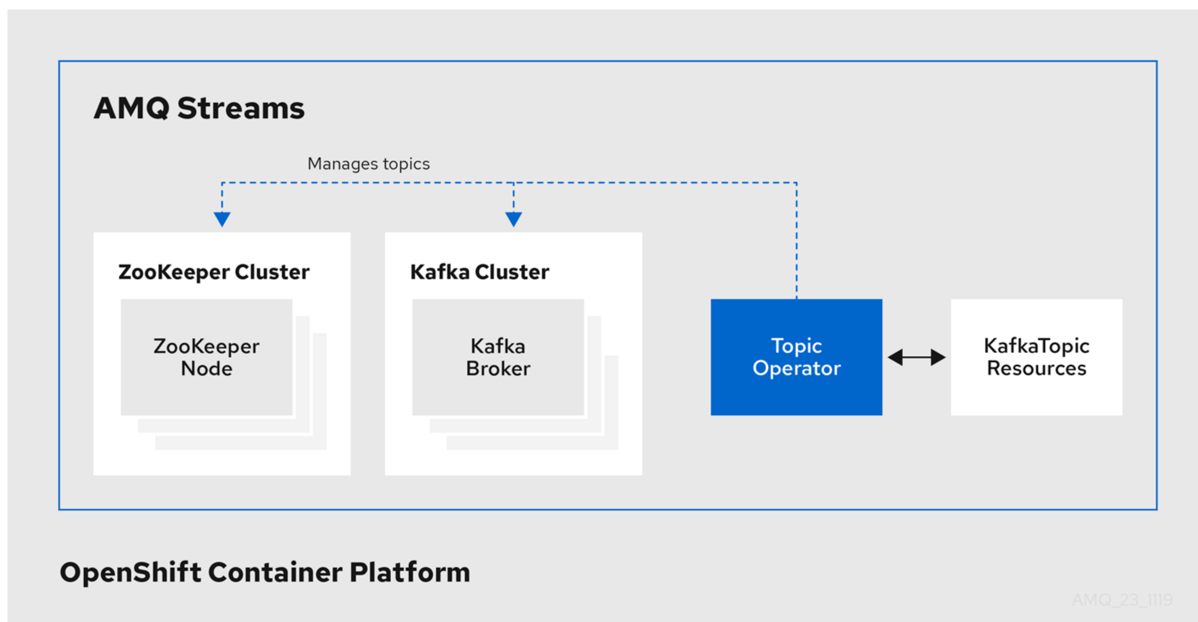
Cluster Operator のアーキテクチャー例



1.4.2. Topic Operator

Topic Operator は、OpenShift リソースより Kafka クラスターのトピックを管理する方法を提供します。

Topic Operator のアーキテクチャー例



Topic Operator の役割は、対応する Kafka トピックと同期して Kafka トピックを記述する **KafkaTopic** OpenShift リソースのセットを保持することです。

KafkaTopic とトピックの関係は次のとおりです。

- **KafkaTopic** が作成されると、Topic Operator によってトピックが作成されます。
- **KafkaTopic** が削除されると、Topic Operator によってトピックが削除されます。
- **KafkaTopic** が変更されると、Topick Operator によってトピックが更新されます。

上記と逆になるトピックと **KafkaTopic** の関係は次のとおりです。

- トピックが Kafka クラスター内で作成されると、Operator によって **KafkaTopic** が作成されます。
- トピックが Kafka クラスターから削除されると、Operator によって **KafkaTopic** が削除されます。
- トピックが Kafka クラスターで変更されると、Operator によって **KafkaTopic** が更新されます。

このため、**KafkaTopic** をアプリケーションのデプロイメントの一部として宣言でき、トピックの作成は Topic Operator によって行われます。アプリケーションは、必要なトピックからの作成または消費のみに対処する必要があります。

Topic Operator は、各トピックの情報を **トピックストア** で維持します。トピックストアは、Kafka トピックまたは OpenShift **KafkaTopic** カスタムリソースからの更新と継続的に同期されます。ローカルのインメモリートピックストアに適用される操作からの更新は、ディスク上のバックアップトピックストアに永続化されます。トピックが再設定されたり、別のブローカーに再割り当てされた場合、**KafkaTopic** は常に最新の状態になります。

1.4.3. User Operator

User Operator は、Kafka ユーザーが記述される **KafkaUser** リソースを監視して Kafka クラスターの Kafka ユーザーを管理し、Kafka ユーザーが Kafka クラスターで適切に設定されるようにします。

たとえば、**KafkaUser** とユーザーの関係は次のようになります。

- **KafkaUser** が作成されると、User Operator によって記述されるユーザーが作成されます。
- **KafkaUser** が削除されると、User Operator によって記述されるユーザーが削除されます。
- **KafkaUser** が変更されると、User Operator によって記述されるユーザーが更新されます。

User Operator は Topic Operator とは異なり、Kafka クラスターからの変更は OpenShift リソースと同期されません。アプリケーションで直接 Kafka トピックを Kafka で作成することは可能ですが、ユーザーが User Operator と同時に直接 Kafka クラスターで管理されることは想定されません。

User Operator では、アプリケーションのデプロイメントの一部として **KafkaUser** リソースを宣言できます。ユーザーの認証および承認メカニズムを指定できます。たとえば、ユーザーがブローカーへのアクセスを独占しないようにするため、Kafka リソースの使用を制御する **ユーザークォータ** を設定することもできます。

ユーザーが作成されると、ユーザークレデンシャルが **Secret** に作成されます。アプリケーションはユーザーとそのクレデンシャルを使用して、認証やメッセージの生成または消費を行う必要があります。

User Operator は 認証のクレデンシャルを管理する他に、**KafkaUser** 宣言にユーザーのアクセス権限の記述を含めることで承認も管理します。

1.4.4. AMQ Streams operator のフィーチャーゲート

フィーチャーゲートを使用して、operator の一部の機能を有効または無効にすることができます。

フィーチャーゲートは Operator の設定で指定され、alpha、beta、または General Availability (GA) の 3 段階の成熟度があります。

詳細は、「Feature gates」 を参照してください。

1.5. AMQ STREAMS のカスタムリソース

AMQ Streams を使用した Kafka コンポーネントの OpenShift クラスターへのデプロイメントは、カスタムリソースの適用により高度な設定が可能です。カスタムリソースは、OpenShift リソースを拡張するために CRD (カスタムリソース定義、Custom Resource Definition) によって追加される API のインスタンスとして作成されます。

CRD は、OpenShift クラスターでカスタムリソースを記述するための設定手順として機能し、デプロイメントで使用する Kafka コンポーネントごとに AMQ Streams で提供されます。CRD およびカスタムリソースは YAML ファイルとして定義されます。YAML ファイルのサンプルは AMQ Streams ディストリビューションに同梱されています。

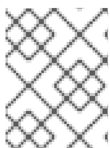
また、CRD を使用すると、CLI へのアクセスや設定検証などのネイティブ OpenShift 機能を AMQ Streams リソースで活用することもできます。

1.5.1. AMQ Streams カスタムリソースの例

AMQ Streams 固有リソースのインスタンス化および管理に使用されるスキーマを定義するため、CRD をクラスターに 1 度インストールする必要があります。

CRD をインストールして新規カスタムリソースタイプをクラスターに追加した後に、その仕様に基づいてリソースのインスタンスを作成できます。

クラスターの設定によりますが、インストールには通常、クラスター管理者権限が必要です。



注記

カスタムリソースの管理は、AMQ Streams 管理者のみが行えます。詳細は、「 AMQ Streams の管理者の指名 」を参照してください。

kind:Kafka などの新しい **kind** リソースは、OpenShift クラスター内で CRD によって定義されます。

Kubernetes API サーバーを使用すると、**kind** を基にしたカスタムリソースの作成が可能になり、カスタムリソースが OpenShift クラスターに追加されたときにカスタムリソースの検証および格納方法を CRD から判断します。

警告

CRD が削除されると、そのタイプのカスタムタイプも削除されます。さらに、Pod や Statefulset などのカスタムリソースによって作成されたリソースも削除されます。

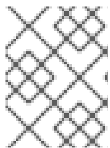
AMQ Streams 固有の各カスタムリソースは、リソースの **kind** の CRD によって定義されるスキーマに準拠します。AMQ Streams コンポーネントのカスタムリソースには、**spec**で定義される共通の設定プロパティーがあります。

CRD とカスタムリソースの関係を理解するため、Kafka トピックの CRD の例を見てみましょう。

Kafka トピックの CRD

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

1. CRD を識別するためのトピック CRD、その名前および名前のメタデータ。
2. この CRD に指定された項目には、トピックの API にアクセスするため URL に使用されるグル ShortNameープ (ドメイン) 名、複数名、およびサポートされるスキーマバージョンが含まれます。他の名前は、CLI のインスタンスリソースを識別するために使用されます。たとえば、**oc get kafkaShortNametopic my-topic** や **oc get kafkatopics** などです。
3. ShortName は CLI コマンドで使用できます。たとえば、**oc get kafkatopic** の代わりに **oc get kt** を略名として使用できます。
4. カスタムリソースで **get** コマンドを使用する場合に示される情報。
5. リソースの スキーマ参照 に記載されている CRD の現在のステータス。
6. openAPIV3Schema 検証によって、トピックカスタムリソースの作成が検証されます。たとえば、トピックには 1 つ以上のパーティションと 1 つのレプリカが必要です。



注記

ファイル名に、インデックス番号とそれに続く「Crd」が含まれるため、AMQ Streams インストールファイルと提供される CRD YAML ファイルを識別できます。

KafkaTopic カスタムリソースに該当する例は次のとおりです。

Kafka トピックカスタムリソース

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

1. **kind** および **apiVersion** によって、インスタンスであるカスタムリソースの CRD が特定されます。
2. トピックまたはユーザーが属する Kafka クラスターの名前 (**Kafka** リソースの名前と同じ) を定義する、**KafkaTopic** および **KafkaUser** リソースのみに適用可能なラベル。
3. 指定内容には、トピックのパーティション数およびレプリカ数や、トピック自体の設定パラメーターが示されています。この例では、メッセージがトピックに保持される期間や、ログのセグメン

トファイルサイズが指定されています。

4 **KafkaTopic** リソースのステータス条件。**lastTransitionTime** で **type** 条件が **Ready** に変更されています。

プラットフォーム CLI からカスタムリソースをクラスターに適用できます。カスタムリソースが作成されると、Kubernetes API の組み込みリソースと同じ検証が使用されます。

KafkaTopic の作成後、Topic Operator は通知を受け取り、該当する Kafka トピックが AMQ Streams で作成されます。

その他のリソース

- 「Extend the Kubernetes API with CustomResourceDefinitions」
- Example configuration files provided with AMQ Streams

1.6. リスナーの設定

リスナーは、Kafka ブローカーへの接続に使用されます。

AMQ Streamsは、**Kafka** リソースを介してリスナーを設定するためのプロパティを備えたジェネリックな **GenericKafkaListener** スキーマを提供しています。

GenericKafkaListener は、リスナー設定に柔軟なアプローチを提供します。プロパティーを指定して、OpenShift クラスター内で接続する **内部** リスナーを設定したり、OpenShift クラスター外部で接続する**外部** リスナーを設定したりできます。

各リスナーは **Kafka** リソースの配列として定義されます。名前とポートが一意であれば、必要なリスナーをいくつでも設定できます。

たとえば、異なる認証メカニズムを必要とするネットワークからのアクセスを処理する場合などに、複数の外部リスナーを設定することがあります。また、OpenShift ネットワークを外部ネットワークに参加させる必要があることがあります。この場合、OpenShift サービスの DNS ドメイン（通常は**.cluster.local**）が使用されないように内部リスナーを構成することができます（**useServiceDnsDomain** プロパティを使用）。

リスナーで利用可能な設定オプションの詳細は、「**GenericKafkaListener スキーマ参照**」を参照してください。

Kafka ブローカーへのアクセスをセキュアにするためのリスナー設定

リスナーを設定して、認証を使用したセキュアな接続を確立できます。詳細は、「Kafka ブローカーへのアクセスのセキュリティー保護」を参照してください。

OpenShift 外部のクライアントアクセスに対する外部リスナーの設定

ロードバランサーなどの指定された接続メカニズムを使用して、OpenShift 環境外部のクライアントアクセスに対して外部リスナーを設定できます。外部クライアントに接続するための設定オプションの詳細は、「OpenShift クラスター外の外部クライアントからの Kafka へのアクセス」を参照してください。

リスナー証明書

TLS 暗号化が有効になっている TLS リスナーまたは外部リスナーの、**Kafka リスナー証明書** と呼ばれる独自のサーバー証明書を提供できます。詳細は「Kafka リスナー証明書」を参照してください。

1.7. 本書の表記慣例

置き換え可能なテキスト

本書では、置き換え可能なテキストは、**monospace** フォントのイタリック体、大文字、およびハイフンで記載されています。

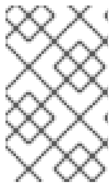
たとえば、以下のコードでは **MY-NAMESPACE** を namespace の名前に置き換えます。

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

第2章 デプロイメント設定

本章では、カスタムリソースを使用してサポートされるデプロイメントのさまざまな側面を設定する方法について説明します。

- Kafka クラスター
- Kafka Connect クラスター
- Kafka MirrorMaker
- Kafka Bridge
- Cruise Control



注記

カスタムリソースに適用されるラベルは、Kafka MirrorMaker を構成する OpenShift リソースにも適用されます。そのため、必要に応じてリソースにラベルが適用されるため便利です。

AMQ Streams では 、 デプロイメントの独自の Kafka コンポーネント設定を構築するときに開始点として機能できる設定ファイルのサンプルが提供されます。

Strimzi デプロイメントの監視

Prometheus および Grafana を使用して、Strimzi デプロイメントを監視できます。詳細は、「Kafka へのメトリクスの導入」を参照してください。

2.1. KAFKA クラスターの設定

ここでは、AMQ Streams クラスターで Kafka デプロイメントを設定する方法を説明します。Kafka クラスターは ZooKeeper クラスターとデプロイされます。デプロイメントには、Kafka トピックおよびユーザーを管理する Topic Operator および User Operator も含まれます。

Kafkaの設定は、**Kafka**リソースを使って行います。設定オプションは、**Kafka** リソース内の ZooKeeper および Entity Operator でも利用できます。Entity Operator は Topic Operator と User Operator で構成されます。

Kafka リソースの完全なスキーマは 「**Kafka** スキーマ参照」 に記載されています。Apache Kafka の詳細は Apache Kafka のドキュメント を参照してください。

リスナーの設定

クライアントを Kafka ブローカーに接続するためのリスナーを設定します。ブローカーに接続するためのリスナーの設定に関する詳細は、「リスナーの設定」を参照してください。

Kafka へのアクセスの承認

ユーザーが実行するアクションを許可または拒否するように Kafka クラスターを設定できます。詳細は、「Kafka ブローカーへのアクセスのセキュリティー保護」を参照してください。

TLS 証明書の管理

Kafka をデプロイする場合、Cluster Operator は自動で TLS 証明書の設定および更新を行い、クラスター内での暗号化および認証を有効にします。必要な場合は、更新期間の終了前にクラスターおよびクライアント CA 証明書を手動で更新できます。クラスターおよびクライアント CA 証明書によって使用

される鍵を置き換えることもできます。詳細は、「CA 証明書の手動更新」および「秘密鍵の置換」を参照してください。

2.1.1. Kafka の設定

Kafka リソースのプロパティーを使用して、Kafka デプロイメントを設定します。

Kafka の設定に加え、ZooKeeper および AMQ Streams Operator の設定を追加することもできます。ロギングやヘルスチェックなどの一般的な設定プロパティーは、コンポーネントごとに独立して設定されます。

この手順では、可能な設定オプションの一部のみを取り上げますが、特に重要なオプションは次のとおりです。

- リソース要求 (CPU/メモリー)
- 最大および最小メモリー割り当ての JVM オプション
- リスナー (およびクライアントの認証)
- 認証
- ストレージ
- ラックアウェアネス (Rack Awareness)
- メトリクス
- Cruise Control によるクラスターのリバランス

Kafka バージョン

Kafka **設定の inter.broker.protocol.version** プロパティーは、指定された Kafka バージョン (spec.**kafka.version**)によってサポートされるバージョン **である必要があります**。このプロパティーは、Kafka クラスターで使用される Kafka プロトコルのバージョンを表します。

Kafka 3.0.0 から、**inter.broker.protocol.version** が **3.0** 以上に設定されていると、**log.message.format.version** オプションは無視され、設定する必要はありません。

Kafka バージョンのアップグレード時に、**inter.broker.protocol.version** への更新が必要です。詳細は「 Kafka のアップグレード 」を参照してください。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

以下をデプロイする手順については、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』を参照してください。

- Cluster Operator
- Kafka クラスター

手順

1. **Kafka** リソースの **spec** プロパティーを編集します。
   設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3 (1)
    version: 3.0.0 (2)
    logging: (3)
      type: inline
      loggers:
        kafka.root.logger.level: "INFO"
    resources: (4)
      requests:
        memory: 64Gi
        cpu: "8"
      limits:
        memory: 64Gi
        cpu: "12"
    readinessProbe: (5)
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe:
      initialDelaySeconds: 15
      timeoutSeconds: 5
    jvmOptions: (6)
      -Xms: 8192m
      -Xmx: 8192m
    image: my-org/my-image:latest (7)
    listeners: (8)
      - name: plain (9)
        port: 9092 (10)
        type: internal (11)
        tls: false (12)
        configuration:
          useServiceDnsDomain: true (13)
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication: (14)
          type: tls
      - name: external (15)
        port: 9094
        type: route
        tls: true
        configuration:
          brokerCertChainAndKey: (16)
            secretName: my-secret
            certificate: my-certificate.crt
```

```
        key: my-key.key
    authorization: (17)
      type: simple
    config: (18)
      auto.create.topics.enable: "false"
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      inter.broker.protocol.version: "3.0"
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (19)
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
    storage: (20)
      type: persistent-claim (21)
      size: 10000Gi (22)
    rack: (23)
      topologyKey: topology.kubernetes.io/zone
    metricsConfig: (24)
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef: (25)
          name: my-config-map
          key: my-key
    # ...
  zookeeper: (26)
    replicas: 3 (27)
    logging: (28)
      type: inline
      loggers:
        zookeeper.root.logger: "INFO"
    resources:
      requests:
        memory: 8Gi
        cpu: "2"
      limits:
        memory: 8Gi
        cpu: "2"
    jvmOptions:
      -Xms: 4096m
      -Xmx: 4096m
    storage:
      type: persistent-claim
      size: 1000Gi
    metricsConfig:
      # ...
  entityOperator: (29)
    tlsSidecar: (30)
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
```

```
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging: (31)
        type: inline
        loggers:
          rootLogger.level: "INFO"
      resources:
        requests:
          memory: 512Mi
          cpu: "1"
        limits:
          memory: 512Mi
          cpu: "1"
    userOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging: (32)
        type: inline
        loggers:
          rootLogger.level: INFO
      resources:
        requests:
          memory: 512Mi
          cpu: "1"
        limits:
          memory: 512Mi
          cpu: "1"
  kafkaExporter: (33)
    # ...
  cruiseControl: (34)
    # ...
    tlsSidecar: (35)
    # ...
```

1. レプリカノードの数。クラスターにトピックがすでに定義されている場合は、クラスターをスケーリング できます。

2. アップグレード手順 に従い、サポートされるバージョンに変更できます。

3. 指定された Kafka loggers and log levels が ConfigMap を介して直接的に (**inline**) または間接的に (**external**) に追加されます。カスタム ConfigMap は、**log4j.properties** キー下に配置する必要があります。Kafka **kafka.root.logger.level** ロガーでは、ログレベルを INFO、ERROR、WARN、TRACE、DEBUG、FATAL または OFF に設定できます。

4. サポートされているリソース（現在は **cpu** と **memory**）の予約と、消費可能な最大リソースを指定するための制限を要求します。

5. コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するためのヘルスチェック。

6. Kafka を実行している仮想マシン (VM) のパフォーマンスを最適化するための JVM 設定オプション。

7. 高度な任意設定: 特別な場合のみ推奨されるコンテナーイメージの設定。

8. リスナーは、ブートストラップアドレスでクライアントが Kafka クラスターに接続する方法を設定します。リスナーは、OpenShift クラスター内部または外部からの接続の **内部** または **外部** リスナーとして設定されます。

9. リスナーを識別するための名前。Kafka クラスター内で一意である必要があります。

10. Kafka 内でリスナーによって使用されるポート番号。ポート番号は指定の Kafka クラスター内で一意である必要があります。許可されるポート番号は 9092 以上ですが、すでに Prometheus および JMX によって使用されているポート 9404 および 9999 以外になります。リスナーのタイプによっては、ポート番号は Kafka クライアントに接続するポート番号と同じではない場合があります。

11. **internal** として、または external リスナーに対して指定されるリスナータイプ（**route**、**loadbalancer**、**nodeport**、または **ingress**）。

12. 各リスナーの TLS 暗号化を有効にします。デフォルトは **false** です。**route** リスナーには TLS 暗号化は必要ありません。

13. クラスターサービスサフィックス（通常は **cluster.local**）を含む完全修飾 DNS 名が割り当てられているかどうかを定義します。

14. 相互 TLS、SCRAM-SHA-512、またはトークンベース OAuth 2.0 として指定される リスナー認証メカニズム。

15. 外部リスナー設定は 、**route**、**loadbalancer**、または **nodeport**からなど、Kafka クラスターが外部の OpenShift に公開される方法を指定します。

16. 外部の認証局によって管理される Kafka リスナー証明書 の任意設定。**brokerCertChainAndKey** は、サーバー証明書および秘密鍵が含まれる **Secret** を指定します。TLS による暗号化が有効な任意のリスナーで Kafka リスナー証明書を設定できます。

17. 承認は Kafka ブローカーで簡易、OAUTH2.0、または OPA 承認を有効にします。簡易承認では、**AclAuthorizer** Kafka プラグインが使用されます。

18. **config**はブローカーの設定を指定します。標準の Apache Kafka 設定が提供されることがあり、AMQ Streams によって直接管理されないプロパティーに限定されます。

19. 特定の **暗号スイート** または TLS バージョンを有効にするための、TLS による暗号化が有効になっているリスナーの SSL プロパティー。

20. Storageは**ephemeral**、**persistent-claim**、**jbod**のいずれかに設定されています。

21. 永続ボリュームのストレージサイズが増やされることがあり、追加の ボリュームが JBOD ストレージに追加されることがあります。

22. パーシステントストレージには、ダイナミックボリュームプロビジョニングのためのストレージ **id** や **class** など、 追加の設定オプションがあります。

23. ラックアウェアネス (Rack awareness) は、異なるラック全体でレプリカを分散するために設定されます。**topologykey** はクラスターノードのラベルと一致する必要があります。

24. Prometheus メトリクス は有効になっています。この例では、メトリクスは Prometheus JMX Exporter (デフォルトのメトリクスエクスポーター) に対して設定されます。

25. Prometheus JMX Exporter 経由でメトリクスを Grafana ダッシュボードにエクスポートする Prometheus ルール。Prometheus JMX Exporter の設定が含まれる ConfigMap を参照することで有効になります。**metricsConfig.valueFrom.configMapKeyRef.key** 配下に

空のファイルが含まれる ConfigMap の参照を使用して、追加設定なしでメトリクスを有効にできます。

26 Kafka 設定と似たプロパティーが含まれる、ZooKeeper 固有の設定。

27 ZooKeeper ノードの数通常、ZooKeeper クラスターまたはアンサンブルは、一般的に 3、5、7 個の奇数個のノードで実行されます。効果的なクォーラムを維持するには、過半数のノードが利用可能である必要があります。ZooKeeper クラスターでクォーラムを損失すると、クライアントへの応答が停止し、Kafka ブローカーが機能しなくなります。AMQ Streams では、 ZooKeeper クラスターの安定性および高可用性が重要になります。

28 指定された ZooKeeper ロガーおよびログレベル。

29 Topic Operator および User Operator の設定を指定する、Entity Operator 設定。

30 Entity Operator の TLS サイドカー設定。Entity Operator は、ZooKeeper とのセキュアな通信に TLS サイドカーを使用します。

31 指定された Topic Operator ロガーおよびログレベル。この例では、**inline** ロギングを使用します。

32 指定された User Operator ロガーおよびログレベル。

33 Kafka Exporter の設定。Kafka Exporter は、特にコンシューマーラグデータなど、Kafka ブローカーからメトリクスデータを抽出するオプションのコンポーネントです。

34 Kafka クラスターのリバランス に使用される Cruise Control の任意設定。

35 Cruise Conrol の TLS サイドカーの設定。Cruise Control は、ZooKeeper とのセキュアな通信に TLS サイドカーを使用します。

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

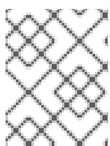
2.1.2. Entity Operator の設定

Entity Operator は、実行中の Kafka クラスターで Kafka 関連のエンティティーを管理します。

Entity Operator は以下と構成されます。

- Kafka トピックを管理する Topic Operator
- Kafka ユーザーを管理する User Operator

Cluster Operator は **Kafka** リソース設定を介して、Kafka クラスターのデプロイ時に、上記の Operator の 1 つまたは両方を含む Entity Operator をデプロイできます。



注記

デプロイされると、デプロイメント設定に応じて、Entity Operator にオペレーターが含まれます。

これらのオペレーターは、Kafka クラスターのトピックおよびユーザーを管理するために自動的に設定されます。

2.1.2.1. Entity Operator の設定プロパティー

Kafka.spec の **entityOperator** プロパティーを使用して Entity Operator を設定します。

entityOperator プロパティーでは複数のサブプロパティーがサポートされます。

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

tlsSidecar プロパティーには、ZooKeeper との通信に使用される TLS サイドカーコンテナーの設定が含まれます。

template プロパティーには、ラベル、アノテーション、アフィニティー、および容認 (Toleration) などの Entity Operator Pod の設定が含まれます。テンプレートの設定に関する詳細は、「OpenShift リソースのカスタマイズ」 を参照してください。

topicOperator プロパティーには、Topic Operator の設定が含まれます。このオプションがないと、Entity Operator は Topic Operator なしでデプロイされます。

userOperator プロパティーには、User Operator の設定が含まれます。このオプションがないと、Entity Operator は User Operator なしでデプロイされます。

Entity Operator の設定に使用されるプロパティーに関する詳細は **EntityUserOperatorSpec** schema reference を参照してください。

両方の Operator を有効にする基本設定の例

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

topicOperator および **userOperator** に空のオブジェクト (**{}**) が使用された場合、すべてのプロパティーでデフォルト値が使用されます。

topicOperator および **userOperator** プロパティーの両方がない場合、Entity Operator はデプロイされません。

2.1.2.2. Topic Operator 設定プロパティー

Topic Operator デプロイメントは、**topicOperator** オブジェクト内で追加オプションを使用すると設定できます。以下のプロパティーがサポートされます。

watchedNamespace

Topic Operator によって **KafkaTopics** が監視される OpenShift namespace。デフォルトは、Kafka クラスターがデプロイされた namespace です。

reconciliationIntervalSeconds

定期的な調整 (reconciliation) の間隔 (秒単位)。デフォルト は **120** です。

zookeeperSessionTimeoutSeconds

ZooKeeper セッションのタイムアウト (秒単位)。デフォルトは **18** です。

topicMetadataMaxAttempts

Kafka からトピックメタデータの取得を試行する回数。各試行の間隔は、指数バックオフとして定義されます。パーティションまたはレプリカの数によって、トピックの作成に時間がかかる可能性がある場合は、この値を大きくすることを検討してください。デフォルトは **6** です。

image

image プロパティーを使用すると、使用されるコンテナーイメージを設定できます。カスタムコンテナーイメージの設定に関する詳細は、「**image**」 を参照してください。

resources

resources プロパティーを使用すると、Topic Operator に割り当てられるリソースの量を設定できます。リソースの要求と制限の設定に関する詳細は、「**resources**」 を参照してください。

ログ

logging プロパティーは、Topic Operator のロギングを設定します。詳細は 「**ログ**」 を参照してください。

Topic Operator の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

2.1.2.3. User Operator 設定プロパティー

User Operator デプロイメントは、**userOperator** オブジェクト内で追加オプションを使用すると設定できます。以下のプロパティーがサポートされます。

watchedNamespace

User Operator によって **KafkaUsers** が監視される OpenShift namespace。デフォルトは、Kafka クラスターがデプロイされた namespace です。

reconciliationIntervalSeconds

定期的な調整 (reconciliation) の間隔 (秒単位)。デフォルト は **120** です。

image

image プロパティーを使用すると、使用されるコンテナーイメージを設定できます。カスタムコンテナーイメージの設定に関する詳細は、「**image**」を参照してください。

resources

resources プロパティーを使用すると、User Operator に割り当てられるリソースの量を設定できます。リソースの要求と制限の設定に関する詳細は、「**resources**」を参照してください。

ログ

logging プロパティーは、User Operator のロギングを設定します。詳細は「ログ」を参照してください。

secretPrefix

secretPrefix プロパティーは、KafkaUser リソースから作成されたすべての Secret の名前にプレフィックスを追加します。例えば、**STRIMZI_SECRET_PREFIX=kafka-**とすると、すべてのシークレット名の前に**kafka-**を付けることができます。そのため、**my-user** という名前の KafkaUser は、**kafka-my-user** という名前の Secret を作成します。

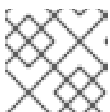
User Operator の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-user-namespace
      reconciliationIntervalSeconds: 60
    # ...
```

2.1.3. Kafka および ZooKeeper のストレージタイプ

Kafka および ZooKeeper はステートフルなアプリケーションであるため、データをディスクに格納する必要があります。AMQ Streams では、3 つのタイプのストレージがサポートされます。

- 一時ストレージ
- 永続ストレージ
- JBOD ストレージ



注記

JBOD ストレージは Kafka でサポートされ、ZooKeeper ではサポートされていません。

Kafka リソースを設定する場合、Kafka ブローカーおよび対応する ZooKeeper ノードによって使用されるストレージのタイプを指定できます。以下のリソースの **storage** プロパティーを使用して、ストレージタイプを設定します。

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

ストレージタイプは **type** フィールドで設定されます。

ストレージ設定プロパティーの詳細は、スキーマ参照を参照してください。

- **EphemeralStorage** スキーマ参照
- **PersistentClaimStorage** スキーマ参照
- **JbodStorage** schema reference

警告

Kafka クラスターをデプロイした後に、ストレージタイプを変更することはできません。

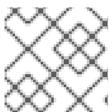
2.1.3.1. データストレージに関する留意事項

効率的なデータストレージインフラストラクチャーは、AMQ Streams のパフォーマンスを最適化するために不可欠です。

ブロックストレージが必要です。NFS などのファイルストレージは、Kafka では機能しません。

ブロックストレージには、以下のいずれかのオプションを選択します。

- Amazon Elastic Block Store (EBS) などのクラウドベースのブロックストレージソリューション。
- ローカルの永続ボリューム。
- **ファイバーチャネル** や **iSCSI** などのプロトコルがアクセスする SAN (ストレージネットワークエリア) ボリューム。



注記

AMQ Streams には OpenShift の raw ブロックボリュームは必要ありません。

2.1.3.1.1. ファイルシステム

XFS ファイルシステムを使用するようにストレージシステムを設定することが推奨されます。AMQ Streams は **ext4** ファイルシステムとも互換性がありますが、最適化するには追加の設定が必要になることがあります。

2.1.3.1.2. Apache Kafka および ZooKeeper ストレージ

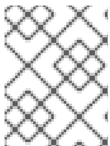
Apache Kafka と ZooKeeper には別々のディスクを使用します。

3 つのタイプのデータストレージがサポートされます。

- 一時データストレージ (開発用のみで推奨されます)
- 永続データストレージ
- JBOD (Just a Bunch of Disks、Kafka のみに適しています)

詳細は「Kafka および ZooKeeper ストレージ」を参照してください。

ソリッドステートドライブ (SSD) は必須ではありませんが、複数のトピックに対してデータが非同期的に送受信される大規模なクラスターで Kafka のパフォーマンスを向上させることができます。SSD は、高速で低レイテンシーのデータアクセスが必要な ZooKeeper で特に有効です。

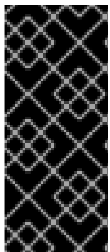


注記

Kafka と ZooKeeper の両方にデータレプリケーションが組み込まれているため、複製されたストレージのプロビジョニングは必要ありません。

2.1.3.2. 一時ストレージ

一時ストレージは **emptyDir** ボリュームを使用してデータを保存します。一時ストレージを使用するには、**type** フィールドを **ephemeral** に設定します。



重要

emptyDir ボリュームは永続的ではなく、保存されたデータは Pod の再起動時に失われます。新規 Pod の起動後に、クラスターの他のノードからすべてのデータを復元する必要があります。一時ストレージは、単一ノードの ZooKeeper クラスターやレプリケーション係数が 1 の Kafka トピックでの使用には適していません。この設定により、データが失われます。

一時ストレージの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: ephemeral
    # ...
  zookeeper:
    # ...
    storage:
      type: ephemeral
    # ...
```

2.1.3.2.1. ログディレクトリー

一時ボリュームは、以下のパスにマウントされるログディレクトリーとして Kafka ブローカーによって使用されます。

```
/var/lib/kafka/data/kafka-logIDX
```

–

IDX は、Kafka ブローカーポッドインデックスです。たとえば、**/var/lib/kafka/data/kafka-log0** のようになります。

2.1.3.3. 永続ストレージ

永続ストレージは Persistent Volume Claim (永続ボリューム要求、PVC) を使用して、データを保存するための永続ボリュームをプロビジョニングします。永続ボリューム要求を使用すると、ボリュームのプロビジョニングを行う ストレージクラス に応じて、さまざまなタイプのボリュームをプロビジョニングできます。永続ボリューム要求と使用できるデータタイプには、多くのタイプの SAN ストレージやローカル永続ボリューム などがあります。

永続ストレージを使用するには、**type** を **persistent-claim** に設定する必要があります。永続ストレージでは、追加の設定オプションがサポートされます。

id (任意)

ストレージ ID 番号。このオプションは、JBOD ストレージ宣言で定義されるストレージボリュームには必須です。デフォルトは **0** です。

size (必須)

永続ボリューム要求のサイズを定義します (例: 1000Gi)。

class (任意)

動的ボリュームプロビジョニングに使用する OpenShift の ストレージクラス。

selector (任意)

使用する特定の永続ボリュームを選択できます。このようなボリュームを選択するラベルを表す key:value ペアが含まれます。

deleteClaim (任意)

クラスターのアンデプロイ時に永続ボリューム要求を削除する必要があるかどうかを指定するブール値。デフォルトは **false** です。

警告

既存の AMQ Streams クラスターで永続ボリュームのサイズを増やすことは、永続ボリュームのサイズ変更をサポートする OpenShift バージョンでのみサポートされます。サイズを変更する永続ボリュームには、ボリューム拡張をサポートするストレージクラスを使用する必要があります。ボリューム拡張をサポートしないその他のバージョンの OpenShift およびストレージクラスでは、クラスターをデプロイする前に必要なストレージサイズを決定する必要があります。既存の永続ボリュームのサイズを縮小することはできません。

size が 1000Gi の永続ストレージ設定の例 (抜粋)

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

以下の例は、ストレージクラスの使用例を示しています。

特定のストレージクラスを指定する永続ストレージ設定の例 (抜粋)

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

最後に、**selector** を使用して特定のラベルが付いた永続ボリュームを選択し、SSD などの必要な機能を提供できます。

セレクターを指定する永続ストレージ設定の例 (抜粋)

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

2.1.3.3.1. ストレージクラスのオーバーライド

デフォルトのストレージクラスを使用する代わりに、1 つ以上の Kafka ブローカー または ZooKeeper ノードに異なるストレージクラスを指定できます。これは、ストレージクラスが、異なるアベイラビリティーゾーンやデータセンターに制限されている場合などに便利です。この場合、**overrides** フィールドを使用できます。

以下の例では、デフォルトのストレージクラスの名前は **my-storage-class** になります。

ストレージクラスのオーバーライドを使用した AMQ Streams クラスターの例

```
apiVersion: kafka.strimzi.io/v1beta2
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
  # ...
  zookeeper:
    replicas: 3
    storage:
      deleteClaim: true
      size: 100Gi
      type: persistent-claim
      class: my-storage-class
      overrides:
        - broker: 0
          class: my-storage-class-zone-1a
        - broker: 1
          class: my-storage-class-zone-1b
        - broker: 2
          class: my-storage-class-zone-1c
  # ...
```

overrides プロパティーが設定され、ボリュームによって以下のストレージクラスが使用されます。

- ZooKeeper ノード 0 の永続ボリュームでは **my-storage-class-zone-1a** が使用されます。
- ZooKeeper ノード 1 の永続ボリュームでは **my-storage-class-zone-1b** が使用されます。
- ZooKeeepr ノード 2 の永続ボリュームでは **my-storage-class-zone-1c** が使用されます。
- Kafka ブローカー 0 の永続ボリュームでは **my-storage-class-zone-1a** が使用されます。
- Kafka ブローカー 1 の永続ボリュームでは **my-storage-class-zone-1b** が使用されます。
- Kafka ブローカー 2 の永続ボリュームでは **my-storage-class-zone-1c** が使用されます。

現在、**overrides** プロパティーは、ストレージクラスの設定をオーバーライドするためのみに使用されます。他のストレージ設定フィールドのオーバーライドは現在サポートされていません。ストレージ設定の他のフィールドは現在サポートされていません。

2.1.3.3.2. Persistent Volume Claim (永続ボリューム要求、PVC) の命名

永続ストレージが使用されると、以下の名前で Persistent Volume Claim (永続ボリューム要求、PVC) が作成されます。

data-cluster-name-kafka-idx
: Kafka ブローカー Pod **idx** のデータを保存するために使用されるボリュームの永続ボリューム要求です。

data-cluster-name-zookeeper-idx
: ZooKeeper ノード Pod **idx** のデータを保存するために使用されるボリュームの永続ボリューム要求です。

2.1.3.3.3. ログディレクトリー

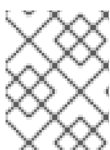
永続ボリュームは、以下のパスにマウントされるログディレクトリーとして Kafka ブローカーによって使用されます。

```
/var/lib/kafka/data/kafka-logIDX
```

IDX は、Kafka ブローカーポッドインデックスです。たとえば、**/var/lib/kafka/data/kafka-log0** のようになります。

2.1.3.4. 永続ボリュームのサイズ変更

既存の AMQ Streams クラスターによって使用される永続ボリュームのサイズを増やすことで、ストレージ容量を増やすことができます。永続ボリュームのサイズ変更は、JBOD ストレージ設定で 1 つまたは複数の永続ボリュームが使用されるクラスターでサポートされます。



注記

永続ボリュームのサイズを拡張することはできますが、縮小することはできません。永続ボリュームのサイズ縮小は、現在 OpenShift ではサポートされていません。

前提条件

- ボリュームのサイズ変更をサポートする OpenShift クラスター。
- Cluster Operator が稼働している必要があります。
- ボリューム拡張をサポートするストレージクラスを使用して作成された永続ボリュームを使用する Kafka クラスター。

手順

1. **Kafka** リソースで、Kafka クラスター、ZooKeeper クラスター、またはその両方に割り当てられた永続ボリュームのサイズを増やします。
   - Kafka クラスターに割り当てられたボリュームサイズを増やすには、**spec.kafka.storage** プロパティーを編集します。
   - ZooKeeper クラスターに割り当てたボリュームサイズを増やすには、**spec.zookeeper.storage** プロパティーを編集します。
     たとえば、ボリュームサイズを **1000Gi** から **2000Gi** に増やすには、以下のように編集します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: persistent-claim
      size: 2000Gi
      class: my-storage-class
    # ...
  zookeeper:
    # ...
```

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

OpenShift では、Cluster Operator からの要求に応じて、選択された永続ボリュームの容量が増やされます。サイズ変更が完了すると、サイズ変更された永続ボリュームを使用するすべての Pod が Cluster Operator によって再起動されます。これは自動的に行われます。

その他のリソース

- OpenShift での永続ボリュームのサイズ変更に関する詳細は、「Resizing Persistent Volumes using Kubernetes」を参照してください。

2.1.3.5. JBOD ストレージの概要

AMQ Streams で、複数のディスクやボリュームのデータストレージ設定である JBOD を使用するように設定できます。JBOD は、Kafka ブローカーのデータストレージを増やす方法の 1 つです。また、パフォーマンスを向上することもできます。

JBOD 設定は 1 つ以上のボリュームによって記述され、各ボリュームは 一時 または 永続 ボリュームのいずれかになります。JBOD ボリューム宣言のルールおよび制約は、一時および永続ストレージのルールおよび制約と同じです。たとえば、永続ストレージのボリュームをプロビジョニング後に縮小することはできません。また、type=ephemeral の場合は **sizeLimit** の値を変更することはできません。

2.1.3.5.1. JBOD の設定

AMQ Streams で JBOD を使用するには、ストレージ **type** を **jbod** に設定する必要があります。**volumes** プロパティーを使用すると、JBOD ストレージアレイまたは設定を構成するディスクを記述できます。以下は、JBOD 設定例の抜粋になります。

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

id は、JBOD ボリュームの作成後に変更することはできません。

ユーザーは JBOD 設定に対してボリュームを追加または削除できます。

2.1.3.5.2. JBOD および 永続ボリューム要求 (PVC)

永続ストレージを使用して JBOD ボリュームを宣言する場合、永続ボリューム要求 (Persistent Volume Claim、PVC) の命名スキームは以下のようになります。

data-id-cluster-name-kafka-idx

id は、Kafka ブローカー Pod **idx** のデータを保存するために使用されるボリュームの ID に置き換えます。

2.1.3.5.3. ログディレクトリー

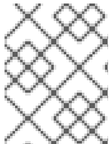
JBOD ボリュームは、以下のパスにマウントされるログディレクトリーとして Kafka ブローカーによって使用されます。

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

id は、Kafka ブローカー Pod **idx** のデータを保存するために使用されるボリュームの ID に置き換えます。たとえば、**/var/lib/kafka/data-0/kafka-log0** のようになります。

2.1.3.6. JBOD ストレージへのボリュームの追加

この手順では、JBOD ストレージを使用するように設定されている Kafka クラスターにボリュームを追加する方法を説明します。この手順は、他のストレージタイプを使用するように設定されている Kafka クラスターには適用できません。



注記

以前使用され、削除された **id** の下に新規ボリュームを追加する場合、以前使用された **PersistentVolumeClaims** が必ず削除されているよう確認する必要があります。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。
- JBOD ストレージのある Kafka クラスター。

手順

1. **Kafka** リソースの **spec.kafka.storage.volumes** プロパティーを編集します。新しいボリュームを **volumes** アレイに追加します。たとえば、id が **2** の新しいボリュームを追加します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 1
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
      - id: 2
```

```
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
    # ...
  zookeeper:
    # ...
```

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

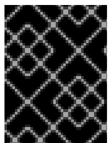
3. 新しいトピックを作成するか、既存のパーティションを新しいディスクに再度割り当てします。

その他のリソース

トピックの再割り当てに関する詳細は 「パーティション再割り当てツール」 を参照してください。

2.1.3.7. JBOD ストレージからのボリュームの削除

この手順では、JBOD ストレージを使用するように設定されている Kafka クラスターからボリュームを削除する方法を説明します。この手順は、他のストレージタイプを使用するように設定されている Kafka クラスターには適用できません。JBOD ストレージには、常に 1 つのボリュームが含まれている必要があります。



重要

データの損失を避けるには、ボリュームを削除する前にすべてのパーティションを移動する必要があります。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。
- 複数のボリュームがある JBOD ストレージのある Kafka クラスター。

手順

1. 削除するディスクからすべてのパーティションを再度割り当てます。削除するディスクに割り当てられたままになっているパーティションのデータは削除される可能性があります。
2. **Kafka** リソースの **spec.kafka.storage.volumes** プロパティーを編集します。**volumes** アレイから 1 つまたは複数のボリュームを削除します。たとえば、ID が **1** と **2** のボリュームを削除します。

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

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

3. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

関連情報

トピックの再割り当てに関する詳細は 「パーティション再割り当てツール」 を参照してください。

2.1.4. クラスターのスケーリング

ブローカーを追加または削除して、Kafka クラスターをスケーリングします。クラスターにトピックがすでに定義されている場合は、パーティションを再割り当てする必要があります。

kafka-reassign-partitions.sh ツールを使用して、パーティションを再割り当てします。このツールは、再割り当てするトピックを指定する再割り当て JSON ファイルを使用します。

特定のパーティションを移動させたい場合は、再割り当てJSONファイルを生成するか、手動でファイルを作成します。

2.1.4.1. ブローカーのスケーリング設定

Kafka.spec.kafka.replicas の設定を行い、ブローカーの数を追加または削減します。

ブローカーの追加
トピックのスループットを向上させる主な方法は、そのトピックのパーティション数を増やすことです。これにより、追加のパーティションによってクラスター内の異なるブローカー間でトピックの負荷が共有されます。ただし、各ブローカーが特定のリソース (通常は I/O) によって制約される場合、パーティションを増やしてもスループットは向上しません。代わりに、ブローカーをクラスターに追加する必要があります。

追加のブローカーをクラスターに追加する場合、Kafka ではパーティションは自動的に割り当てられません。どのパーティションを既存のブローカーから新しいブローカーに再割り当てするかを決めなければなりません。

すべてのブローカーの間でパーティションが再分配されると、各ブローカーのリソース使用量が減少します。

ブローカーの削除
AMQ Streams では **StatefulSets** を使用してブローカー Pod を管理されるため、**あらゆる** Pod を削除できるわけではありません。クラスターから削除できるのは、番号が最も大きい 1 つまたは複数の Pod のみです。たとえば、12 個のブローカーがあるクラスターでは、Pod の名前は **cluster-name-kafka-0** から **cluster-name-kafka-11** になります。1 つのブローカー分をスケールダウンする場合、**cluster-name-kafka-11** が削除されます。

クラスターからブローカーを削除する前に、そのブローカーにパーティションが割り当てられていないことを確認します。また、使用が停止されたブローカーの各パーティションを引き継ぐ、残りのブロー

カーを決める必要もあります。ブローカーに割り当てられたパーティションがなければ、クラスターを安全にスケールダウンできます。

2.1.4.2. パーティション再割り当てツール

現在、Topic Operator はレプリカを別のブローカーに再割当てすることをサポートしないため、ブローカー Pod に直接接続してレプリカをブローカーに再割り当てする必要があります。

ブローカーポッド内では、**kafka-reassign-partitions.sh** ツールを使用して、パーティションを異なるブローカーに再割り当てすることができます。

これには、以下の 3 つのモードがあります。

--generate

トピックとブローカーのセットを取り、**再割り当て JSON ファイル** を生成します。これにより、トピックのパーティションがブローカーに割り当てられます。これはトピック全体で動作するため、一部のトピックのパーティションを再度割り当てする場合は使用できません。

--execute

再割り当て JSON ファイル を取り、クラスターのパーティションおよびブローカーに適用します。その結果、パーティションを取得したブローカーは、パーティションリーダーのフォロワーになります。新規ブローカーが ISR (In-Sync Replica、同期レプリカ) に参加できたら、古いブローカーはフォロワーではなくなり、そのレプリカが削除されます。

--verify

--execute ステップと同じ**再割り当て JSON ファイル**を使用して、**--verify** は、ファイル内のすべてのパーティションが目的のブローカーに移動されたかどうかをチェックします。再割り当てが完了すると、**--verify** は効果のあるトラフィックのスロットル(**--throttle**)も削除します。スロットルを削除しないと、再割り当てが完了した後もクラスターは影響を受け続けます。

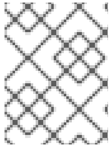
クラスターでは、1 度に 1 つの再割当てのみを実行でき、実行中の再割当てをキャンセルすることはできません。再割り当てをキャンセルする必要がある場合は、割り当てが完了するのを待ってから別の再割り当てを実行し、最初の再割り当ての結果を元に戻します。**kafka-reassign-partitions.sh** によって、元に戻すための再割り当て JSON が出力の一部として生成されます。大規模な再割り当ては、進行中の再割り当てを停止する必要がある場合に備えて、複数の小さな再割り当てに分割するようにしてください。

パーティション再割り当ての JSON ファイル
再割り当て JSON ファイル には特定の構造があります。

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

ここで **<PartitionObjects>** は、以下のようなコンマ区切りのオブジェクトリストになります。

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



注記

Kafka は **"log_dirs"** プロパティーもサポートしますが、AMQ Streams では使用しないでください。

以下は、トピック **topic-a** のパーティション **4** をブローカー **2**、**4**、**7** に割り当て、トピック **topic-b** のパーティション **2** をブローカー **1**、**5**、**7** に割り当てる再割り当て JSON ファイルの例です。

パーティション再割り当てファイルの例

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

JSON に含まれていないパーティションは変更されません。

JBOD ボリューム間のパーティション再割り当て
Kafka クラスターで JBOD ストレージを使用する場合は、特定のボリュームとログディレクトリー (各ボリュームに単一のログディレクトリーがある) との間でパーティションを再割り当てを選択することができます。パーティションを特定のボリュームに再割り当てするには、再割り当て JSON ファイルで **log_dirs** オプションを **<PartitionObjects>** に追加します。

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

log_dirs オブジェクトに含まれるログディレクトリーの数は、**replicas** オブジェクトで指定されるレプリカ数と同じである必要があります。値は、ログディレクトリーへの絶対パスか、**any** キーワードである必要があります。

ログディレクトリーを指定するパーティション再割り当てファイルの例

```
{
      "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" ]
}
```

パーティション再割り当てスロットル
パーティションの再割り当てには、ブローカーの間で大量のデータを転送する必要があるため、処理が遅くなる可能性があります。クライアントへの悪影響を防ぐため、再割り当て処理をスロットルで調整することができます。**--throttle** パラメーターを **kafka-reassign-partitions.sh** ツールと共に使用して、再割り当てをスロットルします。ブローカー間のパーティションの移動の最大しきい値をバイト単位で指定します。たとえば **--throttle 5000000** は、パーティションを移動する最大しきい値を 50 MBps に設定します。

スロットリングにより、再割り当ての完了に時間がかかる場合があります。

- スロットルが低すぎると、新たに割り当てられたブローカーは公開されるレコードに対応できず、再割り当ては完了しません。
- スロットルが高すぎると、クライアントに影響します。

たとえば、プロデューサーの場合は、確認応答を待つ通常のレイテンシーよりも高い可能性があります。コンシューマーの場合は、ポーリング間のレイテンシーが大きいことが原因でスループットが低下する可能性があります。

2.1.4.3. 再割り当て JSON ファイルの生成

この手順では、再割り当て JSON ファイルを生成する方法を説明します。**kafka-reassign-partitions.sh** ツールと共に再割り当てファイルを使用して、Kafka クラスターのスケーリング後にパーティションを再割り当てします。

この手順では、TLS を使用するセキュアな再割り当てプロセスを説明します。TLS による暗号化および認証を使用する Kafka クラスターが必要です。

前提条件

- Cluster Operator が実行中である。
- 内部 TLS 認証および暗号化で設定された **Kafka** リソースを基にして Kafka クラスターが稼働中である必要があります。

TLS での Kafka 設定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    listeners:
      # ...
      - name: tls
        port: 9093
        type: internal
        tls: true (1)
        authentication:
          type: tls (2)
    # ...
```

(1) 内部リスナーの TLS 暗号化を有効にします。

2. 相互 TLS として指定されるリスナー認証メカニズム。

- 稼働中の Kafka クラスターには、再割り当てするトピックおよびパーティションのセットが含まれます。

my-topicのトピック設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 3
  config:
    retention.ms: 7200000
    segment.bytes: 1073741824
    # ...
```

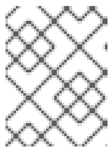
- Kafka ブローカーからトピックを生成および使用するパーミッションを指定する ACL ルールとともに **KafkaUser** を設定している。

my-topic および my-cluster で操作を許可する ACL ルールがある Kafka ユーザー設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication: (1)
    type: tls
  authorization:
    type: simple (2)
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Write
        host: "*"
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Create
        host: "*"
      - resource:
          type: topic
          name: my-topic
```

```
        patternType: literal
      operation: Describe
      host: "*"
    - resource:
        type: cluster
        name: my-cluster
        patternType: literal
      operation: Alter
      host: "*"
    # ...
  # ...
```

1. 相互 **tls** として定義されたユーザー認証メカニズム。
2. ACL ルールのリストを簡易承認と付随します。



注記

トピックへのTLSアクセスには、最低でも **Describe** 操作のパーミッションが必要です。

手順

1. Kafka クラスターの **<cluster_name> -cluster-ca-cert** Secret から、クラスター CA 証明書およびパスワードを抽出します。

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' | base64 -d > ca.p12
```

```
oc get secret <cluster_name>-cluster-ca-cert -o jsonpath='{.data.ca\.password}' | base64 -d > ca.password
```

<cluster_name> を Kafka クラスターの名前に置き換えます。Kafka リソースを使用して **Kafka** をデプロイする場合、クラスター CA 証明書のある Secret が Kafka クラスター名 (**<cluster_name> -cluster-ca-cert**)で作成されます。例: **my-cluster-cluster-ca-cert**

2. AMQ Streams の Kafka イメージを使用して新しい対話型の Pod コンテナーを実行し、稼働中の Kafka ブローカーに接続します。

```
oc run --restart=Never --image=registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1 <interactive_pod_name> -- /bin/sh -c "sleep 3600"
```

<interactive_pod_name> を Pod の名前に置き換えます。

3. クラスター CA 証明書を対話式の pod コンテナーにコピーします。

```
oc cp ca.p12 <interactive_pod_name>:/tmp
```

4. Kafka ブローカーへのアクセス権限を持つ Kafka ユーザーの Secret から、ユーザー CA 証明書およびパスワードを抽出します。

```
oc get secret <kafka_user> -o jsonpath='{.data.user\.p12}' | base64 -d > user.p12
```

```
oc get secret <kafka_user> -o jsonpath='{.data.user\.password}' | base64 -d >
user.password
```

<kafka_user> を Kafka ユーザーの名前に置き換えます。 **KafkaUser** リソースを使用して Kafka ユーザーを作成すると、ユーザー CA 証明書のある Secret が Kafka ユーザー名で作成されます。例： **my-user**

5. ユーザー CA 証明書をインタラクティブな Pod コンテナーにコピーします。

```
oc cp user.p12 <interactive_pod_name>:/tmp
```

CA 証明書を使用すると、対話型の Pod コンテナーが TLS を使用して Kafka ブローカーに接続できます。

6. **config.properties** ファイルを作成し、Kafka クラスターへの認証に使用されるトラストストアおよびキーストアを指定します。
前のステップで展開した証明書とパスワードを使用します。

```
bootstrap.servers=<kafka_cluster_name>-kafka-bootstrap:9093 (1)
security.protocol=SSL (2)
ssl.truststore.location=/tmp/ca.p12 (3)
ssl.truststore.password=<truststore_password> (4)
ssl.keystore.location=/tmp/user.p12 (5)
ssl.keystore.password=<keystore_password> (6)
```

1. Kafka クラスターに接続するためのブートストラップサーバーアドレス。独自の Kafka クラスター名を使用して **<kafka_cluster_name>** を置き換えます。
2. 暗号化に TLS を使用する場合のセキュリティープロトコルオプション。
3. トラストストアの場所には、Kafka クラスターの公開鍵証明書(**ca.p12**)が含まれます。
4. トラストストアにアクセスするためのパスワード(**ca.password**)。
5. キーストアの場所には、Kafka ユーザーの公開鍵証明書(**user.p12**)が含まれます。
6. キーストアにアクセスするためのパスワード(**user.password**)。

7. **config.properties** ファイルを対話式の Pod コンテナーにコピーします。

```
oc cp config.properties <interactive_pod_name>:/tmp/config.properties
```

8. 移動するトピックを指定する **topics.json** という名前の JSON ファイルを準備します。
トピック名をカンマ区切りの一覧として指定します。

topic-a および topic-b のすべてのパーティションを再割り当てする JSON ファイルの例

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

```
    { "topic": "topic-b"}
  ]
}
```

9. **topics.json** ファイルを対話式の Pod コンテナーにコピーします。

```
oc cp topics.json <interactive_pod_name>:/tmp/topics.json
```

10. インタラクティブな Pod コンテナーでシェルプロセスを開始します。

```
oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

<namespace> を Pod が実行されている OpenShift namespace に置き換えます。

11. **kafka-reassign-partitions.sh** コマンドを使用して、再割り当て JSON を生成します。

topic-a と topic-b のすべてのパーティションをブローカー 0、1、2 に移動させるコマンド例

```
bin/kafka-reassign-partitions.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --topics-to-move-json-file /tmp/topics.json \
  --broker-list 0,1,2 \
  --generate
```

関連情報

- Kafka の設定
- Kafka トピックの設定
- Kafka へのユーザーアクセスのセキュア化

2.1.4.4. Kafka クラスターのスケールアップ

再割り当てファイルを使用して Kafka クラスターのブローカーの数を増やします。

再割り当てファイルには、拡大された Kafka クラスターでパーティションをブローカーに再割り当てする方法が記述されます。

この手順では、TLS を使用するセキュアなスケーリングプロセスについて説明します。TLS による暗号化および認証を使用する Kafka クラスターが必要です。

前提条件

- 内部 TLS 認証および暗号化で設定された **Kafka** リソースを基にして Kafka クラスターが稼働中である必要があります。
- **reassignment.json** という名前の再割り当て JSON ファイルを生成している。
- 実行中の Kafka ブローカーに接続されている対話型の Pod コンテナーを実行している。
- Kafka クラスターおよびそのトピックを管理するパーミッションを指定する ACL ルールで設定された **KafkaUser** として接続されている必要があります。

「 再割り当て JSON ファイルの生成 」を参照してください。

手順

1. **kafka.spec.kafka.replicas** 設定オプションを増やして、新しいブローカーを必要なだけ追加します。
2. 新しいブローカー Pod が起動したことを確認します。
3. これを実行していない場合は、インタラクティブな Pod コンテナーを実行して **reassignment.json** という名前の再割り当て JSON ファイルを生成します。
4. **reassignment.json** ファイルをインタラクティブな Pod コンテナーにコピーします。

   ```
   oc cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
   ```

   <interactive_pod_name> を Pod の名前に置き換えます。
5. インタラクティブな Pod コンテナーでシェルプロセスを開始します。

   ```
   oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
   ```

   <namespace> を Pod が実行されている OpenShift namespace に置き換えます。
6. 対話型の Pod コンテナーの **kafka-reassign-partitions.sh** スクリプトを使用して、パーティションの再割り当てを実行します。

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
    <cluster_name>-kafka-bootstrap:9093 \
    --command-config /tmp/config.properties \
    --reassignment-json-file /tmp/reassignment.json \
    --execute
   ```

   <cluster_name> を Kafka クラスターの名前に置き換えます。例： **my-cluster-kafka-bootstrap:9093**

   レプリケーションをスロットルで調整する場合 、**--throttle** オプションにブローカー間のスロットル率（バイト/秒単位）を渡すこともできます。以下は例になります。

   ```
   bin/kafka-reassign-partitions.sh --bootstrap-server
     <cluster_name>-kafka-bootstrap:9093 \
     --command-config /tmp/config.properties \
     --reassignment-json-file /tmp/reassignment.json \
     --throttle 5000000 \
     --execute
   ```

   このコマンドは、2 つの再割り当て JSON オブジェクトを出力します。最初の JSON オブジェクトには、移動されたパーティションの現在の割り当てが記録されます。後で再割り当てを元に戻す必要がある場合に備え、この値をローカルファイル (Pod のファイル以外) に保存します。2 つ目の JSON オブジェクトは、再割り当て JSON ファイルに渡した目的の再割り当てです。

   再割り当て中にスロットルを変更する必要がある場合は、同じコマンドを別のスロットル率で使用することができます。以下は例になります。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

7. ブローカー Pod のいずれかから **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了したことを確認します。これは直前の手順と同じコマンドですが、**--execute** オプションの代わりに **--verify** オプションを使用します。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

   --verify コマンドが移動した各パーティションが正常に完了したことを示すと、再割り当ては終了します。この最終的な **--verify** によって、結果的に再割り当てスロットルも削除されます。

8. 割り当てを元のブローカーに戻すために JSON ファイルを保存した場合は、ここでそのファイルを削除できます。

2.1.4.5. Kafka クラスターのスケールダウン

再割り当てファイルを使用して Kafka クラスターのブローカーの数を減らします。

再割り当てファイルには、パーティションを Kafka クラスターの残りのブローカーに再割り当てする方法が記述されている必要があります。番号が最も多い Pod のブローカーが最初に削除されます。

この手順では、TLS を使用するセキュアなスケーリングプロセスについて説明します。TLS による暗号化および認証を使用する Kafka クラスターが必要です。

前提条件

- 内部 TLS 認証および暗号化で設定された **Kafka** リソースを基にして Kafka クラスターが稼働中である必要があります。
- **reassignment.json** という名前の再割り当て JSON ファイルを生成している。
- 実行中の Kafka ブローカーに接続されている対話型の Pod コンテナーを実行している。
- Kafka クラスターおよびそのトピックを管理するパーミッションを指定する ACL ルールで設定された **KafkaUser** として接続されている必要があります。

「再割り当て JSON ファイルの生成」を参照してください。

手順

1. これを実行していない場合は、インタラクティブな Pod コンテナーを実行して **reassignment.json** という名前の再割り当て JSON ファイルを生成します。
2. **reassignment.json** ファイルをインタラクティブな Pod コンテナーにコピーします。

```
oc cp reassignment.json <interactive_pod_name>:/tmp/reassignment.json
```

<interactive_pod_name> を Pod の名前に置き換えます。

3. インタラクティブな Pod コンテナーでシェルプロセスを開始します。

```
oc exec -n <namespace> -ti <interactive_pod_name> /bin/bash
```

<namespace> を Pod が実行されている OpenShift namespace に置き換えます。

4. 対話型の Pod コンテナーの **kafka-reassign-partitions.sh** スクリプトを使用して、パーティションの再割り当てを実行します。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
 <cluster_name>-kafka-bootstrap:9093 \
 --command-config /tmp/config.properties \
 --reassignment-json-file /tmp/reassignment.json \
 --execute
```

<cluster_name> を Kafka クラスターの名前に置き換えます。例： **my-cluster-kafka-bootstrap:9093**

レプリケーションをスロットルで調整する場合 、**--throttle** オプションにブローカー間のスロットル率（バイト/秒単位）を渡すこともできます。以下は例になります。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 5000000 \
  --execute
```

このコマンドは、2 つの再割り当て JSON オブジェクトを出力します。最初の JSON オブジェクトには、移動されたパーティションの現在の割り当てが記録されます。後で再割り当てを元に戻す必要がある場合に備え、この値をローカルファイル (Pod のファイル以外) に保存します。2 つ目の JSON オブジェクトは、再割り当て JSON ファイルに渡した目的の再割り当てです。

再割り当て中にスロットルを変更する必要がある場合は、同じコマンドを別のスロットル率で使用することができます。以下は例になります。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --throttle 10000000 \
  --execute
```

5. ブローカー Pod のいずれかから **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了したことを確認します。これは直前の手順と同じコマンドですが、**--execute** オプションの代わりに **--verify** オプションを使用します。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
  <cluster_name>-kafka-bootstrap:9093 \
  --command-config /tmp/config.properties \
  --reassignment-json-file /tmp/reassignment.json \
  --verify
```

-

--verify コマンドが移動した各パーティションが正常に完了したことを示すと、再割り当ては終了します。この最終的な **--verify** によって、結果的に再割り当てスロットルも削除されます。

6. 割り当てを元のブローカーに戻すために JSON ファイルを保存した場合は、ここでそのファイルを削除できます。

7. すべてのパーティションの再割り当てが終了すると、削除されるブローカーはクラスター内のいずれのパーティションにも対応しないはずです。これは、ブローカーのデータログディレクトリーにライブパーティションのログが含まれていないことを確認すると検証できます。ブローカーのログディレクトリーに、拡張された正規表現 **[a-zA-Z0-9.-]+\.[a-z0-9]+-delete$** と一致しないディレクトリーが含まれる場合、ブローカーにはライブパーティションがあるため、停止してはなりません。
これを確認するには、以下のコマンドを実行します。

```
oc exec my-cluster-kafka-0 -c kafka -it -- \
  /bin/bash -c \
  "ls -l /var/lib/kafka/kafka-log_<n>_ | grep -E '^d' | grep -vE '[a-zA-Z0-9.-]+\.[a-z0-9]+-
delete$'"
```

ここで 、 n は削除される Pod の数に置き換えます。

上記のコマンドによって出力が生成される場合、ブローカーにはライブパーティションがあります。この場合、再割り当てが終了しなかったか、再割り当て JSON ファイルが正しくありません。

8. ブローカーにライブパーティションがないことを確認したら、Kafka リソースの **Kafka.spec.kafka.replicas** プロパティーを編集してブローカーの数を減らすことができます。

2.1.5. ローリングアップデートのメンテナンス時間枠

メンテナンス時間枠によって、Kafka および ZooKeeper クラスターの特定のローリングアップデートが便利な時間に開始されるようにスケジュールできます。

2.1.5.1. メンテナンス時間枠の概要

ほとんどの場合、Cluster Operator は対応する **Kafka** リソースの変更に対応するために Kafka または ZooKeeper クラスターのみを更新します。これにより、**Kafka** リソースの変更を適用するタイミングを計画し、Kafka クライアントアプリケーションへの影響を最小限に抑えることができます。

ただし、**Kafka** リソースの変更がなくても Kafka および ZooKeeper クラスターの更新が発生することがあります。たとえば、Cluster Operator によって管理される CA (認証局) 証明書が期限切れ直前である場合にローリング再起動の実行が必要になります。

サービスの **可用性** は Pod のローリング再起動による影響を受けないはずですが (ブローカーおよびトピックの設定が適切である場合)、Kafka クライアントアプリケーションの **パフォーマンス** は影響を受ける可能性があります。メンテナンス時間枠によって、Kafka および ZooKeeper クラスターのこのような自発的なアップデートが便利な時間に開始されるようにスケジュールできます。メンテナンス時間枠がクラスターに設定されていない場合は、予測できない高負荷が発生する期間など、不便な時間にこのような自発的なローリングアップデートが行われる可能性があります。

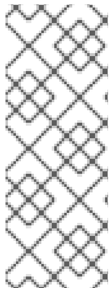
2.1.5.2. メンテナンス時間枠の定義

Kafka.spec.maintenanceTimeWindows プロパティーに文字列の配列を入力して、メンテナンス時間枠を設定します。各文字列は、UTC (協定世界時、Coordinated Universal Time) であると解釈される cron 式 です。UTC は実用的にはグリニッジ標準時と同じです。

以下の例では、日、月、火、水、および木曜日の午前 0 時に開始し、午前 1 時 59 分 (UTC) に終わる、単一のメンテナンス時間枠が設定されます。

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

実際には、必要な CA 証明書の更新が設定されたメンテナンス時間枠内で完了できるように、**Kafka** リソースの **Kafka.spec.clusterCa.renewalDays** および **Kafka.spec.clientsCa.renewalDays** プロパティーとともにメンテナンス期間を設定する必要があります。



注記

AMQ Streams では、指定の期間にしたがってメンテナンス操作を正確にスケジュールしません。その代わりに、調整ごとにメンテナンス期間が現在「オープン」であるかどうかを確認します。これは、特定の時間枠内でのメンテナンス操作の開始が、最大で Cluster Operator の調整が行われる間隔の長さ分、遅れる可能性があることを意味します。したがって、メンテナンス時間枠は最低でもその間隔の長さにする必要があります。

関連情報

- Cluster Operator 設定についての詳細は、「Cluster Operator の設定」 を参照してください。

2.1.5.3. メンテナンス時間枠の設定

サポートされるプロセスによってトリガーされるローリングアップデートのメンテナンス時間枠を設定できます。

前提条件

- OpenShift クラスターが必要です。
- Cluster Operator が稼働している必要があります。

手順

1. **Kafka** リソースの **maintenanceTimeWindows** プロパティー を追加または編集します。たとえば、0800 から 1059 までと、1400 から 1559 までのメンテナンスを可能にするには、以下のように **maintenanceTimeWindows** を設定します。

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

```
maintenanceTimeWindows:
  - "* * 8-10 * * ?"
  - "* * 14-15 * * ?"
```

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

関連情報

ローリングアップデートの実行:

- 「StatefulSet アノテーションを使用したローリングアップデートの実行」
- 「Pod アノテーションを使用したローリングアップデートの実行」

2.1.6. ターミナルからの ZooKeeper への接続

ほとんどの Kafka CLI ツールは Kafka に直接接続できます。したがって、通常の状況では ZooKeeper に接続する必要はありません。ZooKeeper サービスは暗号化および認証でセキュア化され、AMQ Streams の一部でない外部アプリケーションでの使用は想定されていません。

ただし、ZooKeeper への接続を必要とする Kafka CLI ツールを使用する場合は、ZooKeeper コンテナー内でターミナルを使用し、ZooKeeper アドレスとして **localhost:12181** に接続できます。

前提条件

- OpenShift クラスターが利用できる必要があります。
- Kafka クラスターが稼働している必要があります。
- Cluster Operator が稼働している必要があります。

手順

1. OpenShift コンソールを使用してターミナルを開くか、CLI から **exec** コマンドを実行します。
以下は例になります。

```
oc exec -ti my-cluster-zookeeper-0 -- bin/kafka-topics.sh --list --zookeeper localhost:12181
```

必ず **localhost:12181** を使用してください。

ZooKeeper に対して Kafka コマンドを実行できるようになりました。

2.1.7. Kafka ノードの手動による削除

この手順では、OpenShift アノテーションを使用して既存の Kafka ノードを削除する方法を説明します。Kafka ノードの削除するには、Kafka ブローカーが稼働している **Pod** と、関連する **PersistentVolumeClaim** の両方を削除します (クラスターが永続ストレージでデプロイされた場合)。削除後、**Pod** と関連する **PersistentVolumeClaim** は自動的に再作成されます。

警告

PersistentVolumeClaim を削除すると、データが永久に失われる可能性があります。以下の手順は、ストレージで問題が発生した場合にのみ実行してください。

前提条件

以下を実行する方法については、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』を参照してください。

- Cluster Operator
- Kafka クラスター

手順

1. 削除する **Pod** の名前を見つけます。
   Kafka ブローカー Pod の名前は **<cluster-name>**-kafka-**<index>** です。& **lt;index>** はゼロで始まり、レプリカの合計数から 1 を引いた数で終了します。例： **my-cluster-kafka-0**
2. OpenShift で **Pod** リソースにアノテーションを付けます。
   oc annotate を使用します。

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

3. 基盤となる永続ボリューム要求 (Persistent Volume Claim) でアノテーションが付けられた Pod が削除され、再作成されるときに、次の調整の実行を待ちます。

2.1.8. ZooKeeper ノードの手動による削除

この手順では、OpenShift アノテーションを使用して既存の ZooKeeper ノードを削除する方法を説明します。ZooKeeper ノードの削除するには、ZooKeeper が稼働している **Pod** と、関連する **PersistentVolumeClaim** の両方を削除します (クラスターが永続ストレージでデプロイされた場合)。削除後、**Pod** と関連する **PersistentVolumeClaim** は自動的に再作成されます。

警告

PersistentVolumeClaim を削除すると、データが永久に失われる可能性があります。以下の手順は、ストレージで問題が発生した場合にのみ実行してください。

前提条件

以下を実行する方法については、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』を参照してください。

- Cluster Operator

- Kafka クラスター

手順

1. 削除する **Pod** の名前を見つけます。
   ZooKeeper Pod の名前は **<cluster-name>**-zookeeper-**<index>** です。ここで、& **lt;index>** はゼロで始まり、レプリカの合計数から 1 を引いた数で終了します。たとえば、my **-cluster-zookeeper-0** です。

2. OpenShift で **Pod** リソースにアノテーションを付けます。
   oc annotate を使用します。

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

3. 基盤となる永続ボリューム要求 (Persistent Volume Claim) でアノテーションが付けられた Pod が削除され、再作成されるときに、次の調整の実行を待ちます。

2.1.9. Kafka クラスターリソースのリスト

以下のリソースは、OpenShift クラスターの Cluster Operator によって作成されます。

共有リソース

cluster-name-cluster-ca
: クラスター通信の暗号化に使用されるクラスター CA プライベートキーのあるシークレット。

cluster-name-cluster-ca-cert
: クラスター CA 公開鍵のあるシークレット。このキーは、Kafka ブローカーのアイデンティティーの検証に使用できます。

cluster-name-clients-ca
: ユーザー証明書に署名するために使用されるクライアント CA 秘密鍵のあるシークレット。

cluster-name-clients-ca-cert
: クライアント CA 公開鍵のあるシークレット。このキーは、Kafka ユーザーのアイデンティティーの検証に使用できます。

cluster-name-cluster-operator-certs
: Kafka および ZooKeeper と通信するための Cluster Operator キーのあるシークレット。

ZooKeeper ノード

cluster-name-zookeeper
: ZooKeeper ノード Pod の管理を担当する StatefulSet。

cluster-name-zookeeper-idx
: Zookeeper StatefulSet によって作成された Pod。

cluster-name-zookeeper-nodes
: DNS が ZooKeeper Pod の IP アドレスを直接解決するのに必要なヘッドレスサービス。

cluster-name-zookeeper-client
: Kafka ブローカーがクライアントとして ZooKeeper ノードに接続するために使用するサービス。

cluster-name-zookeeper-config

ZooKeeper 補助設定が含まれ、ZooKeeper ノード Pod によってボリュームとしてマウントされる ConfigMap。

cluster-name-zookeeper-nodes

ZooKeeper ノードキーがあるシークレット。

cluster-name-zookeeper

Zookeeper ノードで使用されるサービスアカウント。

cluster-name-zookeeper

ZooKeeper ノードに設定された Pod の Disruption Budget。

cluster-name-network-policy-zookeeper

ZooKeeper サービスへのアクセスを管理するネットワークポリシー。

data-cluster-name-zookeeper-idx

ZooKeeper ノード Pod **idx** のデータを保存するために使用されるボリュームの永続ボリューム要求です。このリソースは、データを保存するために永続ボリュームのプロビジョニングに永続ストレージが選択された場合のみ作成されます。

Kafka ブローカー

cluster-name-kafka

Kafka ブローカー Pod の管理を担当する StatefulSet。

cluster-name-kafka-idx

Kafka StatefulSet によって作成された Pod。

cluster-name-kafka-brokers

DNS が Kafka ブローカー Pod の IP アドレスを直接解決するのに必要なサービス。

cluster-name-kafka-bootstrap

サービスは、OpenShift クラスター内から接続する Kafka クライアントのブートストラップサーバーとして使用できます。

cluster-name-kafka-external-bootstrap

OpenShift クラスター外部から接続するクライアントのブートストラップサービス。このリソースは、外部リスナーが有効な場合にのみ作成されます。リスナー名が **external** でポートが **9094** の場合、後方互換性のために古いサービス名が使用されます。

cluster-name-kafka-pod-id

トラフィックを OpenShift クラスターの外部から個別の Pod にルーティングするために使用されるサービス。このリソースは、外部リスナーが有効な場合にのみ作成されます。リスナー名が **external** でポートが **9094** の場合、後方互換性のために古いサービス名が使用されます。

cluster-name-kafka-external-bootstrap

OpenShift クラスターの外部から接続するクライアントのブートストラップルート。このリソースは、外部リスナーが有効でタイプが **route** に設定されている場合にのみ作成されます。リスナー名が **external** でポートが **9094** の場合、後方互換性のために古いルート名が使用されます。

cluster-name-kafka-pod-id

OpenShift クラスターの外部から個別の Pod へのトラフィックに対するルート。このリソースは、外部リスナーが有効でタイプが **route** に設定されている場合にのみ作成されます。リスナー名が **external** でポートが **9094** の場合、後方互換性のために古いルート名が使用されます。

cluster-name-kafka-listener-name-bootstrap

OpenShift クラスター外部から接続するクライアントのブートストラップサービス。このリソースは、外部リスナーが有効な場合にのみ作成されます。新しいサービス名はその他すべての外部リスナーに使用されます。

cluster-name-kafka-listener-name-pod-id

トラフィックを OpenShift クラスターの外部から個別の Pod にルーティングするために使用されるサービス。このリソースは、外部リスナーが有効な場合にのみ作成されます。新しいサービス名はその他すべての外部リスナーに使用されます。

cluster-name-kafka-listener-name-bootstrap

OpenShift クラスターの外部から接続するクライアントのブートストラップルート。このリソースは、外部リスナーが有効でタイプが **route** に設定されている場合にのみ作成されます。新しいルート名はその他すべての外部リスナーに使用されます。

cluster-name-kafka-listener-name-pod-id

OpenShift クラスターの外部から個別の Pod へのトラフィックに対するルート。このリソースは、外部リスナーが有効でタイプが **route** に設定されている場合にのみ作成されます。新しいルート名はその他すべての外部リスナーに使用されます。

cluster-name-kafka-config

Kafka 補助設定が含まれ、Kafka ブローカー Pod によってボリュームとしてマウントされる ConfigMap。

cluster-name-kafka-brokers

Kafka ブローカーキーのあるシークレット。

cluster-name-kafka

Kafka ブローカーによって使用されるサービスアカウント。

cluster-name-kafka

Kafka ブローカーに設定された Pod の Disruption Budget。

cluster-name-network-policy-kafka

Kafka サービスへのアクセスを管理するネットワークポリシー。

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

Kafka ブローカーによって使用されるクラスターロールバインディング。

cluster-name-jmx

Kafka ブローカーポートのセキュア化に使用される JMX ユーザー名およびパスワードのあるシークレット。このリソースは、Kafka で JMX が有効になっている場合にのみ作成されます。

data-cluster-name-kafka-idx

Kafka ブローカー Pod **idx** のデータを保存するために使用されるボリュームの永続ボリューム要求です。このリソースは、データを保存するために永続ボリュームのプロビジョニングに永続ストレージが選択された場合のみ作成されます。

data-id-cluster-name-kafka-idx

Kafka ブローカー Pod **idx** のデータを保存するために使用されるボリューム **id** の永続ボリューム要求です。このリソースは、永続ボリュームをプロビジョニングしてデータを保存するときに、JBOD ボリュームに永続ストレージが選択された場合のみ作成されます。

Entitiy Operator

これらのリソースは、Cluster Operator を使用して Entity Operator がデプロイされる場合にのみ作成されます。

cluster-name-entity-operator

Topic および User Operator とのデプロイメント。

cluster-name-entity-operator-random-string

Entity Operator デプロイメントによって作成された Pod。

cluster-name-entity-topic-operator-config

Topic Operator の補助設定のある ConfigMap。

cluster-name-entity-user-operator-config

User Operator の補助設定のある ConfigMap。

cluster-name-entity-operator-certs

Kafka および ZooKeeper と通信するための Entity Operator キーのあるシークレット。

cluster-name-entity-operator

Entity Operator によって使用されるサービスアカウント。

strimzi-cluster-name-entity-topic-operator

Entity Topic Operator によって使用されるロールバインディング。

strimzi-cluster-name-entity-user-operator

Entity User Operator によって使用されるロールバインディング。

Kafka Exporter

これらのリソースは、Cluster Operator を使用して Kafka Exporter がデプロイされる場合にのみ作成されます。

cluster-name-kafka-exporter

Kafka Exporter でのデプロイメント。

cluster-name-kafka-exporter-random-string

Kafka Exporter デプロイメントによって作成された Pod。

cluster-name-kafka-exporter

コンシューマーラグメトリクスの収集に使用されるサービス。

cluster-name-kafka-exporter

Kafka Exporter によって使用されるサービスアカウント。

Cruise Control

これらのリソースは、Cluster Operator を使用して Cruise Control がデプロイされた場合のみ作成されます。

cluster-name-cruise-control

Cruise Control でのデプロイメント。

cluster-name-cruise-control-random-string

Cruise Control デプロイメントによって作成された Pod。

cluster-name-cruise-control-config

Cruise Control の補助設定が含まれ、Cruise Control Pod によってボリュームとしてマウントされる ConfigMap。

cluster-name-cruise-control-certs

Kafka および ZooKeeper と通信するための Cruise Control キーのあるシークレット。

cluster-name-cruise-control

Cruise Control との通信に使用されるサービス。

cluster-name-cruise-control

Cruise Control によって使用されるサービスアカウント。

cluster-name-network-policy-cruise-control

Cruise Control サービスへのアクセスを管理するネットワークポリシー。

2.2. KAFKA CONNECT クラスターの設定

このセクションでは、AMQ StreamsクラスターでKafka Connectのデプロイメントを構成する方法について説明します。

Kafka Connect は、コネクタープラグインを使用して Kafka ブローカーと他のシステムの間でデータをストリーミングする統合ツールです。Kafka Connect は、Kafka と、データベースなどの外部データソースまたはターゲットと統合するためのフレームワークを提供し、コネクターを使用してデータをインポートまたはエクスポートします。コネクターは、必要な接続設定を提供するプラグインです。**KafkaConnect** リソースの完全なスキーマは 「**KafkaConnect** スキーマ参照」 に記載されています。

コネクタープラグインのデプロイに関する詳細は、「コネクタープラグイン による Kafka Connect の拡張」を参照してください。

2.2.1. Kafka Connect の設定

Kafka Connect を使用して、Kafka クラスターへの外部データ接続を設定します。**KafkaConnect** リソースのプロパティーを使用して、Kafka Connect デプロイメントを設定します。

Kafka Connector の設定

KafkaConnect リソースを使用すると、Kafka Connect のコネクターインスタンスを OpenShift ネイティブに作成および管理できます。

Kafka Connect 設定では、**strimzi.io/use-connector-resources** アノテーションを追加して、Kafka Connect クラスターの KafkaConnectors を有効にします。また、**build** 構成を追加して、データ接続に必要なコネクタープラグインを備えたコンテナイメージを AMQ Streams が自動的にビルドするようにすることもできます。Kafka Connectコネクタの外部設定は、**externalConfiguration** プロパティで指定します。

コネクターを管理するには、Kafka Connect REST API を使用するか、KafkaConnector カスタムリソースを使用します。KafkaConnector リソースは、リンク先の Kafka Connect クラスターと同じ namespace にデプロイする必要があります。これらの方法を使用してコネクターを作成、再設定、または削除する方法は、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』の「 コネクターの作成および管理 」を参照してください。

コネクター設定は、HTTP リクエストの一部として Kafka Connect に渡され、Kafka 自体に保存されます。ConfigMap およびシークレットは、設定やデータの保存に使用される標準的な OpenShift リソースです。ConfigMap およびシークレットを使用してコネクターの特定の要素を設定できます。その後、HTTP REST コマンドで設定値を参照できます。これにより、必要な場合は設定が分離され、よりセキュアになります。この方法は、ユーザー名、パスワード、証明書などの機密性の高いデータに適用されます。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

以下を実行する方法については、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』を参照してください。

- Cluster Operator
- Kafka クラスター

手順

1. **KafkaConnect** リソースの **spec** プロパティーを編集します。
   設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect (1)
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" (2)
spec:
  replicas: 3 (3)
  authentication: (4)
    type: tls
    certificateAndKey:
      certificate: source.crt
      key: source.key
      secretName: my-user-source
  bootstrapServers: my-cluster-kafka-bootstrap:9092 (5)
  tls: (6)
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        certificate: ca.crt
      - secretName: my-cluster-cluster-cert
        certificate: ca2.crt
  config: (7)
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
  build: (8)
    output: (9)
      type: docker
      image: my-registry.io/my-org/my-connect-cluster:latest
      pushSecret: my-registry-credentials
    plugins: (10)
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb54804
5e115e33a9e69b1b2a352bee24df035a0447cb820077af00c03
      - name: camel-telegram
        artifacts:
          - type: tgz
```

```
        url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-
telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
        sha512sum:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e
8020bf4187870699819f54ef5859c7846ee4081507f48873479
  externalConfiguration: (11)
    env:
      - name: AWS_ACCESS_KEY_ID
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsAccessKey
      - name: AWS_SECRET_ACCESS_KEY
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsSecretAccessKey
  resources: (12)
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  logging: (13)
    type: inline
    loggers:
      log4j.rootLogger: "INFO"
  readinessProbe: (14)
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  metricsConfig: (15)
    type: jmxPrometheusExporter
    valueFrom:
      configMapKeyRef:
        name: my-config-map
        key: my-key
  jvmOptions: (16)
    "-Xmx": "1g"
    "-Xms": "1g"
  image: my-org/my-image:latest (17)
  rack:
    topologyKey: topology.kubernetes.io/zone (18)
  template: (19)
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: application
                    operator: In
```

```
          values:
            - postgresql
            - mongodb
        topologyKey: "kubernetes.io/hostname"
  connectContainer: (20)
    env:
      - name: JAEGER_SERVICE_NAME
        value: my-jaeger-service
      - name: JAEGER_AGENT_HOST
        value: jaeger-agent-name
      - name: JAEGER_AGENT_PORT
        value: "6831"
```

1. **KafkaConnect** を使用します。

2. Kafka Connect クラスターの KafkaConnectors を有効にします。

3. レプリカノードの数。

4. OAuth ベアラートークン、SASL ベースの SCRAM-SHA-512 または PLAIN メカニズムを使用し、ここで示された TLS メカニズム を使用する、Kafka Connect クラスターの認証。デフォルトでは、Kafka Connect はプレーンテキスト接続を使用して Kafka ブローカーに接続します。

5. Kafka Connect クラスターに接続するためのブートストラップサーバー。

6. クラスターの TLS 証明書が X.509 形式で保存されるキー名のある TLS による暗号化。複数の証明書が同じシークレットに保存されている場合は、複数回リストできます。

7. **ワーカー** の Kafka Connect 設定 (コネクターではない)。標準の Apache Kafka 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。

8. コネクタープラグインで自動的にコンテナーイメージをビルドするためのビルド設定プロパティー。

9. (必須) 新しいイメージがプッシュされるコンテナーレジストリーの設定。

10. (必須) 新しいコンテナーイメージに追加するコネクタープラグインとそれらのアーティファクトの一覧。各プラグインは、1 つ以上の **artifact** で設定する必要があります。

11. ここで示す環境変数や、ボリュームを使用した Kafka コネクターの外部設定**設定プロバイダープラグインを使用して、**外部ソースから設定値を読み込む こともできます。

12. サポートされているリソース（現在は **cpu** と **memory**）の予約と、消費可能な最大リソースを指定するための制限を要求します。

13. 指定された Kafka loggers and log levels が ConfigMap を介して直接的に (**inline**) または間接的に (**external**) に追加されます。カスタム ConfigMap は、**log4j.properties** または **log4j2.properties** キー下に配置する必要があります。Kafka Connect **log4j.rootLogger** ロガーでは、ログレベルを INFO、ERROR、WARN、TRACE、DEBUG、FATAL または OFF に設定できます。

14. コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するためのヘルスチェック。

15. Prometheus メトリクス。この例では、Prometheus JMX エクスポーターの設定が含まれ

16. Kafka Connect を実行している仮想マシン (VM) のパフォーマンスを最適化するための JVM 設定オプション。

17. 高度な任意設定: 特別な場合のみ推奨されるコンテナーイメージの設定。

18. ラックアウェアネス (Rack awareness) は、異なるラック全体でレプリカを分散するために設定されます。**topologykey** はクラスターノードのラベルと一致する必要があります。

19. テンプレートのカスタマイズ。ここでは、Pod は非アフィニティーでスケジュールされるため、Pod は同じホスト名のノードではスケジュールされません。

20. 環境変数は、Jaeger を使用した分散トレーシングにも設定 されます。

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONNECT-CONFIG-FILE
```

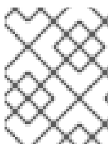
3. Kafka Connect の承認が有効である場合、Kafka Connect ユーザーを設定し、Kafka Connect のコンシューマーグループおよびトピックへのアクセスを有効にします。

2.2.2. 複数インスタンスの Kafka Connect 設定

Kafka Connect のインスタンスを複数実行している場合は、以下の **config** プロパティーのデフォルト設定を変更する必要があります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: connect-cluster (1)
    offset.storage.topic: connect-cluster-offsets (2)
    config.storage.topic: connect-cluster-configs (3)
    status.storage.topic: connect-cluster-status (4)
    # ...
# ...
```

1. Kafka 内の Kafka Connect クラスター ID。
2. コネクターオフセットを保存する Kafka トピック。
3. コネクターおよびタスクステータスの設定を保存する Kafka トピック。
4. コネクターおよびタスクステータスの更新を保存する Kafka トピック。



注記

これら 3 つのトピックの値は、同じ **group.id** を持つすべての Kafka Connect インスタンスで同じする必要があります。

デフォルト設定を変更しないと、同じ Kafka クラスターに接続する各 Kafka Connect インスタンスは同じ値でデプロイされます。その結果、事実上はすべてのインスタンスが結合されてクラスターで実行され、同じトピックが使用されます。

複数の Kafka Connect クラスターが同じトピックの使用を試みると、Kafka Connect は想定どおりに動作せず、エラーが生成されます。

複数の Kafka Connect インスタンスを実行する場合は、インスタンスごとにこれらのプロパティーの値を変更してください。

2.2.3. Kafka Connect のユーザー承認の設定

この手順では、Kafka Connect のユーザーアクセスを承認する方法を説明します。

Kafka でいかなるタイプの承認が使用される場合、Kafka Connect ユーザーは Kafka Connect のコンシューマーグループおよび内部トピックへの読み書きアクセス権限が必要になります。

コンシューマーグループおよび内部トピックのプロパティーは AMQ Streams によって自動設定されますが、**KafkaConnect** リソースの **spec** で明示的に指定することもできます。

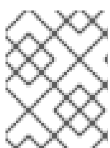
KafkaConnect リソースの設定プロパティーの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster (1)
    offset.storage.topic: my-connect-cluster-offsets (2)
    config.storage.topic: my-connect-cluster-configs (3)
    status.storage.topic: my-connect-cluster-status (4)
    # ...
  # ...
```

1. Kafka 内の Kafka Connect クラスター ID。
2. コネクターオフセットを保存する Kafka トピック。
3. コネクターおよびタスクステータスの設定を保存する Kafka トピック。
4. コネクターおよびタスクステータスの更新を保存する Kafka トピック。

この手順では、**simple** 承認の使用時にアクセス権限が付与される方法を説明します。

簡易承認では、Kafka **AclAuthorizer** プラグインによって処理される ACL ルールを使用し、適切なレベルのアクセス権限が提供されます。**KafkaUser**リソースに簡易認証を使用するように設定する方法については、**AclRule**スキーマリファレンスを参照してください。



注記

複数のインスタンスを実行している場合、コンシューマーグループとトピックのデフォルト値は異なります。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. **KafkaUser** リソースの **authorization** プロパティーを編集し、アクセス権限をユーザーに付与します。
   以下の例では、**literal** の名前の値を使用して Kafka Connect トピックおよびコンシューマーグループにアクセス権限が設定されます。

プロパティー	名前
offset.storage.topic	**connect-cluster-offsets**
status.storage.topic	**connect-cluster-status**
config.storage.topic	**connect-cluster-configs**
group	**connect-cluster**

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      # access to offset.storage.topic
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Write
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Create
        host: "*"
      - resource:
          type: topic
          name: connect-cluster-offsets
          patternType: literal
        operation: Describe
        host: "*"
```

```
- resource:
    type: topic
    name: connect-cluster-offsets
    patternType: literal
  operation: Read
  host: "*"
# access to status.storage.topic
- resource:
    type: topic
    name: connect-cluster-status
    patternType: literal
  operation: Write
  host: "*"
- resource:
    type: topic
    name: connect-cluster-status
    patternType: literal
  operation: Create
  host: "*"
- resource:
    type: topic
    name: connect-cluster-status
    patternType: literal
  operation: Describe
  host: "*"
- resource:
    type: topic
    name: connect-cluster-status
    patternType: literal
  operation: Read
  host: "*"
# access to config.storage.topic
- resource:
    type: topic
    name: connect-cluster-configs
    patternType: literal
  operation: Write
  host: "*"
- resource:
    type: topic
    name: connect-cluster-configs
    patternType: literal
  operation: Create
  host: "*"
- resource:
    type: topic
    name: connect-cluster-configs
    patternType: literal
  operation: Describe
  host: "*"
- resource:
    type: topic
    name: connect-cluster-configs
    patternType: literal
  operation: Read
  host: "*"
```

```
    # consumer group
    - resource:
        type: group
        name: connect-cluster
        patternType: literal
      operation: Read
      host: "*"
```

2. リソースを作成または更新します。

```
oc apply -f KAFKA-USER-CONFIG-FILE
```

2.2.4. Kafka コネクターの再起動の実行

この手順では、OpenShift アノテーションを使用して Kafka コネクターの再起動を手動でトリガーする方法を説明します。

前提条件

- Cluster Operator が稼働している必要があります。

手順

1. 再起動する Kafka コネクターを制御する **KafkaConnector** カスタムリソースの名前を見つけます。

```
oc get KafkaConnector
```

2. コネクターを再起動するには、OpenShift で **KafkaConnector** リソースにアノテーションを付けます。たとえば、**oc annotate** を使用すると以下のようになります。

```
oc annotate KafkaConnector KAFKACONNECTOR-NAME strimzi.io/restart=true
```

3. 次の調整が発生するまで待ちます (デフォルトでは 2 分ごとです)。
アノテーションが調整プロセスで検出されれば、Kafka コネクターは再起動されます。Kafka Connect が再起動リクエストを受け入れると、アノテーションは **KafkaConnector** カスタムリソースから削除されます。

2.2.5. Kafka コネクタータスクの再起動の実行

この手順では、OpenShift アノテーションを使用して Kafka コネクタータスクの再起動を手動でトリガーする方法を説明します。

前提条件

- Cluster Operator が稼働している必要があります。

手順

1. 再起動する Kafka コネクタータスクを制御する **KafkaConnector** カスタムリソースの名前を見つけます。

```
oc get KafkaConnector
```

–

2. **KafkaConnector** カスタムリソースから再起動するタスクの ID を検索します。タスク ID は 0 から始まる負の値ではない整数です。

```
oc describe KafkaConnector KAFKACONNECTOR-NAME
```

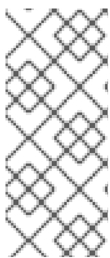
3. コネクタータスクを再起動するには、OpenShift で **KafkaConnector** リソースにアノテーションを付けます。たとえば、**oc annotate** を使用してタスク 0 を再起動します。

```
oc annotate KafkaConnector KAFKACONNECTOR-NAME strimzi.io/restart-task=0
```

4. 次の調整が発生するまで待ちます (デフォルトでは 2 分ごとです)。
   アノテーションが調整プロセスで検出されれば、Kafka コネクタータスクは再起動されます。Kafka Connect が再起動リクエストを受け入れると、アノテーションは **KafkaConnector** カスタムリソースから削除されます。

2.2.6. Kafka Connect API の公開

KafkaConnector リソースを使用してコネクターを管理する代わりに、Kafka Connect REST API を使用します。Kafka Connect REST API は、**<connect_cluster_name>-connect-api:8083** で実行しているサービスとして利用できます。**<connect_cluster_name>** は Kafka Connect クラスターの名前です。サービスは、Kafka Connect インスタンスの作成時に作成されます。



注記

strimzi.io/use-connector-resources アノテーションは KafkaConnectors を有効にします。アノテーションを **KafkaConnect** リソース設定に適用した場合、そのアノテーションを削除して Kafka Connect API を使用する必要があります。それ以外の場合は、Kafka Connect REST API を使用して直接手動で変更した変更は Cluster Operator によって元に戻されます。

コネクター設定を JSON オブジェクトとして追加できます。

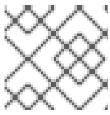
コネクター設定を追加するための curl 要求の例

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"}
}'
```

API には OpenShift クラスター内でのみアクセスできます。OpenShift クラスター外で実行しているアプリケーションに Kafka Connect API にアクセスできるようにするには、以下の機能のいずれかを作成して手動で公開できます。

- **LoadBalancer** または **NodePort** タイプのサービス

- **Ingress** リソース
- OpenShift ルート



注記

接続は安全ではないため、外部アクセスが推奨されています。

サービスを作成する場合には、**<connect_cluster_name>-connect-api** サービスの **セレクター** からラベルを使用して、サービスがトラフィックをルーティングする Pod を設定します。

サービスのセレクター設定

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

1. OpenShift クラスターの Kafka Connect カスタムリソースの名前。
2. Cluster Operator によって作成される Kafka Connect デプロイメントの名前。

また、外部クライアントからの HTTP 要求を許可する **NetworkPolicy** を作成する必要があります。

Kafka Connect API への要求を許可する NetworkPolicy の例

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-custom-connect-network-policy
spec:
  ingress:
  - from:
    - podSelector: (1)
        matchLabels:
          app: my-connector-manager
    ports:
    - port: 8083
      protocol: TCP
  podSelector:
    matchLabels:
      strimzi.io/cluster: my-connect-cluster
      strimzi.io/kind: KafkaConnect
      strimzi.io/name: my-connect-cluster-connect
  policyTypes:
  - Ingress
```

1. API への接続が許可される Pod のラベル。

クラスター外でコネクター設定を追加するには、curl コマンドで API を公開するリソースの URL を使用します。

関連情報

- REST API でサポートされる操作は、Apache Kafka のドキュメント を参照してください。

2.2.7. Kafka Connect クラスターリソースの一覧

以下のリソースは、OpenShift クラスターの Cluster Operator によって作成されます。

connect-cluster-name-connect

Kafka Connect ワーカーノード Pod の作成を担当するデプロイメント。

connect-cluster-name-connect-api

Kafka Connect クラスターを管理するために REST インターフェースを公開するサービス。

connect-cluster-name-config

Kafka Connect 補助設定が含まれ、Kafka ブローカー Pod によってボリュームとしてマウントされる ConfigMap。

connect-cluster-name-connect

Kafka Connect ワーカーノードに設定された Pod の Disruption Budget。

2.2.8. 変更データキャプチャーのための Debezium との統合

Red Hat Debezium は分散型の変更データキャプチャー (change data capture) プラットフォームです。データベースの行レベルの変更をキャプチャーして、変更イベントレコードを作成し、Kafka トピックへレコードをストリーミングします。Debezium は Apache Kafka に構築されます。AMQ Streams で Debezium をデプロイおよび統合できます。AMQ Streams のデプロイ後に、Kafka Connect で Debezium をコネクター設定としてデプロイします。Debezium は変更イベントレコードを OpenShift 上の AMQ Streams に渡します。アプリケーションは **変更イベントストリーム** を読み取りでき、変更イベントが発生した順にアクセスできます。

Debezium には、以下を含む複数の用途があります。

- データレプリケーション。
- キャッシュの更新およびインデックスの検索。
- モノリシックアプリケーションの簡素化。
- データ統合。
- ストリーミングクエリーの有効化。

データベースの変更をキャプチャーするには、Debezium データベースコネクターで Kafka Connect をデプロイします。**KafkaConnector** リソースを設定し、コネクターインスタンスを定義します。

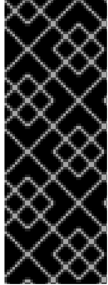
AMQ Streams で Debezium をデプロイするための詳細は、「製品ドキュメント」を参照してください。Debezium のドキュメントの 1 つ が『**Getting Started with Debezium**』で、このガイドはデータベース更新の変更イベントレコードの表示に必要なサービスおよびコネクターの設定方法を説明します。

2.3. KAFKA MIRRORMAKER クラスターの設定

本章では、Kafka クラスター間でデータを複製するために AMQ Streams クラスターで Kafka MirrorMaker デプロイメントを設定する方法を説明します。

AMQ Streams では、MirrorMaker または MirrorMaker 2.0 を使用できます。MirrorMaker 2.0 は最新バージョンで、Kafka クラスター間でより効率的にデータをミラーリングする方法を提供します。

MirrorMaker を使用している場合は、**KafkaMirrorMaker** リソースを設定します。



重要

Kafka MirrorMaker 1（ドキュメントでは **MirrorMaker** と呼ぶ）は Apache Kafka 3.0.0 で非推奨となり、Apache Kafka 4.0.0 で削除されます。そのため、Kafka MirrorMaker 1 のデプロイに使用される **KafkaMirrorMaker** カスタムリソースも AMQ Streams で非推奨となりました。Apache Kafka 4.0.0 の導入時に **KafkaMirrorMaker** リソースは AMQ Streams から削除されます。代わりに、**IdentityReplicationPolicy** で **KafkaMirrorMaker2** カスタムリソースを使用します。

以下の手順は、リソースの設定方法を示しています。

- Kafka MirrorMaker の設定

KafkaMirrorMaker リソースの完全なスキーマは、「KafkaMirrorMaker schema のスキーマ参照」に記載されています。

2.3.1. Kafka MirrorMaker の設定

KafkaMirrorMaker リソースのプロパティーを使用して、Kafka MirrorMaker デプロイメントを設定します。

TLS または SASL 認証を使用して、プロデューサーおよびコンシューマーのアクセス制御を設定できます。この手順では、コンシューマーおよびプロデューサー側で TLS による暗号化および認証を使用する設定を説明します。

前提条件

- 以下を実行する方法については、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』を参照してください。
  - Cluster Operator
  - Kafka クラスター
- ソースおよびターゲットの Kafka クラスターが使用できる必要があります。

手順

1. **KafkaMirrorMaker** リソースの **spec** プロパティーを編集します。
   設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  replicas: 3 (1)
```

```yaml
consumer:
  bootstrapServers: my-source-cluster-kafka-bootstrap:9092 (2)
  groupId: "my-group" (3)
  numStreams: 2 (4)
  offsetCommitInterval: 120000 (5)
  tls: (6)
    trustedCertificates:
    - secretName: my-source-cluster-ca-cert
      certificate: ca.crt
  authentication: (7)
    type: tls
    certificateAndKey:
      secretName: my-source-secret
      certificate: public.crt
      key: private.key
  config: (8)
    max.poll.records: 100
    receive.buffer.bytes: 32768
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (9)
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    ssl.endpoint.identification.algorithm: HTTPS (10)
producer:
  bootstrapServers: my-target-cluster-kafka-bootstrap:9092
  abortOnSendFailure: false (11)
  tls:
    trustedCertificates:
    - secretName: my-target-cluster-ca-cert
      certificate: ca.crt
  authentication:
    type: tls
    certificateAndKey:
      secretName: my-target-secret
      certificate: public.crt
      key: private.key
  config:
    compression.type: gzip
    batch.size: 8192
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (12)
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    ssl.endpoint.identification.algorithm: HTTPS (13)
include: "my-topic|other-topic" (14)
resources: (15)
  requests:
    cpu: "1"
    memory: 2Gi
  limits:
    cpu: "2"
    memory: 2Gi
logging: (16)
  type: inline
  loggers:
    mirrormaker.root.logger: "INFO"
```

```
readinessProbe: 17
  initialDelaySeconds: 15
  timeoutSeconds: 5
livenessProbe:
  initialDelaySeconds: 15
  timeoutSeconds: 5
metricsConfig: 18
 type: jmxPrometheusExporter
 valueFrom:
   configMapKeyRef:
     name: my-config-map
     key: my-key
jvmOptions: 19
  "-Xmx": "1g"
  "-Xms": "1g"
image: my-org/my-image:latest 20
template: 21
  pod:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: application
                  operator: In
                  values:
                    - postgresql
                    - mongodb
            topologyKey: "kubernetes.io/hostname"
  connectContainer: 22
    env:
      - name: JAEGER_SERVICE_NAME
        value: my-jaeger-service
      - name: JAEGER_AGENT_HOST
        value: jaeger-agent-name
      - name: JAEGER_AGENT_PORT
        value: "6831"
tracing: 23
  type: jaeger
```

1. レプリカノードの数。
2. コンシューマーおよびプロデューサーのブートストラップサーバー。
3. コンシューマーのグループ ID。
4. コンシューマーストリームの数。
5. オフセットの自動コミット間隔 (ミリ秒単位)。
6. コンシューマーまたはプロデューサーの TLS 証明書が X.509 形式で保存される、キー名のある TLS による暗号化。複数の証明書が同じシークレットに保存されている場合は、複数回リストできます。
7.

OAuth ベアラートークン、SASL ベースの SCRAM-SHA-512 または PLAIN メカニズムを使用し、ここで示された TLS メカニズム を使用する、コンシューマーおよびプロデュー

8 コンシューマー および プロデューサー の Kafka 設定オプション。

9 TLS バージョンの特定の **暗号スイート** と実行される外部リスナーの SSL プロパティー。

10 **HTTPS** に設定することで、ホスト名の検証が有効になります。空の文字列を指定すると検証が無効になります。

11 **abortOnSendFailure** プロパティが **true** に設定されている場合、メッセージの送信に失敗した後、Kafka MirrorMaker は終了し、コンテナは再起動します。

12 TLS バージョンの特定の **暗号スイート** と実行される外部リスナーの SSL プロパティー。

13 **HTTPS** に設定することで、ホスト名の検証が有効になります。空の文字列を指定すると検証が無効になります。

14 ソースからターゲット Kafka クラスターにミラーリングされた含まれるトピック。

15 サポートされているリソース（現在は **cpu** と **memory**）の予約と、消費可能な最大リソースを指定するための制限を要求します。

16 指定された loggers and log levels が ConfigMap を介して直接的に (**inline**) または間接的に (**external**) に追加されます。カスタム ConfigMap は、**log4j.properties** または **log4j2.properties** キー下に配置する必要があります。MirrorMaker には **mirrormaker.root.logger** と呼ばれる単一のロガーがあります。ログレベルは INFO、ERROR、WARN、TRACE、DEBUG、FATAL、または OFF に設定できます。

17 コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するためのヘルスチェック。

18 Prometheus メトリクス。この例では、Prometheus JMX エクスポーターの設定が含まれる ConfigMap を参照して有効になります。**metricsConfig.valueFrom.configMapKeyRef.key** 配下に空のファイルが含まれる ConfigMap の参照を使用して、追加設定なしでメトリクスを有効にできます。

19 Kafka MirrorMaker を実行している仮想マシン (VM) のパフォーマンスを最適化するための JVM 設定オプション。

20 高度な任意設定: 特別な場合のみ推奨されるコンテナーイメージの設定。

21 テンプレートのカスタマイズ。ここでは、Pod は非アフィニティーでスケジュールされるため、Pod は同じホスト名のノードではスケジュールされません。

22 環境変数は、Jaeger を使用した分散トレーシングにも設定 されます。

23 Jaeger の分散トレーシングは有効になっています。

警告

abortOnSendFailure プロパティーが **false** に設定されると、プロデューサーはトピックの次のメッセージを送信しようとします。失敗したメッセージは再送されないため、元のメッセージが失われる可能性があります。

2. リソースを作成または更新します。

```
oc apply -f <your-file>
```

2.3.2. Kafka MirrorMaker クラスターリソースの一覧

以下のリソースは、OpenShift クラスターの Cluster Operator によって作成されます。

<mirror-maker-name>-mirror-maker
: Kafka MirrorMaker Pod の作成を担当するデプロイメント。

<mirror-maker-name>-config
: Kafka MirrorMaker の補助設定が含まれ、Kafka ブローカー Pod によってボリュームとしてマウントされる ConfigMap。

<mirror-maker-name>-mirror-maker
: Kafka MirrorMaker ワーカーノードに設定された Pod の Disruption Budget。

2.4. KAFKA MIRRORMAKER 2.0 クラスターの設定

ここでは、AMQ Streams クラスターで Kafka MirrorMaker 2.0 デプロイメントを設定する方法を説明します。

MirrorMaker 2.0 は、データセンター内またはデータセンター全体の 2 台以上の Kafka クラスター間でデータを複製するために使用されます。

クラスター全体のデータレプリケーションでは、以下が必要な状況がサポートされます。

- システム障害時のデータの復旧
- 分析用のデータの集計
- 特定のクラスターへのデータアクセスの制限
- レイテンシーを改善するための特定場所でのデータのプロビジョニング

MirrorMaker 2.0 を使用している場合は、**KafkaMirrorMaker2** リソースを設定します。

MirrorMaker 2.0 では、クラスターの間でデータを複製する全く新しい方法が導入されました。

その結果、リソースの設定は MirrorMaker の以前のバージョンとは異なります。MirrorMaker 2.0 の使用を選択した場合、現在、レガシーサポートがないため、リソースを手作業で新しい形式に変換する必要があります。

MirrorMaker 2.0 によってデータが複製される方法は、以下に説明されています。

- MirrorMaker 2.0 データレプリケーション

以下の手順では、MirrorMaker 2.0 に対してリソースが設定される方法について取り上げます。

- MirrorMaker 2.0 を使用した Kafka クラスター間でのデータの同期

KafkaMirrorMaker2 リソースの完全なスキーマは、「KafkaMirrorMaker2 のスキーマ参照」 に記載されています。

2.4.1. MirrorMaker 2.0 のデータレプリケーション

MirrorMaker 2.0 はソースの Kafka クラスターからメッセージを消費して、ターゲットの Kafka クラスターに書き込みます。

MirrorMaker 2.0 は以下を使用します。

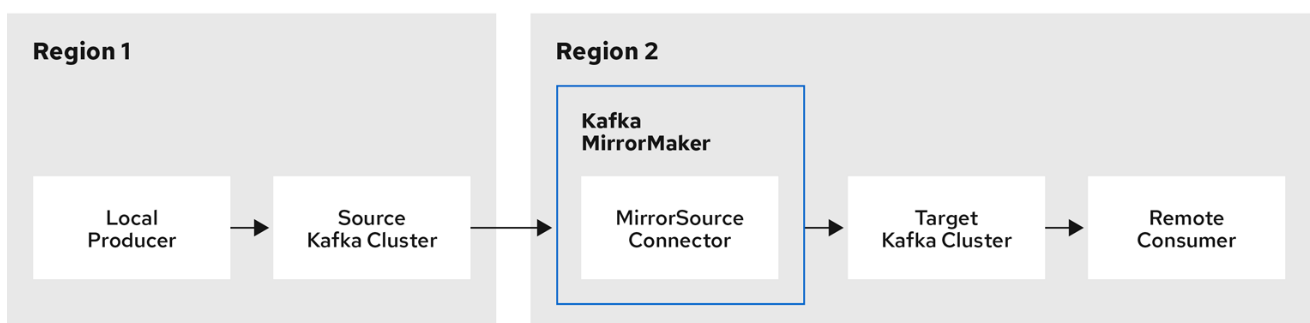
- ソースクラスターからデータを消費するソースクラスターの設定。
- データをターゲットクラスターに出力するターゲットクラスターの設定。

MirrorMaker 2.0 は Kafka Connect フレームワークをベースとし、**コネクター**によってクラスター間のデータ転送が管理されます。MirrorMaker 2.0 の **MirrorSourceConnector** は、ソースクラスターからターゲットクラスターにトピックを複製します。

あるクラスターから別のクラスターにデータを ***ミラーリング*** するプロセスは非同期です。推奨されるパターンは、ソース Kafka クラスターとともにローカルでメッセージが作成され、ターゲットの Kafka クラスターの近くでリモートで消費されることです。

MirrorMaker 2.0 は、複数のソースクラスターで使用できます。

図2.1 2 つのクラスターにおけるレプリケーション



デフォルトでは、ソースクラスターの新規トピックのチェックは 10 分ごとに行われます。頻度は、**refresh.topics.interval.seconds** をソースコネクター設定に追加することで変更できます。ただし、操作の頻度が増えると、全体的なパフォーマンスに影響する可能性があります。

2.4.2. クラスターの設定

active/passive または active/active クラスター設定で MirrorMaker 2.0 を使用できます。

- active/active 設定では、両方のクラスターがアクティブで、同じデータを同時に提供します。これは、地理的に異なる場所で同じデータをローカルで利用可能にする場合に便利です。

- **active/passive** 設定では、アクティブなクラスターからのデータはパッシブなクラスターで複製され、たとえば、システム障害時のデータ復旧などでスタンバイ状態を維持します。

プロデューサーとコンシューマーがアクティブなクラスターのみに接続することを前提とします。

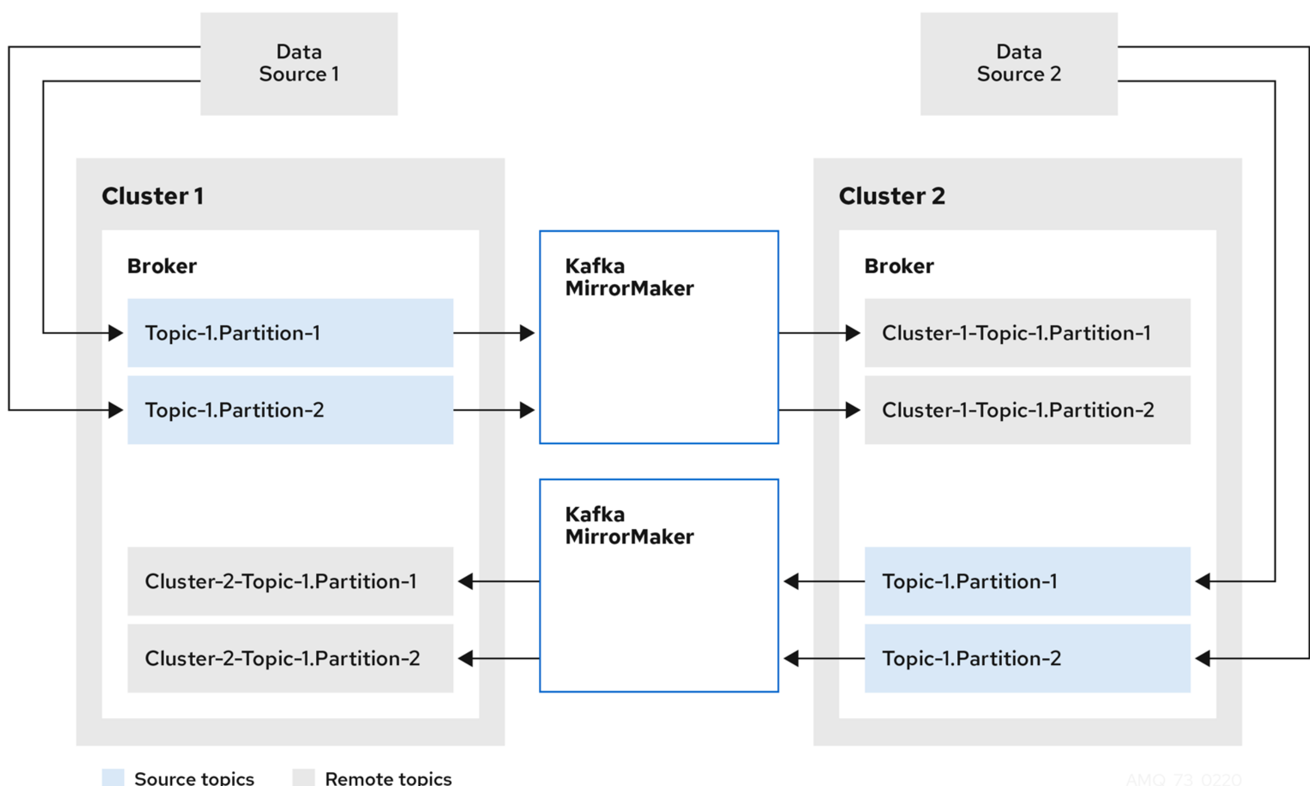
MirrorMaker 2.0 クラスターは、ターゲットの宛先ごとに必要です。

2.4.2.1. 双方向レプリケーション (active/active)

MirrorMaker 2.0 アーキテクチャーでは、**active/active** クラスター設定で双方向レプリケーションがサポートされます。

各クラスターは、 **source** および **remote** トピックの概念を使用して、別のクラスターのデータを複製します。同じトピックが各クラスターに保存されるため、リモートトピックの名前がソースクラスターを表すように自動的に MirrorMaker 2.0 によって変更されます。元のクラスターの名前の先頭には、トピックの名前が追加されます。

図2.2 トピック名の変更



ソースクラスターにフラグを付けると、トピックはそのクラスターに複製されません。

remote トピックを介したレプリケーションの概念は、データの集約が必要なアーキテクチャーの設定に役立ちます。コンシューマーは、同じクラスター内でソースおよびリモートトピックにサブスクライブできます。これに個別の集約クラスターは必要ありません。

2.4.2.2. 一方向レプリケーション (active/passive)

MirrorMaker 2.0 アーキテクチャーでは、**active/passive** クラスター設定で一方向レプリケーションがサポートされます。

active/passiveのクラスター設定を使用してバックアップを作成したり、データを別のクラスターに移行したりできます。この場合、リモートトピックの名前を自動的に変更したくないことがあります。

IdentityReplicationPolicy をソースコネクター設定に追加することで、名前の自動変更をオーバーライドできます。この設定が適用されると、トピックには元の名前が保持されます。

2.4.2.3. トピック設定の同期

トピック設定は、ソースクラスターとターゲットクラスター間で自動的に同期化されます。設定プロパティーを同期化することで、リバランスの必要性が軽減されます。

2.4.2.4. データの整合性

MirrorMaker 2.0 は、ソーストピックを監視し、設定変更をリモートトピックに伝播して、不足しているパーティションを確認および作成します。MirrorMaker 2.0 のみがリモートトピックに書き込みできます。

2.4.2.5. オフセットの追跡

MirrorMaker 2.0 では、**内部トピック**を使用してコンシューマーグループのオフセットを追跡します。

- **オフセット同期** トピックは、複製されたトピックパーティションのソースおよびターゲットオフセットをレコードメタデータからマッピングします。
- **チェックポイント** トピックは、各コンシューマーグループの複製されたトピックパーティションのソースおよびターゲットクラスターで最後にコミットされたオフセットをマッピングします。

チェックポイント トピックのオフセットは、設定によって事前定義された間隔で追跡されます。両方のトピックは、フェイルオーバー時に正しいオフセットの位置からレプリケーションの完全復元を可能にします。

MirrorMaker 2.0 は、**MirrorCheckpointConnector** を使用して、オフセット追跡の **チェックポイント**を生成します。

2.4.2.6. コンシューマーグループオフセットの同期

__consumer_offsets トピックには、各コンシューマーグループのコミットされたオフセットに関する情報が保存されます。オフセットの同期は、ソースクラスターのコンシューマーグループのコンシューマーオフセットをターゲットクラスターのコンシューマーオフセットに定期的に転送します。

オフセットの同期は、特に **active/passive** 設定で便利です。アクティブなクラスターがダウンした場合、コンシューマーアプリケーションはパッシブ (スタンバイ) クラスターに切り替え、最後に転送されたオフセットの位置からピックアップできます。

トピックオフセットの同期を使用するには、sync. **group.offsets.enabled を checkpoint コネクター設定に追加し、プロパティーを true に設定して同期** を有効にします。同期はデフォルトで無効になっています。

ソースコネクターで **IdentityReplicationPolicy** を使用する場合、チェックポイントコネクター設定でも設定する必要があります。これにより、ミラーリングされたコンシューマーオフセットが正しいトピックに適用されます。

コンシューマーオフセットは、ターゲットクラスターでアクティブではないコンシューマーグループに対してのみ同期されます。

同期を有効にすると、ソースクラスターからオフセットの同期が定期的に行われます。この頻度は、**sync.group.offsets.interval.seconds** および **emit.checkpoints.interval.seconds** をチェックポイントコネクター設定に追加することで変更できます。これらのプロパティーは、コンシューマーグルー

プのオフセットが同期される頻度 (秒単位) と、オフセットを追跡するためにチェックポイントが生成される頻度を指定します。両方のプロパティーのデフォルトは 60 秒です。**refresh.groups.interval.seconds** プロパティーを使用して、新規コンシューマーグループのチェック頻度を変更することもできます。デフォルトでは 10 分ごとに実行されます。

同期は時間ベースであるため、コンシューマーによってパッシブクラスターへ切り替えられると、一部のメッセージが重複する可能性があります。

2.4.2.7. 接続性チェック

ハートビート 内部トピックによって、クラスター間の接続性が確認されます。

ハートビート トピックは、ソースクラスターから複製されます。

ターゲットクラスターは、トピックを使用して以下を確認します。

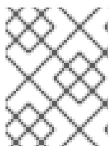
- クラスター間の接続を管理するコネクターが稼働している。
- ソースクラスターが利用可能である。

MirrorMaker 2.0 は **MirrorHeartbeatConnector** を使用して、これらのチェックを実行する **ハートビート** を生成します。

2.4.3. ACL ルールの同期

User Operator を使用して **いない** 場合は、ACL でリモートトピックにアクセスできます。

User Operator なしで if **AclAuthorizer** が使用されている場合、ブローカーへのアクセスを管理する ACL ルールはリモートトピックにも適用されます。ソーストピックを読み取りできるユーザーは、そのリモートトピックを読み取りできます。



注記

OAuth 2.0 での承認は、このようなリモートトピックへのアクセスをサポートしません。

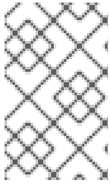
2.4.4. MirrorMaker 2.0 を使用した Kafka クラスター間でのデータの同期

MirrorMaker 2.0 を使用して、設定を介して Kafka クラスター間のデータを同期します。

設定では以下を指定する必要があります。

- 各 Kafka クラスター
- TLS 認証を含む各クラスターの接続情報
- レプリケーションのフローおよび方向
  - クラスター対クラスター
  - トピック対トピック

KafkaMirrorMaker2 リソースのプロパティーを使用して、Kafka MirrorMaker 2.0 のデプロイメントを設定します。



注記

従来のバージョンの MirrorMaker は継続してサポートされます。従来のバージョンに設定したリソースを使用する場合は、MirrorMaker 2.0 でサポートされる形式に更新する必要があります。

MirrorMaker 2.0 によって、レプリケーション係数などのプロパティーのデフォルト設定値が提供されます。デフォルトに変更がない最小設定の例は以下のようになります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.0.0
  connectCluster: "my-cluster-target"
  clusters:
  - alias: "my-cluster-source"
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
  - alias: "my-cluster-target"
    bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector: {}
```

TLS または SASL 認証を使用して、ソースおよびターゲットクラスターのアクセス制御を設定できます。この手順では、ソースおよびターゲットクラスターに対して TLS による暗号化および認証を使用する設定を説明します。

前提条件

- 以下を実行する方法については、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』を参照してください。
    - Cluster Operator
    - Kafka クラスター
- ソースおよびターゲットの Kafka クラスターが使用できる必要があります。

手順

1. **KafkaMirrorMaker2** リソースの **spec** プロパティーを編集します。
   設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.0.0 (1)
  replicas: 3 (2)
  connectCluster: "my-cluster-target" (3)
```

```
clusters: 4
- alias: "my-cluster-source" 5
  authentication: 6
    certificateAndKey:
      certificate: source.crt
      key: source.key
      secretName: my-user-source
    type: tls
  bootstrapServers: my-cluster-source-kafka-bootstrap:9092 7
  tls: 8
    trustedCertificates:
    - certificate: ca.crt
      secretName: my-cluster-source-cluster-ca-cert
- alias: "my-cluster-target" 9
  authentication: 10
    certificateAndKey:
      certificate: target.crt
      key: target.key
      secretName: my-user-target
    type: tls
  bootstrapServers: my-cluster-target-kafka-bootstrap:9092 11
  config: 12
    config.storage.replication.factor: 1
    offset.storage.replication.factor: 1
    status.storage.replication.factor: 1
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 13
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    ssl.endpoint.identification.algorithm: HTTPS 14
  tls: 15
    trustedCertificates:
    - certificate: ca.crt
      secretName: my-cluster-target-cluster-ca-cert
mirrors: 16
- sourceCluster: "my-cluster-source" 17
  targetCluster: "my-cluster-target" 18
  sourceConnector: 19
    tasksMax: 10 20
    config:
      replication.factor: 1 21
      offset-syncs.topic.replication.factor: 1 22
      sync.topic.acls.enabled: "false" 23
      refresh.topics.interval.seconds: 60 24
      replication.policy.separator: "" 25
      replication.policy.class: "io.strimzi.kafka.connect.mirror.IdentityReplicationPolicy" 26
  heartbeatConnector: 27
    config:
      heartbeats.topic.replication.factor: 1 28
  checkpointConnector: 29
    config:
      checkpoints.topic.replication.factor: 1 30
      refresh.groups.interval.seconds: 600 31
```

```
      sync.group.offsets.enabled: true 32
      sync.group.offsets.interval.seconds: 60 33
      emit.checkpoints.interval.seconds: 60 34
      replication.policy.class: "io.strimzi.kafka.connect.mirror.IdentityReplicationPolicy"
    topicsPattern: ".*" 35
    groupsPattern: "group1|group2|group3" 36
  resources: 37
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  logging: 38
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  readinessProbe: 39
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  jvmOptions: 40
    "-Xmx": "1g"
    "-Xms": "1g"
  image: my-org/my-image:latest 41
  template: 42
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: application
                    operator: In
                    values:
                      - postgresql
                      - mongodb
              topologyKey: "kubernetes.io/hostname"
    connectContainer: 43
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger 44
  externalConfiguration: 45
    env:
      - name: AWS_ACCESS_KEY_ID
        valueFrom:
```

```
          secretKeyRef:
            name: aws-creds
            key: awsAccessKey
      - name: AWS_SECRET_ACCESS_KEY
        valueFrom:
          secretKeyRef:
            name: aws-creds
            key: awsSecretAccessKey
```

1. 常に同じになる Kafka Connect と Mirror Maker 2.0 の バージョン。
2. レプリカノードの数。
3. Kafka Connect の Kafka クラスターエイリアス。**ターゲット** Kafka クラスターを指定する必要があります。Kafka クラスターは、その内部トピックのために Kafka Connect によって使用されます。
4. 同期される Kafka クラスターの 指定。
5. ソースの Kafka クラスターの クラスターエイリアス。
6. OAuth ベアラートークン、SASL ベースの SCRAM-SHA-512 または PLAIN メカニズムを使用し、ここで示された TLS メカニズム を使用する、ソースクラスターの認証。
7. ソース Kafka クラスターに接続するための ブートストラップサーバー。
8. ソース Kafka クラスターの TLS 証明書が X.509 形式で保存されるキー名のある TLS による暗号化。複数の証明書が同じシークレットに保存されている場合は、複数回リストできます。
9. ターゲット Kafka クラスターの クラスターエイリアス。
10. ターゲット Kafka クラスターの認証は、ソース Kafka クラスターと同様に設定されます。
11. ターゲット Kafka クラスターに接続するための ブートストラップサーバー。
12. Kafka Connect の設定。標準の Apache Kafka 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。
13. TLS バージョンの特定の **暗号スイート** と実行される外部リスナーの SSL プロパティー。
14. **HTTPS** に設定することで、ホスト名の検証が有効になります。空の文字列を指定すると検証が無効になります。
15. ターゲット Kafka クラスターの TLS による暗号化は、ソース Kafka クラスターと同様に設定されます。
16. MirrorMaker 2.0 コネクター。
17. MirrorMaker 2.0 コネクターによって使用されるソースクラスターの クラスターエイリアス。
18. MirrorMaker 2.0 コネクターによって使用されるターゲットクラスターの クラスターエイリアス。
19. リモートトピックを作成する **MirrorSourceConnector**の設定。デフォルトの設定オプションは **config** によって上書きされます。

20 コネクターによる作成が可能なタスクの最大数。タスクは、データのレプリケーションを処理し、並行して実行されます。インフラストラクチャーが処理のオーバーヘッドをサ

21 ターゲットクラスターで作成されるミラーリングされたトピックのレプリケーション係数。

22 ソースおよびターゲットクラスターのオフセットをマップする **MirrorSourceConnector offset-syncs** 内部トピックのレプリケーション係数。

23 ACL ルールの同期 が有効になっていると、同期されたトピックに ACL が適用されます。デフォルトは **true** です。

24 新規トピックのチェック頻度を変更する任意設定。デフォルトでは 10 分毎にチェックされます。

25 リモートトピック名の変更に使用する区切り文字を定義します。

26 リモートトピック名の自動変更をオーバーライドするポリシーを追加します。その名前の前にソースクラスターの名前を追加する代わりに、トピックが元の名前を保持します。このオプションの設定は、active/passive バックアップおよびデータ移行に役立ちます。トピックオフセットの同期を設定するには、このプロパティーも **checkpointConnector.config** に設定する必要があります。

27 接続チェックを実行する **MirrorHeartbeatConnector** の設定。デフォルトの設定オプションは **config** によって上書きされます。

28 ターゲットクラスターで作成されたハートビートトピックのレプリケーション係数。

29 オフセットを追跡する **MirrorCheckpointConnector**の設定。デフォルトの設定オプションは **config** によって上書きされます。

30 ターゲットクラスターで作成されたチェックポイントトピックのレプリケーション係数。

31 新規コンシューマーグループのチェック頻度を変更する任意設定。デフォルトでは 10 分毎にチェックされます。

32 コンシューマーグループのオフセットを同期する任意設定。これは、active/passive 設定でのリカバリーに便利です。同期はデフォルトでは有効になっていません。

33 コンシューマーグループオフセットの同期が有効な場合は、同期の頻度を調整できます。

34 オフセット追跡のチェック頻度を調整します。オフセット同期の頻度を変更する場合、これらのチェックの頻度も調整する必要がある場合があります。

35 正規表現パターンとして定義されたソースクラスターからのトピックレプリケーション。ここで、すべてのトピックを要求します。

36 正規表現パターンとして定義されたソースクラスターからのコンシューマーグループレプリケーション。ここで、3 つのコンシューマーグループを名前で要求します。コンマ区切りリストを使用できます。

37 サポートされているリソース（現在は **cpu** と **memory**）の予約と、消費可能な最大リソースを指定するための制限を要求します。

38 指定された Kafka loggers and log levels が ConfigMap を介して直接的に (**inline**) または間接的に (**external**) に追加されます。カスタム ConfigMap は、**log4j.properties** または **log4j2.properties** キー下に配置する必要があります。Kafka Connect **log4j.rootLogger**

ロガーでは、ログレベルを INFO、ERROR、WARN、TRACE、DEBUG、FATAL または OFF に設定できます。

39 コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するためのヘルスチェック。

40 Kafka MirrorMaker を実行している仮想マシン (VM) のパフォーマンスを最適化するための JVM 設定オプション。

41 高度な任意設定: 特別な場合のみ推奨されるコンテナーイメージの設定。

42 テンプレートのカスタマイズ。ここでは、Pod は非アフィニティーでスケジュールされるため、Pod は同じホスト名のノードではスケジュールされません。

43 環境変数は、Jaeger を使用した分散トレーシングにも設定 されます。

44 Jaeger の分散トレーシングは有効になっています。

45 環境変数として Kafka MirrorMaker にマウントされた OpenShift Secret の外部設定。**設定プロバイダープラグインを使用して、**外部ソースから設定値を読み込む こともできます。

2. リソースを作成または更新します。

```
oc apply -f MIRRORMAKER-CONFIGURATION-FILE
```

2.4.5. Kafka MirrorMaker 2.0 コネクターの再起動の実行

この手順では、OpenShift アノテーションを使用して Kafka MirrorMaker 2.0 コネクターの再起動を手動でトリガーする方法を説明します。

前提条件

- Cluster Operator が稼働している必要があります。

手順

1. 再起動する Kafka MirrorMaker 2.0 コネクターを制御する **KafkaMirrorMaker2** カスタムリソースの名前を見つけます。

```
oc get KafkaMirrorMaker2
```

2. **KafkaMirrorMaker2** カスタムリソースから再起動される Kafka MirrorMaker 2.0 コネクターの名前を見つけます。

```
oc describe KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME
```

3. コネクターを再起動するには、OpenShift で **KafkaMirrorMaker2** リソースにアノテーションを付けます。この例では、**oc annotate** は **my-source->my-target.MirrorSourceConnector** という名前のコネクターを再起動します。

```
oc annotate KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME "strimzi.io/restart-connector=my-source->my-target.MirrorSourceConnector"
```

4. 次の調整が発生するまで待ちます (デフォルトでは 2 分ごとです)。
アノテーションが調整プロセスで検出されれば、Kafka MirrorMaker 2.0 コネクターは再起動されます。再起動要求が許可されると、アノテーションは **KafkaMirrorMaker2** カスタムリソースから削除されます。

関連情報

- Kafka MirrorMaker 2.0 クラスターの設定

2.4.6. Kafka MirrorMaker 2.0 コネクタータスクの再起動の実行

この手順では、OpenShift アノテーションを使用して Kafka MirrorMaker 2.0 コネクタータスクの再起動を手動でトリガーする方法を説明します。

前提条件

- Cluster Operator が稼働している必要があります。

手順

1. 再起動する Kafka MirrorMaker 2.0 コネクターを制御する **KafkaMirrorMaker2** カスタムリソースの名前を見つけます。

```
oc get KafkaMirrorMaker2
```

2. Kafka MirrorMaker 2.0 コネクターの名前と、**KafkaMirrorMaker2** カスタムリソースから再起動されるタスクの ID を検索します。タスク ID は 0 から始まる負の値ではない整数です。

```
oc describe KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME
```

3. コネクタータスクを再起動するには、OpenShift で **KafkaMirrorMaker2** リソースにアノテーションを付けます。この例では、**oc annotate** は **my-source->my-target.MirrorSourceConnector** という名前のコネクタのタスク 0 を再起動します。

```
oc annotate KafkaMirrorMaker2 KAFKAMIRRORMAKER-2-NAME "strimzi.io/restart-connector-task=my-source->my-target.MirrorSourceConnector:0"
```

4. 次の調整が発生するまで待ちます (デフォルトでは 2 分ごとです)。
アノテーションが調整プロセスで検出されれば、Kafka MirrorMaker 2.0 コネクタータスクは再起動されます。再起動タスクの要求が受け入れられると、**KafkaMirrorMaker2** のカスタムリソースからアノテーションが削除されます。

関連情報

- Kafka MirrorMaker 2.0 クラスターの設定

2.5. KAFKA BRIDGE クラスターの設定

ここでは、AMQ Streams クラスターで Kafka Bridge デプロイメントを設定する方法を説明します。

Kafka Bridge では、HTTP ベースのクライアントと Kafka クラスターを統合するための API が提供されます。

Kafka Bridge を使用している場合は、**KafkaBridge** リソースを設定します。

KafkaBridge リソースの完全なスキーマは 「**KafkaBridge** スキーマ参照」 に記載されています。

2.5.1. Kafka Bridge の設定

Kafka Bridge を使用した Kafka クラスターへの HTTP ベースのリクエスト

KafkaBridge リソースのプロパティーを使用して、Kafka Bridge デプロイメントを設定します。

クライアントのコンシューマーリクエストが異なる Kafka Bridge インスタンスによって処理された場合に発生する問題を防ぐには、アドレスベースのルーティングを利用して、要求が適切な Kafka Bridge インスタンスにルーティングされるようにする必要があります。また、独立した各 Kafka Bridge インスタンスにレプリカが必要です。Kafka Bridge インスタンスには、別のインスタンスと共有されない独自の状態があります。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

以下を実行する方法については、『**OpenShift での AMQ Streams のデプロイおよびアップグレード**』を参照してください。

- Cluster Operator
- Kafka クラスター

手順

1. **KafkaBridge** リソースの **spec** プロパティーを編集します。
   設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  replicas: 3 (1)
  bootstrapServers: my-cluster-kafka-bootstrap:9092 (2)
  tls: (3)
    trustedCertificates:
      - secretName: my-cluster-cluster-cert
        certificate: ca.crt
      - secretName: my-cluster-cluster-cert
        certificate: ca2.crt
  authentication: (4)
    type: tls
    certificateAndKey:
      secretName: my-secret
      certificate: public.crt
      key: private.key
  http: (5)
    port: 8080
```

```
    cors: (6)
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  consumer: (7)
    config:
      auto.offset.reset: earliest
  producer: (8)
    config:
      delivery.timeout.ms: 300000
  resources: (9)
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
      cpu: "2"
      memory: 2Gi
  logging: (10)
    type: inline
    loggers:
      logger.bridge.level: "INFO"
      # enabling DEBUG just for send operation
      logger.send.name: "http.openapi.operation.send"
      logger.send.level: "DEBUG"
  jvmOptions: (11)
    "-Xmx": "1g"
    "-Xms": "1g"
  readinessProbe: (12)
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  image: my-org/my-image:latest (13)
  template: (14)
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: application
                    operator: In
                    values:
                      - postgresql
                      - mongodb
              topologyKey: "kubernetes.io/hostname"
    bridgeContainer: (15)
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
```

1. レプリカノードの数。
2. ターゲット Kafka クラスターに接続するための ブートストラップサーバー。
3. ソース Kafka クラスターの TLS 証明書が X.509 形式で保存されるキー名のある TLS による暗号化。複数の証明書が同じシークレットに保存されている場合は、複数回リストできます。
4. OAuth ベアラートークン、SASL ベースの SCRAM-SHA-512 または PLAIN メカニズムを使用し、ここで示された TLS メカニズム を使用する、Kafka Bridge クラスターの認証。デフォルトでは、Kafka Bridge は認証なしで Kafka ブローカーに接続します。
5. Kafka ブローカーへの HTTP アクセス。
6. 選択されたリソースおよびアクセスメソッドを指定する CORS アクセス。リクエストの追加の HTTP ヘッダーには Kafka クラスターへのアクセスが許可されるオリジンが記述されています。
7. コンシューマー設定 オプション。
8. プロデューサー設定 オプション。
9. サポートされているリソース（現在は **cpu** と **memory**）の予約と、消費可能な最大リソースを指定するための制限を要求します。
10. 指定された Kafka Bridge loggers and log levels が ConfigMap を介して直接的に (**inline**) または間接的に (**external**) に追加されます。カスタム ConfigMap は、**log4j.properties** または **log4j2.properties** キー下に配置する必要があります。Kafka Bridge ロガーでは、ログレベルを INFO、ERROR、WARN、TRACE、DEBUG、FATAL または OFF に設定できます。
11. Kafka Bridge を実行している仮想マシン (VM) のパフォーマンスを最適化するための JVM 設定オプション。
12. コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するためのヘルスチェック。
13. オプション： 特別な 場合のみ 推奨されるコンテナーイメージの設定。
14. テンプレートのカスタマイズ。ここでは、Pod は非アフィニティーでスケジュールされるため、Pod は同じホスト名のノードではスケジュールされません。
15. 環境変数は、Jaeger を使用した分散トレーシングにも設定 されます。

2. リソースを作成または更新します。

```
oc apply -f KAFKA-BRIDGE-CONFIG-FILE
```

2.5.2. Kafka Bridge クラスターリソースのリスト

以下のリソースは、OpenShift クラスターの Cluster Operator によって作成されます。

bridge-cluster-name-bridge

Kafka Bridge ワーカーノード Pod の作成を担当するデプロイメント。

bridge-cluster-name-bridge-service

Kafka Bridge クラスターの REST インターフェースを公開するサービス。

bridge-cluster-name-bridge-config

Kafka Bridge の補助設定が含まれ、Kafka ブローカー Pod によってボリュームとしてマウントされる ConfigMap。

bridge-cluster-name-bridge

Kafka Bridge ワーカーノードに設定された Pod の Disruption Budget。

2.6. OPENSHIFT リソースのカスタマイズ

AMQ Streamsは、**Deployments**、**StatefulSets**、**Pods**、**Service** などの複数の OpenShift リソースを作成し、AMQ Streamsのオペレータが管理します。特定の OpenShift リソースの管理を担当する operator のみがそのリソースを変更できます。operator によって管理される OpenShift リソースを手動で変更しようとすると、operator はその変更を元に戻します。

しかし、operator が管理する OpenShift リソースの変更は、以下のような特定のタスクを実行する場合に役立ちます。

- **Pod** が Istio またはその他のサービスによって処理される方法を制御するカスタムラベルまたはアノテーションの追加
- **Loadbalancer**-type サービスがクラスターによって作成される方法の管理

このような変更は、AMQ Streams カスタムリソースの **template** プロパティーを使用して追加します。**template** プロパティーは以下のリソースでサポートされます。API リファレンスは、カスタマイズ可能フィールドに関する詳細を提供します。

Kafka.spec.kafka

「**KafkaClusterTemplate** スキーマ参照」 を参照

Kafka.spec.zookeeper

「**ZookeeperClusterTemplate** スキーマ参照」 を参照

Kafka.spec.entityOperator

「**EntityOperatorTemplate** スキーマ参照」 を参照してください。

Kafka.spec.kafkaExporter

「**KafkaExporterTemplate** スキーマ参照」 を参照

Kafka.spec.cruiseControl

「**CruiseControlTemplate** スキーマ参照」 を参照

KafkaConnect.spec

「**KafkaConnectTemplate** スキーマ参照」 を参照

KafkaMirrorMaker.spec

「**KafkaMirrorMakerTemplate** スキーマ参照」 を参照

KafkaMirrorMaker2.spec

「**KafkaConnectTemplate** スキーマ参照」 を参照

KafkaBridge.spec

「**KafkaBridgeTemplate** スキーマ参照」 を参照

KafkaUser.spec

「**KafkaUserTemplate**スキーマリファレンス」 を参照

以下の例では、**template** プロパティーを使用して Kafka ブローカーの **StatefulSet** のラベルを変更します。

テンプレートのカスタマイズ例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  labels:
    app: my-cluster
spec:
  kafka:
    # ...
    template:
      statefulset:
        metadata:
          labels:
            mylabel: myvalue
    # ...
```

2.6.1. イメージプルポリシーのカスタマイズ

AMQ Streams では、Cluster Operator によってデプロイされたすべての Pod のコンテナーのイメージプルポリシーをカスタマイズできます。イメージプルポリシーは、Cluster Operator デプロイメントの環境変数 **STRIMZI_IMAGE_PULL_POLICY** を使用して設定されます。**STRIMZI_IMAGE_PULL_POLICY** 環境変数に設定できる値は 3 つあります。

Always

Pod が起動または再起動されるたびにコンテナーイメージがレジストリーからプルされます。

IfNotPresent

以前プルされたことのないコンテナーイメージのみがレジストリーからプルされます。

Never

コンテナーイメージはレジストリーからプルされることはありません。

現在、イメージプルポリシーはすべての Kafka、Kafka Connect、および Kafka MirrorMaker クラスターに対してのみ 1 度にカスタマイズできます。ポリシーを変更すると、すべての Kafka、Kafka Connect、および Kafka MirrorMaker クラスターのローリングアップデートが実行されます。

関連情報

- Cluster Operator の設定に関する詳細は、「Cluster Operator の使用」 を参照してください。
- イメージプルポリシーに関する詳細は、「Disruptions」を参照してください。

2.7. POD スケジューリングの設定

2 つのアプリケーションが同じ OpenShift ノードにスケジュールされた場合、両方のアプリケーションがディスク I/O のように同じリソースを使用し、パフォーマンスに影響する可能性があります。これにより、パフォーマンスが低下する可能性があります。ノードを他の重要なワークロードと共有しないように Kafka Pod をスケジュールする場合、適切なノードを使用したり、Kafka 専用のノードのセットを使用すると、このような問題を適切に回避できます。

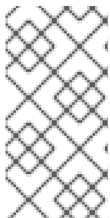
2.7.1. アフィニティー、容認 (Toleration)、およびトポロジー分散制約の指定

アフィニティー、容認 (Toleration)、およびトポロジー分散制約を使用して、kafka リソースの Pod をノードにスケジュールします。アフィニティー、容認(Toleration)、およびトポロジー分散制約は、以下のリソースの **affinity**、**tolerations**、および **topologySpreadConstraint** プロパティーを使用して設定されます。

- **Kafka.spec.kafka.template.pod**
- **Kafka.spec.zookeeper.template.pod**
- **Kafka.spec.entityOperator.template.pod**
- **KafkaConnect.spec.template.pod**
- **KafkaBridge.spec.template.pod**
- **KafkaMirrorMaker.spec.template.pod**
- **KafkaMirrorMaker2.spec.template.pod**

affinity、**tolerations**、および **topologySpreadConstraint** プロパティーの形式は、OpenShift の仕様に準拠します。アフィニティー設定には、さまざまなタイプのアフィニティーを含めることができます。

- Pod のアフィニティーおよび非アフィニティー
- ノードのアフィニティー



注記

OpenShift 1.16 および 1.17 では、**topologySpreadConstraint** のサポートはデフォルトで無効にされています。**topologySpreadConstraint** を使用するには、Kubernetes API サーバーおよびスケジューラーで Even **PodsSpread** 機能ゲートを有効にする必要があります。

関連情報

- Kubernetes ノードおよび Pod のアフィニティーに関するドキュメント
- Kubernetes テイントおよび容認 (Toleration)
- Pod トポロジー分散制約を使用した Pod 配置の制御

2.7.1.1. Pod の非アフィニティーを使用して重要なアプリケーションがノードを共有しないようにする

Pod の非アフィニティーを使用して、重要なアプリケーションが同じディスクにスケジュールされないようにします。Kafka クラスターの実行時に、Pod の非アフィニティーを使用して、Kafka ブローカーがデータベースなどの他のワークロードとノードを共有しないようにすることが推奨されます。

2.7.1.2. ノードのアフィニティーを使用したワークロードの特定ノードへのスケジュール

OpenShift クラスターは、通常多くの異なるタイプのワーカーノードで構成されます。ワークロードが非常に大きい環境の CPU に対して最適化されたものもあれば、メモリー、ストレージ (高速のローカル SSD)、または ネットワークに対して最適化されたものもあります。異なるノードを使用すると、コス

トとパフォーマンスの両面で最適化しやすくなります。最適なパフォーマンスを実現するには、AMQ Streams コンポーネントのスケジューリングで適切なノードを使用できるようにすることが重要です。

OpenShift はノードのアフィニティーを使用してワークロードを特定のノードにスケジュールします。ノードのアフィニティーにより、Pod がスケジュールされるノードにスケジューリングの制約を作成できます。制約はラベルセレクターとして指定されます。**beta.kubernetes.io/instance-type** などの組み込みノードラベルまたはカスタムラベルのいずれかを使用してラベルを指定すると、適切なノードを選択できます。

2.7.1.3. 専用ノードへのノードのアフィニティーと容認 (Toleration) の使用

テイントを使用して専用ノードを作成し、ノードのアフィニティーおよび容認 (Toleration) を設定して専用ノードに Kafka Pod をスケジュールします。

クラスター管理者は、選択した OpenShift ノードをテイントとしてマーク付けできます。テイントのあるノードは、通常のスケジューリングから除外され、通常の Pod はそれらのノードでの実行はスケジュールされません。ノードに設定されたテイントを許容できるサービスのみをスケジュールできます。このようなノードで実行されるその他のサービスは、ログコレクターやソフトウェア定義のネットワークなどのシステムサービスのみです。

専用のノードで Kafka とそのコンポーネントを実行する利点は多くあります。障害の原因になったり、Kafka に必要なリソースを消費するその他のアプリケーションが同じノードで実行されません。これにより、パフォーマンスと安定性が向上します。

2.7.2. それぞれの Kafka ブローカーを別のワーカーノードでスケジュールするための Pod の非アフィニティーの設定

多くの Kafka ブローカーまたは ZooKeeper ノードは、同じ OpenShift ワーカーノードで実行できます。ワーカーノードが失敗すると、それらはすべて同時に利用できなくなります。信頼性を向上させるために、**podAntiAffinity** 設定を使用して、各 Kafka ブローカーまたは ZooKeeper ノードを異なる OpenShift ワーカーノードにスケジュールすることができます。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. クラスターデプロイメントを指定するリソースの **affinity** プロパティーを編集します。ワーカーノードが Kafka ブローカーまたは ZooKeeper ノードで共有されないようにするには、**strimzi.io/name** ラベルを使用します。**topologyKey** を **kubernetes.io/hostname** に設定して、選択した Pod が同じホスト名のノードでスケジュールされないように指定します。これにより、同じワーカーノードを単一の Kafka ブローカーと単一の ZooKeeper ノードで共有できます。以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
```

```
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: strimzi.io/name
                    operator: In
                    values:
                      - CLUSTER-NAME-kafka
              topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: strimzi.io/name
                      operator: In
                      values:
                        - CLUSTER-NAME-zookeeper
                topologyKey: "kubernetes.io/hostname"
    # ...
```

CLUSTER-NAME は、Kafka カスタムリソースの名前です。

2. Kafka ブローカーと ZooKeeper ノードが同じワーカーノードを共有しないようにする場合は、**strimzi.io/cluster** ラベルを使用します。以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: strimzi.io/cluster
                      operator: In
                      values:
                        - CLUSTER-NAME
                topologyKey: "kubernetes.io/hostname"
    # ...
  zookeeper:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
```

```
          - labelSelector:
              matchExpressions:
                - key: strimzi.io/cluster
                  operator: In
                  values:
                    - CLUSTER-NAME
            topologyKey: "kubernetes.io/hostname"
    # ...
```

CLUSTER-NAME は、Kafka カスタムリソースの名前です。

3. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

2.7.3. Kafka コンポーネントでの Pod の非アフィニティーの設定

Pod の非アフィニティー設定は、Kafka ブローカーの安定性とパフォーマンスに役立ちます。**podAntiAffinity** を使用すると、OpenShift は他のワークロードと同じノードで Kafka ブローカーをスケジュールしません。通常、Kafka が他のネットワークと同じワーカーノードで実行されないようにし、データベース、ストレージ、その他のメッセージングプラットフォームなどのストレージを大量に消費するアプリケーションで実行されないようにします。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. クラスターデプロイメントを指定するリソースの **affinity** プロパティーを編集します。ラベルを使用して、同じノードでスケジュールすべきでない Pod を指定します。**topologyKey** を **kubernetes.io/hostname** に設定し、選択した Pod が同じホスト名のノードでスケジュールされてはならないことを指定する必要があります。以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          podAntiAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              - labelSelector:
                  matchExpressions:
                    - key: application
                      operator: In
                      values:
                        - postgresql
                        - mongodb
                topologyKey: "kubernetes.io/hostname"
```

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

2. リソースを作成または更新します。
   oc apply を使用して、これを行うことができます。

```
oc apply -f KAFKA-CONFIG-FILE
```

2.7.4. Kafka コンポーネントでのノードのアフィニティーの設定

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. AMQ Streams コンポーネントをスケジュールする必要のあるノードにラベルを付けます。
   oc label を使用してこれを行うことができます。

```
oc label node NAME-OF-NODE node-type=fast-network
```

   または、既存のラベルによっては再利用が可能です。

2. クラスターデプロイメントを指定するリソースの **affinity** プロパティーを編集します。以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    template:
      pod:
        affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                  - key: node-type
                    operator: In
                    values:
                    - fast-network
    # ...
  zookeeper:
    # ...
```

3. リソースを作成または更新します。
   oc apply を使用して、これを行うことができます。

```
oc apply -f KAFKA-CONFIG-FILE
```

2.7.5. 専用ノードの設定と Pod のスケジューリング

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. 専用ノードとして使用するノードを選択します。
2. これらのノードにスケジュールされているワークロードがないことを確認します。
3. 選択したノードにテイントを設定します。
   oc adm taint を使用してこれを行うことができます。

   ```
   oc adm taint node NAME-OF-NODE dedicated=Kafka:NoSchedule
   ```

4. さらに、選択したノードにラベルも追加します。
   oc label を使用してこれを行うことができます。

   ```
   oc label node NAME-OF-NODE dedicated=Kafka
   ```

5. クラスターデプロイメントを指定するリソースの **affinity** および **tolerations** プロパティーを編集します。
   以下は例になります。

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   spec:
     kafka:
       # ...
       template:
         pod:
           tolerations:
             - key: "dedicated"
               operator: "Equal"
               value: "Kafka"
               effect: "NoSchedule"
           affinity:
             nodeAffinity:
               requiredDuringSchedulingIgnoredDuringExecution:
                 nodeSelectorTerms:
                 - matchExpressions:
                   - key: dedicated
                     operator: In
                     values:
                     - Kafka
       # ...
     zookeeper:
       # ...
   ```

6. リソースを作成または更新します。

oc apply を使用して、これを行うことができます。

```
oc apply -f KAFKA-CONFIG-FILE
```

2.8. ロギングの設定

Kafka コンポーネントおよび AMQ Streams Operator のカスタムリソースでロギングレベルを設定します。ログレベルは、カスタムリソースの **spec.logging** プロパティーに直接指定できます。あるいは、**configMapKeyRef** プロパティを使ってカスタムリソースで参照される ConfigMap でロギングプロパティを定義することもできます。

ConfigMap を使用する利点は、ロギングプロパティーが 1 カ所で維持され、複数のリソースにアクセスできることです。複数のリソースに ConfigMap を再利用することもできます。ConfigMap を使用して AMQ Streams Operator のロガーを指定する場合は、ロギング仕様を追加してフィルターを追加することもできます。

ロギング仕様でロギング **type** を指定します。

- ロギングレベルを直接指定する場合は **inline**
- ConfigMap を参照する場合は **external**

inline ロギングの設定例

```
spec:
  # ...
  logging:
    type: inline
    loggers:
      kafka.root.logger.level: "INFO"
```

external 設定の例

```
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: my-config-map
        key: my-config-map-key
```

ConfigMap の **name** と **key** の値は必須です。**name** や **key** が設定されていない場合は、デフォルトのロギングが使用されます。

2.8.1. Kafka コンポーネントおよび Operator のロギングオプション

特定の Kafka コンポーネントまたは Operator のロギングの設定に関する詳細は、以下のセクションを参照してください。

Kafka コンポーネントのロギング

- Kafka ロギング

- ZooKeeper のロギング
- Kafka Connect および Mirror Maker 2.0 ロギング
- MirrorMaker のロギング
- Kafka Bridge のロギング
- Cruise Control のロギング

Operator のロギング

- Cluster Operator のロギング
- Topic Operator のロギング
- User Operator のロギング

2.8.2. ロギングの ConfigMap の作成

ConfigMap を使用してロギングプロパティーを定義するには、ConfigMap を作成してから、リソースの **spec** にあるロギング定義の一部としてそれを参照します。

ConfigMap には適切なロギング設定が含まれる必要があります。

- Kafka コンポーネント、ZooKeeper、および Kafka Bridge の **log4j.properties**。
- Topic Operator および User Operator の **log4j2.properties**

設定はこれらのプロパティーの配下に配置する必要があります。

この手順では、ConfigMap は Kafka リソースのルートロガーを定義します。

手順

1. ConfigMap を作成します。
   ConfigMap を YAML ファイルとして作成するか、プロパティーファイルから Config Map を作成します。

   Kafka のルートロガー定義が含まれる ConfigMap の例:

   ```
   kind: ConfigMap
   apiVersion: v1
   metadata:
     name: logging-configmap
   data:
     log4j.properties:
       kafka.root.logger.level="INFO"
   ```

   プロパティーファイルを使用している場合は、コマンドラインでファイルを指定します。

   ```
   oc create configmap logging-configmap --from-file=log4j.properties
   ```

   プロパティーファイルではロギング設定が定義されます。

```
# Define the logger
kafka.root.logger.level="INFO"
# ...
```

2. リソースの **spec** にexternal ロギングを定義し、**logging.valueFrom.configMapKeyRef.name** に ConfigMap の名前を、**logging.valueFrom.configMapKeyRef.key** にこの ConfigMap のキーを設定します。

```
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: logging-configmap
        key: log4j.properties
```

3. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

2.8.3. ロギングフィルターの Operator への追加

ConfigMap を使用して AMQ Streams Operator のロギングレベル (log4j2) ロギングレベルを設定する場合、ロギングフィルターを定義して、ログに返される内容も制限できます。

ロギングフィルターは、ロギングメッセージが多数ある場合に役に立ちます。ロガーのログレベルを DEBUG(**rootLogger.level="DEBUG"**)に設定すると仮定します。ロギングフィルターは、このレベルでロガーに対して返されるログ数を減らし、特定のリソースに集中できるようにします。フィルターが設定されると、フィルターに一致するログメッセージのみがログに記録されます。

フィルターは**マーカー**を使用して、ログに含まれる内容を指定します。マーカーの種類、namespace、および名前を指定します。たとえば、Kafka クラスターで障害が発生した場合、種類を **Kafka** に指定してログを分離し、障害が発生しているクラスターの namespace および名前を使用します。

以下の例は、**my-kafka-cluster** という名前の Kafka クラスターのマーカーフィルターを示しています。

基本的なロギングフィルターの設定

```
rootLogger.level="INFO"
appender.console.filter.filter1.type=MarkerFilter (1)
appender.console.filter.filter1.onMatch=ACCEPT (2)
appender.console.filter.filter1.onMismatch=DENY (3)
appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster) (4)
```

1. **MarkerFilter** 型は、フィルターを行うために指定されたマーカーを比較します。
2. **onMatch** プロパティーは、マーカーが一致するとログを受け入れます。
3. **onMismatch** プロパティーは、マーカーが一致しない場合にログを拒否します。
4. フィルター処理に使用されるマーカーの形式は KIND(NAMESPACE/NAME-OF-RESOURCE) です。

フィルターは 1 つまたは複数作成できます。ここでは、ログは 2 つの Kafka クラスターに対してフィルターされます。

複数のロギングフィルターの設定

```
appender.console.filter.filter1.type=MarkerFilter
appender.console.filter.filter1.onMatch=ACCEPT
appender.console.filter.filter1.onMismatch=DENY
appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster-1)
appender.console.filter.filter2.type=MarkerFilter
appender.console.filter.filter2.onMatch=ACCEPT
appender.console.filter.filter2.onMismatch=DENY
appender.console.filter.filter2.marker=Kafka(my-namespace/my-kafka-cluster-2)
```

フィルターの Cluster Operator への追加

フィルターを Cluster Operator に追加するには、そのロギング ConfigMap YAML ファイルを更新します(**install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml**)。

手順

1. **050-ConfigMap-strimzi-cluster-operator.yaml** ファイルを更新して、フィルタープロパティーを ConfigMap に追加します。
   この例では、フィルタープロパティーは **my-kafka-cluster** Kafka クラスターのログのみを返します。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: strimzi-cluster-operator
data:
  log4j2.properties:
    #...
    appender.console.filter.filter1.type=MarkerFilter
    appender.console.filter.filter1.onMatch=ACCEPT
    appender.console.filter.filter1.onMismatch=DENY
    appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster)
```

   または、**ConfigMap** を直接編集することもできます。

```
oc edit configmap strimzi-cluster-operator
```

2. **ConfigMap** を直接編集せずに YAML ファイルを更新する場合は、ConfigMap をデプロイして変更を適用します。

```
oc create -f install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml
```

Topic Operator または User Operator へのフィルターの追加

フィルターを Topic Operator または User Operator に追加するには、ロギング ConfigMap を作成または編集します。

この手順では、ロギング ConfigMap は、Topic Operator のフィルターで作成されます。User Operator に同じアプローチが使用されます。

手順

1. ConfigMap を作成します。
   ConfigMap を YAML ファイルとして作成するか、プロパティーファイルから Config Map を作成します。

   この例では、フィルタープロパティーは **my-topic** トピックに対してのみログを返します。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: logging-configmap
data:
  log4j2.properties:
    rootLogger.level="INFO"
    appender.console.filter.filter1.type=MarkerFilter
    appender.console.filter.filter1.onMatch=ACCEPT
    appender.console.filter.filter1.onMismatch=DENY
    appender.console.filter.filter1.marker=KafkaTopic(my-namespace/my-topic)
```

   プロパティーファイルを使用している場合は、コマンドラインでファイルを指定します。

```
oc create configmap logging-configmap --from-file=log4j2.properties
```

   プロパティーファイルではロギング設定が定義されます。

```
# Define the logger
rootLogger.level="INFO"
# Set the filters
appender.console.filter.filter1.type=MarkerFilter
appender.console.filter.filter1.onMatch=ACCEPT
appender.console.filter.filter1.onMismatch=DENY
appender.console.filter.filter1.marker=KafkaTopic(my-namespace/my-topic)
# ...
```

2. リソースの **spec** にexternal ロギングを定義し、**logging.valueFrom.configMapKeyRef.name** に ConfigMap の名前を、**logging.valueFrom.configMapKeyRef.key** にこの ConfigMap のキーを設定します。
   Topic Operatorについては、**Kafka** リソースの **topicOperator** 設定でロギングを指定します。

```
spec:
  # ...
  entityOperator:
    topicOperator:
      logging:
        type: external
        valueFrom:
          configMapKeyRef:
            name: logging-configmap
            key: log4j2.properties
```

3. Cluster Operator をデプロイして変更を適用します。

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

関連情報

- Kafka の設定
- Cluster Operator のロギング
- Topic Operator のロギング
- User Operator のロギング

第3章 外部ソースからの設定値の読み込み

設定プロバイダープラグインを使用して、外部ソースから設定データを読み込みます。プロバイダーは AMQ Streams とは独立して動作します。これらを使用して、プロデューサーやコンシューマーなど、すべての Kafka コンポーネントの設定データをロードできます。たとえば、Kafka Connect コネクター設定のクレデンシャルを提供する場合に使用します。

OpenShift 設定プロバイダー

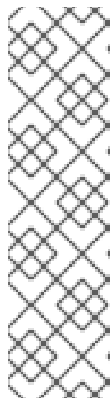
OpenShift Configuration Provider プラグインは、OpenShift シークレットまたは設定マップから設定データを読み込みます。

Kafka namespace 外で管理される **Secret** オブジェクト、または Kafka クラスター外にある Secret オブジェクトがあるとします。OpenShift 設定プロバイダーを使用すると、ファイルを抽出せずに設定の秘密の値を参照できます。使用するシークレットとアクセス権限の提供をプロバイダーに伝える必要があります。プロバイダーは、新しい **Secret** または **ConfigMap** オブジェクトを使用している場合でも、Kafka コンポーネントを再起動することなくデータを読み込みます。この機能により、Kafka Connect インスタンスが複数のコネクターをホストする場合に中断の発生を防ぎます。

環境変数設定プロバイダー

環境変数の設定プロバイダープラグインは、環境変数から設定データを読み込みます。

環境変数の値は、シークレットまたは設定マップからマッピングできます。環境変数設定プロバイダーを使用して、OpenShift シークレットからマップされた環境変数から証明書または JAAS 設定を読み込むことができます。



注記

OpenShift Configuration Provider はマウントされたファイルを使用できません。たとえば、トラストストアまたはキーストアの場所を必要とする値をロードできません。代わりに、設定マップまたはシークレットを環境変数またはボリュームとして Kafka Connect Pod にマウントできます。環境変数の設定プロバイダーを使用して、環境変数の値を読み込むことができます。**KafkaConnect.spec** の externalConfiguration プロパティーを使用して設定を追加します。このアプローチでアクセス権限を設定する必要はありません。ただし、コネクターに新しい **Secret** または **ConfigMap** を使用する場合は、Kafka Connect の再起動が必要になります。これにより、すべての Kafka Connect インスタンスのコネクターが中断されます。

3.1. 設定マップからの設定値の読み込み

この手順では、OpenShift Configuration Provider プラグインを使用する方法を説明します。

この手順では、外部 **ConfigMap** オブジェクトはコネクターの設定プロパティーを提供します。

前提条件

- OpenShift クラスターが利用できる必要があります。
- Kafka クラスターが稼働している必要があります。
- Cluster Operator が稼働している必要があります。

手順

1. 設定プロパティーが含まれる **ConfigMap またはシークレットを作成します。**

この例では、my -connector-configuration という名前の ConfigMap オブジェクトにコネクタープロパティーが含まれます。

コネクタープロパティーのある ConfigMap の例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-connector-configuration
data:
  option1: value1
  option2: value2
```

2. Kafka Connect 設定で OpenShift Configuration Provider を指定します。

ここで示されている仕様は、シークレットおよび設定マップからの値の読み込みをサポートできます。

OpenShift 設定プロバイダーを有効にする Kafka Connect の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
    config.providers: secrets,configmaps (1)
    config.providers.secrets.class: io.strimzi.kafka.KubernetesSecretConfigProvider (2)
    config.providers.configmaps.class:
io.strimzi.kafka.KubernetesConfigMapConfigProvider (3)
  # ...
```

1. 設定プロバイダーのエイリアスは、他の設定パラメーターを定義するために使用されます。プロバイダーパラメーターは config.providers からのエイリアスを使用し、config.providers.${alias}.class の形式を取ります。

2. KubernetesSecretConfigProvider はシークレットから値を提供します。

3. KubernetesConfigMapConfigProvider は設定マップから値を提供します。

3. リソースを作成または更新してプロバイダーを有効にします。

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

4. 外部設定マップの値へのアクセスを許可するロールを作成します。

設定マップから値にアクセスするためのロールの例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: connector-configuration-role
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["my-connector-configuration"]
  verbs: ["get"]
# ...
```

このルールは、my -connector-configuration 設定マップにアクセスするためのロールパーミッションを付与します。

5. ロールバインディングを作成し、設定マップが含まれる namespace へのアクセスを許可します。

設定マップが含まれる namespace にアクセスするためのロールバインディングの例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: connector-configuration-role-binding
subjects:
- kind: ServiceAccount
  name: my-connect-connect
  namespace: my-project
roleRef:
  kind: Role
  name: connector-configuration-role
  apiGroup: rbac.authorization.k8s.io
# ...
```

ロールバインディングは、ロールにmy-project名前空間へのアクセス許可を与えます。

サービスアカウントは、Kafka Connect デプロイメントによって使用されるものと同じである必要があります。サービスアカウント名の形式は <cluster_name>-connect です。<cluster_name> は KafkaConnect カスタムリソースの名前です。

6. コネクター設定で設定マップを参照します。

設定マップを参照するコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
  # ...
  config:
    option: ${configmaps:my-project/my-connector-configuration:option1}
    # ...
# ...
```

設定マップのプロパティー値のプレースホルダーはコネクター設定で参照されます。プレースホルダー構造は configmaps:<path_and_file_name>:<property> です。KubernetesConfigMapConfigProvider は、外部設定マップから option1 プロパティーの値を読み取り、抽出します。

3.2. 環境変数から設定値の読み込み

以下の手順では、環境変数の設定プロバイダープラグインを使用する方法を説明します。

この手順では、環境変数はコネクターの設定プロパティーを提供します。データベースのパスワードは環境変数として指定されます。

前提条件

- OpenShift クラスターが利用できる必要があります。
- Kafka クラスターが稼働している必要があります。
- Cluster Operator が稼働している必要があります。

手順

1. Kafka Connect 設定で環境変数設定プロバイダーを指定します。

   externalConfiguration プロパティーを使用して 環境変数を定義します。

   環境変数設定プロバイダーを有効にする Kafka Connect の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
```

```
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
    config.providers: env (1)
    config.providers.env.class: io.strimzi.kafka.EnvVarConfigProvider (2)
  # ...
  externalConfiguration:
    env:
      - name: DB_PASSWORD (3)
        valueFrom:
          secretKeyRef:
            name: db-creds (4)
            key: dbPassword (5)
  # ...
```

(1) 設定プロバイダーのエイリアスは、他の設定パラメーターを定義するために使用されます。プロバイダーパラメーターは config.providers からのエイリアスを使用し、config.providers.${alias}.class の形式を取ります。

(2) EnvVarConfigProvider は環境変数から値を提供します。

(3) DB_PASSWORD 環境変数がシークレットからパスワードの値を取ります。

(4) 事前に定義されたパスワードが含まれるシークレットの名前。

(5) シークレット内に格納されているパスワードのキー。

2. リソースを作成または更新してプロバイダーを有効にします。

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

3. コネクター設定で環境変数を参照します。

環境変数を参照するコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
  # ...
  config:
    option: ${env:DB_PASSWORD}
    # ...
# ...
```

第4章 OPENSHIFT クラスター外の KAFKA へのアクセス

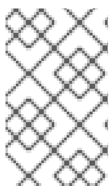
外部リスナーを使用して AMQ Streams の Kafka クラスターを OpenShift 環境外のクライアントに公開します。

外部リスナー設定で Kafka を公開するため type を指定します。

- nodeport は NodePort タイプの Services を使用します。
- loadbalancer が使用する Loadbalancer 型 Services
- ingress はKubernetes Ingress と NGINX Ingress Controller for Kubernetes を使用しています。
- route は、OpenShift Routes と HAProxy ルーターを使用します。

リスナーの設定の詳細については、GenericKafkaListener スキーマリファレンスを参照してください。

各接続タイプの pros および cons に関する詳細情報は、「S trimzi での Apache Kafka へのアクセス」を参照してください。



注記

route は OpenShift でのみサポートされます。

4.1. ノードポートを使用した KAFKA へのアクセス

この手順では、ノードポートを使用して外部クライアントから AMQ Streams Kafka クラスターにアクセスする方法について説明します。

ブローカーに接続するには、Kafka bootstrap アドレスのホスト名とポート番号、および認証に使用される証明書が必要です。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. 外部リスナーを nodeport タイプに設定して Kafka リソースを設定します。

   以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: nodeport
        tls: true
        authentication:
          type: tls
        # ...
    # ...
  zookeeper:
    # ...
```

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

   NodePort タイプのサービスは、各 Kafka ブローカーと、外部のブートストラップサービスのために作成されます。ブートストラップサービスは外部トラフィックを Kafka ブローカーにルーティングします。接続に使用されるノードアドレスは、Kafka カスタムリソースの status に伝搬されます。

   kafka ブローカーの ID を検証するためのクラスター CA 証明書も、Kafka リソースと同じ名前で作成されます。

3.

Kafka リソースのステータスから、Kafka クラスタにアクセスする際に使用するブートストラップアドレスを取得します。

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?(@.type=="external")].bootstrapServers}{"\n"}'
```

4. TLS による暗号化が有効な場合は、ブローカーの認証局の公開証明書を取得します。

```
oc get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

Kafka クライアントで取得した証明書を使用して TLS 接続を設定します。認証が有効になっている場合は、SASL または TLS 認証を設定する必要もあります。

4.2. ロードバランサーを使用した KAFKA へのアクセス

この手順では、ロードバランサーを使用して外部クライアントから AMQ Streams Kafka クラスターにアクセスする方法について説明します。

ブローカーに接続するには、ブートストラップロードバランサーのアドレスと、TLS による暗号化に使用される証明書が必要です。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. 外部リスナーを loadbalancer タイプに設定して Kafka リソースを設定します。

   以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
```

```
    # ...
    listeners:
      - name: external
        port: 9094
        type: loadbalancer
        tls: true
        # ...
    # ...
  zookeeper:
    # ...
```

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

loadbalancer タイプのサービスおよびロードバランサーは、各 Kafka ブローカーと外部 bootstrap service について作成されます。ブートストラップサービスは外部トラフィックをすべての Kafka ブローカーにルーティングします。接続に使用したDNS名やIPアドレスは、各サービスの status に伝わります。

kafka ブローカーの ID を検証するためのクラスター CA 証明書も、Kafka リソースと同じ名前で作成されます。

3. Kafka リソースのステータスから、Kafka クラスタへのアクセスに使用できるブートストラップサービスのアドレスを取得します。

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?(@.type=="external")].bootstrapServers}{"\n"}'
```

4. TLS による暗号化が有効な場合は、ブローカーの認証局の公開証明書を取得します。

```
oc get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

Kafka クライアントで取得した証明書を使用して TLS 接続を設定します。認証が有効になっている場合は、SASL または TLS 認証を設定する必要もあります。

4.3. INGRESS を使用した KAFKA へのアクセス

このの手順では、Nginx Ingress を使用して OpenShift 外部の外部クライアントから AMQ Streams Kafka クラスターにアクセスする方法を説明します。

ブローカーに接続するには、Ingress ブートストラップアドレス のホスト名 (アドバタイズされたアドレス) と、認証に使用される証明書が必要です。

Ingress を使用したアクセスでは、ポートは常に 443 になります。

TLS パススルー

Kafka は TCP 上でバイナリープロトコルを使用しますが、NGINX Ingress Controller for Kubernetes は HTTP プロトコルで動作するように設計されています。Ingress から Kafka コネクションを渡せるようにするため、AMQ Streams では NGINX Ingress Controller for Kubernetes の TLS パススルー機能が使用されます。TLS パススルーが NGINX Ingress Controller for Kubernetes デプロイメントで有効になっているようにしてください。

Ingress を使用して Kafka を公開する場合、TLS パススルー機能を使用するため、TLS による暗号化を無効にできません。

TLS パススルーの有効化に関する詳細は、TLS パススルーのドキュメント を参照してください。

前提条件

- OpenShift クラスターが必要です。
- TLS パススルーが有効になっている、デプロイ済みの NGINX Ingress Controller for Kubernetes。
- 稼働中の Cluster Operator。

手順

1. 外部リスナーを ingress タイプに設定して Kafka リソースを設定します。

   ブートストラップサービスおよび Kafka ブローカーの Ingress ホストを指定します。

以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: ingress
        tls: true
        authentication:
          type: tls
        configuration: (1)
          bootstrap:
            host: bootstrap.myingress.com
          brokers:
          - broker: 0
            host: broker-0.myingress.com
          - broker: 1
            host: broker-1.myingress.com
          - broker: 2
            host: broker-2.myingress.com
    # ...
  zookeeper:
    # ...
```

(1) ブートストラップサービスおよび Kafka ブローカーの Ingress ホスト。

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

Kafka ブローカーごとに ClusterIP タイプのサービスが作成され、さらに bootstrap service も追加されています。これらのサービスは、トラフィックを Kafka ブローカーにルーティングするために Ingress コントローラーによって使用されます。また、Ingressコントローラを使ってサービスを公開するために、各サービスに Ingress リソースを作成します。Ingress ホストは各サービスの status に伝播されます。

kafka ブローカーの ID を検証するためのクラスター CA 証明書も、Kafka リソースと同じ名前で作成されます。

Kafka クラスターに接続するためのブートストラップアドレスとして、configuration で指定したブートストラップホストのアドレスと、Kafka クライアントのポート 443（BOOTSTRAP-HOST:443）を使用します。

3. ブローカーの認証局の公開証明書を取得します。

```
oc get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' |
base64 -d > ca.crt
```

Kafka クライアントで取得した証明書を使用して TLS 接続を設定します。認証が有効になっている場合は、SASL または TLS 認証を設定する必要もあります。

4.4. OPENSHIFT ルートを使用した KAFKA へのアクセス

この手順では、ルートを使用して OpenShift 外部の外部クライアントから AMQ Streams Kafka クラスターにアクセスする方法について説明します。

ブローカーに接続するには、ルートブートストラップアドレス のホスト名と、TLS による暗号化に使用される証明書が必要です。

ルートを使用したアクセスでは、ポートは常に 443 になります。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. 外部リスナーを route タイプに設定した Kafka リソースを設定します。

   以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
```

```
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners:
      - name: listener1
        port: 9094
        type: route
        tls: true
        # ...
    # ...
  zookeeper:
    # ...
```

警告

OpenShift Route アドレスは、Kafka クラスターの名前、リスナーの名前、および作成される namespace の名前で構成されます。たとえば、my-cluster-kafka-listener1-bootstrap-myproject (CLUSTER-NAME-kafka-LISTENER-NAME-bootstrap-NAMESPACE) となります。アドレスの全体の長さが上限の 63 文字を超えないように注意してください。

2. リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

ClusterIP タイプサービスは、各 Kafka ブローカーと、外部 bootstrap service に対して作成されます。サービスは、トラフィックを OpenShift ルートから Kafka ブローカーにルーティングします。また、HAProxyロードバランサーを使ってサービスを公開するために、各サービスにOpenShift Routeリソースを作成します。接続に使用される DNS アドレスは、各サービスの status に伝播されます。

kafka ブローカーの ID を検証するためのクラスター CA 証明書も、Kafka リソースと同じ名前で作成されます。

3. Kafka リソースのステータスから、Kafka クラスタへのアクセスに使用できるブートストラップサービスのアドレスを取得します。

–

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?(@.type=="external")].bootstrapServers}{"\n"}'
```

4. ブローカーの認証局の公開証明書を取得します。

```
oc get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d > ca.crt
```

Kafka クライアントで取得した証明書を使用して TLS 接続を設定します。認証が有効になっている場合は、SASL または TLS 認証を設定する必要もあります。

第5章 KAFKA へのセキュアなアクセスの管理

各クライアントの Kafka ブローカーへのアクセスを管理することで、Kafka クラスターを保護できます。

Kafka ブローカーとクライアント間のセキュアな接続には、以下が含まれます。

- データ交換の暗号化
- アイデンティティー証明に使用する認証
- ユーザーが実行するアクションを許可または拒否する承認

本章では、以下を取り上げ、Kafka ブローカーとクライアント間でセキュアな接続を設定する方法を説明します。

- Kafka クラスターおよびクライアントのセキュリティーオプション
- Kafka ブローカーをセキュアにする方法
- OAuth 2.0 トークンベースの認証および承認に承認サーバーを使用する方法

5.1. KAFKA のセキュリティーオプション

Kafka リソースを使用して、Kafka の認証および承認に使用されるメカニズムを設定します。

5.1.1. リスナー認証

OpenShift クラスター内のクライアントの場合は、plain （暗号化なし）または tls internal リスナーを作成できます。

OpenShift クラスター外のクライアントの場合は、external リスナーを作成

し、nodeport、loadbalancer、ingress、または route（OpenShift 上）などの接続メカニズムを指定します。

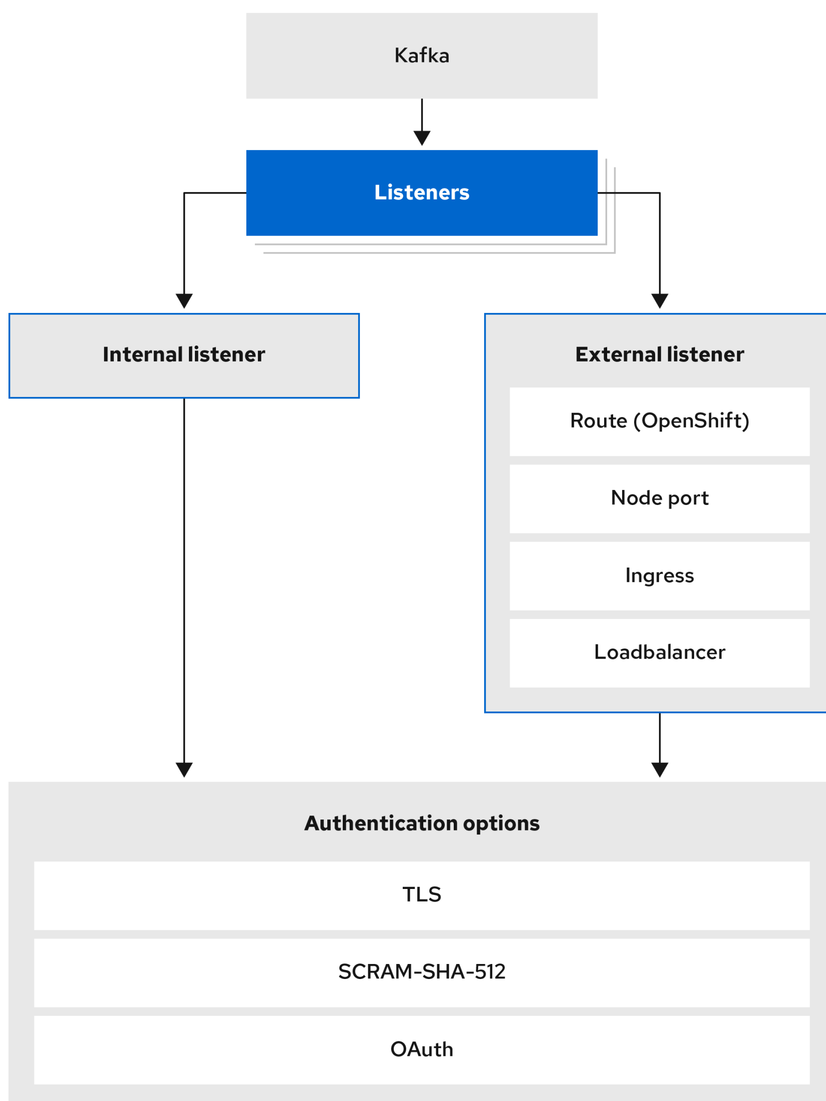
外部クライアントに接続するための設定オプションの詳細は、「OpenShift クラスター外で Kafka へのアクセス」を参照してください。

サポートされる認証オプションは次のとおりです。

1. 相互 TLS 認証 (TLS による暗号化が有効なリスナーのみ)
2. SCRAM-SHA-512 認証
3. OAuth 2.0 のトークンベースの認証

選択する認証オプションは、Kafka ブローカーへのクライアントアクセスを認証する方法によって異なります。

図5.1 Kafka リスナーの認証オプション



リスナーの authentication プロパティーは、そのリスナーに固有の認証メカニズムを指定するために使用されます。

authentication プロパティーが指定されていない場合、リスナーはそのリスナー経由で接続するクライアントを認証しません。認証がないと、リスナーではすべての接続が許可されます。

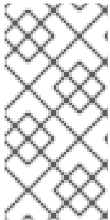
認証は、User Operator を使用して KafkaUsers を管理する場合に設定する必要があります。

以下の例で指定されるものは次のとおりです。

- SCRAM-SHA-512 認証に設定された plain リスナー

- 相互 TLS 認証を使用する tls リスナー
- 相互 TLS 認証を使用する external リスナー

各リスナーは、Kafka クラスター内で一意の名前およびポートで設定されます。



注記

ブローカー間通信 (9091 または 9090) およびメトリクス (9404) 用に確保されたポートを使用するようにリスナーを設定することはできません。

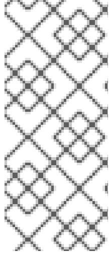
リスナー認証設定の例

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: true
    authentication:
      type: scram-sha-512
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication:
      type: tls
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
# ...
```

5.1.1.1. 相互 TLS 認証

相互 TLS 認証は、Kafka ブローカーと ZooKeeper Pod 間の通信で常に使用されます。

AMQ Streams では、Kafka が TLS (Transport Layer Security) を使用して、相互認証の有無を問わず、Kafka ブローカーとクライアントとの間で暗号化された通信が行われるよう設定できます。相互 (双方向) 認証の場合、サーバーとクライアントの両方が証明書を提示します。相互認証を設定すると、ブローカーはクライアントを認証し (クライアント認証)、クライアントはブローカーを認証します (サーバー認証)。



注記

TLS 認証は一般的には一方向で、一方が他方のアイデンティティーを認証します。たとえば、Web ブラウザーと Web サーバーの間で HTTPS が使用される場合、ブラウザーは Web サーバーのアイデンティティーの証明を取得します。

5.1.1.2. SCRAM-SHA-512 認証

SCRAM (Salted Challenge Response Authentication Mechanism) は、パスワードを使用して相互認証を確立できる認証プロトコルです。AMQ Streams では、Kafka が SASL (Simple Authentication and Security Layer) SCRAM-SHA-512 を使用するよう設定し、暗号化されていないクライアントの接続と暗号化されたクライアントの接続の両方で認証を提供できます。

TLS クライアント接続で SCRAM-SHA-512 認証が使用される場合、TLS プロトコルは暗号化を提供しますが、認証には使用されません。

SCRAM の以下のプロパティーは、暗号化されていない接続でも SCRAM-SHA-512 を安全に使用できるようにします。

- 通信チャネル上では、パスワードはクリアテキストで送信されません。代わりに、クライアントとサーバーはお互いにチャレンジを生成し、認証するユーザーのパスワードを認識していることを証明します。

- サーバーとクライアントは、認証を交換するたびに新しいチャレンジを生成します。よって、この交換はリレー攻撃に対する回復性を備えています。

KafkaUser.spec.authentication.type を scram-sha-512 に設定すると、User Operator は、大文字と小文字の ASCII 文字と数字で構成されるランダムな 12 文字のパスワードを生成します。

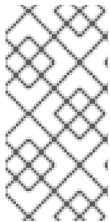
5.1.1.3. ネットワークポリシー

デフォルトでは、AMQ Streams では、Kafka ブローカーで有効になっているリスナーごとに

NetworkPolicy リソースが自動的に作成されます。この NetworkPolicy により、アプリケーションはすべての namespace のリスナーに接続できます。リスナー設定の一部としてネットワークポリシーを使用します。

ネットワークレベルでのリスナーへのアクセスを指定のアプリケーションまたは namespace のみに制限するには、networkPolicyPeers プロパティーを使用します。リスナーごとに、異なる networkPolicyPeers 設定を指定できます。ネットワークポリシーピアの詳細は、NetworkPolicyPeer API reference を参照してください。

カスタムネットワークポリシーを使用する場合は、Cluster Operator 設定で STRIMZI_NETWORK_POLICY_GENERATION 環境変数を false に設定できます。詳細は、Cluster Operator configuration を参照してください。



注記

AMQ Streams でネットワークポリシーを使用するためには、OpenShift の構成が ingress NetworkPolicies をサポートしている必要があります。

5.1.1.4. 追加のリスナー設定オプション

GenericKafkaListenerConfiguration スキーマのプロパティーを使用して、設定をリスナーに追加できます。

5.1.2. Kafka の承認

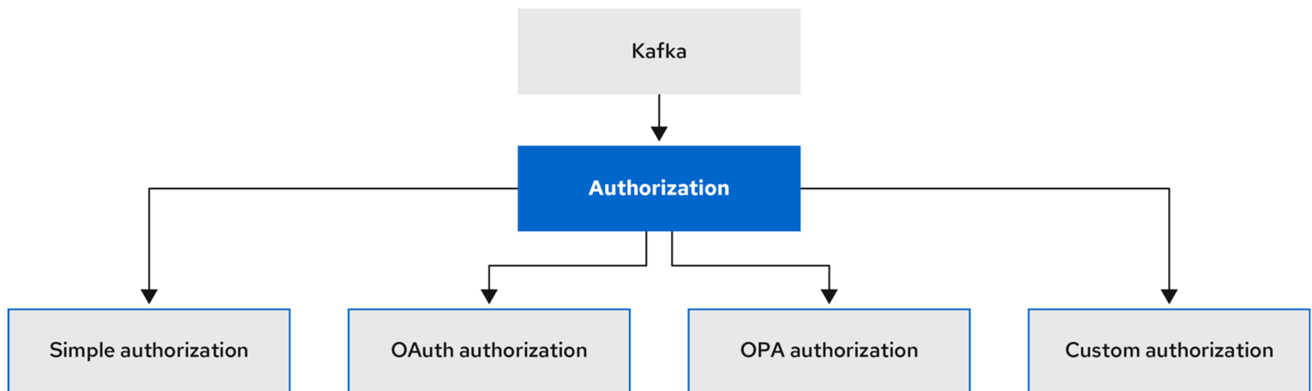
Kafka.spec.kafka リソースの authorization プロパティーを使用すると Kafka ブローカーの承認を設定できます。authorization プロパティーがないと、承認が有効になりず、クライアントには制限がありません。承認を有効にすると、承認は有効なすべてのリスナーに適用されます。承認方法は type フィールドで定義されます。

サポートされる承認オプションは次のとおりです。

- 簡易承認
- OAuth 2.0 での承認 (OAuth 2.0 トークンベースの認証を使用している場合)

- Open Policy Agent (OPA) での承認
- カスタム承認

図5.2 Kafka クラスター承認オプション



5.1.2.1. スーパーユーザー

スーパーユーザーは、アクセスの制限に関係なく Kafka クラスターのすべてのリソースにアクセスでき、すべての承認メカニズムでサポートされます。

Kafka クラスターのスーパーユーザーを指定するには、superUsers プロパティーにユーザープリンシパルのリストを追加します。ユーザーが TLS クライアント認証を使用する場合、ユーザー名は CN= で始まる証明書のサブジェクトの共通名になります。

スーパーユーザーを使用した設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: simple
      superUsers:
        - CN=client_1
```

```
      - user_2
      - CN=client_3
  # ...
```

5.2. KAFKA クライアントのセキュリティーオプション

KafkaUser リソースを使用して、Kafka クライアントの認証メカニズム、承認メカニズム、およびアクセス権限を設定します。セキュリティーの設定では、クライアントはユーザーとして表されます。

Kafka ブローカーへのユーザーアクセスを認証および承認できます。認証によってアクセスが許可され、承認によって許容されるアクションへのアクセスが制限されます。

Kafka ブローカーへのアクセスが制限されない スーパーユーザー を作成することもできます。

認証および承認メカニズムは、Kafka ブローカーへのアクセスに使用されるリスナーの仕様 と一致する必要があります。

Kafka ブローカーへのアクセスをセキュアにするためのユーザーの設定

Kafka ブローカーをセキュアにアクセスできるように KafkaUser リソースを設定する方法については、以下のセクションを参照してください。

- Kafka へのユーザーアクセスのセキュア化
- OpenShift 外クライアントのアクセスの設定

5.2.1. ユーザー処理用の Kafka クラスターの特定

KafkaUser リソースには、このリソースが属する Kafka クラスターに適した名前 (Kafka リソースの名前から派生) を定義するラベルが含まれています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
```

```
name: my-user
labels:
  strimzi.io/cluster: my-cluster
```

このラベルは、KafkaUser リソースを特定し、新しいユーザーを作成するために、User Operator によって使用されます。また、以降のユーザーの処理でも使用されます。

ラベルが Kafka クラスターと一致しない場合、User Operator は KafkaUser を識別できず、ユーザーは作成されません。

KafkaUser リソースの状態が空のままの場合は、ラベルを確認します。

5.2.2. ユーザー認証

ユーザー認証は、KafkaUser.spec の authentication プロパティーを使用して設定されます。ユーザーに有効な認証メカニズムは、type フィールドを使用して指定されます。

サポートされる認証タイプ

- TLSクライアント認証のための tls
- 外部証明書を使用した TLS クライアント認証の tls-external
- scram-sha-512（SCRAM-SHA-512 認証用)

tls または scram-sha-512 が指定された場合、User Operator がユーザーを作成する際に、認証用のクレデンシャルを作成します。tls-external が指定されている場合、ユーザーは引き続き TLS クライアント認証を使用しますが、認証情報は作成されません。独自の証明書を指定する場合は、このオプションを使用します。認証タイプが指定されていない場合、User Operator はユーザーまたはそのクレデンシャルを作成しません。

tls-external を使用して、User Operator 外で発行された証明書を使用して TLS クライアント認証で認証できます。User Operator は TLS 証明書またはシークレットを生成しません。tls メカニズムを使用する場合と同じように、User Operator を使用して ACL ルールおよびクォータを管理できます。これは、ACL ルールおよびクォータを指定する際に CN=USER-NAME 形式を使用することを意味します。USER-NAME は、TLS 証明書で指定した共通名です。

その他のリソース

- クライアントに相互 TLS 認証または SCRAM-SHA 認証を使用する場合
- KafkaUserSpec schema reference

5.2.2.1. TLS クライアント認証

TLSクライアント認証を使用するには、KafkaUser リソースの type フィールドを tls に設定します。

TLS クライアント認証が有効になっているユーザーの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  # ...
```

ユーザーが User Operator によって作成されると、KafkaUser リソースと同じ名前で新しいシークレットが作成されます。シークレットには、TLS クライアント認証の秘密鍵と公開鍵が含まれます。公開鍵は、クライアント認証局 (CA) によって署名されたユーザー証明書に含まれます。

すべての鍵は X.509 形式です。

Secret には、PEM 形式および PKCS #12 形式の秘密鍵と証明書が含まれます。

Kafka と Secret との通信をセキュアにする方法については、12章*TLS 証明書の管理* を参照してください。

ユーザー認証情報を含むシークレットの例

```
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 client CA
  user.crt: # User certificate that contains the public key of the user
  user.key: # Private key of the user
  user.p12: # PKCS #12 archive file for storing certificates and keys
  user.password: # Password for protecting the PKCS #12 archive file
```

5.2.2.2. User Operator の外部で発行された証明書を使用した TLS クライアント認証

User Operator の外部で発行された証明書を使用して TLS クライアント認証を使用するには、KafkaUser リソースの type フィールドを tls-external に設定します。シークレットおよび認証情報はユーザー用には作成されません。

User Operator の外部で発行された証明書を使用する TLS クライアント認証を持つユーザーの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls-external
  # ...
```

5.2.2.3. SCRAM-SHA-512 認証

SCRAM-SHA-512 認証メカニズムを使用するには、KafkaUser リソースの type フィールドを scram-sha-512 に設定します。

SCRAM-SHA-512 認証が有効になっているユーザーの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  # ...
```

ユーザーが User Operator によって作成されると、KafkaUser リソースと同じ名前で新しいシークレットが作成されます。シークレットの password キーには、生成されたパスワードが含まれ、base64 でエンコードされます。パスワードを使用するにはデコードする必要があります。

ユーザー認証情報を含むシークレットの例

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  password: Z2VuZXJhdGVkcGFzc3dvcmQ= (1)
  sasl.jaas.config:
b3JnLmFwYWNoZS5rYWZrYS5jb21tb24uc2VjdXJpdHkuc2NyYW0uU2NyYW1Mb2dpbk1vZHVs
ZSByZXF1aXJlZCB1c2VybmFtZT0ibXktdXNlciIgcGFzc3dvcmQ9ImdlbmVyYXRlZHBhc3N3b3JkIl
jsK (2)
```

(1)

base64 でエンコードされた生成されたパスワード。

2

base64 でエンコードされた SASL SCRAM-SHA-512 認証の JAAS 設定文字列。

生成されたパスワードをデコードします。

```
echo "Z2VuZXJhdGVkcGFzc3dvcmQ=" | base64 --decode
```

5.2.2.3.1. カスタムパスワード設定

ユーザーが作成されると、AMQ Streams は無作為にパスワードを生成します。AMQ Streams によって生成されたパスワードの代わりに、独自のパスワードを使用できます。これを行うには、パスワードでシークレットを作成し、KafkaUser リソースでこれを参照します。

SCRAM-SHA-512 認証に設定されたパスワードを持つユーザーの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
    password:
      valueFrom:
        secretKeyRef:
          name: my-secret 1
          key: my-password 2
  # ...
```

1

事前に定義されたパスワードが含まれるシークレットの名前。

2

シークレット内に格納されているパスワードのキー。

5.2.3. ユーザーの承認

ユーザーの承認は、KafkaUser.spec の authorization プロパティーを使用して設定されます。ユーザーに有効な承認タイプは、type フィールドを使用して指定します。

簡易承認を使用するには、KafkaUser.spec.authorization で type プロパティーを simple に設定します。簡易承認は、Kafka Admin API を使用して Kafka クラスター内で ACL ルールを管理します。User Operator の ACL 管理が有効であるかどうかは、Kafka クラスターの承認設定によって異なります。

- 簡易承認では、ACL 管理が常に有効になります。
- OPA 承認の場合、ACL 管理は常に無効になります。承認ルールは OPA サーバーで設定されます。
- Red Hat Single Sign-On の承認では、Red Hat Single Sign-On で ACL ルールを直接管理できます。設定のフォールバックオプションとして、承認を簡単なオーソライザーに委譲することもできます。簡単なオーソライザーへの委譲が有効になっている場合、User Operator は ACL ルールの管理も有効にします。
- カスタム承認プラグインを使用したカスタム承認では、Kafka カスタムリソースの .spec.kafka.authorization 設定の supportsAdminApi プロパティーを使用して、サポートを有効または無効にする必要があります。

ACL 管理が有効になっていない場合は、AMQ Streams に ACL ルールが含まれる場合はリソースを拒否します。

User Operator のスタンドアロンデプロイメントを使用している場合、ACL 管理はデフォルトで有効にされます。STRIMZI_ACLS_ADMIN_API_SUPPORTED 環境変数を使用してこれを無効にすることができます。

承認が指定されていない場合は、User Operator によるユーザーのアクセス権限のプロビジョニングは行われません。このような KafkaUser がリソースにアクセスできるかどうかは、使用されている

オーソライザーによって異なります。たとえば、AclAuthorizer の場合、これは allow.everyone.if.no.acl.found 設定によって決定されます。

5.2.3.1. ACL ルール

AclAuthorizer は ACL ルールを使用して Kafka ブローカーへのアクセスを管理します。

ACL ルールによって、acls プロパティーで指定したユーザーにアクセス権限が付与されます。

AclRule オブジェクトの詳細は、AclRule schema reference を参照してください。

5.2.3.2. Kafka ブローカーへのスーパーユーザーアクセス

ユーザーを Kafka ブローカー設定のスーパーユーザーのリストに追加すると、KafkaUser の ACL で定義された承認制約に関係なく、そのユーザーにはクラスターへのアクセスが無制限に許可されます。

ブローカーへのスーパーユーザーアクセスの設定に関する詳細は「Kafka の承認」を参照してください。

5.2.3.3. ユーザークォータ

KafkaUser リソースの spec を設定してクォータを強制し、ユーザーが Kafka ブローカーへの設定されたアクセスレベルを超えないようにします。サイズベースのネットワーク使用量と時間ベースの CPU 使用率のしきい値を設定できます。また、パーティション mutation (変更) クォータを追加して、ユーザー要求に対して受け入れられるパーティション変更のリクエストのレートを制御することもできます。

ユーザークォータをともなう KafkaUser の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  quotas:
```

```
producerByteRate: 1048576 (1)
consumerByteRate: 2097152 (2)
requestPercentage: 55 (3)
controllerMutationRate: 10 (4)
```

1. ユーザーが Kafka ブローカーにプッシュできるデータ量の、秒あたりのバイトクォータ。
2. ユーザーが Kafka ブローカーからフェッチできるデータ量の、秒あたりのバイトクォータ。
3. クライアントグループあたりの時間割合で示される、CPU 使用制限。
4. 1 秒あたり許容される同時パーティション作成および削除操作 (mutations) の数

これらのプロパティーの詳細は、KafkaUserQuotas schema reference を参照してください。

5.3. KAFKA ブローカーへのアクセスのセキュア化

Kafka ブローカーへのセキュアなアクセスを確立するには、以下を設定し、適用します。

- 以下を行う Kafka リソース。
  - 指定された認証タイプでリスナーを作成します。
  - Kafka クラスター全体の承認を設定します。
- Kafka ブローカーにリスナー経由でセキュアにアクセスするための KafkaUser リソース。

Kafka リソースを設定して以下を設定します。

- リスナー認証
- Kafka リスナーへのアクセスを制限するネットワークポリシー
- Kafka の承認
- ブローカーへのアクセスが制限されないスーパーユーザー

認証は、リスナーごとに独立して設定されます。承認は、常に Kafka クラスター全体に対して設定されます。

Cluster Operator はリスナーを作成し、クラスターおよびクライアント認証局 (CA) 証明書を設定して Kafka クラスター内で認証を有効にします。

独自の証明書をインストールして、Cluster Operator によって生成された証明書 を置き換えることができます。外部認証局によって管理される Kafka リスナー証明書を使用するようにリスナーを設定することもできます。PKCS #12 形式 (.p12) および PEM 形式 (.crt) の証明書を利用できます。

KafkaUser を使用して、特定のクライアントが Kafka にアクセスするために使用する認証および承認メカニズムを有効にします。

KafkaUser リソースを設定して以下を設定します。

- 有効なリスナー認証と一致する認証
- 有効な Kafka 承認と一致する承認
- クライアントによるリソースの使用を制御するクォータ

User Operator はクライアントに対応するユーザーを作成すると共に、選択した認証タイプに基づいて、クライアント認証に使用されるセキュリティークレデンシャルを作成します。

設定プロパティーへのアクセスに関する詳細は、スキーマ参照を参照してください。

- Kafka スキーマ参照
- KafkaUser スキーマ参照
- GenericKafkaListener schema reference

5.3.1. Kafka ブローカーのセキュア化

この手順では、AMQ Streams の実行時に Kafka ブローカーをセキュアにするためのステップを説明します。

Kafka ブローカーに実装されたセキュリティーは、アクセスを必要とするクライアントに実装されたセキュリティーとの互換性を維持する必要があります。

- Kafka.spec.kafka.listeners[*].authentication matches KafkaUser.spec.authentication
- Kafka.spec.kafka.authorization matches KafkaUser.spec.authorization

この手順では、TLS 認証を使用した簡易承認とリスナーの設定を説明します。リスナーの設定の詳細については、GenericKafkaListener schema reference を参照してください。

代わりに、リスナー認証 には SCRAM-SHA または OAuth 2.0、Kafka 承認 には OAuth 2.0 または OPA を使用することができます。

手順

1. Kafka リソースを設定します。

a. 承認には authorization プロパティーを設定します。

b. listeners プロパティーを設定し、認証でリスナーを作成します。

以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    authorization: (1)
      type: simple
      superUsers: (2)
        - CN=client_1
        - user_2
        - CN=client_3
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls (3)
    # ...
  zookeeper:
    # ...
```

1. Authorizationは、AclAuthorizer Kafka プラグインを使用して、Kafka ブローカーでのsimpleな承認を可能にします。

2. Kafka へのアクセスを制限されないユーザープリンシパルのリスト。CN は、TLS による認証が使用される場合のクライアント証明書の共通名です。

3. リスナーの認証メカニズムは各リスナーに対して設定でき、相互 TLS、SCRAM-SHA-512、またはトークンベース OAuth 2.0 として指定 できます。

外部リスナーを設定している場合、設定は選択した接続のメカニズムによって異なります。

2. Kafka リソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

Kafka クラスターは、TLS 認証を使用する Kafka ブローカーリスナーと共に設定されます。

Kafka ブローカー Pod ごとにサービスが作成されます。

サービスが作成され、Kafka クラスターに接続するための ブートストラップアドレス として機能します。

kafka ブローカーの ID を検証するためのクラスター CA 証明書も、Kafka リソースと同じ名前で作成されます。

5.3.2. Kafka へのユーザーアクセスのセキュア化

KafkaUser リソースのプロパティーを使用して Kafka ユーザーを設定します。

oc apply を使用すると、ユーザーを作成または編集できます。oc delete を使用すると、既存のユーザーを削除できます。

以下は例になります。

- oc apply -f USER-CONFIG-FILE
- oc delete KafkaUser USER-NAME

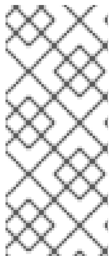
KafkaUser 認証および承認メカニズムを設定する場合、必ず同等の Kafka 設定と一致するようにしてください。

- KafkaUser.spec.authentication は Kafka.spec.kafka.listeners[*].authentication と一致します。

- KafkaUser.spec.authorization は Kafka.spec.kafka.authorization と一致します。

この手順では、TLS 認証でユーザーを作成する方法を説明します。SCRAM-SHA 認証でユーザーを作成することも可能です。

必要な認証は、Kafka ブローカーリスナーに設定された認証のタイプ によって異なります。



注記

Kafka ユーザーと Kafka ブローカー間の認証は、それぞれの認証設定によって異なります。たとえば、TLS が Kafka 設定で有効になっていない場合は、TLS でユーザーを認証できません。

前提条件

- TLS による認証および暗号化を使用して Kafka ブローカーリスナーで設定された 稼働中の Kafka クラスターが必要です。
- 稼働中の User Operator (通常は Entity Operator でデプロイされる) が必要です。

KafkaUser の認証タイプは、Kafka ブローカーに設定された認証と一致する必要があります。

手順

1. KafkaUser リソースを設定します。

   以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication: (1)
```

```
    type: tls
  authorization:
    type: simple ②
    acls:
      - 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
```

①

相互 tls または scram-sha-512 として定義されたユーザー認証メカニズム。

②

ACL ルールのリストが必要な簡易承認。

2. KafkaUser リソースを作成または更新します。

```
oc apply -f USER-CONFIG-FILE
```

KafkaUser リソースと同じ名前の Secret と共に、ユーザーが作成されます。Secret には、TLS クライアント認証の秘密鍵と公開鍵が含まれます。

Kafka ブローカーへの接続をセキュアにするために Kafka クライアントをプロパティーで設定する詳細は、『OpenShift での AMQ Streams のデプロイおよびアップグレード』の「 OpenShift 外クライアントのアクセスの設定 」を参照してください。

5.3.3. ネットワークポリシーを使用した Kafka リスナーへのアクセス制限

networkPolicyPeers プロパティーを使用すると、リスナーへのアクセスを指定のアプリケーションのみに制限できます。

前提条件

前提条件

- Ingress NetworkPolicies をサポートする OpenShift クラスター。
- Cluster Operator が稼働している必要があります。

手順

1. Kafka リソースを開きます。
2. networkPolicyPeers プロパティーで、Kafka クラスターへのアクセスが許可されるアプリケーション Pod または namespace を定義します。

   以下は、ラベル app が kafka-client に設定されているアプリケーションからの接続のみを許可するよう tls リスナーを設定する例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
        networkPolicyPeers:
          - podSelector:
              matchLabels:
                app: kafka-client
    # ...
  zookeeper:
    # ...
```

3. リソースを作成または更新します。

   次のように oc apply を使用します。

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

関連情報

- networkPolicyPeers configuration
- NetworkPolicyPeer API reference

5.4. OAUTH 2.0 トークンベース認証の使用

AMQ Streams は、OAUTHBEARER および PLAIN メカニズムを使用した OAuth 2.0 認証 の使用をサポートします。

OAuth 2.0 は、アプリケーション間で標準的なトークンベースの認証および承認を有効にし、中央の承認サーバーを使用してリソースに制限されたアクセス権限を付与するトークンを発行します。

OAuth 2.0 認証を設定した後に OAuth 2.0 承認 を設定できます。

Kafka ブローカーおよびクライアントの両方が OAuth 2.0 を使用するように設定する必要があります。OAuth 2.0 認証は、simple または OPA ベースの Kafka authorization と併用することもできます。

OAuth 2.0 のトークンベースの認証を使用すると、アプリケーションクライアントはアカウントのクレデンシャルを公開せずにアプリケーションサーバー (リソースサーバー と呼ばれる) のリソースにアクセスできます。

アプリケーションクライアントは、アクセストークンを認証の手段として渡します。アプリケーションサーバーはこれを使用して、付与するアクセス権限のレベルを決定することもできます。承認サーバーは、アクセスの付与とアクセスに関する問い合わせを処理します。

AMQ Streams のコンテキストでは以下が行われます。

- Kafka ブローカーは OAuth 2.0 リソースサーバーとして動作します。
- Kafka クライアントは OAuth 2.0 アプリケーションクライアントとして動作します。

Kafka クライアントは Kafka ブローカーに対して認証を行います。ブローカーおよびクライアント

は、必要に応じて OAuth 2.0 承認サーバーと通信し、アクセストークンを取得または検証します。

AMQ Streams のデプロイメントでは、OAuth 2.0 インテグレーションは以下を提供します。

- Kafka ブローカーのサーバー側 OAuth 2.0 サポート。
- Kafka MirrorMaker、Kafka Connect、および Kafka Bridge のクライアント側 OAuth 2.0 サポート。

5.4.1. OAuth 2.0 認証メカニズム

AMQ Streams は、OAuth 2.0 認証で OAUTHBEARER および PLAIN メカニズムをサポートします。どちらのメカニズムも、Kafka クライアントが Kafka ブローカーで認証されたセッションを確立できるようにします。クライアント、承認サーバー、および Kafka ブローカー間の認証フローは、メカニズムごとに異なります。

可能な限り、OAUTHBEARER を使用するようにクライアントを設定することが推奨されます。OAUTHBEARER では、クライアントクレデンシャルは Kafka ブローカーと共有されることがないため、PLAIN よりも高レベルのセキュリティーが提供されます。OAUTHBEARER をサポートしない Kafka クライアントの場合のみ、PLAIN の使用を検討してください。

必要な場合は、同じ oauth リスナーで OAUTHBEARER と PLAIN を同時に有効にできます。

OAUTHBEARER の概要

Kafka は OAUTHBEARER 認証メカニズムをサポートしますが、明示的に設定する必要があります。また、多くの Kafka クライアントツールでは、プロトコルレベルで OAUTHBEARER の基本サポートを提供するライブラリーを使用します。

AMQ Streams では、アプリケーションの開発を容易にするため、アップストリームの Kafka Client Java ライブラリー用の OAuth コールバックハンドラーが提供されます (ただし、他のライブラリーには提供されません)。そのため、このようなクライアントには独自のコールバックハンドラーを作成する必要はありません。アプリケーションクライアントはコールバックハンドラーを使用してアクセストークンを提供できます。Go などの他言語で書かれたクライアントは、カスタムコードを使用して承認サーバーに接続し、アクセストークンを取得する必要があります。

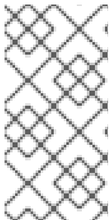
OAUTHBEARER を使用する場合、クライアントはクレデンシャルを交換するために Kafka ブローカーでセッションを開始します。ここで、クレデンシャルはコールバックハンドラーによって提供され

るベアラートークンの形式を取ります。コールバックを使用して、以下の 3 つの方法のいずれかでトークンの提供を設定できます。

- クライアント ID および Secret (OAuth 2.0 クライアントクレデンシャルメカニズムを使用)
- 設定時に手動で取得された有効期限の長いアクセストークン
- 設定時に手動で取得された有効期限の長い更新トークン

OAUTHBEARER は、Kafka ブローカーの oauth リスナー設定で自動的に有効になります。enableOauthBearer プロパティーを true に設定できますが、これは必須ではありません。

```
# ...
authentication:
  type: oauth
  # ...
  enableOauthBearer: true
```



注記

OAUTHBEARER 認証は、プロトコルレベルで OAUTHBEARER メカニズムをサポートする Kafka クライアントでのみ使用できます。

PLAIN の概要

PLAIN は、すべての Kafka クライアントツールによって使用される簡易認証メカニズムです。PLAIN を OAuth 2.0 認証とともに使用できるようにするため、AMQ Streams にはサーバー側のコールバックが含まれ、この OAuth 2.0 over PLAIN を呼び出します。

PLAIN の AMQ Streams 実装では、クライアントのクレデンシャルは ZooKeeper に保存されません。代わりに、OAUTHBEARER 認証が使用される場合と同様に、クライアントのクレデンシャルは準拠した承認サーバーの背後で一元的に処理されます。

OAuth 2.0 over PLAIN コールバックを併用する場合、以下のいずれかの方法を使用して Kafka クライアントは Kafka ブローカーで認証されます。

-

クライアント ID およびシークレット (OAuth 2.0 クライアントクレデンシャルメカニズムを使用)

- 設定時に手動で取得された有効期限の長いアクセストークン

クライアントは、PLAIN認証を使用できるようにして、username と password を提供する必要があります。パスワードに $accessToken: のプレフィックスが付けられ、その後にアクセストークンの値が続く場合、Kafka ブローカーはパスワードをアクセストークンとして解釈します。そうでない場合、Kafkaブローカーは、username をクライアントID、password をクライアントシークレットと解釈します。

アクセストークンとして password が設定されている場合、username はKafka ブローカーがアクセストークンから取得するプリンシパル名と同じものを設定する必要があります。この処理は、userNameClaim、fallbackUserNameClaim、fallbackUsernamePrefix、または userInfoEndpointUri を使ってユーザー名抽出をどのように設定したかによって異なります。また、承認サーバーによっても異なり、特にクライアント ID をアカウント名にマッピングする方法によります。

PLAIN を使用するには、Kafka ブローカーの oauth リスナー設定で有効にする必要があります。

以下の例では、デフォルトで有効になっている OAUTHBEARER に加え、PLAIN も有効になっています。PLAIN のみを使用する場合は、enableOauthBearer を false に設定して OAUTHBEARER を無効にすることができます。

```
# ...
authentication:
  type: oauth
  # ...
  enablePlain: true
  tokenEndpointUri: https://OAUTH-SERVER-ADDRESS/auth/realms/external/protocol/openid-connect/token
```

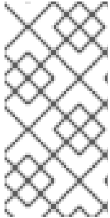
関連情報

- 「Kafka ブローカーの OAuth 2.0 サポートの設定」

5.4.2. OAuth 2.0 Kafka ブローカーの設定

OAuth 2.0 の Kafka ブローカー設定には、以下が関係します。

- 承認サーバーでの OAuth 2.0 クライアントの作成
- Kafka カスタムリソースでの OAuth 2.0 認証の設定



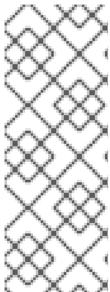
注記

承認サーバーに関連する Kafka ブローカーおよび Kafka クライアントはどちらも OAuth 2.0 クライアントと見なされます。

5.4.2.1. 承認サーバーの OAuth 2.0 クライアント設定

セッションの開始中に受信されたトークンを検証するように Kafka ブローカーを設定するには、承認サーバーで OAuth 2.0 の クライアント 定義を作成し、以下のクライアントクレデンシャルが有効な状態で 機密情報 として設定することが推奨されます。

- kafka のクライアント ID (例)
- 認証メカニズムとしてのクライアント ID およびシークレット



注記

承認サーバーのパブリックでないイントロスペクションエンドポイントを使用する場合のみ、クライアント ID およびシークレットを使用する必要があります。高速のローカル JWT トークンの検証と同様に、パブリック承認サーバーのエンドポイントを使用する場合は、通常クレデンシャルは必要ありません。

5.4.2.2. Kafka クラスターでの OAuth 2.0 認証設定

Kafka クラスターで OAuth 2.0 認証を使用するには、たとえば、認証方法が oauth の Kafka クラスターカスタムリソースの TLS リスナー設定を指定します。

OAuth 2.0 の認証方法タイプの割り当て

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
```

```
kafka:
  # ...
  listeners:
    - name: tls
      port: 9093
      type: internal
      tls: true
      authentication:
        type: oauth
    #...
```

plain、plain、external リスナーを設定することができますが、plain や TLS 暗号化を無効にした external リスナーを OAuth 2.0 で使用すると、ネットワークの盗聴やトークンの盗難による不正アクセスの脆弱性が生じるため、使用しないことをお勧めします。

external リスナーを type: oauth で設定し、セキュアなトランスポート層がクライアントと通信するようにします。

OAuth 2.0 の外部リスナーとの使用

```
# ...
listeners:
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: oauth
    #...
```

tls プロパティーはデフォルトで false に設定されているため、有効にする必要があります。

認証のタイプを OAuth 2.0 として定義した場合、検証のタイプに基づいて、 高速のローカル JWT 検証 または イントロスペクションエンドポイントを使用したトークンの検証 のいずれかとして、設定を追加します。

説明や例を用いてリスナー向けに OAuth 2.0 を設定する手順は、「Kafka ブローカーの OAuth 2.0 サポートの設定」を参照してください。

5.4.2.3. 高速なローカル JWT トークン検証の設定

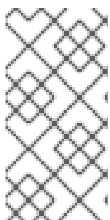
高速なローカル JWT トークンの検証では、JWTトークンの署名がローカルでチェックされます。

ローカルチェックでは、トークンに対して以下が確認されます。

- アクセストークンに Bearer の (typ) 要求値が含まれ、トークンがタイプに準拠することを確認します。
- 有効であるか (期限切れでない) を確認します。
- トークンに validIssuerURI と一致する発行元があることを確認します。

リスナーの設定時に validIssuerURI 属性を指定することで、認証サーバーから発行されていないトークンは拒否されます。

高速のローカル JWT トークン検証の実行中に、承認サーバーの通信は必要はありません。OAuth 2.0 の承認サーバーによって公開されるエンドポイントの jwksEndpointUri 属性を指定して、高速のローカル JWT トークン検証をアクティベートします。エンドポイントには、署名済み JWT トークンの検証に使用される公開鍵が含まれます。これらは、Kafka クライアントによってクレデンシャルとして送信されます。



注記

承認サーバーとの通信はすべて TLS による暗号化を使用して実行する必要があります。

証明書トラストストアを AMQ Streams プロジェクト namespace の OpenShift シークレットとして設定し、tlsTrustedCertificates 属性を使用してトラストストアファイルが含まれる OpenShift シークレットを示すことができます。

JWT トークンからユーザー名を適切に取得するため、userNameClaim の設定を検討してくださ

い。Kafka ACL 承認を使用する場合は、認証中にユーザー名でユーザーを特定する必要があります。JWT トークンの sub 要求は、通常は一意な ID でユーザー名ではありません。

高速なローカル JWT トークン検証の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    #...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          jwksEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-
connect/certs>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
    #...
```

5.4.2.4. OAuth 2.0 イントロスペクションエンドポイントの設定

OAuth 2.0 のイントロスペクションエンドポイントを使用したトークンの検証では、受信したアクセストークンは不透明として対処されます。Kafka ブローカーは、アクセストークンをイントロスペクションエンドポイントに送信します。このエンドポイントは、検証に必要なトークン情報を応答として返します。ここで重要なのは、特定のアクセストークンが有効である場合は最新情報を返すことで、トークンの有効期限に関する情報も返します。

OAuth 2.0 のイントロスペクションベースの検証を設定するには、高速のローカル JWT トークン検証に指定された **jwksEndpointUri** 属性ではなく、**introspectionEndpointUri** 属性を指定します。通常、イントロスペクションエンドポイントは保護されているため、承認サーバーに応じて **clientId** および **clientSecret** を指定する必要があります。

イントロスペクションエンドポイントの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          clientId: kafka-broker
          clientSecret:
            secretName: my-cluster-oauth
            key: clientSecret
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          introspectionEndpointUri: <https://<auth-server-
address>/auth/realms/tls/protocol/openid-connect/token/introspect>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
```

5.4.3. Kafka ブローカーの再認証の設定

KafkaクライアントとKafkaブローカー間のOAuth 2.0セッションにKafka session re-authentication を使用するように、oauth リスナーを設定できます。このメカニズムは、定義された期間後に、クライアントとブローカー間の認証されたセッションを期限切れにします。セッションの有効期限が切れると、クライアントは既存のコネクションを破棄せずに再使用して、新しいセッションを即座に開始します。

セッションの再認証はデフォルトで無効になっています。これを有効にするには、oauth リスナー設定で maxSecondsWithoutReauthentication の時間値を設定します。OAUTHBEARER および PLAIN 認証では、同じプロパティーを使用してセッションの再認証が設定されます。設定例については、「Kafka ブローカーの OAuth 2.0 サポートの設定」 を参照してください。

セッションの再認証は、クライアントによって使用される Kafka クライアントライブラリーによってサポートされる必要があります。

セッションの再認証は、高速ローカル JWT またはイントロスペクションエンドポイントのトークン検証と使用できます。

クライアントの再認証

ブローカーの認証されたセッションが期限切れになると、クライアントは接続を切断せずに新しい有効なアクセストークンをブローカーに送信し、既存のセッションを再認証する必要があります。

トークンの検証に成功すると、既存の接続を使用して新しいクライアントセッションが開始されます。クライアントが再認証に失敗した場合、さらにメッセージを送受信しようとすると、ブローカーは接続を閉じます。ブローカーで再認証メカニズムが有効になっていると、Kafka クライアントライブラリー 2.2 以降を使用する Java クライアントが自動的に再認証されます。

更新トークンが使用される場合、セッションの再認証は更新トークンにも適用されます。セッションが期限切れになると、クライアントは更新トークンを使用してアクセストークンを更新します。その後、クライアントは新しいアクセストークンを使用して既存のセッションに再認証されます。

OAUTHBEARER および PLAIN のセッションの有効期限

セッションの再認証が設定されている場合、OAUTHBEARER と PLAIN 認証ではセッションの有効期限は異なります。

クライアント ID とシークレットによる方法を使用する OAUTHBEARER および PLAIN の場合:

- ブローカーの認証されたセッションは、設定された maxSecondsWithoutReauthentication で期限切れになります。
- アクセストークンが設定期間前に期限切れになると、セッションは設定期間前に期限切れになります。

有効期間の長いアクセストークンによる方法を使用する PLAIN の場合:

- ブローカーの認証されたセッションは、設定された maxSecondsWithoutReauthentication で期限切れになります。
- アクセストークンが設定期間前に期限切れになると、再認証に失敗します。セッションの再認証は試行されますが、PLAIN にはトークンを更新するメカニズムがありません。

maxSecondsWithoutReauthentication が 設定されていない 場合、OAUTHBEARER および

PLAIN クライアントは、再認証しなくてもブローカーへの接続を無限に保持できます。認証されたセッションは、アクセストークンの期限が切れても終了しません。ただし、keycloak 承認を使用したり、カスタムオーソライザーをインストールして、承認を設定する場合に考慮できます。

関連情報

- 「OAuth 2.0 Kafka ブローカーの設定」
- 「Kafka ブローカーの OAuth 2.0 サポートの設定」
- KafkaListenerAuthenticationOAuth スキーマ参照
- KIP-368

5.4.4. OAuth 2.0 Kafka クライアントの設定

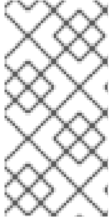
Kafka クライアントは以下のいずれかで設定されます。

- 承認サーバーから有効なアクセストークンを取得するために必要なクレデンシャル (クライアント ID およびシークレット)。
- 承認サーバーから提供されたツールを使用して取得された、有効期限の長い有効なアクセストークンまたは更新トークン。

アクセストークンは、Kafka ブローカーに送信される唯一の情報です。アクセストークンを取得するために承認サーバーでの認証に使用されるクレデンシャルは、ブローカーに送信されません。

クライアントによるアクセストークンの取得後、承認サーバーと通信する必要はありません。

クライアント ID とシークレットを使用した認証が最も簡単です。有効期間の長いアクセストークンまたは更新トークンを使用すると、承認サーバーツールに追加の依存関係があるため、より複雑になります。



注記

有効期間が長いアクセストークンを使用している場合は、承認サーバーでクライアントを設定し、トークンの最大有効期間を長くする必要があります。

Kafka クライアントが直接アクセストークンで設定されていない場合、クライアントは承認サーバーと通信して Kafka セッションの開始中にアクセストークンのクレデンシャルを交換します。Kafka クライアントは以下のいずれかを交換します。

- クライアント ID およびシークレット
- クライアント ID、更新トークン、および (任意の) シークレット

5.4.5. OAuth 2.0 のクライアント認証フロー

ここでは、Kafka セッションの開始時における Kafka クライアント、Kafka ブローカー、および承認ブローカー間の通信フローを説明および可視化します。フローは、クライアントとサーバーの設定によって異なります。

Kafka クライアントがアクセストークンをクレデンシャルとして Kafka ブローカーに送信する場合、トークンを検証する必要があります。

使用する承認サーバーや利用可能な設定オプションによっては、以下の使用が適している場合があります。

- 承認サーバーと通信しない、JWT の署名確認およびローカルトークンのイントロスペクションをベースとした高速なローカルトークン検証。
- 承認サーバーによって提供される OAuth 2.0 のイントロスペクションエンドポイント。

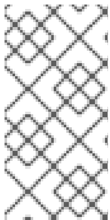
高速のローカルトークン検証を使用するには、トークンでの署名検証に使用される公開証明書のある JWKS エンドポイントを提供する承認サーバーが必要になります。

この他に、承認サーバーで OAuth 2.0 のイントロスペクションエンドポイントを使用することもできます。新しい Kafka ブローカー接続が確立されるたびに、ブローカーはクライアントから受け取った

アクセストークンを承認サーバーに渡し、応答を確認してトークンが有効であるかどうかを確認します。

Kafka クライアントのクレデンシャルは以下に対して設定することもできます。

- 以前に生成された有効期間の長いアクセストークンを使用した直接ローカルアクセス。
- 新しいアクセストークンの発行についての承認サーバーとの通信。



注記

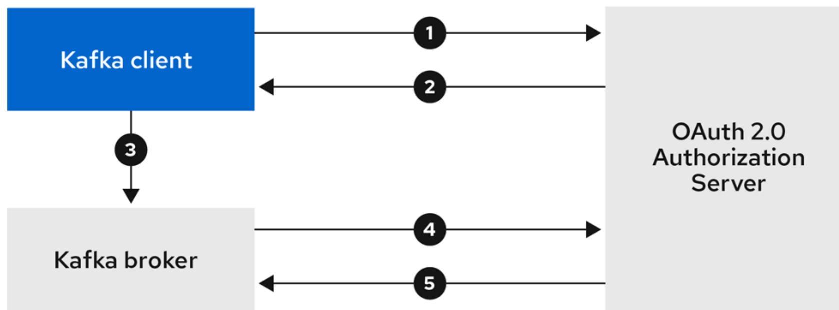
承認サーバーは不透明なアクセストークンの使用のみを許可する可能性があり、この場合はローカルトークンの検証は不可能です。

5.4.5.1. クライアント認証フローの例

Kafka クライアントおよびブローカーが以下に設定されている場合の、Kafka セッション認証中のコミュニケーションフローを確認できます。

- クライアントではクライアント ID とシークレットが使用され、ブローカーによって検証が承認サーバーに委譲される場合。
- クライアントではクライアント ID およびシークレットが使用され、ブローカーによって高速のローカルトークン検証が実行される場合。
- クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって検証が承認サーバーに委譲される場合。
- クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって高速のローカル検証が実行される場合。

クライアントではクライアント ID とシークレットが使用され、ブローカーによって検証が承認サーバーに委譲される場合



1. Kafka クライアントは承認サーバーからアクセストークンを要求します。これにはクライアント ID とシークレットを使用し、任意で更新トークンも使用します。

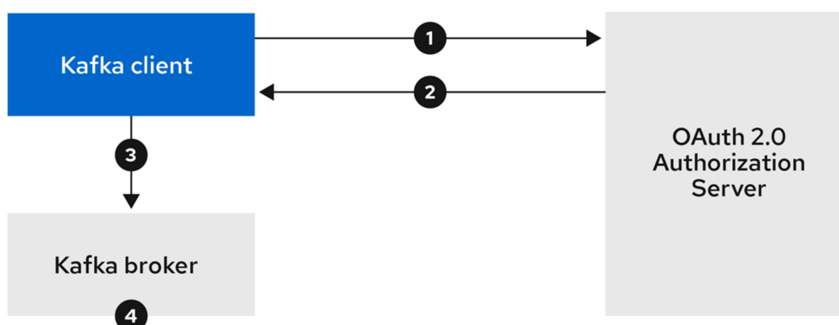
2. 承認サーバーによって新しいアクセストークンが生成されます。

3. Kafka クライアントは SASL OAUTHBEARER メカニズムを使用してアクセストークンを渡し、Kafka ブローカーの認証を行います。

4. Kafka ブローカーは、独自のクライアント ID およびシークレットを使用して、承認サーバーでトークンイントロスペクションエンドポイントを呼び出し、アクセストークンを検証します。

5. トークンが有効な場合は、Kafka クライアントセッションが確立されます。

クライアントではクライアント ID およびシークレットが使用され、ブローカーによって高速のローカルトークン検証が実行される場合



1.

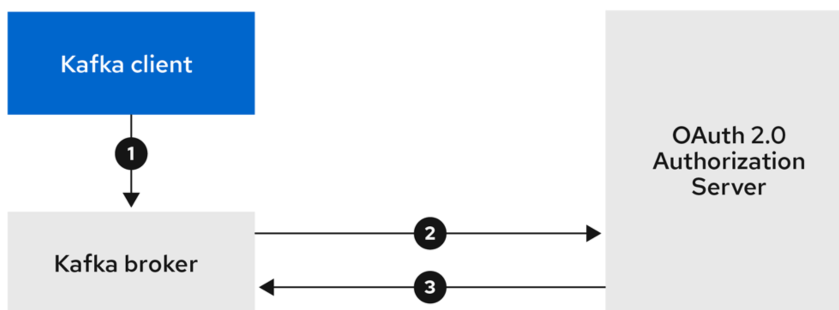
Kafka クライアントは、トークンエンドポイントから承認サーバーの認証を行います。これにはクライアント ID とシークレットが使用され、任意で更新トークンも使用されます。

2. **承認サーバーによって新しいアクセストークンが生成されます。**

3. **Kafka クライアントは SASL OAUTHBEARER メカニズムを使用してアクセストークンを渡し、Kafka ブローカーの認証を行います。**

4. **Kafka ブローカーは、JWT トークン署名チェックおよびローカルトークンイントロスペクションを使用して、ローカルでアクセストークンを検証します。**

クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって検証が承認サーバーに委譲される場合



1. **Kafka クライアントは、SASL OAUTHBEARER メカニズムを使用して有効期限の長いアクセストークンを渡し、Kafka ブローカーの認証を行います。**

2. **Kafka ブローカーは、独自のクライアント ID およびシークレットを使用して、承認サーバーでトークンイントロスペクションエンドポイントを呼び出し、アクセストークンを検証します。**

3. **トークンが有効な場合は、Kafka クライアントセッションが確立されます。**

クライアントでは有効期限の長いアクセストークンが使用され、ブローカーによって高速のローカル検証が実行される場合

1. Kafka クライアントは、SASL OAUTHBEARER メカニズムを使用して有効期限の長いアクセストークンを渡し、Kafka ブローカーの認証を行います。

2. Kafka ブローカーは、JWT トークン署名チェックおよびローカルトークンイントロスペクションを使用して、ローカルでアクセストークンを検証します。

警告

トークンが取り消された場合に承認サーバーとのチェックが行われないため、高速のローカル JWT トークン署名の検証は有効期限の短いトークンにのみ適しています。トークンの有効期限はトークンに書き込まれますが、失効はいつでも発生する可能性があるため、承認サーバーと通信せずに対応することはできません。発行されたトークンはすべて期限切れになるまで有効とみなされます。

5.4.6. OAuth 2.0 認証の設定

OAuth 2.0 は、Kafka クライアントと AMQ Streams コンポーネントとの対話に使用されます。

AMQ Streams に OAuth 2.0 を使用するには、以下を行う必要があります。

1. 承認サーバーをデプロイし、AMQ Streams と統合するためにそのデプロイメントを設定します。

2. OAuth 2.0 を使用するよう設定された Kafka ブローカーリスナーで Kafka クラスターをデプロイまたは更新します。

3. OAuth 2.0 を使用するように Java ベースの Kafka クライアントを更新します。

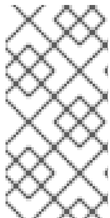
4. OAuth 2.0 を使用するように Kafka コンポーネントクライアントを更新します。

5.4.6.1. OAuth 2.0 承認サーバーとしての Red Hat Single Sign-On の設定

この手順では、Red Hat Single Sign-On を承認サーバーとしてデプロイし、AMQ Streams と統合するための設定方法を説明します。

承認サーバーは、一元的な認証および承認の他、ユーザー、クライアント、およびパーミッションの一元管理を実現します。Red Hat Single Sign-On にはレルムの概念があります。レルム はユーザー、クライアント、パーミッション、およびその他の設定の個別のセットを表します。デフォルトのマスターレルム を使用できますが、新しいレルムを作成することもできます。各レルムは独自の OAuth 2.0 エンドポイントを公開します。そのため、アプリケーションクライアントとアプリケーションサーバーはすべて同じレルムを使用する必要があります。

AMQ Streams で OAuth 2.0 を使用するには、Red Hat Single Sign-On のデプロイメントを使用して認証レルムを作成および管理します。



注記

Red Hat Single Sign-On がすでにデプロイされている場合は、デプロイメントの手順を省略して、現在のデプロイメントを使用できます。

作業を開始する前に

Red Hat Single Sign-On を使用するための知識が必要です。

デプロイメントおよび管理の手順は、以下を参照してください。

- Red Hat Single Sign-On for OpenShift
- Server Administration Guide

前提条件

- AMQ Streams および Kafka が稼働している必要があります。

Red Hat Single Sign-On デプロイメントに関する条件:

- 「Red Hat Single Sign-On でサポートされる構成」を確認しておく必要があります。
- インストールには、system:admin などの cluster-admin ロールを持つユーザーが必要です。

手順

1. Red Hat Single Sign-On を OpenShift クラスターにデプロイします。

   OpenShift Web コンソールでデプロイメントの進捗を確認します。

2. Red Hat Single Sign-On の Admin Console にログインし、AMQ Streams の OAuth 2.0 ポリシーを作成します。

   ログインの詳細は、Red Hat Single Sign-On のデプロイ時に提供されます。

3. レルムを作成し、有効にします。

   既存のマスターレルムを使用できます。

4. 必要に応じて、レルムのセッションおよびトークンのタイムアウトを調整します。

5. kafka-broker というクライアントを作成します。

6. Settings タブで以下を設定します。

- Access Type を Confidential に設定します。

- Standard Flow Enabled を OFF に設定し、このクライアントからの Web ログインを無効にします。

- Service Accounts Enabled を ON に設定し、このクライアントが独自の名前で認証できるようにします。

7. 続行する前に Save クリックします。

8. Credentials タブにある、AMQ Streams の Kafka クラスター設定で使用するシークレットを書き留めておきます。

9. Kafka ブローカーに接続するすべてのアプリケーションクライアントに対して、このクライアント作成手順を繰り返し行います。

   新しいクライアントごとに定義を作成します。

   設定では、名前をクライアント ID として使用します。

次のステップ

承認サーバーのデプロイおよび設定後に、Kafka ブローカーが OAuth 2.0 を使用するように設定します。

5.4.6.2. Kafka ブローカーの OAuth 2.0 サポートの設定

この手順では、ブローカーリスナーが承認サーバーを使用して OAuth 2.0 認証を使用するように、Kafka ブローカーを設定する方法について説明します。

TLS リスナーを設定して、暗号化されたインターフェースで OAuth 2.0 を使用することが推奨されます。プレーンリスナーは推奨されません。

承認サーバーが信頼できる CA によって署名された証明書を使用し、OAuth 2.0 サーバーのホスト名と一致する場合、TLS 接続はデフォルト設定を使用して動作します。それ以外の場合は、プローバー

証明書でトラストストアを設定するか、証明書のホスト名の検証を無効にする必要があります。

Kafka ブローカーの設定する場合、新たに接続された Kafka クライアントの OAuth 2.0 認証中にアクセストークンを検証するために使用されるメカニズムには、以下の 2 つのオプションがあります。

- 高速なローカル JWT トークン検証の設定
- イントロスペクションエンドポイントを使用したトークン検証の設定

作業を開始する前の注意事項

Kafka ブローカーリスナーの OAuth 2.0 認証の設定に関する詳細は、以下を参照してください。

- KafkaListenerAuthenticationOAuth スキーマ参照
- Kafka へのアクセス管理

前提条件

- AMQ Streams および Kafka が稼働している必要があります。
- OAuth 2.0 の承認サーバーがデプロイされている必要があります。

手順

1. エディターで、Kafka リソースの Kafka ブローカー設定 (Kafka.spec.kafka) を更新します。

```
oc edit kafka my-cluster
```

2. Kafka ブローカーの listeners 設定を行います。

   各タイプのリスナーは独立しているため、同じ設定にする必要はありません。

以下は、外部リスナーに設定された設定オプションの例になります。

例 1: 高速なローカル JWT トークン検証の設定

```
#...
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth 1
    validIssuerUri: <https://<auth-server-address>/auth/realms/external> 2
    jwksEndpointUri: <https://<auth-server-
address>/auth/realms/external/protocol/openid-connect/certs> 3
    userNameClaim: preferred_username 4
    maxSecondsWithoutReauthentication: 3600 5
    tlsTrustedCertificates: 6
    - secretName: oauth-server-cert
      certificate: ca.crt
    disableTlsHostnameVerification: true 7
    jwksExpirySeconds: 360 8
    jwksRefreshSeconds: 300 9
    jwksMinRefreshPauseSeconds: 1 10
```

1. oauth に設定されたリスナータイプ。

2. 認証に使用されるトークン発行者の URI。

3. ローカルの JWT 検証に使用される JWKS 証明書エンドポイントの URI。

4. トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される principal です。userNameClaim の値は、使用される認証フローと承認サーバーによって異なります。

5

(任意設定): セッションの有効期限がアクセストークンと同じ期間になるよう強制する Kafka の再認証メカニズムを有効にします。指定された値がアクセストークンの有効期限が切れるまでの残り時間未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

6

(任意設定): 承認サーバーへの TLS 接続用の信用できる証明書。

7

(任意設定): TLS ホスト名の検証を無効にします。デフォルトは false です。

8

JWKS 証明書が期限切れになる前に有効であるとみなされる期間。デフォルトは 360 秒です。デフォルトよりも長い時間を指定する場合は、無効になった証明書へのアクセスが許可されるリスクを考慮してください。

9

JWKS 証明書を更新する間隔。この間隔は、有効期間よりも 60 秒以上短くする必要があります。デフォルトは 300 秒です。

10

JWKS 公開鍵の更新が連続して試行される間隔の最小一時停止時間 (秒単位)。不明な署名キーが検出されると、JWKS キーの更新は、最後に更新を試みてから少なくとも指定された期間は一時停止し、通常の定期スケジュール以外でスケジュールされます。キーの更新は指数バックオフ（指数バックオフ）のルールに従い、jwksRefreshSeconds に到達するまで、一時停止を増やして失敗した更新を再試行します。デフォルト値は 1 です。

例 2: イントロスペクションエンドポイントを使用したトークンの検証の設定

```
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth
    validIssuerUri: <https://<auth-server-address>/auth/realms/external>
    introspectionEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-connect/token/introspect> 1
```

```
clientId: kafka-broker (2)
clientSecret: (3)
  secretName: my-cluster-oauth
  key: clientSecret
userNameClaim: preferred_username (4)
maxSecondsWithoutReauthentication: 3600 (5)
```

(1) トークンイントロスペクションエンドポイントの URI。

(2) クライアントを識別するためのクライアント ID。

(3) 認証にはクライアントシークレットとクライアント ID が使用されます。

(4) トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される principal です。userNameClaim の値は、使用される承認サーバーによって異なります。

(5) (任意設定): セッションの有効期限がアクセストークンと同じ期間になるよう強制する Kafka の再認証メカニズムを有効にします。指定された値がアクセストークンの有効期限が切れるまでの残り時間未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

OAuth 2.0 認証の適用方法や、承認サーバーのタイプによっては、追加 (任意) の設定を使用できます。

```
# ...
authentication:
  type: oauth
  # ...
  checkIssuer: false (1)
  checkAudience: true (2)
  fallbackUserNameClaim: client_id (3)
```

```
    fallbackUserNamePrefix: client-account- (4)
    validTokenType: bearer (5)
    userInfoEndpointUri: https://OAUTH-SERVER-
ADDRESS/auth/realms/external/protocol/openid-connect/userinfo (6)
    enableOauthBearer: false (7)
    enablePlain: true (8)
    tokenEndpointUri: https://OAUTH-SERVER-
ADDRESS/auth/realms/external/protocol/openid-connect/token (9)
    customClaimCheck: "@.custom == 'custom-value'" (10)
    clientAudience: AUDIENCE (11)
    clientScope: SCOPE (12)
```

1. 承認サーバーが iss クレームを提供しない場合は、発行者チェックを行うことができません。このような場合、checkIssuer を false に設定し、validIssuerUri を指定しないようにします。デフォルトは true です。

2. オーソリゼーションサーバーが aud（オーディエンス）クレームを提供していて、オーディエンスチェックを実施したい場合は、checkAudience を true に設定します。オーディエンスチェックによって、トークンの目的の受信者が特定されます。これにより、Kafka ブローカーは aud 要求に clientId を持たないトークンを拒否します。デフォルトは false です。

3. 承認サーバーは、通常ユーザーとクライアントの両方を識別する単一の属性を提供しない場合があります。クライアントが独自の名前で認証される場合、サーバーによってクライアント ID が提供されることがあります。更新トークンまたはアクセストークンを取得するために、ユーザー名およびパスワードを使用してユーザーが認証される場合、サーバーによってクライアント ID の他に ユーザー名 が提供されることがあります。プライマリーユーザー ID 属性が使用できない場合は、このフォールバックオプションで、使用するユーザー名クレーム (属性) を指定します。

4. fallbackUserNameClaim が適用される場合、ユーザー名クレームの値とフォールバックユーザー名クレームの値が競合しないようにする必要もあることがあります。producer というクライアントが存在し、producer という通常ユーザーも存在する場合について考えてみましょう。この 2 つを区別するには、このプロパティーを使用してクライアントのユーザー ID に接頭辞を追加します。

5. (introspectionEndpointUri を使用する場合のみ該当): 使用している認証サーバーによっては、イントロスペクションエンドポイントによってトークンタイプ属性が返されるかどうかは分からず、異なる値が含まれることがあります。イントロスペクションエンドポイントからの応答に含まれなければならない有効なトークンタイプ値を指定できます。

6

(introspectionEndpointUri を使用する場合のみ該当): イントロスペクションエンドポイントの応答に識別可能な情報が含まれないように、承認サーバーが設定または実装されることがあります。ユーザー ID を取得するには、userinfo エンドポイントの URI をフォールバックとして設定します。userNameClaim、fallbackUserNameClaim、および fallbackUserNamePrefix の設定が userinfo エンドポイントの応答に適用されます。

7

これを false に設定してリスナーで OAUTHBEARER メカニズムを無効にします。PLAIN または OAUTHBEARER のいずれかを有効にする必要があります。デフォルトは true です。

8

リスナーで PLAIN 認証を有効にするには、true に設定します。これは、すべてのプラットフォームのすべてのクライアントでサポートされています。Kafkaクライアントは、PLAINメカニズムを有効にし、username と password を設定する必要があります。PLAINは、OAuth アクセストークン、または OAuth のclientId と secret（クライアントの認証情報）を使って認証することができます。動作は、tokenEndpointUri が指定されているかどうかによってさらに制御されます。デフォルトは false です。tokenEndpointUri が指定され、クライアントが password を文字列$accessToken:で始まるように設定した場合、サーバーはパスワードをアクセストークンと解釈し、username をアカウントのユーザー名と解釈します。それ以外の場合は、username が clientId、password が client secret と解釈され、ブローカはこれを使ってクライアント名のアクセストークンを取得します。tokenEndpointUri が指定されていない場合、password は常にアクセストークンとして解釈され、ユーザー名は常にアカウント username として解釈されます。これは、トークンから抽出されるプリンシパル ID と一致する必要があります。これは no-client-credentials モードと呼ばれます。クライアントはアクセストークンを常に単独で取得する必要があり、clientId および secret を使用できません。

9

前述のように clientId と secret を username と password として渡してクライアントを認証できるようにするためのPLAIN機構の追加設定です。指定のない場合、クライアントはアクセストークンを password パラメーターとして渡すことで PLAIN で認証できます。

10

これを JsonPath フィルタークエリーに設定すると、検証中に追加のカスタムルールを JWT アクセストークンに適用できます。アクセストークンに必要なデータが含まれていないと拒否されます。introspectionEndpointUri を使用する場合、カスタムチェックはイントロスペクションエンドポイントの応答 JSON に適用されます。

11

（オプション）トークンエンドポイントに渡される audience パラメーター。オーディエンスは、ブローカー間認証用にアクセストークンを取得する場合に使用されます。また、clientId と secret を使った PLAIN クライアント認証の上にある OAuth 2.0 のクラ

イアント名にも使われています。これは、承認サーバーに応じて、トークンの取得機能とトークンの内容のみに影響します。リスナーによるトークン検証ルールには影響しません。

12

（オプション） scope パラメーターがトークンエンドポイントに渡されます。スコープは、ブローカー間認証用にアクセストークンを取得する場合に使用されます。また、clientId と secret を使った PLAIN クライアント認証の上にある OAuth 2.0 のクライアント名にも使われています。これは、承認サーバーに応じて、トークンの取得機能とトークンの内容のみに影響します。リスナーによるトークン検証ルールには影響しません。

3. エディターを保存して終了し、ローリングアップデートの完了を待ちます。

4. 更新をログで確認するか、または Pod 状態の遷移を監視して確認します。

```
oc logs -f ${POD_NAME} -c ${CONTAINER_NAME}
oc get pod -w
```

ローリングアップデートによって、ブローカーが OAuth 2.0 認証を使用するように設定されます。

次のステップ

- OAuth 2.0 を使用するように Kafka クライアントを設定します。

5.4.6.3. OAuth 2.0 を使用するよう Kafka Java クライアントを設定

この手順では、Kafka ブローカーとの対話に OAuth 2.0 を使用するように Kafka プロデューサーおよびコンシューマー API を設定する方法を説明します。

クライアントコールバックプラグインを pom.xml ファイルに追加し、システムプロパティーを設定します。

前提条件

- AMQ Streams および Kafka が稼働している必要があります。

- OAuth 2.0 承認サーバーがデプロイされ、Kafka ブローカーへの OAuth のアクセスが設定されている必要があります。

- Kafka ブローカーが OAuth 2.0 に対して設定されている必要があります。

手順

1. OAuth 2.0 サポートのあるクライアントライブラリーを Kafka クライアントの pom.xml ファイルに追加します。

```
<dependency>
 <groupId>io.strimzi</groupId>
 <artifactId>kafka-oauth-client</artifactId>
 <version>{oauth-version}</version>
</dependency>
```

2. コールバックのシステムプロパティーを設定します。

   以下は例になります。

```
System.setProperty(ClientConfig.OAUTH_TOKEN_ENDPOINT_URI, “https://<auth-server-address>/auth/realms/master/protocol/openid-connect/token”); (1)
System.setProperty(ClientConfig.OAUTH_CLIENT_ID, "<client-name>"); (2)
System.setProperty(ClientConfig.OAUTH_CLIENT_SECRET, "<client-secret>"); (3)
```

(1) 承認サーバーのトークンエンドポイントの URI です。

(2) クライアント ID。承認サーバーで client を作成するときに使用される名前です。

(3) 承認サーバーで client を作成するときに作成されるクライアントシークレット。

3. Kafka クライアント設定の TLS で暗号化された接続で SASL OAUTHBEARER メカニズムを有効にします。

以下は例になります。

```
props.put("sasl.jaas.config",
"org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required;");
props.put("security.protocol", "SASL_SSL"); (1)
props.put("sasl.mechanism", "OAUTHBEARER");
props.put("sasl.login.callback.handler.class",
"io.strimzi.kafka.oauth.client.JaasClientOauthLoginCallbackHandler");
```

1. この例では、TLS 接続で SASL_SSL を使用します。暗号化されていない接続では SASL_PLAINTEXT を使用します。

4. Kafka クライアントが Kafka ブローカーにアクセスできることを確認します。

次のステップ

- OAuth 2.0 を使用するように Kafka コンポーネントを設定します。

5.4.6.4. Kafka コンポーネントの OAuth 2.0 の設定

この手順では、承認サーバーを使用して OAuth 2.0 認証を使用するように Kafka コンポーネントを設定する方法を説明します。

以下の認証を設定できます。

- Kafka Connect
- Kafka MirrorMaker
- Kafka Bridge

この手順では、Kafka コンポーネントと承認サーバーは同じサーバーで稼働しています。

作業を開始する前の注意事項

Kafka コンポーネントの OAuth 2.0 認証の設定に関する詳細は、以下を参照してください。

- KafkaClientAuthenticationOAuth スキーマ参照

前提条件

- AMQ Streams および Kafka が稼働している必要があります。
- OAuth 2.0 承認サーバーがデプロイされ、Kafka ブローカーへの OAuth のアクセスが設定されている必要があります。
- Kafka ブローカーが OAuth 2.0 に対して設定されている必要があります。

手順

1. クライアントシークレットを作成し、これを環境変数としてコンポーネントにマウントします。

   以下は、Kafka Bridge の Secret を作成する例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Secret
metadata:
 name: my-bridge-oauth
type: Opaque
data:
 clientSecret: MGQ1OTRmMzYtZTllZS00MDY2LWI5OGEtMTM5MzM2NjdlZjQw (1)
```

   (1) clientSecret キーは base64 形式である必要があります。

2. Kafka コンポーネントのリソースを作成または編集し、OAuth 2.0 認証が認証プロパティーに設定されるようにします。

   OAuth 2.0 認証では、以下を使用できます。

- クライアント ID およびシークレット
- クライアント ID および更新トークン
- アクセストークン
- TLS

KafkaClientAuthenticationOAuth スキーマ参照は、それぞれの例を提供します。

以下は、クライアント ID、シークレット、および TLS を使用して OAuth 2.0 が Kafka Bridge クライアントに割り当てられる例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: oauth (1)
    tokenEndpointUri: https://<auth-server-
address>/auth/realms/master/protocol/openid-connect/token (2)
    clientId: kafka-bridge
    clientSecret:
      secretName: my-bridge-oauth
      key: clientSecret
    tlsTrustedCertificates: (3)
    - secretName: oauth-server-cert
      certificate: tls.crt
```

(1) oauth に設定された認証タイプ。

(2) 認証用のトークンエンドポイントの URI。

(3)

承認サーバーへの TLS 接続用の信用できる証明書。

OAuth 2.0 認証の適用方法や、承認サーバーのタイプによって、使用できる追加の設定オプションがあります。

```
# ...
spec:
  # ...
  authentication:
    # ...
    disableTlsHostnameVerification: true (1)
    checkAccessTokenType: false (2)
    accessTokenIsJwt: false (3)
    scope: any (4)
    audience: kafka (5)
```

1. (任意設定): TLS ホスト名の検証を無効にします。デフォルトは false です。

2. 承認サーバーによって、JWT トークン内部で typ (タイプ) 要求が返されない場合は、checkAccessTokenType: false を適用するとトークンタイプがチェックされず次に進むことができます。デフォルトは true です。

3. 不透明なトークンを使用している場合、アクセストークンが JWT トークンとして処理されないように accessTokenIsJwt: false を適用することができます。

4. (任意設定): トークンエンドポイントからトークンを要求するための scope。認証サーバーでは、クライアントによるスコープの指定が必要になることがあります。この場合では any になります。

5. （オプション）トークンエンドポイントからトークンを要求するための audience。認証サーバーでは、クライアントによるオーディエンスの指定が必要になることがあります。今回の場合は kafka です。

3. Kafka リソースのデプロイメントに変更を適用します。

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

4. 更新をログで確認するか、または Pod 状態の遷移を監視して確認します。

```
oc logs -f ${POD_NAME} -c ${CONTAINER_NAME}
oc get pod -w
```

ローリングアップデートでは、OAuth 2.0 認証を使用して Kafka ブローカーと対話するコンポーネントが設定されます。

5.5. OAUTH 2.0 トークンベース承認の使用

トークンベースの認証に OAuth 2.0 と Red Hat Single Sign-On を使用している場合、Red Hat Single Sign-On を使用して承認ルールを設定し、Kafka ブローカーへのクライアントのアクセスを制限することもできます。認証はユーザーのアイデンティティーを確立します。承認は、そのユーザーのアクセスレベルを決定します。

AMQ Streams は、Red Hat Single Sign-On の Authorization Services による OAuth 2.0 トークンベースの承認をサポートします。これにより、セキュリティーポリシーとパーミッションの一元的な管理が可能になります。

Red Hat Single Sign-On で定義されたセキュリティーポリシーおよびパーミッションは、Kafka ブローカーのリソースへのアクセスを付与するために使用されます。ユーザーとクライアントは、Kafka ブローカーで特定のアクションを実行するためのアクセスを許可するポリシーに対して照合されます。

Kafka では、デフォルトですべてのユーザーがブローカーに完全アクセスできます。また、アクセス制御リスト(ACL)を基にして承認を設定するために AclAuthorizer プラグインが提供されます。

ZooKeeper には、 ユーザー名 を基にしてリソースへのアクセスを付与または拒否する ACL ルールが保存されます。ただし、Red Hat Single Sign-On を使用した OAuth 2.0 トークンベースの承認では、より柔軟にアクセス制御を Kafka ブローカーに実装できます。さらに、Kafka ブローカーで OAuth 2.0 の承認および ACL が使用されるように設定することができます。

その他のリソース

- OAuth 2.0 トークンベース認証の使用
- Kafka の承認
- Red Hat Single Sign-On のドキュメント

5.5.1. OAuth 2.0 の承認メカニズム

AMQ Streams の OAuth 2.0 での承認では、Red Hat Single Sign-On サーバーの Authorization Services REST エンドポイントを使用して、Red Hat Single Sign-On を使用するトークンベースの認証が拡張されます。これは、定義されたセキュリティーポリシーを特定のユーザーに適用し、そのユーザーの異なるリソースに付与されたパーミッションの一覧を提供します。ポリシーはロールとグループを使用して、パーミッションをユーザーと照合します。OAuth 2.0 の承認では、Red Hat Single Sign-On の Authorization Services から受信した、ユーザーに付与された権限のリストを基にして、権限がローカルで強制されます。

5.5.1.1. Kafka ブローカーのカスタムオーソライザー

AMQ Streams では、Red Hat Single Sign-On の オーソライザー (KeycloakRBACAuthorizer) が提供されます。Red Hat Single Sign-On によって提供される Authorization Services で Red Hat Single Sign-On REST エンドポイントを使用できるようにするには、Kafka ブローカーでカスタムオーソライザーを設定します。

オーソライザーは必要に応じて付与された権限のリストを承認サーバーから取得し、ローカルで Kafka ブローカーに承認を強制するため、クライアントの要求ごとに迅速な承認決定が行われます。

5.5.2. OAuth 2.0 承認サポートの設定

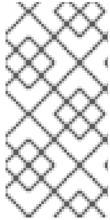
この手順では、Red Hat Single Sign-On の Authorization Services を使用して、OAuth 2.0 承認を使用するように Kafka ブローカーを設定する方法を説明します。

作業を開始する前に

特定のユーザーに必要なアクセス、または制限するアクセスについて検討してください。Red Hat Single Sign-On では、Red Hat Single Sign-On の グループ、ロール、クライアント、および ユーザー の組み合わせを使用して、アクセスを設定できます。

通常、グループは組織の部門または地理的な場所を基にしてユーザーを照合するために使用されます。また、ロールは職務を基にしてユーザーを照合するために使用されます。

Red Hat Single Sign-On を使用すると、ユーザーおよびグループを LDAP で保存できますが、クライアントおよびロールは LDAP で保存できません。ユーザーデータへのアクセスとストレージを考慮して、承認ポリシーの設定方法を選択する必要がある場合があります。



注記

スーパーユーザー は、Kafka ブローカーに実装された承認にかかわらず、常に制限なく Kafka ブローカーにアクセスできます。

前提条件

- AMQ Streams は、トークンベースの認証 に Red Hat Single Sign-On と OAuth 2.0 を使用するように設定されている必要があります。承認を設定するときに、同じ Red Hat Single Sign-On サーバーエンドポイントを使用する必要があります。
- OAuth 2.0 認証は、再認証を有効にするために maxSecondsWithoutReauthentication オプションで設定する必要があります。

手順

1. Red Hat Single Sign-On の Admin Console にアクセスするか、Red Hat Single Sign-On の Admin CLI を使用して、OAuth 2.0 認証の設定時に作成した Kafka ブローカークライアントの Authorization Services を有効にします。
2. 承認サービスを使用して、クライアントのリソース、承認スコープ、ポリシー、およびパーミッションを定義します。
3. ロールとグループをユーザーとクライアントに割り当てて、パーミッションをユーザーとクライアントにバインドします。
4. エディターで Kafka リソースの Kafka ブローカー設定 (Kafka.spec.kafka) を更新して、Kafka ブローカーで Red Hat Single Sign-On による承認が使用されるように設定します。

```
oc edit kafka my-cluster
```

5. Kafka ブローカーの kafka 設定を指定して、keycloak による承認を使用し、承認サーバー

と Red Hat Single Sign-On の Authorization Services にアクセスできるようにします。

以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    authorization:
      type: keycloak (1)
      tokenEndpointUri: <https://<auth-server-
address>/auth/realms/external/protocol/openid-connect/token> (2)
      clientId: kafka (3)
      delegateToKafkaAcls: false (4)
      disableTlsHostnameVerification: false (5)
      superUsers: (6)
      - CN=fred
      - sam
      - CN=edward
      tlsTrustedCertificates: (7)
      - secretName: oauth-server-cert
        certificate: ca.crt
      grantsRefreshPeriodSeconds: 60 (8)
      grantsRefreshPoolSize: 5 (9)
    #...
```

(1) タイプ keycloak によって Red Hat Single Sign-On の承認が有効になります。

(2) Red Hat Single Sign-On トークンエンドポイントの URI。本番環境では常に HTTP を使用してください。トークンベースの oauth 認証を設定する場合、jwksEndpointUri をローカル JWT 検証の URI として指定します。tokenEndpointUri URI のホスト名は同じである必要があります。

(3) 承認サービスが有効になっている Red Hat Single Sign-On の OAuth 2.0 クライアント定義のクライアント ID。通常、kafka が ID として使用されます。

(4) (オプション) Red Hat Single Sign-On Authorization Services ポリシーでアクセスが拒否された場合、Kafka AclAuthorizer に権限を委譲します。デフォルトは false です。

5

(任意設定): TLS ホスト名の検証を無効にします。デフォルトは false です。

6

(任意設定): 指定の スーパーユーザー。

7

(任意設定): 承認サーバーへの TLS 接続用の信用できる証明書。

8

(任意設定): 連続する付与 (Grants) 更新実行の間隔。これは、アクティブなセッションが Red Hat Single Sign-On でユーザーのパーミッション変更を検出する最大時間です。デフォルト値は 60 です。

9

(任意設定): アクティブなセッションの付与 (Grants) の更新 (並行して) に使用するスレッドの数。デフォルト値は 5 です。

6. エディターを保存して終了し、ローリングアップデートの完了を待ちます。

7. 更新をログで確認するか、または Pod 状態の遷移を監視して確認します。

```
oc logs -f ${POD_NAME} -c kafka
oc get pod -w
```

ローリングアップデートによって、ブローカーが OAuth 2.0 承認を使用するように設定されます。

8. クライアントまたは特定のロールを持つユーザーとして Kafka ブローカーにアクセスして、設定したパーミッションを検証し、必要なアクセス権限があり、付与されるべきでないアクセス権限がないことを確認します。

5.5.3. Red Hat Single Sign-On の Authorization Services でのポリシーおよびパーミッションの管理

本セクションでは、Red Hat Single Sign-On Authorization Services および Kafka によって使用される承認モデルについて説明し、各モデルの重要な概念を定義します。

Kafka にアクセスするためのパーミッションを付与するには、Red Hat Single Sign-On で OAuth クライアント仕様を作成して、Red Hat Single Sign-On Authorization Services オブジェクトを Kafka リソースにマップできます。Kafka パーミッションは、Red Hat Single Sign-On Authorization Services ルールを使用して、ユーザーアカウントまたはサービスアカウントに付与されます。

トピックの作成や一覧表示など、一般的な Kafka 操作に必要なさまざまなユーザーパーミッションの例を紹介します。

5.5.3.1. Kafka および Red Hat Single Sign-On 承認モデルの概要

Kafka および Red Hat Single Sign-On Authorization Services は、異なる承認モデルを使用します。

Kafka 承認モデル

Kafka の承認モデルはリソース型を使用します。Kafka クライアントがブローカーでアクションを実行すると、ブローカーは設定済みの KeycloakRBACAuthorizer を使用して、アクションおよびリソースタイプを基にしてクライアントのパーミッションをチェックします。

Kafka は 5 つのリソースタイプを使用してアクセスを制御します（Topic、Group、Cluster、TransactionalId、および DelegationToken ）。各リソースタイプには、利用可能なパーミッションセットがあります。

トピック

- 作成
- Write
- 読み取り
- Delete

- **Describe**
- **DescribeConfigs**
- **Alter**
- **AlterConfigs**

グループ

- **読み取り**
- **Describe**
- **Delete**

クラスター

- **作成**
- **Describe**
- **Alter**
- **DescribeConfigs**
- **AlterConfigs**

- **IdempotentWrite**
- **ClusterAction**

TransactionalId

- **Describe**
- **Write**

DelegationToken

- **Describe**

Red Hat Single Sign-On の Authorization Services モデル

Red Hat Single Sign-On の Authorization Services には、パーミッションを定義および付与するための 4 つの概念があります。これらは リソース、承認スコープ、ポリシー、および パーミッションです。

リソース

リソースは、リソースを許可されたアクションと一致するために使用されるリソース定義のセットです。リソースは、個別のトピックであったり、名前が同じプレフィックスで始まるすべてのトピックであったりします。リソース定義は、利用可能な承認スコープのセットに関連付けられます。これは、リソースで利用可能なすべてのアクションのセットを表します。多くの場合、これらのアクションのサブセットのみが実際に許可されます。

承認スコープ

承認スコープは、特定のリソース定義で利用可能なすべてのアクションのセットです。新規リソースを定義するとき、すべてのスコープのセットからスコープを追加します。

ポリシー

ポリシーは、アカウントのリストと照合するための基準を使用する承認ルールです。ポリシーは以下と一致できます。

- クライアント ID またはロールに基づくサービスアカウント
- ユーザー名、グループ、またはロールに基づくユーザーアカウント

パーミッション

パーミッションは、特定のリソース定義の承認スコープのサブセットをユーザーのセットに付与します。

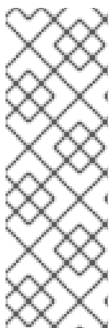
関連情報

- Kafka 承認モデル

5.5.3.2. Red Hat Single Sign-On Authorization Services の Kafka 承認モデルへのマッピング

Kafka 承認モデルは、Kafka へのアクセスを制御する Red Hat Single Sign-On ロールおよびリソースを定義するベースとして使用されます。

ユーザーアカウントまたはサービスアカウントに Kafka パーミッションを付与するには、まず Kafka ブローカーの Red Hat Single Sign-On に OAuth クライアント仕様を作成します。次に、クライアントに Red Hat Single Sign-On の Authorization Services ルールを指定します。通常、ブローカーを表す OAuth クライアントのクライアント ID は kafka です。AMQ Streamsで提供されている設定ファイルの例では、OAuthのクライアントIDとしてkafkaを使用しています。



注記

複数の Kafka クラスターがある場合は、それらすべてに単一の OAuth クライアント(kafka)を使用できます。これにより、承認ルールを定義および管理するための単一の統合されたスペースが提供されます。ただし、異なる OAuth クライアント ID（例 my-cluster-kafka または cluster-dev-kafka）を使用し、各クライアント設定内の各クラスターの承認ルールを定義することもできます。

Kafka クライアント 定義では、Red Hat Single Sign-On 管理コンソールで Authorization Enabled オプションが有効になっている必要があります。

すべてのパーミッションは、kafka クライアントのスコープ内に存在します。異なる OAuth クライアント ID で異なる Kafka クラスターを設定した場合、同じ Red Hat Single Sign-On レルムの一部であっても、それぞれに個別のパーミッションセットが必要です。

Kafka クライアントが OAUTHBEARER 認証を使用する場合、Red Hat Single Sign-On オーソライザー(KeycloakRBACAuthorizer) は現在のセッションのアクセストークンを使用して、Red Hat Single Sign-On サーバーからグラントのリストを取得します。許可を取得するために、オーソライザーは Red Hat Single Sign-On の Authorization Services ポリシーおよびパーミッションを評価します。

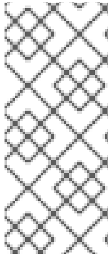
Kafka パーミッションの承認スコープ

通常、Red Hat Single Sign-On 初期設定では、承認スコープをアップロードして、各 Kafka リソースタイプで実行できるすべての可能なアクションのリストを作成します。この手順は、パーミッションを定義する前に 1 度のみ実行されます。承認スコープをアップロードする代わりに、手動で追加できます。

承認スコープには、リソースタイプに関係なく、可能なすべての Kafka パーミッションが含まれる必要があります。

- 作成
- Write
- 読み取り
- Delete
- Describe
- Alter
- DescribeConfig
- AlterConfig
- ClusterAction

- **IdempotentWrite**



注記

パーミッションが必要ない場合（例: IdempotentWrite）、承認スコープの一覧から省略できます。ただし、そのパーミッションは Kafka リソースをターゲットにすることはできません。

パーミッションチェックのリソースパターン

リソースパターンは、パーミッションチェックの実行時にターゲットリソースに対するパターンの照合に使用されます。一般的なパターン形式は **RESOURCE-TYPE:PATTERN-NAME** です。

リソースタイプは Kafka 承認モデルをミラーリングします。このパターンでは、次の 2 つの一致オプションが可能です。

- 完全一致（パターンが * で終了しない場合）
- プレフィックス一致（パターンが * で終了する）

リソースのパターン例

```
Topic:my-topic
Topic:orders-*
Group:orders-*
Cluster:*
```

さらに、一般的なパターンフォーマットは、kafka-cluster:CLUSTER-NAMEの前にコンマを付けることができ、CLUSTER-NAMEはKafkaカスタムリソースのmetadata.nameを参照します。

クラスタープレフィックスが付けられたリソースのパターン例

```
kafka-cluster:my-cluster,Topic:*
kafka-cluster:*,Group:b_*
```

kafka-cluster の接頭辞がない場合は、kafka-cluster:* とみなします。

リソースを定義するときに、リソースに関連する可能な承認スコープのリストを関連付けることができます。ターゲットリソースタイプに妥当なアクションを設定します。

任意の承認スコープを任意のリソースに追加できますが、リソースタイプでサポートされるスコープのみがアクセス制御の対象として考慮されます。

アクセスパーミッションを適用するポリシー

ポリシーは、1 つ以上のユーザーアカウントまたはサービスアカウントにパーミッションをターゲットにするために使用されます。以下がターゲットの対象になります。

- 特定のユーザーまたはサービスアカウント
- レルムロールまたはクライアントロール
- ユーザーグループ
- クライアント IP アドレスに一致する JavaScript ルール

ポリシーには一意の名前が割り当てられ、複数のリソースに対して複数の対象パーミッションを指定するために再使用できます。

アクセスを付与するためのパーミッション

詳細なパーミッションを使用して、ユーザーへのアクセスを付与するポリシー、リソース、および承認スコープをまとめます。

各パーミッションの名前によって、どのユーザーにどのパーミッションが付与されるかが明確に定義される必要があります。例えば、Dev Team B は x で始まるトピックから読むことができます。

関連情報

- Red Hat Single Sign-On の Authorization Services でパーミッションを設定する方法の詳細は、「Red Hat Single Sign-On の Authorization Services の試行」 を参照してください。

5.5.3.3. Kafka 操作に必要なパーミッションの例

以下の例は、Kafka で一般的な操作を実行するために必要なユーザーパーミッションを示しています。

トピックを作成します

トピックを作成するには、特定のトピック、または Cluster:kafka-cluster に対して Create パーミッションが必要です。

```
bin/kafka-topics.sh --create --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

トピックの一覧表示

指定のトピックでユーザーに Describe パーミッションがある場合には、トピックが一覧表示されます。

```
bin/kafka-topics.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

トピックの詳細の表示

トピックの詳細を表示するには、トピックに対して Describe および DescribeConfigs の権限が必要です。

```
bin/kafka-topics.sh --describe --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

トピックへのメッセージの生成

トピックへのメッセージを作成するには、トピックに対する **Describe** と **Write** の権限が必要です。

トピックが作成されておらず、トピックの自動生成が有効になっている場合は、トピックを作成するパーミッションが必要になります。

```
bin/kafka-console-producer.sh  --topic my-topic \
  --broker-list my-cluster-kafka-bootstrap:9092 --producer.config=/tmp/config.properties
```

トピックからのメッセージの消費

トピックからのメッセージを消費するためには、トピックに **Describe** と **Read** のパーミッションが必要です。通常、トピックからの消費は、コンシューマグループにコンシューマオフセットを格納することに依存しており、これにはコンシューマグループに対する追加の **Describe** および **Read** 権限が必要です。

マッチングには2つの **resources** が必要です。以下は例になります。

```
Topic:my-topic
Group:my-group-*
```

```
bin/kafka-console-consumer.sh --topic my-topic --group my-group-1 --from-beginning \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --consumer.config
/tmp/config.properties
```

べき等プロデューサーを使用したトピックへのメッセージの生成

トピックへの生成のためのパーミッションと同様に、追加の **IdempotentWrite** パーミッションが **Cluster** リソースに必要です。

マッチングには2つの **resources** が必要です。以下は例になります。

```
Topic:my-topic
Cluster:kafka-cluster
```

```
bin/kafka-console-producer.sh  --topic my-topic \
  --broker-list my-cluster-kafka-bootstrap:9092 --producer.config=/tmp/config.properties --
producer-property enable.idempotence=true --request-required-acks -1
```

コンシューマーグループのリスト

コンシューマーグループの一覧表示時に、ユーザーが **Describe** 権限を持っているグループのみが

返されます。また、ユーザーが Cluster:kafka-cluster に対して Describe パーミッションを持っている場合は、すべてのコンシューマーグループが返されます。

```
bin/kafka-consumer-groups.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-
config=/tmp/config.properties
```

コンシューマーグループの詳細の表示

　コンシューマグループの詳細を表示するには、グループとグループに関連するトピックに対して Describe 権限が必要です。

```
bin/kafka-consumer-groups.sh --describe --group my-group-1 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-
config=/tmp/config.properties
```

トピック設定の変更

　トピックの構成を変更するには、トピックに Describe と Alter の権限が必要です。

```
bin/kafka-topics.sh --alter --topic my-topic --partitions 2 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-
config=/tmp/config.properties
```

Kafka ブローカー設定の表示

　kafka-configs.sh を使ってブローカーの設定を取得するためには、Cluster:kafka-cluster に DescribeConfigs パーミッションが必要です。

```
bin/kafka-configs.sh --entity-type brokers --entity-name 0 --describe --all \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-
config=/tmp/config.properties
```

Kafka ブローカー設定の変更

　Kafkaブローカーの構成を変更するには、Cluster:kafka-cluster に DescribeConfigs および AlterConfigs パーミッションが必要です。

```
bin/kafka-configs --entity-type brokers --entity-name 0 --alter --add-config
log.cleaner.threads=2 \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-
config=/tmp/config.properties
```

トピックを削除します

トピックを削除するには、トピックにDescribe と Delete の権限が必要です。

```
bin/kafka-topics.sh --delete --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-
config=/tmp/config.properties
```

リードパーティションの選択

トピックパーティションのリーダー選択を実行するには、Cluster:kafka-cluster に Alter パーミッションが必要です。

```
bin/kafka-leader-election.sh --topic my-topic --partition 0 --election-type PREFERRED  /
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --admin.config /tmp/config.properties
```

パーティションの再割り当て

パーティション再割り当てファイルを生成するためには、関係するトピックに対して Describe 権限が必要です。

```
bin/kafka-reassign-partitions.sh --topics-to-move-json-file /tmp/topics-to-move.json --broker-
list "0,1" --generate \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config
/tmp/config.properties > /tmp/partition-reassignment.json
```

パーティションの再割り当てを実行するには、Cluster:kafka-cluster に対して Describe と Alter のパーミッションが必要です。また、関係するトピックには、Describeのパーミッションが必要です。

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --
execute \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config
/tmp/config.properties
```

パーティションの再割り当てを確認するには、Cluster:kafka-clusterおよび関連する各トピックに対してDescribeおよびAlterConfigsのパーミッションが必要です。

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --
verify \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config
/tmp/config.properties
```

5.5.4. Red Hat Single Sign-On の Authorization Services の試行

この例では、Red Hat Single Sign-On Authorization Services をkeycloak認証で使用する方法を説

明します。Red Hat Single Sign-On の Authorization Services を使用して、Kafka クライアントにアクセス制限を強制します。Red Hat Single Sign-On の Authorization Services では、承認スコープ、ポリシー、およびパーミッションを使用してアクセス制御をリソースに定義および適用します。

Red Hat Single Sign-On の Authorization Services REST エンドポイントは、認証されたユーザーのリソースに付与されたパーミッションの一覧を提供します。許可 (パーミッション) のリストは、Kafka クライアントによって認証されたセッションが確立された後に最初のアクションとして Red Hat Single Sign-On サーバーから取得されます。付与の変更が検出されるように、バックグラウンドで一覧が更新されます。付与は、各ユーザーセッションが迅速な承認決定を提供するために、Kafka ブローカーにてローカルでキャッシュおよび適用されます。

AMQ Streams では 、設定ファイルのサンプル が提供されます。これには、Red Hat Single Sign-On を設定するための以下のサンプルファイルが含まれます。

kafka-ephemeral-oauth-single-keycloak-authz.yaml

Red Hat Single Sign-On を使用して OAuth 2.0 トークンベースの承認に設定された Kafka カスタムリソースの例。カスタムリソースを使用して、keycloak 承認およびトークンベースの oauth 認証を使用する Kafka クラスターをデプロイできます。

kafka-authz-realm.json

サンプルグループ、ユーザー、ロール、およびクライアントで設定された Red Hat Single Sign-On レルムの例。レルムを Red Hat Single Sign-On インスタンスにインポートし、Kafka にアクセスするための詳細なパーミッションを設定できます。

Red Hat Single Sign-On で例を試す場合は、これらのファイルを使用して、本セクションの順序で説明したタスクを実行します。

1. Red Hat Single Sign-On 管理コンソールへのアクセス

2. Red Hat Single Sign-On 承認をでの Kafka クラスターのデプロイメント

3. CLI Kafka クライアントセッションの TLS 接続の準備

4. CLI Kafka クライアントセッションを使用した Kafka への承認されたアクセスの確認

認証

トークンベースの oauth 認証を設定する場合、jwksEndpointUri をローカル JWT 検証の URI として指定します。keycloak 承認を設定するとき、a tokenEndpointUri を Red Hat Single Sign-On トークンエンドポイントの URI として指定します。両方の URI のホスト名は同じである必要があります。

グループまたはロールポリシーを使用した対象パーミッション

Red Hat Single Sign-On では、サービスアカウントが有効になっている機密性の高いクライアントを、クライアント ID とシークレットを使用して、独自の名前のサーバーに対して認証できます。これは、通常、特定ユーザーのエージェント (Web サイトなど) としてではなく、独自の名前で動作するマイクロサービスに便利です。サービスアカウントには、通常のユーザーと同様にロールを割り当てることができます。ただし、グループを割り当てることはできません。そのため、サービスアカウントを使用してマイクロサービスへのパーミッションをターゲットにする場合は、グループポリシーを使用できないため、代わりにロールポリシーを使用する必要があります。逆に、ユーザー名およびパスワードを使用した認証が必要な通常のユーザーアカウントにのみ特定のパーミッションを制限する場合は、ロールポリシーではなく、グループポリシーを使用すると、副次的に実現することができます。これは、ClusterManager で始まるパーミッションの例で使用されるものです。通常、クラスター管理の実行は CLI ツールを使用して対話的に行われます。結果的に生成されるアクセストークンを使用して Kafka ブローカーに対して認証を行う前に、ユーザーのログインを要求することは妥当です。この場合、アクセストークンはクライアントアプリケーションではなく、特定のユーザーを表します。

5.5.4.1. Red Hat Single Sign-On 管理コンソールへのアクセス

Red Hat Single Sign-On を設定してから、管理コンソールに接続し、事前設定されたレルムを追加します。kafka-authz-realm.json ファイルのサンプルを使用して、レルムをインポートします。管理コンソールのレルムに定義された承認ルールを確認できます。このルールは、Red Hat Single Sign-On レルムの例を使用するよう設定された Kafka クラスターのリソースへのアクセスを許可します。

前提条件

- 実行中の OpenShift クラスター。
- 事前設定されたレルムが含まれる AMQ Streams の examples/security/keycloak-authorization/kafka-authz-realm.json ファイル。

手順

1. Red Hat Single Sign-On ドキュメントの 「Server Installation and Configuration」で説明されているように、Red Hat Single Sign-On Operator を使用して Red Hat Single Sign-On サーバーをインストールします。
2. Red Hat Single Sign-On インスタンスが実行されるまで待ちます。

3. 管理コンソールにアクセスできるように外部ホスト名を取得します。

```
NS=sso
oc get ingress keycloak -n $NS
```

この例では、Red Hat Single Sign-On サーバーが sso namespace で実行されていることを前提としています。

4. admin ユーザーのパスワードを取得します。

```
oc get -n $NS pod keycloak-0 -o yaml | less
```

パスワードはシークレットとして保存されるため、Red Hat Single Sign-On インスタンスの設定 YAML ファイルを取得して、シークレット名(secretKeyRef.name)を特定します。

5. シークレットの名前を使用して、クリアテキストのパスワードを取得します。

```
SECRET_NAME=credential-keycloak
oc get -n $NS secret $SECRET_NAME -o yaml | grep PASSWORD | awk '{print $2}' |
base64 -D
```

この例では、シークレットの名前が credential-keycloak であることを前提としています。

6. ユーザー名 admin と取得したパスワードを使用して、管理コンソールにログインします。

https://HOSTNAME を使用して OpenShift Ingress にアクセスします。

管理コンソールを使用して、サンプルレルムを Red Hat Single Sign-On にアップロードできるようになりました。

7. Add Realm をクリックして、サンプルレルムをインポートします。

8. examples/security/keycloak-authorization/kafka-authz-realm.json ファイルを追加してから Create をクリックします。

これで、管理コンソールの現在のレルムとして **kafka-authz** が含まれるようになりました。

デフォルトビューには、**Master** レルムが表示されます。

9. **Red Hat Single Sign-On** 管理コンソールで **Clients > kafka > Authorization > Settings** の順に移動し、**Decision Strategy** が **Affirmative** に設定されていることを確認します。

   肯定的な (Affirmative) ポリシーとは、クライアントが Kafka クラスターにアクセスするためには少なくとも 1 つのポリシーが満たされている必要があることを意味します。

10. **Red Hat Single Sign-On** 管理コンソールで、**Groups**、**Users**、**Roles**、および **Clients** と移動して、レルム設定を表示します。

グループ

Groups は、ユーザーグループの作成やユーザー権限の設定に使用します。グループは、名前が割り当てられたユーザーのセットです。地域、組織、または部門単位に区分するために使用されます。グループは LDAP アイデンティティープロバイダーにリンクできます。Kafka リソースにパーミッションを付与するなど、カスタム LDAP サーバー管理ユーザーインターフェースを使用して、ユーザーをグループのメンバーにすることができます。

ユーザー

Users は、ユーザーを作成するために使用されます。この例では、aliceとbobが定義されています。aliceはClusterManagerグループのメンバーであり、bobはClusterManager-my-clusterグループのメンバーです。ユーザーは LDAP アイデンティティープロバイダーに保存できます。

ロール

Rolesは、ユーザーやクライアントが特定の権限を持っていることを示すものです。ロールはグループに似た概念です。通常ロールは、組織ロールでユーザーを タグ付け するために使用され、必要なパーミッションを持ちます。ロールは LDAP アイデンティティープロバイダーに保存できません。LDAP が必須である場合は、代わりにグループを使用し、Red Hat Single Sign-On ロールをグループに追加して、ユーザーにグループを割り当てるときに対応するロールも取得するようにします。

Clients

Clientsは特定の構成を持つことができます。この例では、kafka、kafka-cli、team-a-client、team-b-clientの各クライアントが設定されています。

-

kafkaクライアントは、Kafkaブローカーがアクセストークンの検証に必要なOAuth 2.0 通信を行うために使用されます。このクライアントには、Kafka ブローカーで承認を実行するために使用される承認サービスリソース定義、ポリシー、および承認スコープも含まれます。認証設定はkafkaクライアントの Authorization タブで定義され、Settings タブでAuthorization Enabled をオンにすると表示されます。

- **kafka-cli クライアントは、アクセストークンまたは更新トークンを取得するためにユーザー名とパスワードを使用して認証するときに Kafka コマンドラインツールによって使用されるパブリッククライアントです。**

- **team-a-client および team-b-client クライアントは、特定の Kafka トピックに部分的にアクセスできるサービスを表す機密クライアントです。**

11. **Red Hat Single Sign-On 管理コンソールで、Authorization > Permissions の順に移動し、レルムに定義されたリソースおよびポリシーを使用する付与されたパーミッションを確認します。**

    たとえば、kafka クライアントには以下のパーミッションがあります。

```
Dev Team A can write to topics that start with x_ on any cluster
Dev Team B can read from topics that start with x_ on any cluster
Dev Team B can update consumer group offsets that start with x_ on any cluster
ClusterManager of my-cluster Group has full access to cluster config on my-cluster
ClusterManager of my-cluster Group has full access to consumer groups on my-cluster
ClusterManager of my-cluster Group has full access to topics on my-cluster
```

Dev Team A

Dev チーム A レルムロールは、任意のクラスターで x_ で始まるトピックに書き込みできます。これは、Topic:x_*というリソース、DescribeとWriteのスコープ、そしてDev Team Aのポリシーを組み合わせたものです。Dev Team Aポリシーは、Dev Team Aというレルムロールを持つすべてのユーザーにマッチします。

Dev Team B

Dev チーム B レルムロールは、任意のクラスターで x_ で始まるトピックから読み取ることができます。これは、Topic:x_*、 Group:x_*のリソース、DescribeとReadのスコープ、およびDev Team Bのポリシーを組み合わせたものです。Dev Team Bポリシーは、Dev Team Bというレルムロールを持つすべてのユーザーにマッチします。一致するユーザーおよびクライアントはトピックから読み取りでき、名前が x_ で始まるトピックおよびコンシューマーグループの消費されたオフセットを更新できます。

5.5.4.2. Red Hat Single Sign-On 承認をでの Kafka クラスターのデプロイメント

Red Hat Single Sign-On サーバーに接続するように設定された Kafka クラスターをデプロイします。サンプルのkafka-ephemeral-oauth-single-keycloak-authz.yamlファイルを使用して、Kafkaカスタムリソースとして Kafkaクラスタを展開します。この例では、keycloak 承認と oauth 認証を使用して単一ノードの Kafka クラスターをデプロイします。

前提条件

- Red Hat Single Sign-On 承認サーバーが OpenShift クラスターにデプロイされ、サンプルレルムでロードされている。
- Cluster Operator が OpenShift クラスターにデプロイされている。
- AMQ Streams の examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml カスタムリソース。

手順

1. デプロイした Red Hat Single Sign-On インスタンスのホスト名を使用して、Kafka ブローカーのトラストストア証明書を準備し、Red Hat Single Sign-On サーバーと通信します。

```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk
' /BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.crt
```

OpenShift Ingress はセキュアな (HTTPS) 接続の確立に使用されるため、証明書が必要です。

2. シークレットとして OpenShift に証明書をデプロイします。

```
oc create secret generic oauth-server-cert --from-file=/tmp/sso.crt -n $NS
```

3. ホスト名を環境変数として設定します。

```
SSO_HOST=SSO-HOSTNAME
```

4. サンプル Kafka クラスターを作成およびデプロイします。

```
cat examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml | sed -E 's#\${SSO_HOST}'"#$SSO_HOST#" | oc create -n $NS -f -
```

5.5.4.3. CLI Kafka クライアントセッションの TLS 接続の準備

対話型 CLI セッション用の新規 Pod を作成します。TLS 接続用の Red Hat Single Sign-On 証明書を使用してトラストストアを設定します。トラストストアは、Red Hat Single Sign-On および Kafka ブローカーに接続します。

前提条件

- Red Hat Single Sign-On 承認サーバーが OpenShift クラスターにデプロイされ、サンプルレルムでロードされている。

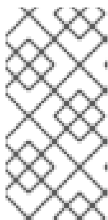
  Red Hat Single Sign-On 管理コンソールで、クライアントに割り当てられたロールが Clients > Service Account Roles に表示されることを確認します。

- Red Hat Single Sign-On に接続するように設定された Kafka クラスターが OpenShift クラスターにデプロイされている。

手順

1. AMQ Streams の Kafka イメージを使用して新しい対話型の Pod コンテナーを実行し、稼働中の Kafka ブローカーに接続します。

```
NS=sso
oc run -ti --restart=Never --image=registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1 kafka-cli -n $NS -- /bin/sh
```



注記

イメージのダウンロードの待機中に oc がタイムアウトする場合、その後の試行によって an AlreadyExists エラーが発生することがあります。

2. Pod コンテナーにアタッチします。

```
oc attach -ti kafka-cli -n $NS
```

-

3. Red Hat Single Sign-On インスタンスのホスト名を使用して、TLS を使用してクライアントコネクションの証明書を準備します。

```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk
' /BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.crt
```

4. Kafka ブローカーへの TLS 接続のトラストストアを作成します。

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias sso -storepass
$STOREPASS -import -file /tmp/sso.crt -noprompt
```

5. Kafka ブートストラップアドレスを Kafka ブローカーのホスト名および tls リスナーポート(9093)のホスト名として使用し、Kafka ブローカーの証明書を準備します。

```
KAFKA_HOST_PORT=my-cluster-kafka-bootstrap:9093
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $KAFKA_HOST_PORT 2>/dev/null |
awk ' /BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/my-cluster-
kafka.crt
```

6. Kafka ブローカーの証明書をトラストストアに追加します。

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias my-cluster-kafka -
storepass $STOREPASS -import -file /tmp/my-cluster-kafka.crt -noprompt
```

承認されたアクセスを確認するために、セッションを開いたままにします。

5.5.4.4. CLI Kafka クライアントセッションを使用した Kafka への承認されたアクセスの確認

対話型 CLI セッションを使用して、Red Hat Single Sign-On レルムを通じて適用される承認ルールを確認します。Kafka のサンプルプロデューサーおよびコンシューマークライアントを使用してチェックを適用し、異なるレベルのアクセスを持つユーザーおよびサービスアカウントでトピックを作成します。

team-a-client クライアントおよび team-b-client クライアントを使用して、承認ルールを確認しま

す。alice admin ユーザーを使用して、Kafka で追加の管理タスクを実行します。

この例で使用される AMQ Streams Kafka イメージには、Kafka プロデューサーおよびコンシューマーバイナリーが含まれます。

前提条件

- ZooKeeper および Kafka は OpenShift クラスターで実行され、メッセージを送受信できる。
- 対話型 CLI Kafka クライアントセッションが開始される。

  Apache Kafka のダウンロード。

クライアントおよび管理ユーザーの設定

1. team-a-client クライアントの認証プロパティーで Kafka 設定ファイルを準備します。

```
SSO_HOST=SSO-HOSTNAME

cat > /tmp/team-a-client.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginM
odule required \
  oauth.client.id="team-a-client" \
  oauth.client.secret="team-a-client-secret" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-
authz/protocol/openid-connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginC
allbackHandler
EOF
```

   SASL OAUTHBEARER メカニズムが使用されます。このメカニズムにはクライアント ID とクライアントシークレットが必要です。これは、クライアントが最初に Red Hat Single Sign-On サーバーに接続してアクセストークンを取得することを意味します。その後、クライアントは Kafka ブローカーに接続し、アクセストークンを使用して認証します。

2. team-b-client クライアントの認証プロパティーで Kafka 設定ファイルを準備します。

```
cat > /tmp/team-b-client.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginM
odule required \
  oauth.client.id="team-b-client" \
  oauth.client.secret="team-b-client-secret" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-
authz/protocol/openid-connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginC
allbackHandler
EOF
```

3. curl を使用して管理者ユーザー alice を認証し、パスワード付与認証を実行して更新トークンを取得します。

```
USERNAME=alice
PASSWORD=alice-password

GRANT_RESPONSE=$(curl -X POST "https://$SSO_HOST/auth/realms/kafka-
authz/protocol/openid-connect/token" -H 'Content-Type: application/x-www-form-
urlencoded' -d
"grant_type=password&username=$USERNAME&password=$PASSWORD&client_id=
kafka-cli&scope=offline_access" -s -k)

REFRESH_TOKEN=$(echo $GRANT_RESPONSE | awk -F "refresh_token\":\"" '{printf
$2}' | awk -F "\"" '{printf $1}')
```

更新トークンは、有効期間がなく、期限切れにならないオフライントークンです。

4. admin ユーザー alice の認証プロパティーで Kafka 設定ファイルを準備します。

```
cat > /tmp/alice.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginM
```

```
odule required \
  oauth.refresh.token="$REFRESH_TOKEN" \
  oauth.client.id="kafka-cli" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-
authz/protocol/openid-connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOauthLoginC
allbackHandler
EOF
```

kafka-cli パブリッククライアントは、sasl.jaas .config の oauth.client. id に使用されます。これはパブリッククライアントであるため、シークレットは必要ありません。クライアントは直前の手順で認証された更新トークンで認証されます。更新トークンは背後でアクセストークンを要求します。これは、認証のために Kafka ブローカーに送信されます。

承認されたアクセスでのメッセージの生成

team-a-clientの設定を使って、a_やx_で始まるトピックへのメッセージを作成できるかどうかを確認します。

1. トピックmy-topicに書き込みます。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic
my-topic \
  --producer.config=/tmp/team-a-client.properties
First message
```

以下のリクエストは、Not authorized to access topics: [my-topic] エラーを返します。

team-a-clientはDev Team Aロールを持っており、a_で始まるトピックに対してサポートされているすべてのアクションを実行する権限を与えられていますが、x_で始まるトピックへの書き込みのみ可能です。my-topic という名前のトピックは、これらのルールのいずれにも一致しません。

2. トピックa_messagesに書き込む。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic
a_messages \
  --producer.config /tmp/team-a-client.properties
First message
Second message
```

メッセージは Kafka に正常に生成されます。

3. CTRL+C を押して CLI アプリケーションを終了します。

4. リクエストについて、Kafka コンテナーログで Authorization GRANTED のデバッグログを確認します。

```
oc logs my-cluster-kafka-0 -f -n $NS
```

承認されたアクセスでのメッセージの消費

team-a-client 設定を使用して、トピック a_messages からメッセージを消費します。

1. トピック a_messages からメッセージをフェッチします。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties
```

team-a-client の Dev Team A ロールは、名前が a_ で始まるコンシューマーグループのみにアクセスできるため、リクエストはエラーを返します。

2. team-a-client プロパティーを更新し、使用が許可されているカスタムコンシューマーグループを指定します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group a_consumer_group_1
```

コンシューマーは a_messages トピックからすべてのメッセージを受信します。

承認されたアクセスでの Kafka の管理

team-a-client はクラスターレベルのアクセスのないアカウントですが、一部の管理操作と使用することができます。

1. トピックを一覧表示します。

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties --list
```

a_messages トピックが返されます。

2. コンシューマーグループを一覧表示します。

```
bin/kafka-consumer-groups.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties --list
```

a_consumer_group_1 コンシューマーグループが返されます。

クラスター設定の詳細を取得します。

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties \
  --entity-type brokers --describe --entity-default
```

操作には team-a-client にないクラスターレベルのパーミッションが必要なため、リクエストはエラーを返します。

異なるパーミッションを持つクライアントの使用

team-b-client 設定を使用して、b_ で始まるトピックにメッセージを生成します。

1. トピックa_messagesに書き込む。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic a_messages \
  --producer.config /tmp/team-b-client.properties
Message 1
```

以下のリクエストは、Not authorized to access topics: [a_messages] エラーを返します。

2. トピックb_messagesに書き込む。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic
b_messages \
  --producer.config /tmp/team-b-client.properties
Message 1
Message 2
Message 3
```

メッセージは Kafka に正常に生成されます。

3. トピックx_messagesに書き込む。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --producer.config /tmp/team-b-client.properties
Message 1
```

Not authorized to access topics: [x_messages] エラーが返され、team-b-client はトピック x_messages からのみ読み取りできます。

4. team-a-client を使用してトピック x_messages に書き込みます。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --producer.config /tmp/team-a-client.properties
Message 1
```

このリクエストは、Not authorized to access topics: [x_messages] エラーを返します。team-a-client は x_messages トピックに書き込みできますが、トピックが存在しない場合に作成するパーミッションがありません。team-a-client が x_messages トピックに書き込みできるようにするには、管理者 power user はパーティションやレプリカの数などの適切な設定で作成する必要があります。

承認された管理ユーザーでの Kafka の管理

管理者ユーザー alice を使用して Kafka を管理します。alice は、すべての Kafka クラスターのすべての管理にフルアクセスできます。

1. aliceとしてx_messagesトピックを作成します。

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/alice.properties \
  --topic x_messages --create --replication-factor 1 --partitions 1
```

トピックが正常に作成されました。

2. aliceとしてすべてのトピックを一覧表示します。

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/alice.properties --list
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-a-client.properties --list
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/team-b-client.properties --list
```

管理者ユーザーのaliceはすべてのトピックを一覧表示できますが、team-a-clientとteam-b-clientは自分がアクセスできるトピックのみを一覧表示できます。

Dev Team AロールとDev Team Bロールは、どちらもx_ で始まるトピックに対するDescribe権限を持っていますが、他のチームのトピックに対するDescribe権限を持っていないため、他のチームのトピックを見ることができません。

3. team-a-client を使用して、x_messages トピックにメッセージを生成します。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic x_messages \
  --producer.config /tmp/team-a-client.properties
Message 1
Message 2
Message 3
```

alice が x_messages トピックを作成すると、メッセージが正常に Kafka に生成されます。

4. team-b-client を使って、x_messages トピックにメッセージを生成します。

```
bin/kafka-console-producer.sh --broker-list my-cluster-kafka-bootstrap:9093 --topic x_messages \
  --producer.config /tmp/team-b-client.properties
Message 4
Message 5
```

このリクエストは、Not authorized to access topics: [x_messages] エラーを返します。

5. team-b-clientを使って、x_messagesトピックからメッセージを消費します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --
topic x_messages \
  --from-beginning --consumer.config /tmp/team-b-client.properties --group
x_consumer_group_b
```

コンシューマーは、x_messagesトピックからすべてのメッセージを受け取ります。

6. team-a-clientを使って、x_messagesトピックからメッセージを消費します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --
topic x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
x_consumer_group_a
```

このリクエストは、Not authorized to access topics: [x_messages] エラーを返します。

7. team-a-clientを使って、a_で始まるコンシューマーグループからのメッセージを消費します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --
topic x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
a_consumer_group_a
```

このリクエストは、Not authorized to access topics: [x_messages] エラーを返します。

Dev Team Aには、x_で始まるトピックのRead権限がありません。

8. aliceを使って、x_messagesトピックへのメッセージを生成します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic x_messages \
  --from-beginning --consumer.config /tmp/alice.properties
```

メッセージは Kafka に正常に生成されます。

alice は、すべてのトピックに対して読み取りまたは書き込みを行うことができます。

9. alice を使用してクラスター設定を読み取ります。

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config /tmp/alice.properties \
  --entity-type brokers --describe --entity-default
```

この例のクラスター設定は空です。

関連情報

- Server Installation and Configuration
- Red Hat Single Sign-On Authorization Services の Kafka 承認モデルへのマッピング

第6章 AMQ STREAMS OPERATOR の使用

AMQ Streams の operator を使用して Kafka クラスターと Kafka トピックおよびユーザーを管理します。

6.1. CLUSTER OPERATOR の使用

Cluster Operator は Kafka クラスターや他の Kafka コンポーネントをデプロイするために使用されます。

Cluster Operator のデプロイメントに関する詳細は、「Cluster Operator のデプロイ」を参照してください。

6.1.1. Cluster Operator の設定

Cluster Operator は、サポートされる環境変数を使用してロギング設定から設定できます。

環境変数は、Cluster Operator イメージのデプロイメンのコンテナー設定に関連します。image設定の詳細については、「image」 を参照してください。

STRIMZI_NAMESPACE

Operator が操作する namespace のカンマ区切りのリスト。設定されていない場合や、空の文字列や * に設定された場合、Cluster Operator はすべての namespace で操作します。Cluster Operator デプロイメントでは OpenShift Downward API を使用して、これを Cluster Operator がデプロイされる namespace に自動設定することがあります。

Cluster Operator namespace の設定例

```
env:
  - name: STRIMZI_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_FULL_RECONCILIATION_INTERVAL_MS

任意設定、デフォルトは 120000 ミリ秒です。定期的な調整の間隔 (秒単位)。

STRIMZI_OPERATION_TIMEOUT_MS

任意設定、デフォルトは 300000 ミリ秒です。内部操作のタイムアウト (ミリ秒単位)。この値は、標準の OpenShift 操作の時間が通常よりも長いクラスターで (Docker イメージのダウンロードが遅い場合など) AMQ Streams を使用する場合に増やす必要があります。

STRIMZI_OPERATIONS_THREAD_POOL_SIZE

任意設定で、デフォルトは 10 です。クラスターオペレーターによって実行されるさまざまな非同期およびブロッキング操作に使用されるワーカースレッドのプールサイズです。

STRIMZI_OPERATOR_NAMESPACE

AMQ Streams Cluster Operator が稼働している namespace の名前。この変数は手動で設定しないでください。OpenShift Downward API を使用します。

```
env:
  - name: STRIMZI_OPERATOR_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_OPERATOR_NAMESPACE_LABELS

任意設定。AMQ Streams Cluster Operator が稼働している namespace のラベル。namespace ラベルは、ネットワークポリシーで namespace セレクターを設定するために使用されます。これにより、AMQ Streams Cluster Operator はこれらのラベルを持つ namespace からのオペランドのみにアクセスできます。設定されていない場合、ネットワークポリシーの namespace セレクターは、OpenShift クラスターのすべての namespace から AMQ Streams Cluster Operator にアクセスできるように設定されます。

```
env:
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS
    value: label1=value1,label2=value2
```

STRIMZI_LABELS_EXCLUSION_PATTERN

任意設定、デフォルトの正規表現パターンは ^app.kubernetes.io/(?!part-of).* です。メインカスタムリソースからサブリソースにラベル伝搬をフィルターするために使用される正規表現除外パターンを指定します。ラベル除外フィルターは、spec.kafka.template.pod.metadata.labels などのテンプレートセクションのラベルには適用されません。

```
env:
  - name: STRIMZI_LABELS_EXCLUSION_PATTERN
    value: "^key1.*"
```

STRIMZI_CUSTOM_{COMPONENT_NAME}_LABELS

オプション。{COMPONENT_NAME} カスタムリソースによって作成されるすべての Pod に適用する 1 つ以上のカスタムラベル。Cluster Operator は、カスタムリソースを作成するか、または次に調整される際に Pod にラベルを付けます。

以下のコンポーネントには環境変数が存在します。

- KAFKA
- KAFKA_CONNECT
- KAFKA_CONNECT_BUILD
- ZOOKEEPER
- ENTITY_OPERATOR
- KAFKA_MIRROR_MAKER2
- KAFKA_MIRROR_MAKER
- CRUISE_CONTROL
- KAFKA_BRIDGE
- KAFKA_EXPORTER

STRIMZI_CUSTOM_RESOURCE_SELECTOR

オプション。Operator によって処理されるカスタムリソースのフィルタリングに使用されるラベルセレクターを指定します。Operator は、指定されたラベルが設定されているカスタムリソースでのみ動作します。これらのラベルのないリソースは Operator によって認識されません。ラベルセレクターは、Kafka、KafkaConnect、KafkaBridge、KafkaMirrorMaker、および

KafkaMirrorMaker2 リソースに適用されます。KafkaRebalance と KafkaConnector リソースは、対応する Kafka および Kafka Connect クラスターに一致するラベルがある場合にのみ操作されます。

```
env:
  - name: STRIMZI_CUSTOM_RESOURCE_SELECTOR
    value: label1=value1,label2=value2
```

STRIMZI_KAFKA_IMAGES

必須。Kafka バージョンから、そのバージョンの Kafka ブローカーが含まれる該当の Docker イメージへのマッピングが提供されます。必要な構文は、空白またはカンマ区切りの <version>=<image> ペアです。例： 2.8.0=registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:2.0.1, 3.0.0=registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1.これはKafka.spec.kafka.versionプロパティが指定されていて、KafkaリソースのKafka.spec.kafka.imageが指定されていない場合に使用されます。

STRIMZI_DEFAULT_KAFKA_INIT_IMAGE

Optional, default registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.1.Kafkaリソースのkafka-init-image としてイメージが指定されていない場合に、初期設定作業（ラックサポート）のためにブローカーの前に開始される init コンテナのデフォルトとして使用するイメージ名。

STRIMZI_KAFKA_CONNECT_IMAGES

必須。Kafka バージョンから、そのバージョンの Kafka Connect が含まれる該当の Docker イメージへのマッピングが提供されます。必要な構文は、空白またはカンマ区切りの <version>=<image> ペアです。例： 2.8.0=registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:2.0.1, 3.0.0=registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1.これは、KafkaConnect.spec.versionプロパティが指定され、KafkaConnect.spec.imageが指定されていない場合に使用されます。

STRIMZI_KAFKA_MIRROR_MAKER_IMAGES

必須。Kafka バージョンから、そのバージョンの Kafka Mirror Maker が含まれる該当の Docker イメージへのマッピングが提供されます。必要な構文は、空白またはカンマ区切りの <version>=<image> ペアです。例： 2.8.0=registry.redhat.io/amq7/amq-streams-kafka-28-rhel8:2.0.1, 3.0.0=registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1.これは、KafkaMirrorMaker.spec.version プロパティーが指定されていても KafkaMirrorMaker.spec.image プロパティーが指定されていない場合に使用されます。

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

Optional, default registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.1.Kafka リソースのKafka.spec.entityOperator.topicOperator.image として指定されたイメージがない場合に、Topic Operator のデプロイ時にデフォルトとして使用するイメージ名。

STRIMZI_DEFAULT_USER_OPERATOR_IMAGE

Optional, default registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.1.KafkaリソースのKafka.spec.entityOperator.userOperator.imageにイメージが指定されていない場合に、ユー

ザーオペレーターをデプロイする際にデフォルトで使用するイメージ名です。

STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE

Optional, default registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1.KafkaリソースのKafka.spec.entityOperator.tlsSidecar.imageにイメージが指定されていない場合に、Entity OperatorのTLSサポートを提供するサイドカーコンテナをデプロイする際にデフォルトで使用するイメージ名です。

STRIMZI_IMAGE_PULL_POLICY

オプション。AMQ Streams の Cluster Operator によって管理されるすべての Pod のコンテナーに適用される ImagePullPolicy。有効な値は Always、IfNotPresent、および Never です。指定のない場合、OpenShift のデフォルトが使用されます。ポリシーを変更すると、すべての Kafka、Kafka Connect、および Kafka MirrorMaker クラスターのローリングアップデートが実行されます。

STRIMZI_IMAGE_PULL_SECRETS

オプション。Secret 名のカンマ区切りのリスト。ここで参照されるシークレットには、コンテナーイメージがプルされるコンテナーレジストリーへのクレデンシャルが含まれます。シークレットは、Cluster Operator によって作成されるすべての Pods の imagePullSecrets フィールドで使用されます。このリストを変更すると、Kafka、Kafka Connect、および Kafka MirrorMaker のすべてのクラスターのローリングアップデートが実行されます。

STRIMZI_KUBERNETES_VERSION

オプション。API サーバーから検出された OpenShift バージョン情報をオーバーライドします。

OpenShift バージョンオーバーライドの設定例

```
env:
  - name: STRIMZI_KUBERNETES_VERSION
    value: |
           major=1
           minor=16
           gitVersion=v1.16.2
           gitCommit=c97fe5036ef3df2967d086711e6c0c405941e14b
           gitTreeState=clean
           buildDate=2019-10-15T19:09:08Z
           goVersion=go1.12.10
           compiler=gc
           platform=linux/amd64
```

KUBERNETES_SERVICE_DNS_DOMAIN

オプション。デフォルトの OpenShift DNS サフィックスを上書きします。

デフォルトでは、OpenShfit クラスターで割り当てられるサービスに、デフォルトのサフィックス cluster.local を使用する DNS ドメイン名があります。

ブローカーが kafka-0 の場合の例は次のとおりです。

```
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers.<namespace>.svc.cluster.local
```

DNS ドメイン名は、ホスト名の検証に使用される Kafka ブローカー証明書に追加されます。

クラスターで異なる DNS サフィックスを使用している場合、Kafka ブローカーとの接続を確立するために、KUBERNETES_SERVICE_DNS_DOMAIN 環境変数をデフォルトから現在使用中の DNS サフィックスに変更します。

STRIMZI_CONNECT_BUILD_TIMEOUT_MS

任意設定、デフォルトは 300000 ミリ秒です。追加のコネクターで新しい Kafka Connect イメージをビルドする場合のタイムアウト (ミリ秒単位)。AMQ Streams を使用して多くのコネクターが含まれるコンテナーイメージをビルドしたり、低速なコンテナーレジストリーを使用する場合は、この値を大きくする必要があります。

STRIMZI_NETWORK_POLICY_GENERATION

任意設定、デフォルトは true です。AMQ Streams がネットワークポリシーリソースを生成するかどうかを制御します。ネットワークポリシーにより、Kafka コンポーネント間の接続が許可されます。

ネットワークポリシーの生成を無効にするには、この環境変数を false に設定します。たとえば、カスタムのネットワークポリシーを使用する場合は、これを行うことができます。カスタムネットワークポリシーを使用すると、コンポーネント間の接続をより詳細に制御できます。

STRIMZI_FEATURE_GATES

オプション。フィーチャーゲートによって制御される機能を有効または無効にします。各フィーチャーゲートについての詳細は、「フィーチャーゲート」 を参照してください。

6.1.1.1. フィーチャーゲート

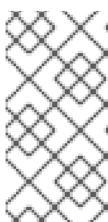
AMQ Streams Operator は、特定の機能および機能を有効または無効にする フィーチャーゲート をサポートします。フィーチャーゲートを有効にすると、関連する operator の動作が変更され、AMQStreams デプロイメントに機能が導入されます。

フィーチャーゲートのデフォルトの状態は enabled または disabled のいずれかになります。機能ゲートのデフォルト状態を変更するには、Operator の設定で STRIMZI_FEATURE_GATES 環境変数を使用します。この 1 つの環境変数を使用して、複数のフィーチャーゲートを変更することができます。

フィーチャーゲートには、3 段階の成熟度があります。

- Alpha: 通常はデフォルトで無効
- Beta: 通常はデフォルトで有効
- General Availability (GA): 通常はデフォルトで有効

Alpha ステージの機能は実験的で不安定である可能性があり、変更される可能性があり、実稼働用に十分にテストされていない可能性があります。Beta ステージの機能は、十分にテストされており、その機能は変更されない可能性が高くなります。GA ステージの機能は安定しており、今後変更されることはないでしょう。Alpha または Bata ステージの機能は、有用であることが証明されない場合は削除されます。



注記

フィーチャーゲートは、GA に達した時点で削除される可能性があります。これは、この機能が AMQ Streams コア機能に組み込まれ、無効にできないことを意味します。

表6.1 Alpha、Beta、または GA に移行したときのすべてのフィーチャーゲートおよび AMQ Streams バージョン

フィーチャーゲート	Alpha	Beta	GA
ControlPlaneListener	1.8	-	-
ServiceAccountPatching	1.8	-	-

フィーチャーゲートの設定

Operator の設定で **STRIMZI_FEATURE_GATES** 環境変数を使用して、機能ゲートを設定します。フィーチャーゲート名とプレフィックスのコンマ区切りリストを指定します。**+** プレフィックスはフィーチャーゲートを有効にし、**-** プレフィックスを無効にします。

FeatureGate1を有効にし、FeatureGate2を無効にするフィーチャーゲートの設定例

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

6.1.1.1.1. コントロールプレーンリスナーフィーチャーゲート

ControlPlaneListener機能ゲートを使用して、Kafkaクラスタ内のブローカー間通信に使用される通信パスを変更します。

OpenShift コントロールプレーンは、ワーカーノードで実行されるワークロードを管理します。Kubernetes API サーバーやコントローラーマネージャーなどのサービスは、コントロールプレーンで実行されます。OpenShift データプレーン は、CPU、メモリー、ネットワーク、およびストレージなどのコンテナーにリソースを提供します。

AMQ Streams では、コントロールプレーンのトラフィックは、Kafka クラスターの必要な状態を維持するコントローラーコネクションで構成されます。データプレーントラフィックは、主にリーダーブローカーとフォロワーブローカー間のデータレプリケーションで構成されます。

ControlPlaneListener 機能ゲートが無効になっている場合、コントロールプレーンおよびデータプレーンのトラフィックはポート9091 の同じ内部リスナーを通過します。これは、フィーチャーゲート導入前のデフォルトの動作でした。

ControlPlaneListener が有効にされている場合、コントロールプレーンのトラフィックはポート 9090 の専用の コントロールプレーンリスナー を通過します。データプレーントラフィックは、引き続きポート 9091 で内部リスナーを使用します。

コントロールプレーンリスナーを使用すると、パーティションリーダーシップの変更などの重要なコントローラーコネクションが、ブローカー全体のデータレプリケーションによって遅延されないた

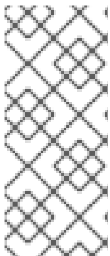
め、パフォーマンスが向上する可能性があります。

コントロールプレーンリスナーフィーチャーゲートの有効化

ControlPlaneListener機能のゲートはアルファ段階であり、デフォルトの状態はdisabledです。これを有効にするには、Cluster Operator 設定の STRIMZI_FEATURE_GATES 環境変数で +ControlPlaneListener を指定します。

以下の場合にフィーチャーゲートを無効にする必要があります。

- AMQ Streams 1.7 以前からのアップグレード
- AMQ Streams 1.7 以前へのダウングレード



注記

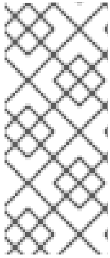
ControlPlaneListener 機能ゲートは AMQ Streams 1.8 に導入されました。これは、ベータステージに移行する前に、複数のリリースのアルファステージ内に留まることが予想されます。

6.1.1.1.2. サービスアカウントパッチ適用のフィーチャーゲート

デフォルトで、Cluster Operator はサービスアカウントを更新しません。Cluster Operator による更新を適用できるようにするには、ServiceAccountPatching 機能ゲートを有効にします。

Cluster Operator 設定の STRIMZI_FEATURE_GATES 環境変数に +ServiceAccountPatching を追加します。

フィーチャーゲートは現在アルファフェーズにあり、デフォルトでは無効になっています。フィーチャーゲートを有効にすると、Cluster Operator は調整ごとに更新をサービスアカウント設定に適用します。たとえば、オペランドが作成された後に、サービスアカウントのラベルおよびアノテーションを変更できます。



注記

ServiceAccountPatching 機能ゲートは AMQ Streams 1.8 に導入されました。これは、ベータフェーズに移行する前に、多くのリリースのアルファフェーズ内に留まり、デフォルトで有効になっています。

6.1.1.2. ConfigMap による設定のロギング

Cluster Operator のロギングは、strimzi-cluster-operator ConfigMap によって設定されます。

ロギング設定が含まれる ConfigMap は、Cluster Operator のインストール時に作成されます。この ConfigMap は、install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml ファイルに記述されます。このConfigMapのデータフィールドlog4j2.propertiesを変更することで、Cluster Operatorのロギングを設定します。

ロギング設定を更新するには、050-ConfigMap-strimzi-cluster-operator.yaml ファイルを編集し、以下のコマンドを実行します。

```
oc create -f install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml
```

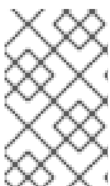
または、ConfigMap を直接編集することもできます。

```
oc edit configmap strimzi-cluster-operator
```

リロード間隔の頻度を変更するには、作成された ConfigMap の monitorInterval オプションで秒単位の時間を設定します。

クラスタオペレータのデプロイ時にConfigMapがない場合、デフォルトのロギング値が使用されます。

Cluster Operator のデプロイ後に ConfigMap が誤って削除される場合、最後に読み込まれたロギング設定が使用されます。新規のロギング設定を読み込むために新規 ConfigMap を作成します。



注記

ConfigMap から monitorInterval オプションを削除しないでください。

6.1.1.3. ネットワークポリシーによる Cluster Operator アクセスの制限

Cluster Operator は、管理するリソースと同じ namespace または別の namespace で実行できます。デフォルトでは、STRIMZI_OPERATOR_NAMESPACE環境変数は、OpenShift Downward APIを使用して、クラスターオペレーターがどのネームスペースで実行されているかを見つけるように構成されています。Cluster Operator がリソースと同じ namespace で実行されている場合は、ローカルアクセスのみが必要で、AMQ Sreams によって許可されます。

Cluster Operator が管理するリソースとは別の namespace で実行されている場合、ネットワークポリシーが設定されている場合を除き、OpenShift クラスターのすべての namespace は Cluster Operator へのアクセスが許可されます。オプションの STRIMZI_OPERATOR_NAMESPACE_LABELS 環境変数を使用して、namespace ラベルを使用して Cluster Operator のネットワークポリシーを確立します。namespace ラベルを追加すると、Cluster Operator へのアクセスは指定された namespace に限定されます。

Cluster Operator デプロイメントに設定されたネットワークポリシー

```
#...
env:
  # ...
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS
    value: label1=value1,label2=value2
  #...
```

6.1.1.4. 定期的な調整

Cluster Operator は OpenShift クラスターから受信する必要なクラスターリソースに関するすべての通知に対応しますが、Operator が実行されていない場合や、何らかの理由で通知が受信されない場合、必要なリソースは実行中の OpenShift クラスターの状態と同期しなくなります。

フェイルオーバーを適切に処理するために、Cluster Operator によって定期的な調整プロセスが実行され、必要なリソースすべてで一貫した状態になるように、必要なリソースの状態を現在のクラスターデプロイメントと比較できます。[STRIMZI_FULL_RECONCILIATION_INTERVAL_MS] 変数を使用して、定期的な調整の時間間隔を設定できます。

6.1.2. ロールベースアクセス制御 (RBAC) のプロビジョニング

クラスターオペレーターが機能するためには、OpenShiftクラスター内で、Kafka、KafkaConnectなどのリソース

や、ConfigMaps、Pod、Deployments、StatefulSets、Servicesなどの管理されたリソースとやりとりする権限が必要です。このようなパーミッションは、OpenShift のロールベースアクセス制御 (RBAC) リソースに記述されます。

- ServiceAccount
- Role および ClusterRole
- RoleBinding および ClusterRoleBinding

Cluster Operator は、ClusterRoleBinding を使用して独自の ServiceAccount で実行される他に、OpenShift リソースへのアクセスを必要とするコンポーネントの RBAC リソースを管理します。

また OpenShift には、ServiceAccount で動作するコンポーネントが、その ServiceAccount にはない他の ServiceAccounts の権限を付与しないようにするための特権昇格の保護機能も含まれています。Cluster Operator は、ClusterRoleBindings と、それが管理するリソースで必要な RoleBindings を作成できる必要があるため、Cluster Operator にも同じ権限が必要です。

6.1.2.1. 委譲された権限

Cluster Operator が必要な Kafka リソースのリソースをデプロイする場合、以下のように ServiceAccounts、RoleBindings、および ClusterRoleBindings も作成します。

- Kafka ブローカー Pod は、cluster-name-kafka という ServiceAccount を使用します。
    - ラック機能が使用されると、strimzi-cluster-name-kafka-init ClusterRoleBinding は、strimzi-kafka-broker と呼ばれる ClusterRole 経由で、クラスター内のノードへの ServiceAccount アクセスを付与するために使用されます。
    - ラック機能が使用されておらず、クラスターがノードポートを介して公開されていない場合、バインディングは作成されません。
- ZooKeeper Pod では cluster-name-zookeeper という ServiceAccount が使用されます。
-

Entity Operator Pod では cluster-name-entity-operator という ServiceAccount が使用されます。

  - Topic Operator はステータス情報のある OpenShift イベントを生成するため、ServiceAccount は strimzi-entity-operator という ClusterRole にバインドされ、strimzi-entity-operator RoleBinding 経由でこのアクセス権限を付与します。

- KafkaConnectリソースのPodは、cluster-name--cluster-connect という ServiceAccountを使用します。

- KafkaMirrorMaker の Pod は、cluster-name-mirror-maker という ServiceAccount を使用します。

- KafkaMirrorMaker2 の Pod は、cluster-name-mirrormaker2という ServiceAccount を使用します。

- KafkaBridge の Pod は、cluster-name-bridge という ServiceAccount を使用します。

6.1.2.2. ServiceAccount

Cluster Operator は ServiceAccount を使用して最適に実行されます。

Cluster Operator の ServiceAccount の例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
```

その後、Cluster Operator の Deployment で、これを spec.template.spec.serviceAccountName に指定する必要があります。

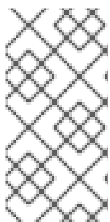
Cluster Operator の Deployment の部分的な例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-cluster-operator
  labels:
    app: strimzi
spec:
  replicas: 1
  selector:
    matchLabels:
      name: strimzi-cluster-operator
      strimzi.io/kind: cluster-operator
  template:
      # ...
```

12 行目で、strimzi-cluster-operator ServiceAccount が serviceAccountName として指定されています。

6.1.2.3. ClusterRoles

Cluster Operator は、必要なリソースへのアクセス権限を付与する ClusterRole を使用して操作する必要があります。OpenShift クラスターの設定によっては、クラスター管理者が ClusterRoles を作成する必要があることがあります。



注記

クラスター管理者の権限は ClusterRoles の作成にのみ必要です。Cluster Operator はクラスター管理者アカウントで実行されません。

ClusterRoles は、 最小権限の原則に従い、Kafka、Kafka Connect、および ZooKeeper クラスターを操作するために Cluster Operator が必要とする権限のみが含まれます。最初に割り当てられた一連の権限により、Cluster Operator で StatefulSets、Deployments、Pods、および ConfigMaps などの OpenShift リソースを管理できます。

Cluster Operator は ClusterRoles を使用して、namespace スコープリソースのレベルおよびクラスタースコープリソースのレベルで権限を付与します。

Cluster Operator の namespaced リソースのある ClusterRole

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-namespaced
  labels:
    app: strimzi
rules:
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to access and manage rolebindings to grant Strimzi
components cluster permissions
      - rolebindings
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to access and manage roles to grant the entity operator
permissions
      - roles
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - ""
    resources:
      # The cluster operator needs to access and delete pods, this is to allow it to monitor pod
health and coordinate rolling updates
      - pods
      # The cluster operator needs to access and manage service accounts to grant Strimzi
components cluster permissions
      - serviceaccounts
      # The cluster operator needs to access and manage config maps for Strimzi components
configuration
      - configmaps
      # The cluster operator needs to access and manage services and endpoints to expose
Strimzi components to network traffic
      - services
      - endpoints
```

```
    # The cluster operator needs to access and manage secrets to handle credentials
    - secrets
    # The cluster operator needs to access and manage persistent volume claims to bind them
to Strimzi components for persistent data
    - persistentvolumeclaims
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
 - apiGroups:
    - "kafka.strimzi.io"
  resources:
    # The cluster operator runs the KafkaAssemblyOperator, which needs to access and
manage Kafka resources
    - kafkas
    - kafkas/status
    # The cluster operator runs the KafkaConnectAssemblyOperator, which needs to access
and manage KafkaConnect resources
    - kafkaconnects
    - kafkaconnects/status
    # The cluster operator runs the KafkaConnectorAssemblyOperator, which needs to access
and manage KafkaConnector resources
    - kafkaconnectors
    - kafkaconnectors/status
    # The cluster operator runs the KafkaMirrorMakerAssemblyOperator, which needs to
access and manage KafkaMirrorMaker resources
    - kafkamirrormakers
    - kafkamirrormakers/status
    # The cluster operator runs the KafkaBridgeAssemblyOperator, which needs to access
and manage BridgeMaker resources
    - kafkabridges
    - kafkabridges/status
    # The cluster operator runs the KafkaMirrorMaker2AssemblyOperator, which needs to
access and manage KafkaMirrorMaker2 resources
    - kafkamirrormaker2s
    - kafkamirrormaker2s/status
    # The cluster operator runs the KafkaRebalanceAssemblyOperator, which needs to access
and manage KafkaRebalance resources
    - kafkarebalances
    - kafkarebalances/status
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
 - apiGroups:
    # The cluster operator needs the extensions api as the operator supports Kubernetes
version 1.11+
    # apps/v1 was introduced in Kubernetes 1.14
```

```
    - "extensions"
  resources:
    # The cluster operator needs to access and manage deployments to run deployment
based Strimzi components
    - deployments
    - deployments/scale
    # The cluster operator needs to access replica sets to manage Strimzi components and to
determine error states
    - replicasets
    # The cluster operator needs to access and manage replication controllers to manage
replicasets
    - replicationcontrollers
    # The cluster operator needs to access and manage network policies to lock down
communication between Strimzi components
    - networkpolicies
    # The cluster operator needs to access and manage ingresses which allow external
access to the services in a cluster
    - ingresses
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
 - apiGroups:
    - "apps"
  resources:
    # The cluster operator needs to access and manage deployments to run deployment
based Strimzi components
    - deployments
    - deployments/scale
    - deployments/status
    # The cluster operator needs to access and manage stateful sets to run stateful sets based
Strimzi components
    - statefulsets
    # The cluster operator needs to access replica-sets to manage Strimzi components and to
determine error states
    - replicasets
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
 - apiGroups:
    - ""
  resources:
    # The cluster operator needs to be able to create events and delegate permissions to do so
    - events
  verbs:
    - create
 - apiGroups:
```

```
    # Kafka Connect Build on OpenShift requirement
    - build.openshift.io
  resources:
    - buildconfigs
    - buildconfigs/instantiate
    - builds
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - networking.k8s.io
  resources:
    # The cluster operator needs to access and manage network policies to lock down
communication between Strimzi components
    - networkpolicies
    # The cluster operator needs to access and manage ingresses which allow external
access to the services in a cluster
    - ingresses
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - route.openshift.io
  resources:
    # The cluster operator needs to access and manage routes to expose Strimzi components
for external access
    - routes
    - routes/custom-host
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - policy
  resources:
    # The cluster operator needs to access and manage pod disruption budgets this limits the
number of concurrent disruptions
    # that a Strimzi component experiences, allowing for higher availability
    - poddisruptionbudgets
  verbs:
    - get
    - list
```

```
      - watch
      - create
      - delete
      - patch
      - update
```

2 番目の一連の権限には、クラスタースコープリソースに必要な権限が含まれます。

Cluster Operator のクラスタースコープリソースのある ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-global
  labels:
    app: strimzi
rules:
  - apiGroups:
      - "rbac.authorization.k8s.io"
    resources:
      # The cluster operator needs to create and manage cluster role bindings in the case of an
install where a user
      # has specified they want their cluster role bindings generated
      - clusterrolebindings
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
  - apiGroups:
      - storage.k8s.io
    resources:
      # The cluster operator requires "get" permissions to view storage class details
      # This is because only a persistent volume of a supported storage class type can be
resized
      - storageclasses
    verbs:
      - get
  - apiGroups:
      - ""
    resources:
      # The cluster operator requires "list" permissions to view all nodes in a cluster
      # The listing is used to determine the node addresses when NodePort access is
configured
```

```
    # These addresses are then exposed in the custom resource states
    - nodes
  verbs:
    - list
```

strimzi-kafka-broker ClusterRole は、ラック機能に使用される Kafka Pod の init コンテナーが必要とするアクセス権限を表します。「委譲された権限」で説明したように、このアクセスを委譲できるようにするには、このロールも Cluster Operator に必要です。

Cluster Operator の ClusterRole により、OpenShift ノードへのアクセスを Kafka ブローカー Pod に委譲できます。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-kafka-broker
  labels:
    app: strimzi
rules:
  - apiGroups:
      - ""
    resources:
      # The Kafka Brokers require "get" permissions to view the node they are on
      # This information is used to generate a Rack ID that is used for High Availability
configurations
      - nodes
    verbs:
      - get
```

strimzi-topic-operator の ClusterRole は、Topic Operator が必要とするアクセスを表します。「委譲された権限」で説明したように、このアクセスを委譲できるようにするには、このロールも Cluster Operator に必要です。

Cluster Operator のClusterRole により、イベントへのアクセスを Topic Operator に委譲できます。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
```

```
  name: strimzi-entity-operator
  labels:
    app: strimzi
rules:
  - apiGroups:
      - "kafka.strimzi.io"
    resources:
      # The entity operator runs the KafkaTopic assembly operator, which needs to access and
manage KafkaTopic resources
      - kafkatopics
      - kafkatopics/status
      # The entity operator runs the KafkaUser assembly operator, which needs to access and
manage KafkaUser resources
      - kafkausers
      - kafkausers/status
    verbs:
      - get
      - list
      - watch
      - create
      - patch
      - update
      - delete
  - apiGroups:
      - ""
    resources:
      - events
    verbs:
      # The entity operator needs to be able to create events
      - create
  - apiGroups:
      - ""
    resources:
      # The entity operator user-operator needs to access and manage secrets to store
generated credentials
      - secrets
    verbs:
      - get
      - list
      - watch
      - create
      - delete
      - patch
      - update
```

strimzi-kafka-client ClusterRole は、クライアントのラックアウェアネスを使用する Kafka クライアントをベースとしたコンポーネントが必要とするアクセス権限を表します。「委譲された権限」 で説明したように、このアクセスを委譲できるようにするには、このロールも Cluster Operator に必要です。

Cluster Operator の ClusterRole により、OpenShift ノードへのアクセスを Kafka クライアントベースの Pod に委譲できます。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-kafka-client
  labels:
    app: strimzi
rules:
  - apiGroups:
      - ""
    resources:
      # The Kafka clients (Connect, Mirror Maker, etc.) require "get" permissions to view the
node they are on
      # This information is used to generate a Rack ID (client.rack option) that is used for
consuming from the closest
      # replicas when enabled
      - nodes
    verbs:
      - get
```

6.1.2.4. ClusterRoleBindings

Operator には ClusterRoleBindings が必要であり、Operator の ClusterRole を ServiceAccount と関連付ける RoleBindings も必要です。ClusterRoleBindings はクラスタースコープリソースが含まれる ClusterRoles に必要です。

Cluster Operator の ClusterRoleBinding の例

```
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 は、委譲に必要な ClusterRole にも必要です。

Kafka ブローカーラックアウェアネスの Cluster Operator の ClusterRoleBinding の例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator-kafka-broker-delegation
  labels:
    app: strimzi
# The Kafka broker cluster role must be bound to the cluster operator service account so that
it can delegate the cluster role to the Kafka brokers.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-broker
  apiGroup: rbac.authorization.k8s.io
```

および

Kafka クライアントラックアウェアネスの Cluster Operator の ClusterRoleBinding の例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: strimzi-cluster-operator-kafka-client-delegation
  labels:
    app: strimzi
# The Kafka clients cluster role must be bound to the cluster operator service account so that
it can delegate the
# cluster role to the Kafka clients using it for consuming from closest replica.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
```

```
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-kafka-client
  apiGroup: rbac.authorization.k8s.io
```

namespaced リソースのみが含まれる ClusterRoles は、RoleBindings のみを使用してバインドされます。

```
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
# The Entity Operator cluster role must be bound to the cluster operator service account so that it can delegate the cluster role to the Entity Operator.
# This must be done to avoid escalating privileges which would be blocked by Kubernetes.
subjects:
  - kind: ServiceAccount
    name: strimzi-cluster-operator
    namespace: myproject
roleRef:
  kind: ClusterRole
  name: strimzi-entity-operator
  apiGroup: rbac.authorization.k8s.io
```

6.1.3. デフォルトのプロキシー設定を使用した Cluster Operator の設定

HTTP プロキシーの背後で Kafka クラスターを実行している場合は、クラスターとの間でデータを

出し入れできます。たとえば、プロキシー外からデータをプッシュおよびプルするコネクターで Kafka Connect を実行できます。または、プロキシーを使用して承認サーバーに接続できます。

プロキシー環境変数を指定するように Cluster Operator デプロイメントを設定します。クラスタオペレータは標準的なプロキシ設定(HTTP_PROXY、HTTPS_PROXY、NO_PROXY)を環境変数として受け入れます。プロキシー設定はすべての AMQ Streams コンテナーに適用されます。

プロキシーアドレスの形式は http://IP-ADDRESS:PORT-NUMBER です。名前とパスワードでプロキシーを設定する場合、形式は http://USERNAME:PASSWORD@IP-ADDRESS:PORT-NUMBER です。

前提条件

この手順では、CustomResourceDefinitions、ClusterRoles、および ClusterRoleBindings を作成できる OpenShift ユーザーアカウントを使用する必要があります。通常、OpenShift クラスターでロールベースアクセス制御 (RBAC) を使用する場合、これらのリソースを作成、編集、および削除する権限を持つユーザーは system:admin などの OpenShift クラスター管理者に限定されます。

手順

1. クラスタオペレータにプロキシ環境変数を追加するには、そのDeployment構成（install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml）を更新します。

   Cluster Operator のプロキシー設定の例

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
        # ...
        - name: "HTTP_PROXY"
          value: "http://proxy.com" 1
        - name: "HTTPS_PROXY"
          value: "https://proxy.com" 2
        - name: "NO_PROXY"
          value: "internal.com, other.domain.com" 3
  # ...
```

1. プロキシーサーバーのアドレス。

2. プロキシーサーバーの安全なアドレス。

3. プロキシーサーバーの例外として直接アクセスされるサーバーのアドレス。URL はカンマで区切られます。

または、Deployment を直接編集します。

```
oc edit deployment strimzi-cluster-operator
```

2. Deployment を直接編集せずに YAML ファイルを更新する場合は、変更を適用します。

```
oc create -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
```

関連情報

- ホストエイリアス
- AMQ Streams の管理者の指名

6.2. TOPIC OPERATOR の使用

KafkaTopic リソースを使用してトピックを作成、編集、または削除する場合、Topic Operator によって変更が確実に Kafka クラスターで反映されます。

『OpenShift での AMQ Streams のデプロイおよびアップグレード』には、Topic Operator をデプロイする手順が記載されています。

- Cluster Operator を使用 (推奨)
- AMQ Streams によって管理されない Kafka クラスターで操作するためのスタンドアロン

6.2.1. Kafka トピックリソース

KafkaTopic リソースは、パーティションやレプリカの数を含む、トピックの設定に使用されます。

KafkaTopic の完全なスキーマは、「KafkaTopic スキーマ参照」で確認できます。

6.2.1.1. トピック処理用の Kafka クラスターの特定

KafkaTopic リソースには、このリソースが属する Kafka クラスターの適した名前 (Kafka リソースの名前から派生) を定義するラベルが含まれます。

以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
  labels:
    strimzi.io/cluster: my-cluster
```

ラベルは、KafkaTopic リソースを特定し、新しいトピックを作成するために、Topic Operator によって使用されます。また、以降のトピックの処理でも使用されます。

ラベルが Kafka クラスターと一致しない場合、Topic Operator は KafkaTopic を識別できず、トピックは作成されません。

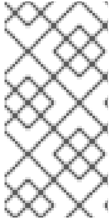
6.2.1.2. Kafka トピックの使用に関する推奨事項

トピックを使用する場合は、整合性を保ちます。常に KafkaTopic リソースで作業を行うか、直接 OpenShift でトピックを扱います。特定のトピックで、両方の方法を頻繁に切り替えないでください。

トピックの性質を反映するトピック名を使用し、後で名前を変更できないことに注意してくださ

い。

Kafka でトピックを作成する場合は、有効な OpenShift リソース名である名前を使用します。それ以外の場合は、Topic Operator は対応する KafkaTopic を OpenShift ルールに準じた名前で作成する必要があります。



注記

OpenShift の識別子および名前の推奨事項については、OpenShift コミュニティーの記事「Identifiers and Names」を参照してください。

6.2.1.3. Kafka トピックの命名規則

Kafka と OpenShift では、Kafka と KafkaTopic.metadata.name でのトピックの命名にそれぞれ独自の検証ルールを適用します。トピックごとに有効な名前があり、他のトピックには無効です。

spec.topicName プロパティーを使用すると、OpenShift の Kafka トピックでは無効な名前を使用して、Kafka で有効なトピックを作成できます。

spec.topicName プロパティーは Kafka の命名検証ルールを継承します。

- 249 文字を超える名前は使用できません。
- Kafka トピックの有効な文字は ASCII 英数字、.、_、および - です。
- 名前を . または .. にすることはできませんが、. は exampleTopic. や .exampleTopic のように名前で使用できます。

spec.topicName は変更しないでください。

以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
```

```
  name: topic-name-1
spec:
  topicName: topicName-1 (1)
  # ...
```

(1) OpenShift では大文字は無効です。

上記は下記のように変更できません。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: name-2
  # ...
```

注記

Kafka Streams など一部の Kafka クライアントアプリケーションは、プログラムを使用して Kafka でトピックを作成できます。これらのトピックに、OpenShift リソース名として無効な名前がある場合、Topic Operator はそれらのトピックに Kafka 名に基づく有効な metadata.name を提供します。無効な文字が置き換えられ、ハッシュが名前に追加されます。以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: mytopic---c55e57fe2546a33f9e603caf57165db4072e827e
spec:
  topicName: myTopic
  # ...
```

6.2.2. Topic Operator のトピックストア

Topic Operator は Kafka を使用して、トピック設定をキーと値のペアとして記述するトピックメタデータを保存します。トピックストアは、Kafka トピックを使用して状態を永続化する Kafka Streams のキーバリューメカニズムを基にしています。

トピックメタデータはインメモリーでキャッシュされ、Topic Operator 内にてローカルでアクセスされます。ローカルのインメモリーキャッシュに適用される操作からの更新は、ディスク上のバックアップトピックストアに永続化されます。トピックストアは、Kafka トピックまたは OpenShift KafkaTopic カスタムリソースからの更新と継続的に同期されます。操作は、このような方法で設定さ

れたトピックストアで迅速に処理されますが、インメモリーキャッシュがクラッシュした場合は、永続ストレージから自動的にデータが再入力されます。

6.2.2.1. 内部トピックストアトピック

内部トピックは、トピックストアでのトピックメタデータの処理をサポートします。

__strimzi_store_topic

トピックメタデータを保存するための入力トピック

__strimzi-topic-operator-kstreams-topic-store-changelog

圧縮されたトピックストア値のログの維持

警告

これらのトピックは、Topic Operator の実行に不可欠であるため、削除しないでください。

6.2.2.2. ZooKeeper からのトピックメタデータの移行

これまでのリリースの AMQ Streams では、トピックメタデータは ZooKeeper に保存されていました。新しいプロセスによってこの要件は除外されたため、メタデータは Kafka クラスターに取り込まれ、Topic Operator の制御下となります。

AMQ Streams 2.0 にアップグレードする場合、Topic Operator によってトピックストアが制御されるようにシームレスに移行されます。メタデータは ZooKeeper から検出および移行され、古いストアは削除されます。

6.2.2.3. ZooKeeper を使用してトピックメタデータを保存する AMQ Streams バージョンへのダウングレード

トピックメタデータの保存に ZooKeeper を使用する 1.7 よりも前のバージョンの AMQ Streams に戻す場合は、Cluster Operator を以前のバージョンにダウングレードしてから、Kafka ブローカーとクライアントアプリケーションを以前の Kafka バージョンにダウングレードします。

ただし、Kafka クラスターのブートストラップアドレスを指定して、kafka-admin コマンドを使用してトピックストア用に作成されたトピックを削除する必要もあります。以下に例を示します。

```
oc run kafka-admin -ti --image=registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1 --
rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic
__strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --
bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

このコマンドは、Kafka クラスターへのアクセスに使用されるリスナーおよび認証のタイプに対応している必要があります。

Topic Operator は、Kafka のトピックの状態から ZooKeeper トピックメタデータを再構築します。

6.2.2.4. Topic Operator トピックのレプリケーションおよびスケーリング

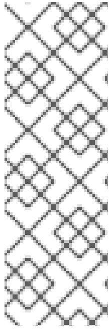
Topic Operator によって管理されるトピックには、トピックレプリケーション係数を 3 に設定し、最低でも 2 つの In-Sync レプリカを設定することが推奨されます。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10 (1)
  replicas: 3 (2)
  config:
    min.insync.replicas: 2 (3)
  #...
```

1. トピックのパーティション数。

2. レプリカトピックパーティションの数。現在のところ、これはKafkaTopicリソースでは変更できませんが、kafka-reassign-partitions.shツールを使って変更することができます。

3. メッセージが正常に書き込まれる必要があるレプリカパーティションの最小数。この条件を満たさない場合は例外が発生します。



注記

インシンクレプリカは、プロデューサーアプリケーションのacks設定と組み合わせて使用します。acks設定は、メッセージが正常に受信されたことを確認するまでに、メッセージを複製しなければならないフォロワーパーティションの数を決定します。トピックオペレータはacks=allで動作します。これにより、メッセージは同期しているすべてのレプリカに確認されなければなりません。

ブローカーを追加または削除して Kafka クラスターをスケーリングする場合、レプリケーション係数設定は変更されず、レプリカは自動的に再割り当てされません。しかし、kafka-reassign-partitions.shツールを使ってレプリケーション係数を変更し、手動でレプリカをブローカーに再割り当てすることができます。

また、AMQ Streams の Cruise Control の統合ではトピックのレプリケーション係数を変更することはできませんが、Kafka をリバランスするために生成された最適化プロポーザルには、パーティションレプリカを転送し、パーティションリーダーを変更するコマンドが含まれます。

6.2.2.5. トピック変更の処理

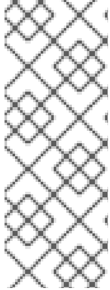
Topic Operator にとって解決しなければならない基本的な問題として、信頼できる唯一の情報源 (SSOT: single source of truth) がないことがあります。KafkaTopic リソースと Kafka トピックは、Topic Operator に関係なく変更できます。面倒なことに、Topic Operator は KafkaTopic リソースと Kafka トピックで変更を常にリアルタイムで監視できるとは限りません。たとえば、Topic Operator が停止した場合などがこれに該当します。

これを解決するために、Topic Operator はトピックストアの各トピックに関する情報を維持します。Kafka クラスターまたは OpenShift で変更が生じると、他のシステムの状態とトピックストアの両方を確認し、すべての同期が保たれるように何を変更する必要があるかを判断します。同じことが Topic Operator の起動時に必ず実行され、また Topic Operator の稼働中にも定期的に行われます。

たとえば、Topic Operator が実行されていないときに my-topic という KafkaTopic が作成された場合を考えてみましょう。Topic Operator が起動すると、トピックストアには my-topic に関する情報が含まれないため、最後に実行された後に KafkaTopic が作成されたと推測できます。Topic Operator によって my-topic に対応するトピックが作成され、さらにトピックストアに my-topic のメタデータも格納されます。

Kafka トピック設定を更新するか、KafkaTopic カスタムリソースで変更を適用する場合、Kafka クラスターの調整後にトピックストアが更新されます。

また、トピックストアにより、トピックオペレーターは、Kafkaトピックでトピック構成が変更され、OpenShiftKafkaTopicのカスタムリソースを通じて更新 されるシナリオを、変更内容に互換性がない限り、管理することができます。たとえば、同じトピック設定キーに変更を加えることはできますが、別の値への変更のみが可能です。互換性のない変更については、Kafka の設定が優先され、それに応じて KafkaTopic が更新されます。



注記

KafkaTopic リソースを使用して、oc delete -f KAFKA-TOPIC-CONFIG-FILE コマンドを使用してトピックを削除できます。これを実現するには、Kafkaリソースのspec.kafka.configでdelete.topic.enableをtrue（デフォルト）に設定する必要があります。

その他のリソース

- AMQ Streams のダウングレード
- プロデューサー設定のチューニングとデータの持続性
- クラスターおよびパーティション再割り当てのスケーリング
- Cruise Control によるクラスターのリバランス

6.2.3. Kafka トピックの設定

KafkaTopic リソースのプロパティーを使用して Kafka トピックを設定します。

oc apply を使用すると、トピックを作成または編集できます。oc delete を使用すると、既存のトピックを削除できます。

以下に例を示します。

- oc apply -f <topic-config-file>

- oc delete KafkaTopic <topic-name>

この手順では、10 個のパーティションと 2 つのレプリカがあるトピックを作成する方法を説明します。

作業を開始する前の注意事項

以下を考慮してから変更を行うことが重要になります。

- Kafka は KafkaTopic リソースによる以下の変更をサポートしません。
  - spec.topicName を使用したトピック名の変更
  - spec.partitions を使用したパーティションサイズの減少
- spec.replicas を使用して最初に指定したレプリカの数を変更することはできません。
- キーのあるトピックの spec.partitions を増やすと、レコードをパーティション化する方法が変更されます。これは、トピックがセマンティックパーティションを使用するとき、特に問題になる場合があります。

前提条件

- TLS 認証および暗号化を使用してリスナーで設定された 稼働中の Kafka クラスターが必要です。
- 稼働中の Topic Operator が必要です (通常は Entity Operator でデプロイされます)。
- トピックを削除する場合は、 Kafka リソースの spec.kafka.config が delete.topic.enable=true (デフォルト) である必要があります。

手順

1. 作成する KafkaTopic が含まれるファイルを準備します。

KafkaTopic の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: orders
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 10
  replicas: 2
```

ヒント

トピックを変更する場合、現行バージョンのリソースは、oc get kafkatopic orders -o yaml を使用して取得できます。

2. OpenShift で KafkaTopic リソースを作成します。

```
oc apply -f TOPIC-CONFIG-FILE
```

6.2.4. リソース要求および制限のある Topic Operator の設定

CPU やメモリーなどのリソースを Topic Operator に割り当て、Topic Operator が消費できるリソースの量に制限を設定できます。

前提条件

- Cluster Operator が稼働している必要があります。

手順

1. 必要に応じてエディターで Kafka クラスター設定を更新します。

```
oc edit kafka MY-CLUSTER
```

2. Kafka リソースの spec.entityOperator.topicOperator.resources プロパティーで、Topic Operator のリソース要求および制限を設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # Kafka and ZooKeeper sections...
  entityOperator:
    topicOperator:
      resources:
        requests:
          cpu: "1"
          memory: 500Mi
        limits:
          cpu: "1"
          memory: 500Mi
```

3. 新しい設定を適用してリソースを作成または更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

6.3. USER OPERATOR の使用

KafkaUser リソースを使用してユーザーを作成、編集、または削除する場合、User Operator によって変更が確実に Kafka クラスターで反映されます。

『OpenShift での AMQ Streams のデプロイおよびアップグレード』には、User Operator をデプロイする手順が記載されています。

- Cluster Operator を使用 (推奨)
- AMQ Streams によって管理されない Kafka クラスターで操作するためのスタンドアロン

スキーマの詳細については、KafkaUser schema reference を参照してください。

Kafka へのアクセスの認証および承認

KafkaUser を使用して、特定のクライアントが Kafka にアクセスするために使用する認証および承

認メカニズムを有効にします。

KafkUserを使用してユーザーを管理し、Kafkaブローカーへのアクセスを確保する方法については、Securing access to Kafka brokers を参照してください。

6.3.1. リソース要求および制限のある User Operator の設定

CPU やメモリーなどのリソースを User Operator に割り当て、User Operator が消費できるリソースの量に制限を設定できます。

前提条件

- Cluster Operator が稼働している必要があります。

手順

1. 必要に応じてエディターで Kafka クラスター設定を更新します。

```
oc edit kafka MY-CLUSTER
```

2. Kafka リソースの spec.entityOperator.userOperator.resources プロパティーで、User Operator のリソース要求および制限を設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # Kafka and ZooKeeper sections...
  entityOperator:
    userOperator:
      resources:
        requests:
          cpu: "1"
          memory: 500Mi
        limits:
          cpu: "1"
          memory: 500Mi
```

ファイルを保存して、エディターを終了します。Cluster Operator によって変更が自動的に適用されます。

6.4. PROMETHEUS メトリクスを使用した OPERATOR の監視

AMQ Streams の operator は Prometheus メトリクスを公開します。メトリクスは自動で有効になり、以下の情報が含まれます。

- 調整の数
- operator が処理しているカスタムリソースの数
- 調整の期間
- operator からの JVM メトリクス

この他に、Grafana ダッシュボードのサンプルが提供されます。

Prometheus の詳細は、『OpenShift での AMQ Streams のデプロイおよびアップグレード』 の「Kafka へのメトリクスの導入 」を参照してください。

第7章 KAFKA BRIDGE

本章では、AMQ Streams Kafka Bridge について概説し、その REST API を使用して AMQ Streams と対話するために役立つ情報を提供します。

- ローカル環境で Kafka Bridge を試すには、本章で後述する「Kafka Bridge クイックスタート」を参照してください。
- 詳細な設定手順は、「Kafka Bridge クラスターの設定」を参照してください。

7.1. KAFKA BRIDGE API ドキュメント

リクエストやレスポンスの例などを含む REST API エンドポイントおよび説明の完全リストは、「Kafka Bridge API reference」を参照してください。

7.2. KAFKA BRIDGE の概要

AMQ Streams Kafka Bridge をインターフェースとして使用し、Kafka クラスターに対して特定タイプの HTTP リクエストを行うことができます。

7.2.1. Kafka Bridge インターフェース

Kafka Bridge では、HTTP ベースのクライアントと Kafka クラスターとの対話を可能にする RESTful インターフェースが提供されます。 また、クライアントアプリケーションによる Kafka プロトコルの変換は必要なく、Web API コネクションの利点が AMQ Streams に提供されます。

API には consumers と topics の 2 つの主なリソースがあります。これらのリソースは、Kafka クラスターでコンシューマーおよびプロデューサーと対話するためにエンドポイント経由で公開され、アクセスが可能になります。リソースと関係があるのは Kafka ブリッジのみで、Kafka に直接接続されたコンシューマーやプロデューサーとは関係はありません。

7.2.1.1. HTTP リクエスト

Kafka Bridge は、以下の方法で Kafka クラスターへの HTTP リクエストをサポートします。

- トピックにメッセージを送信する。

- トピックからメッセージを取得する。
- トピックのパーティションリストを取得する。
- コンシューマーを作成および削除する。
- コンシューマーをトピックにサブスクライブし、このようなトピックからメッセージを受信できるようにする。
- コンシューマーがサブスクライブしているトピックの一覧を取得する。
- トピックからコンシューマーのサブスクライブを解除する。
- パーティションをコンシューマーに割り当てる。
- コンシューマーオフセットの一覧をコミットする。
- パーティションで検索して、コンシューマーが最初または最後のオフセットの位置、または指定のオフセットの位置からメッセージを受信できるようにする。

上記の方法で、JSON 応答と HTTP 応答コードのエラー処理を行います。メッセージは JSON またはバイナリー形式で送信できます。

クライアントは、ネイティブの Kafka プロトコルを使用する必要なくメッセージを生成して使用できます。

その他のリソース

- リクエストおよび応答の例など、API ドキュメントを確認するには、『Strimzi Kafka Bridge Documentation』を参照してください。

7.2.2. Kafka Bridge でサポートされるクライアント

Kafka Bridge を使用して、内部および外部の HTTP クライアントアプリケーションの両方を Kafka クラスターに統合できます。

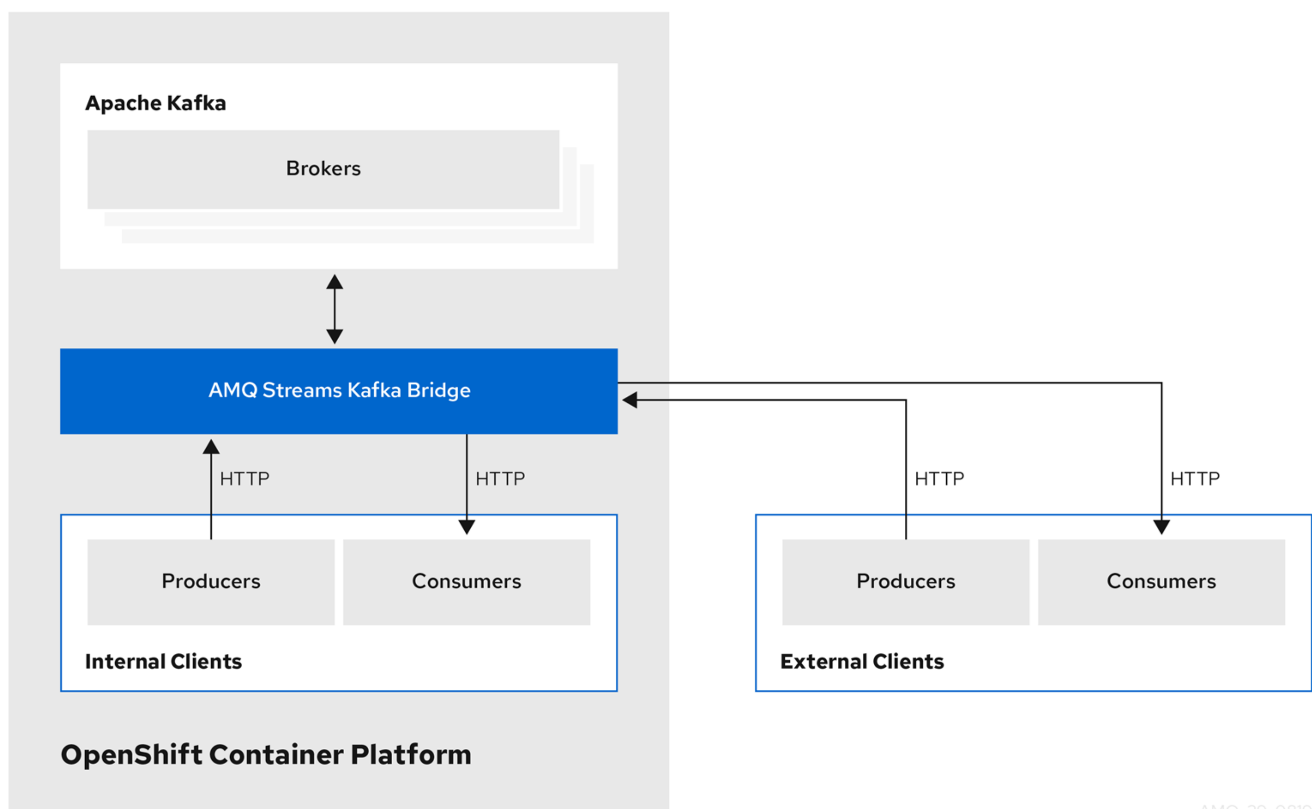
内部クライアント

内部クライアントとは、Kafka Bridge 自体と同じ OpenShift クラスターで実行されるコンテナーベースの HTTP クライアントのことです。内部クライアントは、ホストの Kafka Bridge および KafkaBridge のカスタムリソースで定義されたポートにアクセスできます。

外部クライアント

外部クライアントとは、Kafka Bridge がデプロイおよび実行される OpenShift クラスター外部で実行される HTTP クライアントのことです。外部クライアントは、OpenShift Route、ロードバランサーサービス、または Ingress を使用して Kafka Bridge にアクセスできます。

HTTP 内部および外部クライアントの統合



7.2.3. Kafka Bridge のセキュリティー保護

AMQ Streams には、現在 Kafka Bridge の暗号化、認証、または承認は含まれていません。そのため、外部クライアントから Kafka Bridge に送信されるリクエストは以下のようになります。

- 暗号化されず、HTTPS ではなく HTTP を使用する必要がある。
- 認証なしで送信される。

ただし、以下のような他の方法で Kafka Bridge をセキュアにできます。

- Kafka Bridge にアクセスできる Pod を定義する OpenShift ネットワークポリシー。
- 認証または承認によるリバースプロキシー (例: OAuth2 プロキシー)。
- API ゲートウェイ。
- TLS 終端をともなう Ingress または OpenShift ルート。

Kafka Bridge では、Kafka Broker への接続時に TLS 暗号化と、TLS および SASL 認証がサポートされます。OpenShift クラスター内で以下を設定できます。

- Kafka Bridge と Kafka クラスター間の TLS または SASL ベースの認証。
- Kafka Bridge と Kafka クラスター間の TLS 暗号化接続。

詳細は、「Kafka Bridge の設定」 を参照してください。

Kafka ブローカーで ACL を使用することで、Kafka Bridge を使用して消費および生成できるトピックを制限することができます。

7.2.4. OpenShift 外部の Kafka Bridge へのアクセス

デプロイメント後、AMQ Streams Kafka Bridge には同じ OpenShift クラスターで実行しているアプリケーションのみがアクセスできます。これらのアプリケーションは <kafka_bridge_name>-

bridge-service サービスを使用して API にアクセスします。

OpenShift クラスター外で実行しているアプリケーションに Kafka Bridge にアクセスできるようにするには、以下の機能のいずれかを作成して手動で公開できます。

- LoadBalancer または NodePort タイプのサービス
- Ingress リソース
- OpenShift ルート

サービスを作成する場合には、<kafka_bridge_name>-bridge-service サービスの セレクター から ラベルを使用して、サービスがトラフィックをルーティングする Pod を設定します。

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

(1) OpenShift クラスターでの Kafka Bridge カスタムリソースの名前。

7.2.5. Kafka Bridge へのリクエスト

データ形式と HTTP ヘッダーを指定し、有効なリクエストが Kafka Bridge に送信されるようにします。

7.2.5.1. コンテンツタイプヘッダー

API リクエストおよびレスポンス本文は、常に JSON としてエンコードされます。

- コンシューマー操作の実行時に、POST リクエストの本文が空でない場合は、以下の Content-Type ヘッダーが含まれている必要があります。

```
Content-Type: application/vnd.kafka.v2+json
```

- プロデューサー操作を行う場合、POST リクエストには、生成されるメッセージの埋め込みデータ形式を示すContent-Typeヘッダーを指定する必要があります。これは json または binary のいずれかになります。

埋め込みデータ形式	Content-Type ヘッダー
JSON	**Content-Type: application/vnd.kafka.json.v2+json**
バイナリー	**Content-Type: application/vnd.kafka.binary.v2+json**

次のセクションで説明どおり、埋め込みデータ形式はコンシューマーごとに設定されます。

POSTリクエストのボディが空の場合は、Content-Typeを設定しないでください。空のボディーを使用して、デフォルト値のコンシューマーを作成できます。

7.2.5.2. 埋め込みデータ形式

埋め込みデータ形式は、Kafka メッセージが Kafka Bridge によりプロデューサーからコンシューマーに HTTP で送信される際の形式です。サポートされる埋め込みデータ形式には、JSON とバイナリーの 2 種類があります。

/consumers/groupid エンドポイントを使用してコンシューマーを作成する場合、POST リクエスト本文で JSON またはバイナリーいずれかの埋め込みデータ形式を指定する必要があります。これは、以下の例のように format フィールドで指定します。

```
{
  "name": "my-consumer",
  "format": "binary", (1)
...
}
```

(1) バイナリー埋め込みデータ形式。

コンシューマーの作成時に指定する埋め込みデータ形式は、コンシューマーが消費する Kafka メッセージのデータ形式と一致する必要があります。

バイナリー埋め込みデータ形式を指定する場合は、以降のプロデューサーリクエストで、リクエスト本文にバイナリーデータが Base64 でエンコードされた文字列として含まれる必要があります。たとえば、/topics/topicname エンドポイントを使用してメッセージを送信する場合は、records.value を Base64 でエンコードする必要があります。

```
{
  "records": [
    {
      "key": "my-key",
      "value": "ZWR3YXJkdGhldGhyZWVsZWdnZWRjYXQ="
    },
  ]
}
```

プロデューサーリクエストには、埋め込みデータ形式に対応する Content-Type ヘッダーも含まれる必要があります (例: Content-Type: application/vnd.kafka.binary.v2+json)。

7.2.5.3. メッセージの形式

/topics エンドポイントを使用してメッセージを送信する場合は、records パラメーターのリクエストボディーにメッセージペイロードを入力します。

records パラメーターには、以下のオプションフィールドを含めることができます。

- Message headers
- Message key
- Message value
- Destination partition

/topicsへのPOSTリクエストの例

```
curl -X POST \
  http://localhost:8080/topics/my-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
```

```
-d '{
  "records": [
      {
        "key": "my-key",
        "value": "sales-lead-0001"
        "partition": 2
        "headers": [
          {
            "key": "key1",
            "value": "QXBhY2hlIEthZmthIGlzIHRoZSBib21iIQ==" (1)
          }
        ]
      },
  ]
}'
```

(1) バイナリー形式のヘッダー値。Base64 としてエンコードされます。

7.2.5.4. Accept ヘッダー

コンシューマーを作成したら、以降のすべての GET リクエストには Accept ヘッダーが以下のような形式で含まれる必要があります。

```
Accept: application/vnd.kafka.EMBEDDED-DATA-FORMAT.v2+json
```

EMBEDDED-DATA-FORMAT は json または binary です。

たとえば、サブスクライブされたコンシューマーのレコードを JSON 埋め込みデータ形式で取得する場合、この Accept ヘッダーが含まれるようにします。

```
Accept: application/vnd.kafka.json.v2+json
```

7.2.6. CORS

CORS (Cross-Origin Resource Sharing) を使用すると、Kafka Bridge HTTP の設定 で Kafka クラスターにアクセスするために許可されるメソッドおよび元の URL を指定できます。

Kafka Bridge の CORS 設定例

```
# ...
cors:
  allowedOrigins: "https://strimzi.io"
  allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  # ...
```

CORS では、異なるドメイン上のオリジンソース間での シンプルな リクエストおよび プリフライト リクエストが可能です。

シンプルなリクエストは、GET、HEAD、POSTの各メソッドを使った標準的なリクエストに適しています。

プリフライトリクエストは、実際のリクエストが安全に送信できることを確認する最初のチェックとして HTTP OPTIONS リクエストを送信します。確認時に、実際のリクエストが送信されます。プリフライトリクエストは、PUTやDELETEなど、より高い安全性が求められるメソッドや、非標準のヘッダーを使用するメソッドに適しています。

すべての要求には、HTTP 要求のソースであるヘッダーの Origin 値が必要です。

7.2.6.1. シンプルなリクエスト

たとえば、この単純なリクエストヘッダーは、オリジンを https://strimzi.io と指定します。

```
Origin: https://strimzi.io
```

ヘッダー情報がリクエストに追加されます。

```
curl -v -X GET HTTP-ADDRESS/bridge-consumer/records \
-H 'Origin: https://strimzi.io'\
-H 'content-type: application/vnd.kafka.v2+json'
```

Kafka Bridgeからの応答では、Access-Control-Allow-Originヘッダーが返されます。

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: * (1)
```

(1) アスタリスク(*)を返すと、リソースをどのドメインでもアクセスできることが分かります。

7.2.6.2. プリフライトリクエスト

最初のプリフライトリクエストは、OPTIONSメソッドを使ってKafka Bridgeに送信されます。HTTP OPTIONS リクエストはヘッダー情報を送信し、Kafka Bridge が実際のリクエストを許可することを確認します。

ここでは、プリフライトリクエストは https://strimzi.io から POST リクエストが有効であることを確認します。

```
OPTIONS /my-group/instances/my-user/subscription HTTP/1.1
Origin: https://strimzi.io
Access-Control-Request-Method: POST (1)
Access-Control-Request-Headers: Content-Type (2)
```

(1) Kafka Bridge では、実際のリクエストが POST リクエストであるとアラートされます。

(2) 実際のリクエストは Content-Type ヘッダーと共に送信されます。

OPTIONS は、プリフライトリクエストのヘッダー情報に追加されます。

```
curl -v -X OPTIONS -H 'Origin: https://strimzi.io' \
-H 'Access-Control-Request-Method: POST' \
-H 'content-type: application/vnd.kafka.v2+json'
```

Kafka Bridge は最初のリクエストに応答し、リクエストが受け入れられることを確認します。応答ヘッダーは、許可されるオリジン、メソッド、およびヘッダーを返します。

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://strimzi.io
Access-Control-Allow-Methods: GET,POST,PUT,DELETE,OPTIONS,PATCH
```

```
Access-Control-Allow-Headers: content-type
```

オリジンまたはメソッドが拒否されると、エラーメッセージが返されます。

プリフライトリクエストで確認されたため、実際のリクエストには Access-Control-Request-Method ヘッダーは必要ありませんが、元のヘッダーが必要です。

```
curl -v -X POST HTTP-ADDRESS/topics/bridge-topic \
-H 'Origin: https://strimzi.io' \
-H 'content-type: application/vnd.kafka.v2+json'
```

応答は、送信元 URL が許可されることを示します。

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: https://strimzi.io
```

関連情報

- fetch CORS 仕様

7.2.7. Kafka Bridge デプロイメント

Cluster Operator を使用して、Kafka Bridge を OpenShift クラスターにデプロイします。

Kafka Bridge をデプロイすると、Cluster Operator により OpenShift クラスターに Kafka Bridge オブジェクトが作成されます。オブジェクトには、デプロイメント、サービス、および Pod が含まれ、それぞれ Kafka Bridge のカスタムリソースに付与された名前が付けられます。

関連情報

- デプロイメントの手順は、『OpenShift での AMQ Streams のデプロイおよびアップグレード』の「Kafka Bridge の OpenShift クラスターへのデプロイ 」を参照してください。
- Kafka Bridge の設定に関する詳細は、「Kafka Bridge クラスターの設定」 を参照してください。
- KafkaBridge リソースのホストおよびポートの設定に関する詳細は、「Kafka Bridge の設

定」を参照してください。

- 外部クライアントの統合に関する詳細は、「OpenShift 外部の Kafka Bridge へのアクセス」を参照してください。

7.3. KAFKA BRIDGE クイックスタート

このクイックスタートを使用して、ローカルの開発環境で AMQ Streams の Kafka Bridge を試すことができます。以下の方法について説明します。

- OpenShift クラスターに Kafka Bridge をデプロイする。
- ポート転送を使用して Kafka Bridge サービスをローカルマシンに公開する。
- Kafka クラスターのトピックおよびパーティションへのメッセージを生成する。
- Kafka Bridge コンシューマーを作成する。
- 基本的なコンシューマー操作を実行する (たとえば、コンシューマーをトピックにサブスクライブする、生成したメッセージを取得するなど)。

このクイックスタートでは、HTTP リクエストはターミナルにコピーおよび貼り付けできる curl コマンドを使用します。OpenShift クラスターへのアクセスが必要です。

前提条件を確認し、本章に指定されている順序でタスクを行うようにしてください。

データ形式について

このクイックスタートでは、バイナリーではなく JSON 形式でメッセージを生成および消費します。リクエスト例で使用されるデータ形式および HTTP ヘッダーの詳細は、「Kafka Bridge へのリクエスト」を参照してください。

クイックスタートの前提条件

-

ローカルまたはリモート OpenShift クラスターにアクセスできるクラスター管理者権限が必要です。

- AMQ Streams がインストールされている必要があります。

- Cluster Operator によってデプロイされた稼働中の Kafka クラスターが OpenShift namespace に必要です。

- Entity Operator がデプロイされ、Kafka クラスターの一部として稼働している必要があります。

7.3.1. OpenShift クラスターへの Kafka Bridge のデプロイメント

AMQ Streams には、AMQ Streams Kafka Bridge の設定を指定する YAML サンプルが含まれています。このファイルに最小限の変更を加え、Kafka Bridge のインスタンスを OpenShift クラスターにデプロイします。

手順

1. examples/bridge/kafka-bridge.yaml ファイルを編集します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: quickstart 1
spec:
  replicas: 1
  bootstrapServers: <cluster-name>-kafka-bootstrap:9092 2
  http:
    port: 8080
```

1. Kafka Bridge のデプロイ時に、デプロイメントの名前およびその他の関連するリソースの名前に -bridge が追加されます。この例では、Kafka Bridge デプロイメントには quickstart-bridge という名前が付けられ、付随する Kafka Bridge サービスには quickstart-bridge-service という名前が付けられます。

2. bootstrapServers プロパティーで、Kafka クラスターの名前を <cluster-name> として入力します。

2. Kafka Bridge を OpenShift クラスターにデプロイします。

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

quickstart-bridge デプロイメント、サービス、および他の関連リソースが OpenShift クラスターに作成されます。

3. Kafka Bridge が正常にデプロイされたことを確認します。

```
oc get deployments
```

```
NAME                        READY  UP-TO-DATE  AVAILABLE  AGE
quickstart-bridge            1/1    1           1          34m
my-cluster-connect           1/1    1           1          24h
my-cluster-entity-operator   1/1    1           1          24h
#...
```

次のステップ

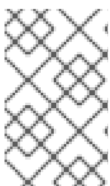
Kafka Bridge を OpenShift クラスターにデプロイしたら、Kafka Bridge サービスをローカルマシンに公開します。

関連情報

- Kafka Bridge の設定に関する詳細は、「Kafka Bridge クラスターの設定」 を参照してください。

7.3.2. Kafka Bridge サービスのローカルマシンへの公開

ポート転送を使用して AMQ Streams Kafka Bridge サービスを http://localhost:8080 のローカルマシンに公開します。



注記

ポート転送は、開発およびテストの目的でのみ適切です。

手順

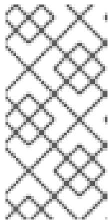
1. OpenShift クラスターの Pod の名前をリストします。

```
oc get pods -o name

pod/kafka-consumer
# ...
pod/quickstart-bridge-589d78784d-9jcnr
pod/strimzi-cluster-operator-76bcf9bc76-8dnfm
```

2. ポート 8080 で quickstart-bridge Pod に接続します。

```
oc port-forward pod/quickstart-bridge-589d78784d-9jcnr 8080:8080 &
```



注記

ローカルマシンのポート 8080 がすでに使用中の場合は、代わりの HTTP ポート (8008 など) を使用します。

これで、API リクエストがローカルマシンのポート 8080 から Kafka Bridge Pod のポート 8080 に転送されるようになります。

7.3.3. トピックおよびパーティションへのメッセージの作成

Kafka Bridge をデプロイし、そのサービスを公開した後、topics エンドポイントを使用して、トピックへのメッセージを JSON 形式で生成できます。以下に示すように、メッセージの宛先パーティションをリクエスト本文に指定できます。partitions エンドポイントは、全メッセージの単一の宛先パーティションをパスパラメーターとして指定する代替方法を提供します。

手順

1. テキストエディターを使用して、3 つのパーティションがある Kafka トピックの YAML 定義を作成します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: bridge-quickstart-topic
  labels:
    strimzi.io/cluster: <kafka-cluster-name> (1)
spec:
  partitions: 3 (2)
```

```
replicas: 1
config:
  retention.ms: 7200000
  segment.bytes: 1073741824
```

1. Kafka Bridge がデプロイされる Kafka クラスターの名前。

2. トピックのパーティション数。

2. ファイルを bridge-quickstart-topic.yaml として examples/topic ディレクトリーに保存します。

3. OpenShift クラスターにトピックを作成します。

```
oc apply -f examples/topic/bridge-quickstart-topic.yaml
```

4. Kafka Bridge を使用して、作成したトピックに 3 つのメッセージを生成します。

```
curl -X POST \
  http://localhost:8080/topics/bridge-quickstart-topic \
  -H 'content-type: application/vnd.kafka.json.v2+json' \
  -d '{
    "records": [
        {
            "key": "my-key",
            "value": "sales-lead-0001"
        },
        {
            "value": "sales-lead-0002",
            "partition": 2
        },
        {
            "value": "sales-lead-0003"
        }
    ]
}'
```

- sales-lead-0001 は、キーのハッシュに基づいてパーティションに送信されます。

- sales-lead-0002 は、パーティション 2 に直接送信されます。
- sales-lead-0003 は、ラウンドロビン方式を使用して bridge-quickstart-topic トピックのパーティションに送信されます。

5. リクエストが正常に行われると、Kafka Bridge は offsets アレイを 200 コードと application/vnd.kafka.v2+json の content-type ヘッダーとともに返します。各メッセージで、offsets アレイは以下を記述します。
   - メッセージが送信されたパーティション。
   - パーティションの現在のメッセージオフセット。

   応答例

```
#...
{
  "offsets":[
    {
      "partition":0,
      "offset":0
    },
    {
      "partition":2,
      "offset":0
    },
    {
      "partition":0,
      "offset":1
    }
  ]
}
```

追加のトピック要求

他の curl 要求を実行して、トピックおよびパーティションに関する情報を検索します。

トピックの一覧表示

```
curl -X GET \
  http://localhost:8080/topics
```

応答例

```
[
  "__strimzi_store_topic",
  "__strimzi-topic-operator-kstreams-topic-store-changelog",
  "bridge-quickstart-topic",
  "my-topic"
]
```

トピック設定およびパーティションの詳細の取得

```
curl -X GET \
  http://localhost:8080/topics/bridge-quickstart-topic
```

応答例

```
{
  "name": "bridge-quickstart-topic",
  "configs": {
    "compression.type": "producer",
    "leader.replication.throttled.replicas": "",
    "min.insync.replicas": "1",
    "message.downconversion.enable": "true",
    "segment.jitter.ms": "0",
    "cleanup.policy": "delete",
    "flush.ms": "9223372036854775807",
    "follower.replication.throttled.replicas": "",
    "segment.bytes": "1073741824",
    "retention.ms": "604800000",
    "flush.messages": "9223372036854775807",
    "message.format.version": "2.8-IV1",
    "max.compaction.lag.ms": "9223372036854775807",
    "file.delete.delay.ms": "60000",
    "max.message.bytes": "1048588",
    "min.compaction.lag.ms": "0",
    "message.timestamp.type": "CreateTime",
    "preallocate": "false",
    "index.interval.bytes": "4096",
    "min.cleanable.dirty.ratio": "0.5",
    "unclean.leader.election.enable": "false",
    "retention.bytes": "-1",
    "delete.retention.ms": "86400000",
```

```
    "segment.ms": "604800000",
    "message.timestamp.difference.max.ms": "9223372036854775807",
    "segment.index.bytes": "10485760"
  },
  "partitions": [
    {
      "partition": 0,
      "leader": 0,
      "replicas": [
        {
          "broker": 0,
          "leader": true,
          "in_sync": true
        },
        {
          "broker": 1,
          "leader": false,
          "in_sync": true
        },
        {
          "broker": 2,
          "leader": false,
          "in_sync": true
        }
      ]
    },
    {
      "partition": 1,
      "leader": 2,
      "replicas": [
        {
          "broker": 2,
          "leader": true,
          "in_sync": true
        },
        {
          "broker": 0,
          "leader": false,
          "in_sync": true
        },
        {
          "broker": 1,
          "leader": false,
          "in_sync": true
        }
      ]
    },
    {
      "partition": 2,
      "leader": 1,
      "replicas": [
        {
          "broker": 1,
          "leader": true,
          "in_sync": true
        },
```

```
        {
          "broker": 2,
          "leader": false,
          "in_sync": true
        },
        {
          "broker": 0,
          "leader": false,
          "in_sync": true
        }
      ]
    }
  ]
}
```

特定のトピックのパーティションを一覧表示する

```
curl -X GET \
  http://localhost:8080/topics/bridge-quickstart-topic/partitions
```

応答例

```
[
  {
    "partition": 0,
    "leader": 0,
    "replicas": [
      {
        "broker": 0,
        "leader": true,
        "in_sync": true
      },
      {
        "broker": 1,
        "leader": false,
        "in_sync": true
      },
      {
        "broker": 2,
        "leader": false,
        "in_sync": true
      }
    ]
  },
  {
    "partition": 1,
    "leader": 2,
    "replicas": [
```

```
      {
        "broker": 2,
        "leader": true,
        "in_sync": true
      },
      {
        "broker": 0,
        "leader": false,
        "in_sync": true
      },
      {
        "broker": 1,
        "leader": false,
        "in_sync": true
      }
    ]
  },
  {
    "partition": 2,
    "leader": 1,
    "replicas": [
      {
        "broker": 1,
        "leader": true,
        "in_sync": true
      },
      {
        "broker": 2,
        "leader": false,
        "in_sync": true
      },
      {
        "broker": 0,
        "leader": false,
        "in_sync": true
      }
    ]
  }
]
```

特定のトピックパーティションの詳細を一覧表示します。

```
curl -X GET \
  http://localhost:8080/topics/bridge-quickstart-topic/partitions/0
```

応答例

```
{
```

```
    "partition": 0,
    "leader": 0,
    "replicas": [
      {
        "broker": 0,
        "leader": true,
        "in_sync": true
      },
      {
        "broker": 1,
        "leader": false,
        "in_sync": true
      },
      {
        "broker": 2,
        "leader": false,
        "in_sync": true
      }
    ]
  }
```

特定のトピックパーティションのオフセットを一覧表示します。

```
curl -X GET \
  http://localhost:8080/topics/bridge-quickstart-topic/partitions/0/offsets
```

応答例

```
{
  "beginning_offset": 0,
  "end_offset": 1
}
```

次のステップ

トピックおよびパーティションへのメッセージを作成したら、Kafka Bridge コンシューマーを作成します。

その他のリソース

- API リファレンスドキュメントの「POST /topics/{topicname}」

- API リファレンスドキュメントの「POST /topics/{topicname}/partitions/{partitionid}」

7.3.4. Kafka Bridge コンシューマーの作成

Kafka クラスターで何らかのコンシューマー操作を実行するには、まず consumers エンドポイントを使用してコンシューマーを作成する必要があります。コンシューマーは Kafka Bridge コンシューマー と呼ばれます。

手順

1. bridge-quickstart-consumer-group という名前の新しいコンシューマーグループに Kafka Bridge コンシューマーを作成します。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "name": "bridge-quickstart-consumer",
    "auto.offset.reset": "earliest",
    "format": "json",
    "enable.auto.commit": false,
    "fetch.min.bytes": 512,
    "consumer.request.timeout.ms": 30000
  }'
```

- コンシューマーには bridge-quickstart-consumer という名前を付け、埋め込みデータ形式は json として設定します。

- 一部の基本的な設定が定義されます。

- コンシューマーはログへのオフセットに自動でコミットしません。これは、enable.auto.commit が false に設定されているからです。このクイックスタートでは、オフセットを跡で手作業でコミットします。

リクエストが正常に行われると、Kafka Bridge はレスポンス本文でコンシューマー ID (instance_id) とベース URL (base_uri) を 200 コードとともに返します。

応答例

```
#...
{
  "instance_id": "bridge-quickstart-consumer",
  "base_uri":"http://<bridge-name>-bridge-service:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer"
}
```

2. ベース URL (base_uri) をコピーし、このクイックスタートの他のコンシューマー操作で使用します。

次のステップ

上記で作成した Kafka Bridge コンシューマーをトピックにサブスクライブできます。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid}」

7.3.5. Kafka Bridge コンシューマーのトピックへのサブスクライブ

Kafka Bridge コンシューマーを作成したら、subscription エンドポイントを使用して、1 つ以上のトピックにサブスクライブします。サブスクライブすると、コンシューマーはトピックに生成されたすべてのメッセージの受信を開始します。

手順

- 前述の「トピックおよびパーティションへのメッセージの作成」の手順ですでに作成した bridge-quickstart-topic トピックに、コンシューマーをサブスクライブします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/subscription \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "topics": [
        "bridge-quickstart-topic"
    ]
}'
```

topics アレイには、例のような単一のトピック、または複数のトピックを含めることができます。正規表現に一致する複数のトピックにコンシューマーをサブスクライブする場合は、topics アレイの代わりに topic_pattern 文字列を使用できます。

リクエストが正常に行われると、Kafka Bridge によって 204 (No Content) コードのみが返されます。

次のステップ

Kafka Bridge コンシューマーをトピックにサブスクライブしたら、コンシューマーからメッセージを取得できます。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/subscription」

7.3.6. Kafka Bridge コンシューマーからの最新メッセージの取得

records エンドポイントからデータを要求して、Kafka Bridge コンシューマーから最新のメッセージを取得します。実稼働環境では、HTTP クライアントはこのエンドポイントを繰り返し (ループで) 呼び出すことができます。

手順

1. 「トピックおよびパーティションへのメッセージの生成」の説明に従い、Kafka Bridge コンシューマーに新たなメッセージを生成します。

2. GET リクエストを records エンドポイントに送信します。

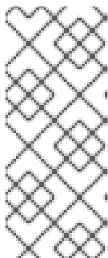
```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/records \
  -H 'accept: application/vnd.kafka.json.v2+json'
```

   Kafka Bridge コンシューマーを作成し、サブスクライブすると、最初の GET リクエストによって空のレスポンスが返されます。これは、ポーリング操作がリバランスプロセスを開始してパーティションを割り当てるからです。

3. 手順 2 を繰り返し、Kafka Bridge コンシューマーからメッセージを取得します。

Kafka Bridge は、レスポンス本文でメッセージのアレイ (トピック名、キー、値、パーティション、オフセットの記述) を 200 コードとともに返します。メッセージはデフォルトで最新のオフセットから取得されます。

```
HTTP/1.1 200 OK
content-type: application/vnd.kafka.json.v2+json
#...
[
  {
    "topic":"bridge-quickstart-topic",
    "key":"my-key",
    "value":"sales-lead-0001",
    "partition":0,
    "offset":0
  },
  {
    "topic":"bridge-quickstart-topic",
    "key":null,
    "value":"sales-lead-0003",
    "partition":0,
    "offset":1
  },
#...
```



注記

空のレスポンスが返される場合は、「トピックおよびパーティションへのメッセージの生成」の説明に従い、コンシューマーに対して追加のレコードを生成し、メッセージの取得を再試行します。

次のステップ

Kafka Bridge コンシューマーからメッセージを取得したら、ログへのオフセットをコミットします。

その他のリソース

- API リファレンスドキュメントの「GET /consumers/{groupid}/instances/{name}/records」

7.3.7. ログへのオフセットのコミット

offsets エンドポイントを使用して、Kafka Bridge コンシューマーによって受信されるすべてのメッセージのオフセットを手動でログにコミットします。この操作が必要なのは、前述の「Kafka Bridge コンシューマーの作成」で作成した Kafka Bridge コンシューマー が enable.auto.commit の設定で false に指定されているからです。

手順

- bridge-quickstart-consumer のオフセットをログにコミットします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer/offsets
```

リクエスト本文は送信されないので、オフセットはコンシューマーによって受信されたすべてのレコードに対してコミットされます。この代わりに、リクエスト本文に、オフセットをコミットするトピックおよびパーティションを指定するアレイ (OffsetCommitSeekList) を含めることができます。

リクエストが正常に行われると、Kafka Bridge は 204 コードのみを返します。

次のステップ

オフセットをログにコミットしたら、オフセットをシークのエンドポイントを試行します。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/offsets」

7.3.8. パーティションのオフセットのシーク

positions エンドポイントを使用して、Kafka Bridge コンシューマーを設定して、パーティションのメッセージを特定のオフセットから取得し、さらに最新のオフセットから取得します。これは Apache Kafka では、シーク操作と呼ばれます。

手順

1. quickstart-bridge-topic トピックで、パーティション 0 の特定のオフセットをシークします。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-
```

```
group/instances/bridge-quickstart-consumer/positions \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "offsets": [
      {
        "topic": "bridge-quickstart-topic",
        "partition": 0,
        "offset": 2
      }
    ]
}'
```

リクエストが正常に行われると、Kafka Bridge は 204 コードのみを返します。

2. GET リクエストを records エンドポイントに送信します。

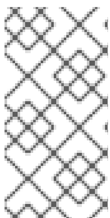
```
curl -X GET http://localhost:8080/consumers/bridge-quickstart-consumer-
group/instances/bridge-quickstart-consumer/records \
  -H 'accept: application/vnd.kafka.json.v2+json'
```

Kafka Bridge は、シークしたオフセットからのメッセージを返します。

3. 同じパーティションの最後のオフセットをシークし、デフォルトのメッセージ取得動作を復元します。この時点で、positions/end エンドポイントを使用します。

```
curl -X POST http://localhost:8080/consumers/bridge-quickstart-consumer-
group/instances/bridge-quickstart-consumer/positions/end \
  -H 'content-type: application/vnd.kafka.v2+json' \
  -d '{
    "partitions": [
      {
        "topic": "bridge-quickstart-topic",
        "partition": 0
      }
    ]
}'
```

リクエストが正常に行われると、Kafka Bridge は別の 204 コードを返します。



注記

また、positions/beginning エンドポイントを使用して、1 つ以上のパーティションの最初のオフセットをシークすることもできます。

次のステップ

このクイックスタートでは、AMQ Streams Kafka Bridge を使用して Kafka クラスターの一般的な操作をいくつか実行しました。これで、すでに作成した Kafka Bridge コンシューマーを削除 できます。

その他のリソース

- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/positions」
- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/positions/beginning」
- API リファレンスドキュメントの「POST /consumers/{groupid}/instances/{name}/positions/end」

7.3.9. Kafka Bridge コンシューマーの削除

このクイックスタート全体で使用した Kafka Bridge コンシューマーを削除します。

手順

- DELETE リクエストを instances エンドポイントに送信し、Kafka Bridge コンシューマーを削除します。

```
curl -X DELETE http://localhost:8080/consumers/bridge-quickstart-consumer-group/instances/bridge-quickstart-consumer
```

  リクエストが正常に行われると、Kafka Bridge は 204 コードを返します。

その他のリソース

- API リファレンスドキュメントの「DELETE /consumers/{groupid}/instances/{name}」

第8章 3SCALE での KAFKA BRIDGE の使用

Red Hat 3scale API Management をデプロイし、AMQ Streams の Kafka Bridge と統合できます。

8.1. 3SCALE での KAFKA BRIDGE の使用

Kafka Bridge のプレーンデプロイメントでは、認証または承認のプロビジョニングがなく、TLS 暗号化による外部クライアントへの接続はサポートされません。

3scale を使用すると、TLS によって Kafka Bridge のセキュリティーが保護され、認証および承認も提供されます。また、3scale との統合により、メトリクス、流量制御、請求などの追加機能も利用できるようになります。

3scale では、AMQ Streams へのアクセスを希望する外部クライアントからのリクエストに対して、各種タイプの認証を使用できます。3scale では、以下のタイプの認証がサポートされます。

標準 API キー

識別子およびシークレットトークンとして機能する、ランダムな単一文字列またはハッシュ。

アプリケーション ID とキーのペア

イミュータブルな識別子およびミュータブルなシークレットキー文字列。

OpenID Connect

委譲された認証のプロトコル。

既存の 3scale デプロイメントを使用する場合

3scale がすでに OpenShift にデプロイされており、Kafka Bridge と併用する場合は、正しく設定されていることを確認してください。

設定については、「Kafka Bridge を使用するための 3scale のデプロイメント」 を参照してください。

8.1.1. Kafka Bridge のサービス検出

3scale は、サービス検出を使用して統合されますが、これには 3scale が AMQ Streams および

Kafka Bridge と同じ OpenShift クラスターにデプロイされている必要があります。

AMQ Streams Cluster Operator デプロイメントには、以下の環境変数が設定されている必要があります。

- STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_LABELS
- STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_ANNOTATIONS

Kafka Bridge をデプロイすると、Kafka Bridge の REST インターフェースを公開するサービスは、3scale による検出にアノテーションとラベルを使用します。

- 3scale によって discovery.3scale.net=true ラベルが使用され、サービスが検出されます。
- アノテーションによってサービスに関する情報が提供されます。

OpenShift コンソールで設定を確認するには、Kafka Bridge インスタンスの Services に移動します。Annotations に、Kafka Bridge の OpenAPI 仕様へのエンドポイントが表示されます。

8.1.2. 3scale APIcast ゲートウェイポリシー

3scale は 3scale APIcast と併用されます。3scale APIcast は、Kafka Bridge の単一エントリーポイントを提供する 3scale とデプロイされる API ゲートウェイです。

APIcast ポリシーは、ゲートウェイの動作をカスタマイズするメカニズムを提供します。3scale には、ゲートウェイ設定のための標準ポリシーのセットが含まれています。また、独自のポリシーを作成することもできます。

APIcast ポリシーの詳細は、3scale ドキュメントの「API ゲートウェイの管理」を参照してください。

Kafka Bridge の APIcast ポリシー

3scale と Kafka Bridge との統合のポリシー設定例は policies_config.json ファイルに含まれてお

り、このファイルでは以下を定義します。

- Anonymous Access (匿名アクセス)
- Header Modification (ヘッダー変更)
- Routing (ルーティング)
- URL Rewriting (URL の書き換え)

ゲートウェイポリシーは、このファイルを使用して有効または無効に設定します。

この例をひな形として使用し、独自のポリシーを定義できます。

Anonymous Access (匿名アクセス)

Anonymous Access ポリシーでは、認証をせずにサービスが公開され、HTTP クライアントがデフォルトのクレデンシャル (匿名アクセス用) を提供しない場合に、このポリシーによって提供されます。このポリシーは必須ではなく、認証が常に必要であれば無効または削除できます。

Header Modification (ヘッダー変更)

Header Modification ポリシーを使用すると、既存の HTTP ヘッダーを変更したり、ゲートウェイを通過するリクエストまたはレスポンスへ新規ヘッダーを追加したりすることができます。3scale の統合では、このポリシーによって、HTTP クライアントから Kafka Bridge までゲートウェイを通過するすべてのリクエストにヘッダーが追加されます。

Kafka Bridge は、新規コンシューマー作成のリクエストを受信すると、URI のある base_uri フィールドが含まれる JSON ペイロードを返します。コンシューマーは後続のすべてのリクエストにこの URI を使用する必要があります。以下に例を示します。

```
{
  "instance_id": "consumer-1",
  "base_uri":"http://my-bridge:8080/consumers/my-group/instances/consumer1"
}
```

APIcast を使用する場合、クライアントは以降のリクエストをすべてゲートウェイに送信し、Kafka Bridge には直接送信しません。そのため URI には、ゲートウェイの背後にある Kafka

Bridge のアドレスではなく、ゲートウェイのホスト名が必要です。

Header Modification ポリシーを使用すると、ヘッダーが HTTP クライアントからリクエストに追加されるので、Kafka Bridge はゲートウェイホスト名を使用します。

たとえば、Forwarded: host=my-gateway:80;proto=http ヘッダーを適用すると、Kafka Bridge は以下をコンシューマーに提供します。

```
{
   "instance_id": "consumer-1",
   "base_uri":"http://my-gateway:80/consumers/my-group/instances/consumer1"
}
```

X-Forwarded-Path ヘッダーには、クライアントからゲートウェイへのリクエストに含まれる元のパスが含まれています。このヘッダーは、ゲートウェイが複数の Kafka Bridge インスタンスをサポートする場合に適用される Routing ポリシーに密接に関連します。

Routing (ルーティング)

Routing ポリシーは、複数の Kafka Bridge インスタンスがある場合に適用されます。コンシューマーが最初に作成された Kafka Bridge インスタンスにリクエストを送信する必要があるため、適切な Kafka Bridge インスタンスにリクエストを転送するようゲートウェイのルートをリクエストに指定する必要があります。

Routing ポリシーは各ブリッジインスタンスに名前を付け、ルーティングはその名前を使用して実行されます。Kafka Bridge のデプロイ時に、KafkaBridge カスタムリソースで名前を指定します。

たとえば、コンシューマーから以下への各リクエスト (X-Forwarded-Path を使用) について考えてみましょう。

http://my-gateway:80/my-bridge-1/consumers/my-group/instances/consumer1

この場合、各リクエストは以下に転送されます。

http://my-bridge-1-bridge-service:8080/consumers/my-group/instances/consumer1

URL Rewriting ポリシーはブリッジ名を削除しますが、これは、リクエストをゲートウェイから Kafka Bridge に転送するときにこのポリシーが使用されないからです。

URL Rewriting (URL の書き換え)

URL Rewiring ポリシーは、ゲートウェイから Kafka Bridge にリクエストが転送されるとき、クライアントから特定の Kafka Bridge インスタンスへのリクエストにブリッジ名が含まれないようにします。

ブリッジ名は、ブリッジが公開するエンドポイントで使用されません。

8.1.3. TLS の検証

TLS の検証用に APIcast を設定できます。これにはテンプレートを使用した APIcast の自己管理によるデプロイメントが必要になります。apicast サービスがルートとして公開されます。

TLS ポリシーを Kafka Bridge API に適用することもできます。

TLS 設定の詳細は、3scale ドキュメントの「API ゲートウェイの管理」を参照してください。

8.1.4. 3scale ドキュメント

3scale を Kafka Bridge と使用するためにデプロイする手順は、3scale をある程度理解していることを前提としています。

詳細は、3scale の製品ドキュメントを参照してください。

- 「Red Hat 3scale API Management 製品ドキュメント」

8.2. KAFKA BRIDGE を使用するための 3SCALE のデプロイメント

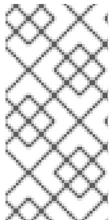
3scale を Kafka Bridge で使用するには、まず 3scale をデプロイし、次に Kafka Bridge API の検出を設定します。

また、3scale APIcast および 3scale toolbox も使用します。

- APIcast は、HTTP クライアントが Kafka Bridge API サービスに接続するための NGINX ベースの API ゲートウェイとして、3scale により提供されます。

- 3scale toolbox は設定ツールで、Kafka Bridge サービスの OpenAPI 仕様を 3scale にインポートするために使用されます。

このシナリオでは、AMQ Streams、Kafka、Kafka Bridge、および 3scale/APIcast を、同じ OpenShift クラスターで実行します。



注記

3scale がすでに Kafka Bridge と同じクラスターにデプロイされている場合は、デプロイメントの手順を省略して、現在のデプロイメントを使用できます。

前提条件

- AMQ Streams および Kafka が稼働している必要があります。

- Kafka Bridge がデプロイされている必要があります。

3scale デプロイメントの場合:

- 「Red Hat 3scale API Management Supported Configurations」を確認します。

- インストールには、cluster-admin ロール (system:admin など) を持つユーザーが必要です。

- 以下が記述されている JSON ファイルにアクセスできる必要があります。

  - Kafka Bridge OpenAPI 仕様 (openAPIV2.json)

- Kafka Bridge のヘッダー変更および Routing ポリシー (policies_config.json)

  GitHub で JSON ファイルを探します。

手順

1. 3scale API Management を OpenShift クラスターにデプロイします。

   a. 新規プロジェクトを作成するか、または既存プロジェクトを使用します。

   ```
   oc new-project my-project \
     --description="description" --display-name="display_name"
   ```

   b. 3scale をデプロイします。

   「3scale のインストール」 ガイドに記載の情報に従い、テンプレートまたは Operator を使用して OpenShift に 3scale をデプロイします。

   どの方法を使用する場合も、WILDCARD_DOMAIN パラメーターが OpenShift クラスターのドメインに設定されていることを確認してください。

   3scale 管理ポータルにアクセスするために表示される URL およびクレデンシャルを書き留めておきます。

2. 3scale が Kafka Bridge サービスを検出するように承認を付与します。

   ```
   oc adm policy add-cluster-role-to-user view system:serviceaccount:my-project:amp
   ```

3. 3scale が OpenShift コンソールまたは CLI から Openshift クラスターに正常にデプロイされたことを確認します。

   以下に例を示します。

   ```
   oc get deployment 3scale-operator
   ```

4. 3scale toolbox を設定します。

   a. 『Operating 3scale』 に記載の情報を使用して、3scale toolbox をインストールします。

   b. 3scale と対話できるように環境変数を設定します。

```
export REMOTE_NAME=strimzi-kafka-bridge (1)
export SYSTEM_NAME=strimzi_http_bridge_for_apache_kafka (2)
export TENANT=strimzi-kafka-bridge-admin (3)
export PORTAL_ENDPOINT=$TENANT.3scale.net (4)
export TOKEN=3scale access token (5)
```

(1) REMOTE_NAME は、3scale 管理ポータルのリモートアドレスに割り当てられた名前です。

(2) SYSTEM_NAME は、3scale toolbox で OpenAPI 仕様をインポートして作成される 3scale サービス/API の名前です。

(3) TENANT は、3scale 管理ポータルのテナント名です (https://$TENANT.3scale.net)。

(4) PORTAL_ENDPOINT は、3scale 管理ポータルを実行するエンドポイントです。

(5) TOKEN は、3scale toolbox または HTTP リクエストを介して対話するために 3scale 管理ポータルによって提供されるアクセストークンです。

   c. 3scale toolbox のリモート Web アドレスを設定します。

```
3scale remote add $REMOTE_NAME https://$TOKEN@$PORTAL_ENDPOINT/
```

これで、toolbox を実行するたびに、3scale 管理ポータルのエンドポイントアドレスを指定する必要がなくなりました。

5. Cluster Operator デプロイメントに、3scale が Kafka Bridge サービスを検出するために必要なラベルプロパティーおよびアノテーションプロパティーがあることを確認します。

```
#...
env:
- name: STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_LABELS
  value: |
  discovery.3scale.net=true
- name: STRIMZI_CUSTOM_KAFKA_BRIDGE_SERVICE_ANNOTATIONS
  value: |
  discovery.3scale.net/scheme=http
  discovery.3scale.net/port=8080
  discovery.3scale.net/path=/
  discovery.3scale.net/description-path=/openapi
#...
```

設定されていない場合は、OpenShift コンソールからプロパティーを追加するか、Cluster Operator および Kafka Bridge を再デプロイします。

6. 3scale で Kafka Bridge API サービスを検出します。

a. 3scale をデプロイしたときに提供されたクレデンシャルを使用して、3scale 管理ポータルにログインします。

b. 3scale 管理ポータルから、New API → Import from OpenShiftに移動します。ここで、Kafka Bridge サービスが表示されます。

c. Create Service をクリックします。

ページを更新して Kafka Bridge サービスを表示することが必要な場合もあります。

ここで、サービスの設定をインポートする必要があります。エディターからインポートしますが、ポータルを開いたまま正常にインポートされたことを確認します。

7. OpenAPI 仕様 (JSON ファイル) の Host フィールドを編集して、Kafka Bridge サービスの

ベース URL を使用します。

以下に例を示します。

```
"host": "my-bridge-bridge-service.my-project.svc.cluster.local:8080"
```

host URL に以下が正しく含まれることを確認します。

- Kafka Bridge 名 (my-bridge)
- プロジェクト名 (my-project)
- Kafka Bridge のポート (8080)

8. 3scale toolbox を使用して、更新された OpenAPI 仕様をインポートします。

```
3scale import openapi -k -d $REMOTE_NAME openapiv2.json -t myproject-my-bridge-bridge-service
```

9. サービスの Header Modification および Routing ポリシー (JSON ファイル) をインポートします。

   a. 3scale で作成したサービスの ID を特定します。

      ここでは、`jq` ユーティリティー を使用します。

```
export SERVICE_ID=$(curl -k -s -X GET
"https://$PORTAL_ENDPOINT/admin/api/services.json?access_token=$TOKEN" |
jq ".services[] | select(.service.system_name | contains(\"$SYSTEM_NAME\")) |
.service.id")
```

      ポリシーをインポートするときにこの ID が必要です。

   b. ポリシーをインポートします。

```
curl -k -X PUT
"https://$PORTAL_ENDPOINT/admin/api/services/$SERVICE_ID/proxy/policies.jso
n" --data "access_token=$TOKEN" --data-urlencode
policies_config@policies_config.json
```

10. 3scale 管理ポータルから、Integration → Configuration に移動し、Kafka Bridge サービスのエンドポイントとポリシーが読み込まれていることを確認します。

11. アプリケーションプランを作成するために、Applications → Create Application Plan に移動します。

12. アプリケーションを作成するために、Audience → Developer → Applications → Create Application に移動します。

    認証のユーザーキーを取得するためにアプリケーションが必要になります。

13. 実稼働環境用の手順: 実稼働環境のゲートウェイで API を利用可能にするには、設定をプロモートします。

```
3scale proxy-config promote $REMOTE_NAME $SERVICE_ID
```

14. API テストツールを使用して、コンシューマーの作成に呼び出しを使用する APIcast ゲートウェイと、アプリケーションに作成されたユーザーキーで、Kafka Bridge にアクセスできることを検証します。

    以下に例を示します。

```
https//my-project-my-bridge-bridge-service-3scale-apicast-
staging.example.com:443/consumers/my-group?
user_key=3dfc188650101010ecd7fdc56098ce95
```

    Kafka Bridge からペイロードが返されれば、コンシューマーが正常に作成されています。

```
{
 "instance_id": "consumer1",
 "base uri": "https//my-project-my-bridge-bridge-service-3scale-apicast-
staging.example.com:443/consumers/my-group/instances/consumer1"
}
```

ベース URI は、クライアントが以降のリクエストで使用するアドレスです。

第9章 CRUISE CONTROL によるクラスターのリバランス

Cruise Control を AMQ Streams クラスターにデプロイし、Kafka クラスターのリバランスに使用できます。

Cruise Control は、クラスターワークロードの監視、事前定義の制約を基にしたクラスターのリバランス、異常の検出および修正などの Kafka の操作を自動化するオープンソースのシステムです。Cruise Control は Load Monitor、Analyzer、Anomaly Detector、および Executor の主な 4 つのコンポーネントと、クライアントの対話に使用される REST API で構成されます。AMQ Streams は REST API を使用して、以下の Cruise Control 機能をサポートします。

- 複数の最適化ゴールから、最適化プロポーザルを生成します。
- 最適化プロポーザルを基にして Kafka クラスターのリバランスを行います。

異常検出、通知、独自ゴールの作成、トピックレプリケーション係数の変更などの、その他の Cruise Control の機能は現在サポートされていません。

AMQ Streams では 、設定ファイルのサンプル が提供されます。Cruise Control の YAML 設定ファイルのサンプルは、examples/cruise-control/ にあります。

9.1. CRUISE CONTROL とは

Cruise Control は、分散された Kafka クラスターを効率的に実行するための時間および労力を削減します。

通常、クラスターの負荷は時間とともに不均等になります。大量のメッセージトラフィックを処理するパーティションは、使用可能なブローカー全体で不均等に分散される可能性があります。クラスターを再分散するには、管理者はブローカーの負荷を監視し、トラフィックの多いパーティションを容量に余裕のあるブローカーに手作業で再割り当てします。

Cruise Control はクラスターのリバランス処理を自動化します。CPU、ディスク、およびネットワーク負荷を基にして、クラスターにおけるリソース使用のワークロードモデルを構築し、パーティションの割り当てをより均等にする、最適化プロポーザル (承認または拒否可能) を生成します。これらのプロポーザルの算出には、設定可能な最適化ゴールが複数使用されます。

最適化プロポーザルを承認すると、Cruise Control はそのプロポーザルを Kafka クラスターに適用します。クラスターのリバランス操作が完了すると、ブローカー Pod はより効率的に使用され、Kafka クラスターはより均等に分散されます。

その他のリソース

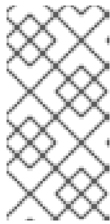
- Cruise Control の Wiki

9.2. 最適化ゴールの概要

Cruise Control は Kafka クラスターをリバランスするために、最適化ゴールを使用して、承認または拒否可能な最適化プロポーザルを生成します。

最適化ゴールは、Kafka クラスター全体のワークロード再分散およびリソース使用の制約です。AMQ Streams は、Cruise Control プロジェクトで開発された最適化ゴールのほとんどをサポートします。以下に、サポートされるゴールをデフォルトの優先度順に示します。

1. ラックアウェアネス (Rack Awareness)
2. トピックのセットに対するブローカーごとのリーダーレプリカの最小数
3. レプリカの容量
4. 容量: ディスク容量、ネットワークインバウンド容量、ネットワークアウトバウンド容量、CPU 容量
5. レプリカの分散
6. 潜在的なネットワーク出力
7. リソース分散: ディスク使用率の分散、ネットワークインバウンド使用率の分散、ネットワークアウトバウンド使用率の分散、CPU 使用率の分散。



注記

リソース分散ゴールは、ブローカーリソースで 容量制限 を使用して制御されます。

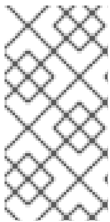
8. リーダーへの単位時間あたりバイト流入量の分散

9. トピックレプリカの分散

10. リーダーレプリカの分散

11. 優先リーダーの選択

各最適化ゴールの詳細は、Cruise Control Wiki の「Goals」を参照してください。



注記

ブローカー内ディスクゴール、独自のゴール、および Kafka アサイナーゴールはサポートされていません。

AMQ Streams カスタムリソースでのゴールの設定

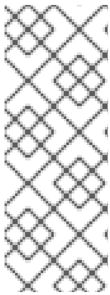
Kafka および KafkaRebalance カスタムリソースで最適化ゴールを設定します。Cruise Control には、必ず満たさなければならない ハード 最適化ゴールの設定と、メイン、デフォルト、および ユーザー提供の最適化ゴールの設定があります。リソースディストリビューションの最適化ゴール (ディスク、ネットワークインバウンド、ネットワークアウトバウンド、および CPU) は、ブローカーリソースの 容量制限 の対象となります。

以下のセクションでは、各ゴール設定の詳細を説明します。

ハードゴールおよびソフトゴール

ハードゴールは最適化プロポーザルで必ず満たさなければならないゴールです。ハードゴールとして設定されていないゴールはソフトゴールと呼ばれます。ソフトゴールは ベストエフォート 型のゴールと解釈できます。最適化プロポーザルで満たす必要はありませんが、最適化の計算に含まれます。すべてのハードゴールを満たし、1 つ以上のソフトゴールに違反する最適化プロポーザルは有効です。

Cruise Control は、すべてのハードゴールを満たし、優先度順にできるだけ多くのソフトゴールを満たす最適化プロポーザルを算出します。すべてのハードゴールを満たさない最適化プロポーザルは Cruise Control によって拒否され、ユーザーには送信されません。



注記

たとえば、クラスター全体でトピックのレプリカを均等に分散するソフトゴールがあるとします (トピックレプリカ分散のゴール)。このソフトゴールを無視すると、設定されたハードゴールがすべて有効になる場合、Cruise Control はこのソフトゴールを無視します。

Cruise Control では、以下のメイン最適化ゴールがハードゴールとして事前設定されています。

> RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal

Kafka.spec.cruiseControl.config の hard.goals プロパティーを編集し、Cruise Control のデプロイメント設定でハードゴールを設定します。

- Cruise Control から事前設定されたハードゴールを継承する場合は、Kafka.spec.cruiseControl.config に hard.goals プロパティーを指定しないでください。
- 事前設定されたハードゴールを変更するには、完全修飾ドメイン名を使用して、希望のゴールを hard.goals プロパティーに指定します。

ハード最適化ゴールの Kafka 設定例

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

```
brokerCapacity:
  inboundNetwork: 10000KB/s
  outboundNetwork: 10000KB/s
config:
  hard.goals: >
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
    com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
  # ...
```

ハードゴールの数を増やすと、Cruise Control が有効な最適化プロポーザルを生成する可能性が低くなります。

skipHardGoalCheck: true が KafkaRebalance カスタムリソースに指定された場合、Cruise Control はユーザー提供の最適化ゴールのリスト (KafkaRebalance.spec.goals 内) に設定済みのハードゴール (hard.goals) がすべて含まれていることをチェックしません。そのため、すべてではなく一部のユーザー提供の最適化ゴールが hard.goals リストにある場合、skipHardGoalCheck: true が指定されていてもハードゴールとして処理されます。

メイン最適化ゴール

メイン最適化ゴールはすべてのユーザーが使用できます。メイン最適化ゴールにリストされていないゴールは、Cruise Control 操作で使用できません。

Cruise Control の デプロイメント設定を変更しない限り、AMQ Streams は以下のメイン最適化ゴールを優先度順 (降順) に Cruise Control から継承します。

```
RackAwareGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal;
NetworkOutboundCapacityGoal; CpuCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal;
DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal;
NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; TopicReplicaDistributionGoal;
LeaderReplicaDistributionGoal; LeaderBytesInDistributionGoal; PreferredLeaderElectionGoal
```

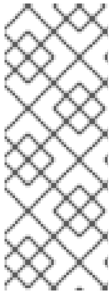
これらのゴールの 6 個が ハードゴール として事前設定されます。

複雑さを軽減するため、1 つ以上のゴールを KafkaRebalance リソースでの使用から完全に 除外する必要がある場合を除き、継承される主な最適化ゴールを使用することが推奨されます。必要な場合、メイン最適化ゴールの優先順位は デフォルトの最適化ゴール の設定で変更できます。

Cruise Control のデプロイメント設定で、必要に応じてメインの最適化ゴールを設定します(Kafka.spec.cruiseControl.config.goals)。

- 継承された主な最適化ゴールを許可する場合は、goals プロパティーを Kafka.spec.cruiseControl.config に指定しないでください。

- 継承した主な最適化目標を変更する必要がある場合は、goals設定オプションで、優先順位の高い順に目標のリストを指定します。



注記

継承された主な最適化ゴールを変更する場合、Kafka.spec.cruiseControl.config の hard.goals プロパティーに設定されたハードゴールがあれば、設定済みの主な最適化ゴールのサブセットになるようにする必要があります。そうでないと、最適化プロポーザルの生成時にエラーが発生します。

デフォルトの最適化ゴール

Cruise Conrol はデフォルトの最適化ゴール を使用して キャッシュされた最適化プロポーザル を生成します。キャッシュされた最適化プロポーザルの詳細は、「最適化プロポーザルの概要」 を参照してください。

ユーザー提供の最適化ゴール を KafkaRebalance カスタムリソースに設定すると、デフォルトの最適化ゴールを上書きできます。

Cruise Control のデプロイメント設定でdefault.goalsを指定しない限り、メインの最適化目標がデフォルトの最適化目標として使用されます。この場合、メイン最適化ゴールを使用して、キャッシュされた最適化プロポーザルが生成されます。

- 主な最適化目標をデフォルトの目標として使用するには、Kafka.spec.cruiseControl.configにdefault.goalsプロパティを指定しないでください。

- デフォルトの最適化ゴールを編集するには、Kafka.spec.cruiseControl.config の default.goals プロパティーを編集します。メイン最適化ゴールのサブセットを使用する必要があります。

デフォルト最適化ゴールの Kafka 設定例

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

```
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
      # ...
```

デフォルトの最適化ゴールの指定がない場合、メイン最適化ゴールを使用して、キャッシュされたプロポーザルが生成されます。

ユーザー提供の最適化ゴール

ユーザー提供の最適化ゴールは、特定の最適化プロポーザルの設定済みのデフォルトゴールを絞り込みます。必要に応じて、**KafkaRebalance**のカスタムリソースの**spec.goals**で設定することができます。

```
KafkaRebalance.spec.goals
```

ユーザー提供の最適化ゴールは、さまざまな状況の最適化プロポーザルを生成できます。たとえば、ディスクの容量やディスクの使用率を考慮せずに、Kafka クラスター全体でリーダーレプリカの分布を最適化したい場合があります。この場合、リーダーレプリカ分布の単一のユーザー提供ゴールが含まれる **KafkaRebalance** カスタムリソースを作成します。

ユーザー提供の最適化ゴールには以下が必要になります。

- 設定済みのハードゴールがすべて含まれるようにする必要があります。そうでないと、エラーが発生します。

- メイン最適化ゴールのサブセットである必要があります。

最適化プロポーザルの生成時に設定済みのハードゴールを無視するには、skipHardGoalCheck: true プロパティーを KafkaRebalance カスタムリソースに追加します。「最適化プロポーザルの生成」 を参照してください。

その他のリソース

- 「Cruise Control の設定」
- Cruise Control Wiki の「Configurations」

9.3. 最適化プロポーザルの概要

最適化プロポーザルは、パーティションのワークロードをブローカー間でより均等に分散することで、Kafka クラスターの負荷をより均等にするために提案された変更の概要です各最適化プロポーザルは、そのプロポーザルの生成に使用された 最適化ゴール のセットが基になっており、ブローカーリソースの設定済みの容量制限 の対象になります。

最適化プロポーザルは KafkaRebalance カスタムリソースの Status.Optimization Result プロパティーに含まれます。提供される情報は完全な最適化プロポーザルの概要になります。概要を使用して以下を決定します。

- 最適化プロポーザルの承認。プロポーザルを Kafka クラスターに適用し、クラスターリバランス操作を開始するよう Cruise Control が指示されます。
- 最適化プロポーザルの拒否。最適化ゴールを変更し、別のプロポーザルを生成できます。

最適化プロポーザルはすべてドライランです。最適化プロポーザルを最初に生成しないと、クラスターのリバランスを承認できません。生成できる最適化プロポーザルの数に制限はありません。

キャッシュされた最適化プロポーザル

Cruise Control は、設定済みのデフォルト最適化ゴールを基にして キャッシュされた最適化プロポーザル を維持します。キャッシュされた最適化プロポーザルはワークロードモデルから生成され、Kafka クラスターの現在の状況を反映するために 15 分ごとに更新されます。デフォルトの最適化ゴー

ルを使用して最適化プロポーザルを生成する場合、Cruise Control は最新のキャッシュされたプロポーザルを返します。

キャッシュされた最適化プロポーザルの更新間隔を変更するには、Cruise Control デプロイメント設定の proposal.expiration.ms 設定を編集します。更新間隔を短くすると、Cruise Control サーバーの負荷が増えますが、変更が頻繁に行われるクラスターでは、更新間隔を短くするよう考慮してください。

最適化プロポーザルの内容

最適化プロポーザルは 2 つのメインセクションで構成されます。

- summaryは、KafkaRebalanceリソースのstatusに格納されます。
- ブローカーの負荷は、データが JSON 文字列として含まれる ConfigMap に保存されます。

サマリーは、提案されたクラスターリバランスの概要を提供し、関係する変更の規模を示します。ブローカーの負荷は提案されたリバランスの前と後の値を表示するため、クラスターの各ブローカーへの影響を確認できます。

サマリー

以下の表は、最適化プロポーザルのサマリーセクションに含まれるプロパティーについて説明しています。

表9.1 最適化プロポーザルに含まれるプロパティー

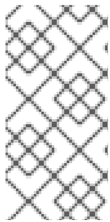
JSON プロパティー	説明
numIntraBrokerReplicaMovements	ディスクとクラスターのブローカーとの間で転送されるパーティションレプリカの合計数。 **リバランス操作中のパフォーマンスへの影響度**: 比較的高いが、**numReplicaMovements** よりも低い。
excludedBrokersForLeadership	サポートされていません。空のリストが返されます。
numReplicaMovements	個別のブローカー間で移動されるパーティションレプリカの数。 **リバランス操作中のパフォーマンスへの影響度**: 比較的高い。

JSON プロパティー	説明
onDemandBalancednessScoreBefore, onDemandBalancednessScoreAfter	最適化プロポーザルの生成前および生成後における、Kafka クラスターの全体的な **分散度** (balancedness) の値。 スコアは、違反した各ソフトゴールの **BalancednessScore** の合計を 100 から引いて算出されます。Cruise Control は、複数の要因を基にして **BalancednessScore** を各最適化ゴールに割り当てます。要因には、**default.goals** またはユーザー提供ゴールのリストでゴールの位置を示す優先順位が含まれます。 **Before** スコアは、Kafka クラスターの現在の設定を基にします。**After** スコアは、生成された最適化プロポーザルを基にします。
intraBrokerDataToMoveMB	同じブローカーのディスク間で移動される各パーティションレプリカのサイズの合計 (**numIntraBrokerReplicaMovements** も参照してください。 **リバランス操作中のパフォーマンスへの影響度**: 場合による。値が大きいほど、クラスターのリバランスの完了にかかる時間が長くなります。大量のデータを移動する場合、同じブローカーのディスク間で移動する方が個別のブローカー間で移動するよりも影響度が低くなります (**dataToMoveMB** 参照)。
recentWindows	最適化プロポーザルの基になるメトリクスウインドウの数。
dataToMoveMB	個別のブローカーに移動される各パーティションレプリカのサイズの合計 (**numReplicaMovements** も参照してください。 **リバランス操作中のパフォーマンスへの影響度**: 場合による。値が大きいほど、クラスターのリバランスの完了にかかる時間が長くなります。
monitoredPartitionsPercentage	最適化プロポーザルの対象となる Kafka クラスターのパーティションの割合 (パーセント)。**excludedTopics** の数が影響します。
excludedTopics	**KafkaRebalance** リソースの **spec.excludedTopicsRegex** プロパティーに正規表現を指定した場合、その式と一致するすべてのトピック名がここにリストされます。これらのトピックは、最適化プロポーザルではパーティションレプリカとリーダーの移動の計算からは除外されます。
numLeaderMovements	リーダーが別のレプリカに切り替えられるパーティションの数。ZooKeeper 設定の変更を伴います。 **リバランス操作中のパフォーマンスへの影響度**: 比較的低い。
excludedBrokersForReplicaMove	サポートされていません。空のリストが返されます。

ブローカーの負荷

ブローカーの負荷は、JSON 形式の文字列として ConfigMap (KafkaRebalance カスタムリソースと

同じ名前) に保存されます。この JSON 文字列は、各ブローカーのいくつかのメトリクスにリンクする各ブローカー ID のキーを持つ JSON オブジェクトで構成されます。各メトリクスは 3 つの値で構成されます。1 つ目は、最適化プロポーザルの適用前のメトリクスの値です。2 つ目はプロポーザルの適用後に期待される値、3 つ目は、最初の 2 つの値の差 (後の値から前の値を引いた) です。



注記

ConfigMap は、KafkaRebalance リソースが ProposalReady 状態にあると表示され、リバランスが完了すると残ります。

ConfigMap から JSON 文字列を抽出するには、jq コマンドラインの JSON パーサーツールを使用する次のコマンドを使用できます。

```
oc get configmap MY-REBALANCE -o json | jq '.["data"]["brokerLoad.json"]|fromjson|.'
```

以下の表は、最適化プロポーザルのブローカー負荷 ConfigMap に含まれるプロパティーについて説明しています。

JSON プロパティー	説明
leaders	パーティションリーダーであるこのブローカーのレプリカ数。
replicas	このブローカーのレプリカ数。
cpuPercentage	定義された容量の割合をパーセントで表す CPU 使用率。
diskUsedPercentage	定義された容量の割合をパーセントで表す ディスク 使用率。
diskUsedMB	絶対ディスク使用量 (MB 単位)
networkOutRate	ブローカーのネットワーク出力レートの合計。
leaderNetworkInRate	このブローカーのすべてのパーティションリーダーレプリカに対するネットワーク入力レート。
followerNetworkInRate	このブローカーのすべてのフォロワーレプリカに対するネットワーク入力レート。
potentialMaxNetworkOutRate	このブローカーが現在ホストしているレプリカすべてのリーダーであった場合に実現される、仮定上の最大ネットワーク出力レート。

関連情報

- 「最適化ゴールの概要」
- 「最適化プロポーザルの生成」
- 「最適化プロポーザルの承認」

9.4. リバランスパフォーマンスチューニングの概要

クラスターリバランスのパフォーマンスチューニングオプションを調整できます。これらのオプションは、リバランスのパーティションレプリカおよびリーダーシップの移動が実行される方法を制御し、また、リバランス操作に割り当てられた帯域幅も制御します。

パーティション再割り当てコマンド

最適化プロポーザル は、個別のパーティション再割り当てコマンドで構成されています。プロポーザルを 承認 すると、Cruise Control サーバーはこれらのコマンドを Kafka クラスターに適用します。

パーティション再割り当てコマンドは、以下のいずれかの操作で構成されます。

- パーティションの移動: パーティションレプリカとそのデータを新しい場所に転送します。パーティションの移動は、以下の 2 つの形式のいずれかになります。
  - ブローカー間の移動: パーティションレプリカを、別のブローカーのログディレクトリーに移動します。
  - ブローカー内の移動: パーティションレプリカを、同じブローカーの異なるログディレクトリーに移動します。
- リーダーシップの移動: パーティションのレプリカのリーダーを切り替えます。

Cruise Control によって、パーティション再割り当てコマンドがバッチで Kafka クラスターに発行されます。リバランス中のクラスターのパフォーマンスは、各バッチに含まれる各タイプの移動数に影響されます。

レプリカの移動ストラテジー

クラスターリバランスのパフォーマンスは、パーティション再割り当てコマンドのバッチに適用される レプリカ移動ストラテジー の影響も受けます。デフォルトでは、Cruise Control は BaseReplicaMovementStrategy を使用します。これは、生成された順序でコマンドを適用します。ただし、プロポーザルの初期に非常に大きなパーティションの再割り当てがある場合、このストラテジーによって他の再割り当ての適用が遅くなる可能性があります。

Cruise Control は、最適化プロポーザルに適用できる代替のレプリカ移動ストラテジーを 4 つ提供します。

- PrioritizeSmallReplicaMovementStrategy: サイズの昇順で再割り当てを並べ替えます。
- PrioritizeLargeReplicaMovementStrategy: サイズの降順で再割り当ての順序。
- PostponeUrpReplicaMovementStrategy: 非同期レプリカがないパーティションのレプリカの再割り当てを優先します。
- PrioritizeMinIsrWithOfflineReplicasStrategy: オフラインレプリカを持つ (At/Under) MinISR パーティションで再割り当てを優先します。この戦略は、Kafkaカスタムリソースの仕様でcruiseControl.config.concurrency.adjuster.min.isr.check.enabledがtrueに設定されている場合にのみ機能します。

これらのストラテジーをシーケンスとして設定できます。最初のストラテジーは、内部ロジックを使用して 2 つのパーティション再割り当ての比較を試みます。再割り当てが同等である場合は、順番を決定するために再割り当てをシーケンスの次のストラテジーに渡します。

リバランスチューニングオプション

Cruise Control には、上記のリバランスパラメーターを調整する設定オプションが複数あります。これらのチューニングオプションは、Cruise Control サーバー または 最適化プロポーザル レベルのいずれかに設定できます。

- クルーズコントロールのサーバー設定は、Kafkaのカスタムリソースである Kafka.spec.cruseControl.configで設定できます。
- 個々のリバランスのパフォーマンス設定は、KafkaRebalance.specで設定できます。

関連する設定の概要を以下に示します。

サーバーおよび KafkaRebalance の設定	説明	デフォルト値
num.concurrent.partition.movements.per.broker **concurrentPartitionMovementsPerBroker**	各パーティション再割り当てバッチでのブローカー間パーティション移動の最大数。	5
num.concurrent.intra.broker.partition.movements **concurrentIntraBrokerPartitionMovements**	各パーティション再割り当てバッチでのブローカー内パーティション移動の最大数。	2
num.concurrent.leader.movements **concurrentLeaderMovements**	各パーティション再割り当てバッチにおけるパーティションリーダー変更の最大数。	1000
default.replication.throttle **replicationThrottle**	パーティションの再割り当てに割り当てられる帯域幅 (バイト/秒単位)。	制限なし
default.replica.movement.strategies **replicaMovementStrategies**	パーティション再割り当てコマンドが、生成されたプロポーザルに対して実行される順番を決定するために使用されるストラテジー (優先順位順) の一覧。 サーバーの設定には、ストラテジークラスの完全修飾名をカンマ区切りの文字列で指定します（各クラス名の先頭に**com.linkedin.kafka.cruisecontrol.executor.strategy.**を追加します）。**KafkaRebalance** リソース設定には、YAML 配列のストラテジークラス名を使用します。	**BaseReplicaMovementStrategy**

デフォルト設定を変更すると、リバランスの完了までにかかる時間と、リバランス中の Kafka クラスターの負荷に影響します。値を小さくすると負荷は減りますが、かかる時間は長くなり、その逆も同様です。

その他のリソース

- 「CruiseControlSpec スキーマ参照」
- 「KafkaRebalanceSpec スキーマ参照」

9.5. CRUISE CONTROL の設定

Kafka.spec.cruiseControl の config プロパティーには設定オプションがキーとして含まれ、それらの値は以下の JSON タイプの 1 つになります。

- 文字列
- 数値
- ブール値

AMQ Streams によって直接管理されるオプション以外は、Cruise Control ドキュメント の「Configurations」セクションにリストされているすべてのオプションを指定および設定できます。ここに示されているキーの 1 つと同等の設定オプションまたはキーの 1 つで始まる設定オプションは、編集できません。

制限されたオプションが指定された場合、そのオプションは無視され、警告メッセージが Cluster Operator のログファイルに出力されます。すべてのサポートされるオプションは Cruise Control に渡されます。

Cruise Control の設定例

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

```
    default.goals: >
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal
    cpu.balance.threshold: 1.1
    metadata.max.age.ms: 300000
    send.buffer.bytes: 131072
  # ...
```

CORS (Corss-Origin Resource Sharing) の設定

CORS (Cross-Origin Resource Sharing) を使用すると、REST API へのアクセスに許可されるメソッドおよびアクセス元 URL を指定できます。

デフォルトでは、Cruise Control REST API の CORS は無効になっています。有効にすると、Kafka クラスターの状態の読み取り専用アクセスに対するGET リクエストのみが許可されます。そのため、AMQ Streams コンポーネントとは異なるオリジンで実行されている外部アプリケーションは、Cruise Control API への POST リクエストを行うことができません。ただし、これらのアプリケーションは、現在のクラスター負荷や最新の最適化プロポーザルなどの Kafka クラスターに関する読み取り専用情報へアクセスするための GET リクエストを行うことができます。

Cruise Control の CORS の有効化

Kafka.spec.cruiseControl.config で CORS を有効化および設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    config:
      webserver.http.cors.enabled: true
      webserver.http.cors.origin: "*"
      webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
    # ...
```

詳細は、Cruise Control Wiki の「REST APIs」を参照してください。

容量の設定

Cruise Control は 容量制限 を使用して、リソース分散の最適化ゴールが破損しているかどうかを判断します。このタイプには 4 つのゴールがあります。

- **DiskUsageDistributionGoal - Disk utilization distribution**
- **CpuUsageDistributionGoal - CPU utilization distribution**
- **NetworkInboundUsageDistributionGoal - Network inbound utilization distribution**
- **NetworkOutboundUsageDistributionGoal - Network outbound utilization distribution**

Kafka ブローカーリソースの容量制限は、Kafka.spec.cruiseControl の brokerCapacity プロパティーに指定します。これらはデフォルトで有効になっており、デフォルト値を変更できます。容量制限は、標準の OpenShift バイト単位 (K、M、G、および T) または同等 (2 のべき乗) の bibyte (Ki、Mi、Gi、および Ti) を使用して、以下のブローカーリソースに設定できます。

- disk: ブローカーごとのディスクストレージ（デフォルトは 100000 Mi）
- cpuUtilization: パーセントで表した CPU 使用率（デフォルトは 100）
- inboundNetwork: バイト毎秒単位のインバウンドネットワークスループット（デフォルトは 10000 KiB/s）
- outboundNetwork: バイト毎秒単位のアウトバウンドネットワークスループット（デフォルトは 10000 KiB/s）

AMQ Streams の Kafka ブーカーは同種であるため、Cruise Control は監視している各ブローカーに同じ容量制限を適用します。

bibyte 単位での Cruise Control brokerCapacity の設定例

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

```
cruiseControl:
  # ...
  brokerCapacity:
    disk: 100Gi
    cpuUtilization: 100
    inboundNetwork: 10000KiB/s
    outboundNetwork: 10000KiB/s
  # ...
```

その他のリソース

詳細は 「BrokerCapacity スキーマー参照」 を参照してください。

ロギングの設定

Cruise Control には独自の設定可能なロガーがあります。

- rootLogger.level

Cruise Control では Apache log4j 2 ロガー実装が使用されます。

logging プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は log4j.properties を使用して記述されます。logging.valueFrom.configMapKeyRef.name および logging.valueFrom.configMapKeyRef.key プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ここで、inline および external ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
```

```
spec:
  cruiseControl:
    # ...
    logging:
      type: inline
      loggers:
        rootLogger.level: "INFO"
    # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
  cruiseControl:
    # ...
    logging:
      type: external
      valueFrom:
        configMapKeyRef:
          name: customConfigMap
          key: cruise-control-log4j.properties
    # ...
```

Cruise Control REST API のセキュリティー

Cruise Control REST API は HTTP Basic 認証および SSL でセキュリティー保護され、Kafka ブローカーの停止などの破壊的な Cruise Control 操作からクラスターを保護します。

AMQ Streams の Cruise Control は、これらの設定を有効にしてのみ 使用することが推奨されます。以下で説明されている組み込み HTTP Basic 認証または SSL 設定を無効にしないでください。

- ビルトイン HTTP Basic 認証を無効にするには、webserver.security.enable を false に設定します。
- ビルトイン SSL を無効にするには、webserver.ssl.enable を false に設定します。

API 承認、認証、および SSL を無効にする Cruise Control の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    config:
      webserver.security.enable: false
      webserver.ssl.enable: false
# ...
```

9.6. CRUISE CONTROL のデプロイ

Cruise Control を AMQ Streams クラスターにデプロイするいは、Kafka リソースの cruiseControl プロパティーを使用して設定を定義した後、リソースを作成または更新します。

Kafka クラスターごとに Cruise Control のインスタンスを 1 つデプロイします。

前提条件

- OpenShift クラスター。
- 稼働中の Cluster Operator。

手順

1. Kafka リソースを編集し、cruiseControl プロパティーを追加します。

   設定可能なプロパティーは以下の例のとおりです。

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

```
spec:
  # ...
  cruiseControl:
    brokerCapacity: (1)
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
      # ...
    config: (2)
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal
        # ...
      cpu.balance.threshold: 1.1
      metadata.max.age.ms: 300000
      send.buffer.bytes: 131072
      # ...
    resources: (3)
      requests:
        cpu: 1
        memory: 512Mi
      limits:
        cpu: 2
        memory: 2Gi
    logging: (4)
        type: inline
        loggers:
          rootLogger.level: "INFO"
    template: (5)
      pod:
        metadata:
          labels:
            label1: value1
        securityContext:
          runAsUser: 1000001
          fsGroup: 0
        terminationGracePeriodSeconds: 120
    readinessProbe: (6)
      initialDelaySeconds: 15
      timeoutSeconds: 5
    livenessProbe: (7)
      initialDelaySeconds: 15
      timeoutSeconds: 5
# ...
```

1

ブローカーリソースの容量制限を指定します。詳細は、容量の設定 を参照してください。

2

クルーズコントロールの設定を定義します。デフォルトの最適化目標（default.goals）と、メインの最適化目標（goals）やハード目標（hard.goals）のカスタマイズを含みます。AMQ Streams によって直接管理されるものを除き、標準の Cruise

Cntrol 設定オプション をすべて提供できます。最適化ゴールの設定に関する詳細は、「最適化ゴールの概要」 を参照してください。

3

Cruise Control によって予約された CPU およびメモリーリソース。詳細は、「resources」 を参照してください。

4

ConfigMap より直接的 (inline) または間接的 (external) に追加されたロガーおよびログレベルを定義します。カスタム ConfigMap は、log4j.properties キー下に配置する必要があります。Cruise Controlには、rootLogger.levelという名前の単一のロガーがあります。ログレベルは INFO、ERROR、WARN、TRACE、DEBUG、FATAL、または OFF に設定できます。詳細は、「ロギングの設定」を参照してください。

5

デプロイメントテンプレートおよび Pod のカスタマイズ。

6

ヘルスチェックの readiness プローブ。

7

ヘルスチェックの liveness プローブ。

2. リソースを作成または更新します。

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

3. Cruise Control が正常にデプロイされたことを確認します。

```
oc get deployments -l app.kubernetes.io/name=cruise-control
```

自動作成されたトピック

以下の表は、Cruise Control のデプロイ時に自動作成される 3 つのトピックを表しています。これらのトピックは、Cruise Control が適切に動作するために必要であるため、削除または変更しないでください。

表9.2 自動作成されたトピック

自動作成されたトピック	作成元	機能
strimzi.cruisecontrol.metrics	AMQ Streams の Metrics Reporter	Metrics Reporter からの raw メトリクスを各 Kafka ブローカーに格納します。
strimzi.cruisecontrol.partitionmetricsamples	Cruise Control	各パーティションの派生されたメトリクスを格納します。これらは Metric Sample Aggregator によって作成されます。
strimzi.cruisecontrol.modeltrainingsamples	Cruise Control	クラスターワークロードモデル の作成に使用されるメトリクスサンプルを格納します。

Cruise Control に必要なレコードを削除しないようにするため、自動作成されたトピックではログの圧縮は無効になっています。

次のステップ

Cruise Control を設定およびデプロイした後、最適化プロポーザルを生成できます。

関連情報

「CruiseControlTemplate スキーマ参照」

9.7. 最適化プロポーザルの生成

KafkaRebalance リソースを作成または更新すると、Cruise Control は 設定済みの最適化ゴールを基にして、Kafka クラスターの 最適化プロポーザル を生成します。

最適化プロポーザルの情報を分析して、プロポーザルを承認するかどうかを決定します。

前提条件

- AMQ Streams クラスターに Cruise Control がデプロイされている 必要があります。
- 最適化ゴール が設定され、任意で ブローカーリソースに容量制限 が設定されている必要があります。

手順

手順

1. KafkaRebalance リソースを作成します。

   a. Kafka リソースに定義された デフォルトの最適化ゴール を使用するには、spec プロパティーを空のままにします。

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec: {}
   ```

   b. デフォルトのゴールを使用する代わりに ユーザー定義の最適化ゴール を設定するには、goals プロパティーを追加し、1 つ以上のゴールを入力します。

   以下の例では、ラックアウェアネス (Rack Awareness) およびレプリカの容量はユーザー定義の最適化ゴールとして設定されています。

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     goals:
       - RackAwareGoal
       - ReplicaCapacityGoal
   ```

   c. 設定されたハードゴールを無視するには、skipHardGoalCheck: trueプロパティを追加します。

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaRebalance
   metadata:
     name: my-rebalance
     labels:
       strimzi.io/cluster: my-cluster
   spec:
     goals:
       - RackAwareGoal
       - ReplicaCapacityGoal
     skipHardGoalCheck: true
   ```

2. リソースを作成または更新します。

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

Cluster Operator は Cruise Control から最適化プロポーザルを要求します。Kafka クラスターのサイズによっては処理に数分かかることがあります。

3. KafkaRebalance リソースの状態をチェックします。

```
oc describe kafkarebalance rebalance-cr-name
```

Cruise Control は以下の 2 つの状態の 1 つを返します。

- PendingProposal: 最適化プロポーザルが準備できているかどうかを確認するために、リバランス operator が Cruise Control API をポーリングしています。
- ProposalReady: 最適化プロポーザルを確認し、希望する場合は承認することができます。最適化プロポーザルは KafkaRebalance カスタムリソースの Status.Optimization Result プロパティーに含まれます。

4. 最適化プロポーザルを確認します。

```
oc describe kafkarebalance rebalance-cr-name
```

以下はプロポーザルの例になります。

```
Status:
  Conditions:
    Last Transition Time:  2020-05-19T13:50:12.533Z
    Status:                ProposalReady
    Type:                  State
  Observed Generation:     1
  Optimization Result:
    Data To Move MB:  0
    Excluded Brokers For Leadership:
    Excluded Brokers For Replica Move:
    Excluded Topics:
    Intra Broker Data To Move MB:       0
    Monitored Partitions Percentage:    100
```

```
  Num Intra Broker Replica Movements:   0
  Num Leader Movements:                 0
  Num Replica Movements:                26
  On Demand Balancedness Score After:   81.8666802863978
  On Demand Balancedness Score Before:  78.01176356230222
  Recent Windows:                       1
Session Id:                             05539377-ca7b-45ef-b359-e13564f1458c
```

Optimization Result セクションのプロパティーには、保留クラスターリバランス操作の詳細が表示されます。各プロパティーの説明は、「最適化プロポーザルの内容」を参照してください。

次のステップ

「最適化プロポーザルの承認」

その他のリソース

- 「最適化プロポーザルの概要」

9.8. 最適化プロポーザルの承認

状態が ProposalReady の場合、Cruise Control によって生成された最適化プロポーザルを承認できます。その後、Cruise Control は最適化プロポーザルを Kafka クラスターに適用して、パーティションをブローカーに再割り当てし、パーティションのリーダーを変更します。

注意

これはドライランではありません。最適化プロポーザルを承認する前に、以下を行う必要があります。

- 最新でない可能性があるため、プロポーザルを更新します。
- プロポーザルの内容を注意して確認します。

前提条件

- Cruise Control から 最適化プロポーザルを生成済み である必要があります。

- KafkaRebalance カスタムリソースの状態が ProposalReady である必要があります。

手順

承認する最適化プロポーザルに対して、以下の手順を実行します。

1. 最適化プロポーザルが新規生成された場合を除き、プロポーザルが Kafka クラスターの状態に関する現在の情報を基にしていることを確認します。これには、最適化プロポーザルを更新し、必ず最新のクラスターメトリクスを使用するようにします。

   a. OpenShift の KafkaRebalance リソースに refresh アノテーションを付けます。

   ```
   oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=refresh
   ```

   b. KafkaRebalance リソースの状態をチェックします。

   ```
   oc describe kafkarebalance rebalance-cr-name
   ```

   c. 状態が ProposalReady に変わるまで待ちます。

2. Cruise Control が適用する最適化プロポーザルを承認します。

   OpenShift の KafkaRebalance リソースにアノテーションを付けます。

   ```
   oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=approve
   ```

3. Cluster Operator は アノテーションが付けられたリソースを検出し、Cruise Control に Kafka クラスターのリバランスを指示します。

4. KafkaRebalance リソースの状態をチェックします。

   ```
   oc describe kafkarebalance rebalance-cr-name
   ```

5. Cruise Control は以下の 3 つの状態の 1 つを返します。

- Rebalaning: クラスターリバランス操作の実行中です。

- Ready: クラスターリバランス操作が正常に完了しました。同じKafkaRebalanceカスタムリソースを使用して別の最適化提案を生成するには、カスタムリソースにrefreshアノテーションを適用します。これにより、カスタムリソースはPendingProposalまたはProposalReadyの状態に移行します。その後、最適化プロポーザルを確認し、必要に応じて承認することができます。

- NotReady: エラーの発生については、「KafkaRebalance リソースの問題の修正」 を参照してください。

関連情報

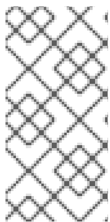
- 「最適化プロポーザルの概要」

- 「クラスターリバランスの停止」

9.9. クラスターリバランスの停止

クラスターリバランス操作を開始すると、完了まで時間がかかることがあり、Kafka クラスターの全体的なパフォーマンスに影響します。

実行中のクラスターリバランス操作を停止するには、stop アノテーションを KafkaRebalance カスタムリソースに適用します。これにより、現在のパーティション再割り当てのバッチ処理を完了し、リバランスを停止するよう Cruise Control が指示されます。リバランスの停止時、完了したパーティションの再割り当てはすで適用されています。そのため、Kafka クラスターの状態は、リバランス操作の開始前とは異なります。さらなるリバランスが必要な場合は、新しい最適化プロポーザルを生成してください。



注記

中間 (停止) 状態の Kafka クラスターのパフォーマンスは、初期状態の場合よりも悪くなる可能性があります。

前提条件

- KafkaRebalance カスタムリソースに approve アノテーションを付けて 最適化プロポーザルが承認済みである必要があります。

- KafkaRebalance カスタムリソースの状態が Rebalancing である必要があります。

手順

1. OpenShift の KafkaRebalance リソースにアノテーションを付けます。

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=stop
```

2. KafkaRebalance リソースの状態をチェックします。

```
oc describe kafkarebalance rebalance-cr-name
```

3. 状態が Stopped に変わるまで待ちます。

関連情報

- 「最適化プロポーザルの概要」

9.10. KAFKAREBALANCE リソースの問題の修正

KafkaRebalance リソースの作成時や、Cruise Control との対話中に問題が発生した場合、エラーとその修正方法の詳細がリソースの状態で報告されます。また、リソースも NotReady の状態に変わります。

クラスタのリバランス操作を続行するには、KafkaRebalanceリソース自体の問題、またはCruise Controlのデプロイメント全体の問題を解決する必要があります。問題には以下が含まれる可能性があります。

- KafkaRebalance リソースのパラメーターが正しく構成されていません。
- KafkaRebalance リソースに Kafka クラスターを指定するための strimzi.io/cluster ラベルがありません。
-

KafkaリソースのcruiseControlプロパティが見つからないため、Cruise Controlサーバーがデプロイされません。

- Cruise Control サーバーに接続できない。

問題の修正後、refresh アノテーションを KafkaRebalance リソースに付ける必要があります。「refresh」(更新) 中、Cruise Control サーバーから新しい最適化プロポーザルが要求されます。

前提条件

- 最適化プロポーザルが承認済みである必要があります。
- リバランス操作の KafkaRebalance カスタムリソースの状態が NotReady である必要があります。

手順

1. KafkaRebalance の状態からエラーに関する情報を取得します。

```
oc describe kafkarebalance rebalance-cr-name
```

2. KafkaRebalance リソースで問題の解決を試みます。
3. OpenShift の KafkaRebalance リソースにアノテーションを付けます。

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=refresh
```

4. KafkaRebalance リソースの状態をチェックします。

```
oc describe kafkarebalance rebalance-cr-name
```

5. 状態が PendingProposal になるまで待つか、直接 ProposalReady になるまで待ちます。

関連情報

- 「最適化プロポーザルの概要」

第10章 SERVICE REGISTRY を使用したスキーマの検証

AMQ Streams では、Red Hat Service Registry を使用できます。

Service Registry は、API およびイベント駆動型アーキテクチャー全体で標準的なイベントスキーマおよび API 設計を共有するためのデータストアです。Service Registry を使用して、クライアントアプリケーションからデータの構造を切り離し、REST インターフェースを使用して実行時にデータ型と API の記述を共有および管理できます。

Service Registry では、メッセージをシリアライズおよびデシリアライズするために使用されるスキーマが保存されます。その後、クライアントアプリケーションからスキーマを参照して、送受信されるメッセージとこれらのスキーマの互換性を維持するようにします。Service Registry によって、Kafka プロデューサーおよびコンシューマーアプリケーションの Kafka クライアントシリアライザーおよびデシリアライザーが提供されます。Kafka プロデューサーアプリケーションは、シリアライザーを使用して、特定のイベントスキーマに準拠するメッセージをエンコードします。Kafka コンシューマーアプリケーションはデシリアライザーを使用して、特定のスキーマ ID に基づいてメッセージが適切なスキーマを使用してシリアライズされたことを検証します。

アプリケーションがレジストリーからスキーマを使用できるようにすることができます。これにより、スキーマが一貫して使用されるようにし、実行時にデータエラーが発生しないようにします。

関連情報

- Service Registry のドキュメント
- Service Registry は、GitHub の Apicurio/Apicurio-registry で利用可能な Apicurio Registry オープンソースコミュニティープロジェクトで構築されます。

第11章 分散トレーシング

分散トレーシングを使用すると、分散システムのアプリケーション間で実行されるトランザクションの進捗を追跡できます。マイクロサービスのアーキテクチャーでは、トレーシングはサービス間のトランザクションの進捗を追跡します。トレースデータは、アプリケーションのパフォーマンスを監視し、ターゲットシステムおよびエンドユーザーアプリケーションの問題を調べるのに役立ちます。

AMQ Streams では、トレーシングによってメッセージのエンドツーエンドの追跡が容易になります。これは、ソースシステムから Kafka、さらに Kafka からターゲットシステムおよびアプリケーションへのメッセージの追跡です。これは、Grafana ダッシュボード で表示できるメトリクスやコンポーネントロガーを補完します。

AMQ Streams によるトレーシングのサポート方法

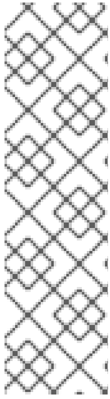
トレーシングのサポートは、以下のコンポーネントに組み込まれています。

- Kafka Connect
- MirrorMaker
- MirrorMaker 2.0
- AMQ Streams Kafka Bridge

カスタムリソースのテンプレート設定プロパティーを使用して、これらのコンポーネントのトレーシングを有効化および設定します。

Kafka プロデューサー、コンシューマー、および Kafka Streams API アプリケーションでトレーシングを有効にするには、AMQ Streams に含まれる OpenTracing Apache Kafka Client Instrumentation ライブラリーを使用してアプリケーションコードを インストルメント化 します。インストルメント化されると、クライアントはメッセージのトレースデータを生成します (メッセージの作成時やログへのオフセットの書き込み時など)。

トレースは、サンプリングストラテジーに従いサンプル化され、Jaeger ユーザーインターフェースで可視化されます。



注記

トレーシングは Kafka ブローカーではサポートされません。

AMQ Streams 以外のアプリケーションおよびシステムにトレーシングを設定する方法については、本章の対象外となります。この件についての詳細は、OpenTracing ドキュメント を参照し、「inject and extrac」を検索してください。

手順の概要

AMQ Streams のトレーシングを設定するには、以下の手順を順番に行います。

- クライアントのトレーシングを設定します。
  - Kafka クライアントの Jaeger トレーサーを初期化します。
- トレーサーでクライアントをインストルメント化します。
  - プロデューサーおよびコンシューマーをトレーシング用にインストルメント化します。
  - Kafka Streams アプリケーションをトレーシング用にインストルメント化します。
- MirrorMaker、Kafka Connect、Kafka Bridge のトレーシングを設定します。

前提条件

- Jaeger バックエンドコンポーネントが OpenShift クラスターにデプロイされている必要があります。デプロイメント手順の詳細は、Jaeger デプロイメントのドキュメントを参照してください。

11.1. OPENTRACING および JAEGER の概要

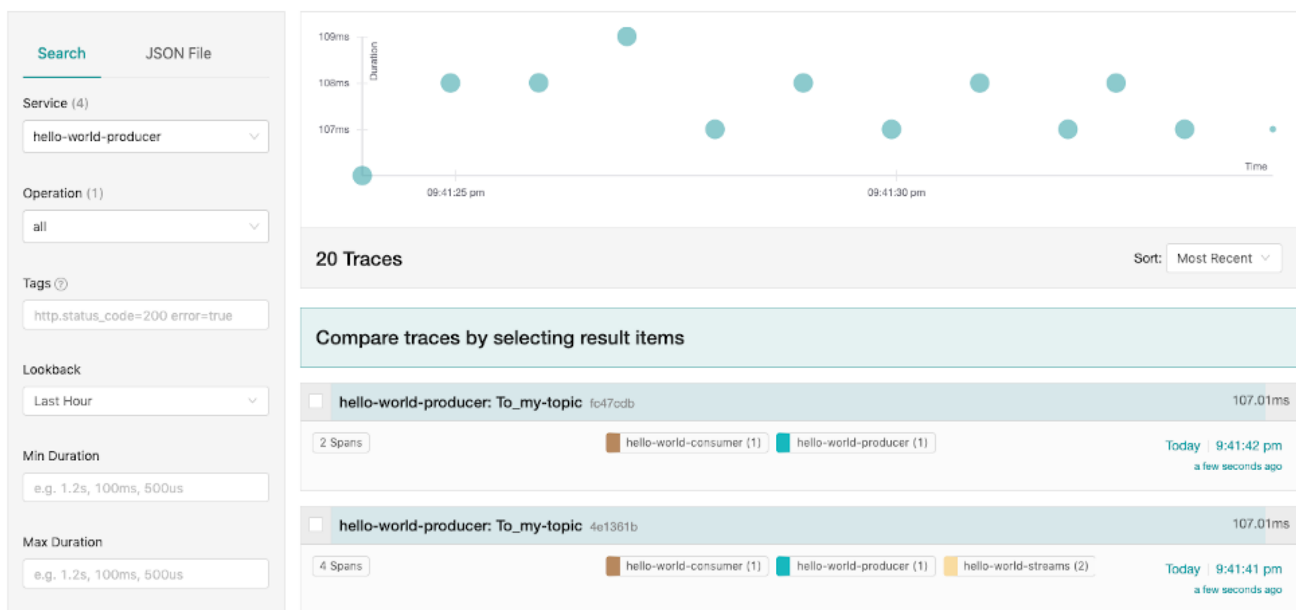
AMQ Streams では OpenTracing および Jaeger プロジェクトが使用されます。

OpenTracing は、トレーシングまたは監視システムに依存しない API 仕様です。

- **OpenTracing API は、アプリケーションコードを インストルメント化 するために使用されます。**
- **インストルメント化されたアプリケーションは、分散システム全体で個別のトランザクションの トレース を生成します。**
- **トレースは、特定の作業単位を定義する スパン で構成されます。**

Jaeger はマイクロサービスベースの分散システムのトレーシングシステムです。

- **Jaeger は OpenTracing API を実装し、インストルメント化のクライアントライブラリーを提供します。**
- **Jaeger ユーザーインターフェースを使用すると、トレースデータをクエリー、フィルター、および分析できます。**



関連情報

- OpenTracing
- Jaeger

11.2. KAFKA クライアントのトレーシング設定

Jaeger トレーサーを初期化し、分散トレーシング用にクライアントアプリケーションをインストルメント化します。

11.2.1. Kafka クライアント用の Jaeger トレーサーの初期化

一連のトレーシング環境変数を使用して、Jaeger トレーサーを設定および初期化します。

手順

各クライアントアプリケーションで以下を行います。

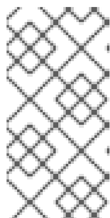
1. Jaeger の Maven 依存関係をクライアントアプリケーションの pom.xml ファイルに追加します。

```
<dependency>
    <groupId>io.jaegertracing</groupId>
    <artifactId>jaeger-client</artifactId>
    <version>1.5.0.redhat-00001</version>
</dependency>
```

2. トレーシング環境変数を使用して Jaeger トレーサーの設定を定義します。

3. 2. で定義した環境変数から、Jaeger トレーサーを作成します。

```
Tracer tracer = Configuration.fromEnv().getTracer();
```



注記

別の Jaeger トレーサーの初期化方法については、Java OpenTracing ライブラリーのドキュメントを参照してください。

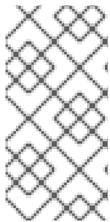
4. Jaeger トレーサーをグローバルトレーサーとして登録します。

```
GlobalTracer.register(tracer);
```

これで、Jaeger トレーサーはクライアントアプリケーションが使用できるように初期化されました。

11.2.2. トレーシングの環境変数

ここに示す環境変数は、Kafka クライアントに Jaeger トレーサーを設定するときに使用します。



注記

トレーシング環境変数は Jaeger プロジェクトの一部で、変更される場合があります。最新の環境変数については、Jaeger ドキュメントを参照してください。

プロパティー	必要性	説明
JAEGER_SERVICE_NAME	必要	Jaeger トレーサーサービスの名前。
JAEGER_AGENT_HOST	不要	UDP (User Datagram Protocol) を介した **jaeger-agent** との通信のためのホスト名。
JAEGER_AGENT_PORT	不要	UDP を介した **jaeger-agent** との通信に使用されるポート。
JAEGER_ENDPOINT	不要	**traces** エンドポイント。クライアントアプリケーションが **jaeger-agent** を迂回し、**jaeger-collector** に直接接続する場合にのみ、この変数を定義します。
JAEGER_AUTH_TOKEN	不要	エンドポイントに bearer トークンとして送信する認証トークン。
JAEGER_USER	不要	Basic 認証を使用する場合にエンドポイントに送信するユーザー名。

プロパティー	必要性	説明
JAEGER_PASSWORD	不要	Basic 認証を使用する場合にエンドポイントに送信するパスワード。
JAEGER_PROPAGATION	不要	トレースコンテキストの伝播に使用するカンマ区切りの形式リスト。デフォルトは標準の Jaeger 形式です。有効な値は、**jaeger**、**b3**、および **w3c** です。
JAEGER_REPORTER_LOG_SPANS	不要	レポーターがスパンも記録する必要があるかどうかを示します。
JAEGER_REPORTER_MAX_QUEUE_SIZE	不要	レポーターの最大キューサイズ。
JAEGER_REPORTER_FLUSH_INTERVAL	不要	レポーターのフラッシュ間隔 (ミリ秒単位)。Jaeger レポーターがスパンバッチをフラッシュする頻度を定義します。
JAEGER_SAMPLER_TYPE	不要	クライアントトレースに使用するサンプリングストラテジー。 • Constant • Probabilistic • Rate Limiting • Remote (デフォルト) すべてのトレースをサンプリングするには、Constant サンプリングストラテジーを使用し、パラメーターを 1 にします。 詳細は、Jaeger ドキュメントを参照してください。
JAEGER_SAMPLER_PARAM	不要	サンプラーのパラメーター (数値)。
JAEGER_SAMPLER_MANAGER_HOST_PORT	不要	リモートサンプリングストラテジーを選択する場合に使用するホスト名およびポート。

プロパティー	必要性	説明
JAEGER_TAGS	不要	報告されたすべてのスパンに追加されるトレーサーレベルのタグのカンマ区切りリスト。 また、**${envVarName:default}** という形式で環境変数を参照することもできます。**:default** はオプションで、環境変数が見つからない場合に使用する値を指定します。

関連情報

- [「Kafka クライアント用の Jaeger トレーサーの初期化」]

11.3. トレーサーでの KAFKA クライアントのインストルメント化

Kafka プロデューサーとコンシューマークライアント、および Kafka Streams API アプリケーションを分散トレーシング用にインストルメント化します。

11.3.1. トレーシングのための Kafka プロデューサーおよびコンシューマーのインストルメント化

Decorator パターンまたは Interceptor を使用して、Java プロデューサーおよびコンシューマーアプリケーションコードをトレーシング用にインストルメント化します。

手順

各プロデューサーおよびコンシューマーアプリケーションのアプリケーションコードで以下を行います。

1. OpenTracing の Maven 依存関係を、プロデューサーまたはコンシューマーの pom.xml ファイルに追加します。

```
<dependency>
   <groupId>io.opentracing.contrib</groupId>
   <artifactId>opentracing-kafka-client</artifactId>
   <version>0.1.15.redhat-00002</version>
</dependency>
```

2. Decorator パターンまたは Interceptor のいずれかを使用して、クライアントアプリケーションコードをインストルメント化します。

- Decorator パターンを使用する場合は以下を行います。

```
// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer:
TracingKafkaProducer<Integer, String> tracingProducer = new
TracingKafkaProducer<>(producer,
        tracer);

// Send:
tracingProducer.send(...);

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>
(consumerProps);

// Create an instance of the TracingKafkaConsumer:
TracingKafkaConsumer<Integer, String> tracingConsumer = new
TracingKafkaConsumer<>(consumer,
        tracer);

// Subscribe:
tracingConsumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = tracingConsumer.poll(1000);

// Retrieve SpanContext from polled record (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext =
TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

- Interceptor を使用する場合は以下を使用します。

```
// Register the tracer with GlobalTracer:
GlobalTracer.register(tracer);

// Add the TracingProducerInterceptor to the sender properties:
senderProps.put(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG,
          TracingProducerInterceptor.class.getName());

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Send:
producer.send(...);
```

```
// Add the TracingConsumerInterceptor to the consumer properties:
consumerProps.put(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG,
        TracingConsumerInterceptor.class.getName());

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>
(consumerProps);

// Subscribe:
consumer.subscribe(Collections.singletonList("messages"));

// Get messages:
ConsumerRecords<Integer, String> records = consumer.poll(1000);

// Retrieve the SpanContext from a polled message (consumer side):
ConsumerRecord<Integer, String> record = ...
SpanContext spanContext =
TracingKafkaUtils.extractSpanContext(record.headers(), tracer);
```

11.3.1.1. Decorator パターンのカスタムスパン名

スパン は Jaeger の論理作業単位で、操作名、開始時間、および期間が含まれます。

プロデューサーとコンシューマーのアプリケーションをインストルメントするためにDecoratorパターンを使用するには、TracingKafkaProducerおよびTracingKafkaConsumerオブジェクトを作成する際に、追加の引数としてBiFunctionオブジェクトを渡して、カスタムスパン名を定義します。OpenTracing の Apache Kafka Client Instrumentation ライブラリーには、複数の組み込みスパン名が含まれています。

例: カスタムスパン名を使用した Decorator パターンでのクライアントアプリケーションコードのインストルメント化

```
// Create a BiFunction for the KafkaProducer that operates on (String operationName,
ProducerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ProducerRecord, String> producerSpanNameProvider =
    (operationName, producerRecord) -> "CUSTOM_PRODUCER_NAME";

// Create an instance of the KafkaProducer:
KafkaProducer<Integer, String> producer = new KafkaProducer<>(senderProps);

// Create an instance of the TracingKafkaProducer
TracingKafkaProducer<Integer, String> tracingProducer = new TracingKafkaProducer<>
(producer,
        tracer,
        producerSpanNameProvider);
```

```
// Spans created by the tracingProducer will now have "CUSTOM_PRODUCER_NAME" as the span name.

// Create a BiFunction for the KafkaConsumer that operates on (String operationName, ConsumerRecord consumerRecord) and returns a String to be used as the name:

BiFunction<String, ConsumerRecord, String> consumerSpanNameProvider =
    (operationName, consumerRecord) -> operationName.toUpperCase();

// Create an instance of the KafkaConsumer:
KafkaConsumer<Integer, String> consumer = new KafkaConsumer<>(consumerProps);

// Create an instance of the TracingKafkaConsumer, passing in the consumerSpanNameProvider BiFunction:

TracingKafkaConsumer<Integer, String> tracingConsumer = new TracingKafkaConsumer<>
(consumer,
        tracer,
        consumerSpanNameProvider);

// Spans created by the tracingConsumer will have the operation name as the span name, in upper-case.
// "receive" -> "RECEIVE"
```

11.3.1.2. ビルトインスパン名

カスタムスパン名を定義するとき、**ClientSpanNameProvider** クラスで以下の **BiFunctions** を使用できます。**spanNameProvider** を指定しないと、**CONSUMER_OPERATION_NAME** および **PRODUCER_OPERATION_NAME** が使用されます。

BiFunction	説明
CONSUMER_OPERATION_NAME、PRODUCER_OPERATION_NAME	**operationName** をスパン名として返します。コンシューマーには「receive」、プロデューサーの場合は「send」を返します。
CONSUMER_PREFIXED_OPERATION_NAME(String prefix), PRODUCER_PREFIXED_OPERATION_NAME(String prefix)	**prefix** および **operationName** の文字列連結を返します。
CONSUMER_TOPIC, PRODUCER_TOPIC	メッセージの送信先または送信元となったトピックの名前を **(record.topic())** 形式で返します。
PREFIXED_CONSUMER_TOPIC(String prefix), PREFIXED_PRODUCER_TOPIC(String prefix)	**prefix** およびトピック名の文字列連結を **(record.topic())** 形式で返します。

BiFunction	説明
CONSUMER_OPERATION_NAME_TOPIC, PRODUCER_OPERATION_NAME_TOPIC	操作名およびトピック名を "operationName - record.topic()" で返します。
CONSUMER_PREFIXED_OPERATION_NAME_TOPIC(String prefix), PRODUCER_PREFIXED_OPERATION_NAME_TOPIC(String prefix)	prefix および "operationName - record.topic()" の文字列連結を返します。

11.3.2. Kafka Streams アプリケーションをトレース用にインストルメント化

本セクションでは、分散トレーシングのために Kafka Streams API アプリケーションをインストルメント化する方法を説明します。

手順

各 Kafka Streams API アプリケーションで以下を行います。

1. opentracing-kafka-streams 依存関係を、Kafka Streams API アプリケーションの pom.xml ファイルに追加します。

```
<dependency>
  <groupId>io.opentracing.contrib</groupId>
  <artifactId>opentracing-kafka-streams</artifactId>
  <version>0.1.15.redhat-00002</version>
</dependency>
```

2. TracingKafkaClientSupplier サプライヤーインターフェースのインスタンスを作成します。

```
KafkaClientSupplier supplier = new TracingKafkaClientSupplier(tracer);
```

3. サプライヤーインターフェースを KafkaStreams に提供します。

```
KafkaStreams streams = new KafkaStreams(builder.build(), new
StreamsConfig(config), supplier);
streams.start();
```

11.4. MIRRORMAKER、KAFKA CONNECT、および KAFKA BRIDGE のトレーシング設定

分散トレーシングは、MirrorMaker、MirrorMaker 2.0、Kafka Connect、および AMQ Streams Kafka Bridge でサポートされます。

MirrorMaker および MirrorMaker 2.0 でのトレーシング

MirrorMaker および MirrorMaker 2.0 では、メッセージはソースクラスターからターゲットクラスターにトレーシングされます。トレースデータは、MirrorMaker または MirrorMaker 2.0 コンポーネントを出入りするメッセージを記録します。

Kafka Connect でのトレーシング

Kafka Connect により生成および消費されるメッセージのみがトレーシングされます。Kafka Connect と外部システム間で送信されるメッセージをトレーシングするには、これらのシステムのコネクターでトレーシングを設定する必要があります。詳細は、「Kafka Connect の設定」 を参照してください。

Kafka Bridge でのトレーシング

Kafka Bridge によって生成および消費されるメッセージがトレーシングされます。Kafka Bridge を介してメッセージを送受信するクライアントアプリケーションから受信する HTTP リクエストもトレーシングされます。エンドツーエンドのトレーシングを設定するために、HTTP クライアントでトレーシングを設定する必要があります。

11.4.1. MirrorMaker、Kafka Connect、および Kafka Bridge リソースでのトレーシングの有効化

KafkaMirrorMaker、KafkaMirrorMaker2、KafkaConnect、KafkaBridgeのカスタムリソースの設定を更新し、各リソースにJaegerトレーサーサービスを指定および設定する。OpenShift クラスターでトレーシングが有効になっているリソースを更新すると、2 つのイベントがトリガーされます。

- インターセプタークラスは、MirrorMaker、MirrorMaker 2.0、Kafka Connect、または AMQ Streams Kafka Bridge の統合されたコンシューマーおよびプロデューサーで更新されます。

- MirrorMaker、MirrorMaker 2.0 および Kafka Connect では、リソースに定義されたトレーシング設定に基づいて、Jaeger トレーサーがトレーシングエージェントによって初期化されます。

- Kafka Bridge では、リソースに定義されたトレーシング設定に基づいて、Jaeger トレーサーが Kafka Bridge によって初期化されます。

手順

以下の手順を、KafkaMirrorMaker、KafkaMirrorMaker2、KafkaConnect、およびKafkaBridgeリソースごとに実行します。

1. spec.template プロパティーで、Jaeger トレーサーサービスを設定します。以下に例を示します。

Kafka Connect の Jaeger トレーサー設定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  template:
    connectContainer: (1)
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing: (2)
    type: jaeger
  #...
```

MirrorMaker の Jaeger トレーサー設定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  #...
  template:
    mirrorMakerContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
```

```
      - name: JAEGER_AGENT_PORT
        value: "6831"
  tracing:
    type: jaeger
#...
```

MirrorMaker 2.0 の Jaeger トレーサー設定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mm2-cluster
spec:
  #...
  template:
    connectContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
        - name: JAEGER_AGENT_PORT
          value: "6831"
  tracing:
    type: jaeger
#...
```

Kafka Bridge の Jaeger トレーサー設定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  #...
  template:
    bridgeContainer:
      env:
        - name: JAEGER_SERVICE_NAME
          value: my-jaeger-service
        - name: JAEGER_AGENT_HOST
          value: jaeger-agent-name
```

```
          - name: JAEGER_AGENT_PORT
            value: "6831"
    tracing:
      type: jaeger
#...
```

1. トレーシング環境変数をテンプレートの設定プロパティーとして使用します。

2. spec.tracing.type プロパティーを jaeger に設定します。

2. リソースを作成または更新します。

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

関連情報

- 「ContainerTemplate スキーマ参照」
- 「OpenShift リソースのカスタマイズ」

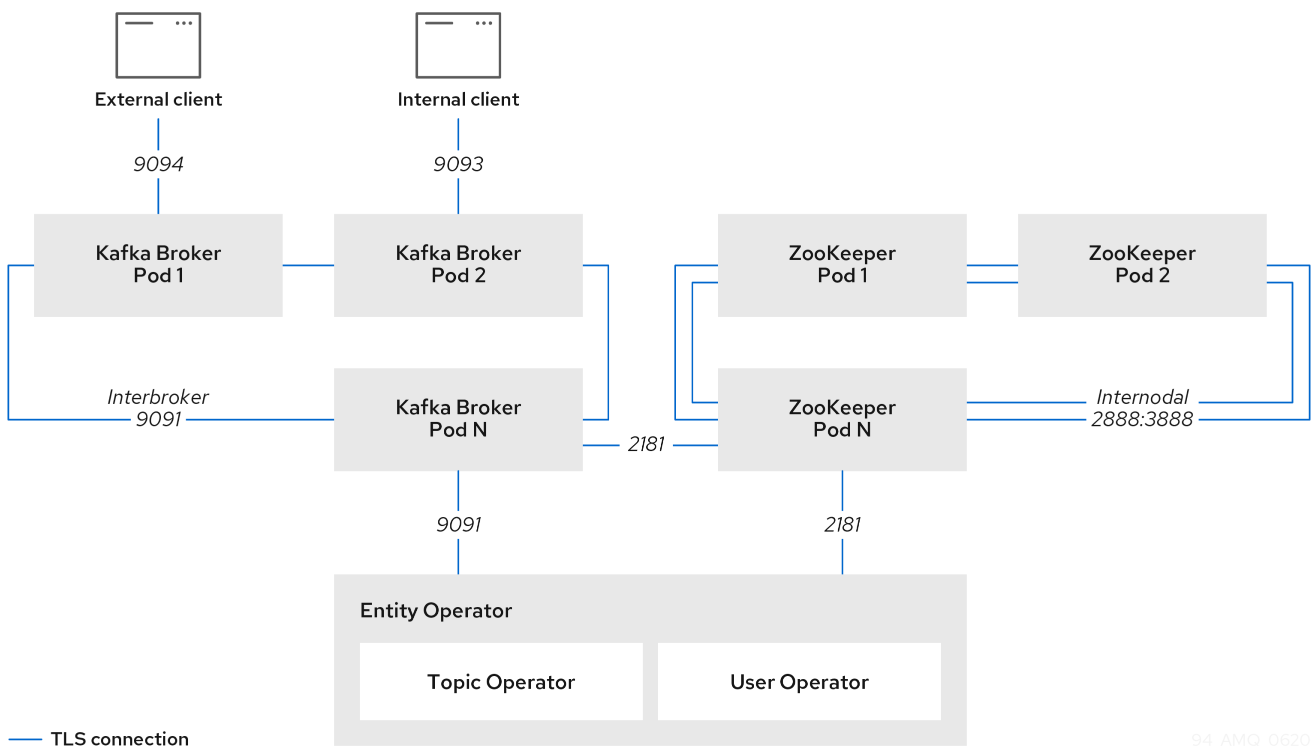
第12章 TLS 証明書の管理

AMQ Streams は、Kafka と AMQ Streams コンポーネントとの間で TLS プロトコルを使用して暗号化された通信をサポートします。Kafka ブローカー間の通信 (Interbroker 通信)、ZooKeeper ノード間の通信 (Internodal 通信)、およびこれらのコンポーネントと AMQ Streams Operator 間の通信は、常に暗号化されます。Kafka クライアントと Kafka ブローカーとの間の通信は、クラスターが設定された方法に応じて暗号化されます。Kafka および AMQ Streams コンポーネントでは、TLS 証明書も認証に使用されます。

Cluster Operator は、自動で TLS 証明書の設定および更新を行い、クラスター内での暗号化および認証を有効にします。また、Kafka ブローカーとクライアントとの間の暗号化または TLS 認証を有効にする場合、他の TLS 証明書も設定されます。ユーザーが用意した証明書は更新されません。

TLS 暗号化が有効になっている TLS リスナーまたは外部リスナーの、Kafka リスナー証明書 と呼ばれる独自のサーバー証明書を提供できます。詳細は、「Kafka リスナー証明書」 を参照してください。

図12.1 TLS によってセキュリティーが保護された通信のアークテクチャー例



12.1. 認証局

暗号化のサポートには、AMQ Streams コンポーネントごとに固有の秘密鍵と公開鍵証明書が必要です。すべてのコンポーネント証明書は、クラスター CA と呼ばれる内部認証局 (CA) により署名されます。

同様に、TLS クライアント認証を使用して AMQ Streams に接続する各 Kafka クライアントアプリケーションは、秘密鍵と証明書を提供する必要があります。クライアント CA という第 2 の内部 CA を使用して、Kafka クライアントの証明書に署名します。

12.1.1. CA 証明書

クラスター CA とクライアント CA の両方には、自己署名の公開鍵証明書があります。

Kafka ブローカーは、クラスター CA またはクライアント CA のいずれかが署名した証明書を信頼するように設定されます。クライアントによる接続が不要なコンポーネント (ZooKeeper など) のみが、クラスター CA によって署名された証明書を信頼します。外部リスナーの TLS 暗号化が無効でない限り、クライアントアプリケーションはクラスター CA により署名された証明書を必ず信頼する必要があります。これは、相互 TLS 認証 を実行するクライアントアプリケーションにも当てはまります。

デフォルトで、AMQ Streams はクラスター CA またはクライアント CA によって発行された CA 証明書を自動で生成および更新します。これらの CA 証明書の管理は、Kafka.spec.clusterCa および Kafka.spec.clientsCa オブジェクトで設定できます。ユーザーが用意した証明書は更新されません。

クラスター CA またはクライアント CA に、独自の CA 証明書を提供できます。詳細は、「独自の CA 証明書のインストール」 を参照してください。独自の証明書を提供する場合は、証明書の更新が必要なときに手作業で更新する必要があります。

12.1.2. 独自の CA 証明書のインストール

この手順では、Cluster Operator で生成される CA 証明書と鍵を使用する代わりに、独自の CA 証明書と秘密鍵をインストールする方法について説明します。

Cluster Operator は以下のシークレットを自動的に生成し、更新します。

CLUSTER-NAME-cluster-ca

クラスター CA の秘密鍵が含まれるクラスターシークレット。

CLUSTER-NAME-cluster-ca-cert

クラスター CA 証明書が含まれるクラスターシークレット。証明書には、Kafka ブローカーの ID を検証する公開鍵が含まれます。

CLUSTER-NAME-clients-ca

クライアント CA の秘密鍵が含まれるクライアントシークレット。

CLUSTER-NAME-clients-ca-cert

クライアント CA 証明書が含まれるクライアントシークレットです。証明書には、Kafka ブローカーにアクセスするクライアントの ID を検証する公開鍵が含まれます。

AMQ Streams はデフォルトでこれらのシークレットを使用します。

この手順では、独自のクラスターまたはクライアント CA 証明書を使用するシークレットを置き換える手順を説明します。

前提条件

- Cluster Operator が稼働している必要があります。
- Kafka クラスターがデプロイされていない必要があります。
- クラスター CA またはクライアントの、PEM 形式による独自の X.509 証明書および鍵が必要です。
  - ルート CA ではないクラスターまたはクライアント CA を使用する場合、証明書ファイルにチェーン全体を含める必要があります。チェーンの順序は以下のとおりです。
    1. クラスターまたはクライアント CA
    2. 1 つ以上の中間 CA
    3. ルート CA
  - チェーン内のすべての CA は、X509v3 基本制約拡張を使用して設定する必要があります。Basic Constraints は、証明書チェーンのパスの長さを制限します。

- 証明書を変換するための OpenSSL TLS 管理ツール。

作業を開始する前に

クラスターオペレーターは、CLUSTER-NAME-cluster-ca-cert シークレット用に以下のファイルを生成します。

- ca.crt クラスター証明書（PEM 形式）
- PKCS #12 形式の ca.p12 クラスター証明書
- PKCS #12 ファイルにアクセスするための ca.password

一部のアプリケーションは PEM 証明書を使用できず、PKCS #12 証明書のみに対応します。独自のクラスター証明書を PKCS #12 形式で追加することもできます。

PKCS #12 形式のクラスター証明書がない場合は、OpenSSL TLS 管理ツールを使用して ca.crt ファイルからこれを生成します。

証明書生成コマンドの例

```
openssl pkcs12 -export -in ca.crt --nokeys -out ca.p12 -password pass:P12-PASSWORD -
caname ca.crt
```

P12-PASSWORD は、自身のパスワードに置き換えます。

CLUSTER-NAME-clients-ca-certシークレットにも同様の操作を行うことができます。このシークレットには、デフォルトでPEMおよびPKCS #12形式の証明書も含まれています。

手順

1. Cluster Operator によって生成される CA 証明書を置き換えます。

   a. 既存のシークレットを削除します。

```
oc delete secret CA-CERTIFICATE-SECRET
```

CA-CERTIFICATE-SECRETは，Secret の名称です。

- CLUSTER-NAME-cluster-ca-cert: クラスタCA証明書
- CLUSTER-NAME-clients-ca-cert: クライアントのCA証明書

CLUSTER-NAME は、Kafka クラスターの名前に置き換えます。

「Not Exists」エラーを無視します。

   b. 新規シークレットを作成します。

PEM 形式の証明書を使用したクライアントシークレットの作成

```
oc create secret generic CLUSTER-NAME-clients-ca-cert --from-file=ca.crt=ca.crt
```

PEM および PKCS #12 形式の証明書を使用したクラスターシークレットの作成

```
oc create secret generic CLUSTER-NAME-cluster-ca-cert \
  --from-file=ca.crt=ca.crt \
  --from-file=ca.p12=ca.p12 \
  --from-literal=ca.password=P12-PASSWORD
```

2. Cluster Operator によって生成される秘密鍵を置き換えます。

   a. 既存のシークレットを削除します。

   ```
   oc delete secret CA-KEY-SECRET
   ```

   CA-KEY-SECRET は CA キーの名前です。

   - CLUSTER-NAME-cluster-ca（クラスタCAキー）
   - CLUSTER-NAME-clients-ca（クラスタCAキー）

   b. 新規シークレットを作成します。

   ```
   oc create secret generic CA-KEY-SECRET --from-file=ca.key=ca.key
   ```

3. シークレットにラベルを付けます。

```
oc label secret CA-CERTIFICATE-SECRET strimzi.io/kind=Kafka
strimzi.io/cluster=CLUSTER-NAME
```

```
oc label secret CA-KEY-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=CLUSTER-
NAME
```

- ラベル strimzi.io/kind=Kafka は Kafka カスタムリソースを識別します。
- ラベル strimzi.io/cluster=CLUSTER-NAME は Kafka クラスターを識別します。

4. クラスターの Kafka リソースを作成し、生成された CA を 使用しない ように Kafka.spec.clusterCa または Kafka.spec.clientsCa オブジェクトを設定します。

独自指定の証明書を使用するようにクラスター CA を設定する Kafka リソースの例 (抜粋)

```
kind: Kafka
version: kafka.strimzi.io/v1beta2
spec:
  # ...
  clusterCa:
    generateCertificateAuthority: false
```

関連情報

- 以前インストールした CA 証明書を更新する場合は「独自の CA 証明書の更新」を参照してください。
- 「独自の Kafka リスナー証明書の指定」

12.2. SECRET

AMQ Streams は Secret を使用して、Kafka クラスターコンポーネントおよびクライアントの秘密鍵および証明書を格納します。Secrets は、Kafka ブローカー間およびブローカーとクライアント間で TLS で暗号化された接続を確立するために使用されます。Secret は相互 TLS 認証にも使用されます。

- Cluster Secret には、Kafka ブローカー証明書に署名するためのクラスター CA 証明書が含まれます。また、接続クライアントによって、Kafka クラスターとの TLS 暗号化接続を確立してブローカー ID を検証するために使用されます。
- Client Secret にはクライアント CA 証明書が含まれ、これによりユーザーは独自のクライアント証明書に署名し、Kafka クラスターに対する相互認証が可能になります。ブローカーはクライアント CA 証明書を使用してクライアント ID を検証します。
- User Secret には、新規ユーザーの作成時にクライアント CA 証明書によって生成および署名される秘密鍵と証明書が含まれています。この鍵と証明書は、クラスターへのアクセス時の認証および承認に使用されます。

Secret には、PEM 形式および PKCS #12 形式の秘密鍵と証明書が含まれます。PEM 形式の秘密鍵

と証明書を使用する場合、ユーザーは Secret からそれらの秘密鍵と証明書を取得し、Java アプリケーションで使用するために対応するトラストストア (またはキーストア) を生成します。PKCS #12 ストレージは、直接使用できるトラストストア (またはキーストア) を提供します。

すべての鍵のサイズは 2048 ビットです。

12.2.1. PKCS #12 ストレージ

PKCS #12 は、暗号化オブジェクトをパスワードで保護された単一のファイルに格納するためのアーカイブファイル形式 (.p12) を定義します。PKCS #12 を使用して、証明書および鍵を一元的に管理できます。

各 Secret には、PKCS #12 特有のフィールドが含まれています。

- .p12 フィールドには、証明書と鍵が含まれます。
- .password フィールドは、アーカイブを保護するパスワードです。

12.2.2. クラスター CA Secret

以下の表は、Kafka クラスターの Cluster Operator によって管理される Cluster Secret を表しています。

クライアントは <cluster>-cluster-ca-cert Secret のみを使用する必要があります。他のすべての Secret は、AMQ Streamsコンポーネントがアクセスする必要があるだけです。これは、必要な場合に OpenShift のロールベースアクセス制御を使用して強制できます。

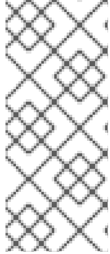
表12.1 <cluster>-cluster-ca Secret のフィールド

フィールド	説明
ca.key	クラスター CA の現在の秘密鍵。

表12.2 <cluster>-cluster-ca-cert Secret のフィールド

フィールド	説明
ca.p12	証明書および鍵を格納するための PKCS #12 アーカイブファイル。

フィールド	説明
ca.password	PKCS #12 アーカイブのファイルを保護するパスワード。
ca.crt	クラスター CA の現在の証明書。



注記

TLS を介した Kafka ブローカーへの接続時に Kafka ブローカー証明書を検証するため、<cluster>-cluster-ca-cert の CA 証明書は Kafka クライアントアプリケーションによって信頼される必要があります。

表12.3 <cluster>-kafka-brokers Secret のフィールド

フィールド	説明
<cluster>-kafka-<num>.p12	証明書および鍵を格納するための PKCS #12 アーカイブファイル。
<cluster>-kafka-<num>.password	PKCS #12 アーカイブのファイルを保護するパスワード。
<cluster>-kafka-<num>.crt	Kafka ブローカー Pod <num> の証明書。**<cluster>-cluster-ca** で現行または以前のクラスター CA の秘密鍵により署名されます。
<cluster>-kafka-<num>.key	Kafka ブローカー Pod **<num> の秘密鍵**。

表12.4 <cluster>-zookeeper-nodes Secret のフィールド

フィールド	説明
<cluster>-zookeeper-<num>.p12	証明書および鍵を格納するための PKCS #12 アーカイブファイル。
<cluster>-zookeeper-<num>.password	PKCS #12 アーカイブのファイルを保護するパスワード。
<cluster>-zookeeper-<num>.crt	ZooKeeper ノード <num> の証明書。**<cluster>-cluster-ca** で現行または以前のクラスター CA の秘密鍵により署名されます。
<cluster>-zookeeper-<num>.key	ZooKeeper Pod **<num> の秘密鍵**。

表12.5 <cluster>-entity-operator-certs Secret のフィールド

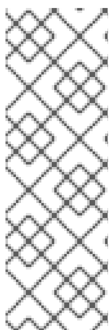
フィールド	説明
entity-operator_.p12	証明書および鍵を格納するための PKCS #12 アーカイブファイル。
entity-operator_.password	PKCS #12 アーカイブのファイルを保護するパスワード。
entity-operator_.crt	Entity Operator と Kafka または ZooKeeper との間の TLS 通信の証明書。**<cluster>-cluster-ca** で現行または以前のクラスター CA の秘密鍵により署名されます。
entity-operator.key	Entity Operator と、Kafka または ZooKeeper との間の TLS 通信の秘密鍵。

12.2.3. クライアント CA Secret

表12.6 <cluster> で Cluster Operator により管理されるクライアント CA Secrets

Secret 名	Secret 内のフィールド	説明
<cluster>-clients-ca	**ca.key**	クライアント CA の現在の秘密鍵。
<cluster>-clients-ca-cert	**ca.p12**	証明書および鍵を格納するための PKCS #12 アーカイブファイル。
	ca.password	PKCS #12 アーカイブのファイルを保護するパスワード。
	ca.crt	クライアント CA の現在の証明書。

<cluster>-clients-ca-cert の証明書は、Kafka ブローカーが信頼する証明書です。



注記

<cluster>-clients-ca は、クライアントアプリケーションの証明書への署名に使用されます。また AMQ Streams コンポーネントにアクセスできる必要があり、User Operator を使わずにアプリケーション証明書を発行する予定であれば管理者のアクセス権限が必要です。これは、必要な場合に OpenShift のロールベースのアクセス制御を使用して強制できます。

12.2.4. ラベルおよびアノテーションの Secret への追加

Kafkaカスタムリソースでclusterカ CaCertテンプレートプロパティを構成することで、クラスタオペレータが作成したクラスタCAシークレットにカスタムラベルやアノテーションを追加することができ

ます。ラベルとアノテーションは、オブジェクトを特定し、コンテキスト情報を追加するのに便利です。AMQ Streams カスタムリソースでテンプレートプロパティーを設定します。

ラベルおよびアノテーションを Secret に追加するテンプレートのカスタマイズ例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      clusterCaCert:
        metadata:
          labels:
            label1: value1
            label2: value2
          annotations:
            annotation1: value1
            annotation2: value2
    # ...
```

テンプレートプロパティーの設定に関する詳細は、「OpenShift リソースのカスタマイズ」 を参照してください。

12.2.5. CA Secret での ownerReference の無効化

デフォルトでは、クラスターおよびクライアント CA Secret は、Kafka カスタムリソースに設定される ownerReference プロパティーで作成されます。つまり、Kafka カスタムリソースが削除されると、OpenShift によって CA シークレットも削除（ガベッジコレクション）されます。

新しいクラスターでCAを再利用したい場合は、Kafkaの設定でClusterおよびClient CA SecretsのgenerateSecretOwnerReferenceプロパティをfalseに設定することで、ownerReferenceを無効にすることができます。ownerReference が無効になっていると、対応する Kafka カスタムリソースが削除されると、CA Secret は OpenShift によって削除されません。

クラスターおよびクライアント CA の ownerReference が無効になっている Kafka 設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    generateSecretOwnerReference: false
  clientsCa:
    generateSecretOwnerReference: false
# ...
```

その他のリソース

- CertificateAuthority schema reference

12.2.6. User Secret

表12.7 User Operator によって管理される Secrets

Secret 名	Secret 内のフィールド	説明
<user>	user.p12	証明書および鍵を格納するための PKCS #12 アーカイブファイル。
	user.password	PKCS #12 アーカイブのファイルを保護するパスワード。
	user.crt	ユーザーの証明書、クライアント CA により署名されます。
	user.key	ユーザーの秘密鍵。

12.3. 証明書の更新および有効期間

クラスター CA およびクライアント CA の証明書は、限定された期間、すなわち有効期間に限り有効です。通常、この期間は証明書の生成からの日数として定義されます。

Cluster Operator によって自動作成される CA 証明書の場合、以下の有効期間を設定できます。

- クラスター CA 証明書の場合は Kafka.spec.clusterCa.validityDays。

- クライアント CA 証明書の場合は Kafka.spec.clientsCa.validityDays。

デフォルトの有効期間は、両方の証明書で 365 日です。手動でインストールした CA 証明書には、独自の有効期間が定義されている必要があります。

CA 証明書の期限が切れると、その証明書を信頼しているコンポーネントおよびクライアントは、その CA 秘密鍵で署名された証明書を持つ相手からの TLS 接続を受け入れません。代わりに、コンポーネントおよびクライアントは 新しい CA 証明書を信頼する必要があります。

サービスを中断せずに CA 証明書を更新できるようにするため、Cluster Operator は古い CA 証明書が期限切れになる前に証明書の更新を開始します。

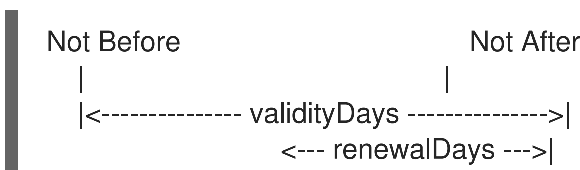
Cluster Operator によって作成される証明書の更新期間を設定できます。

- Kafka.spec.clusterCa.renewalDaysのクラスター CA 証明書
- Kafka.spec.clientsCa.renewalDaysのクライアント CA 証明書

デフォルトの更新期間は、両方の証明書とも 30 日です。

更新期間は、現在の証明書の有効期日から逆算されます。

更新期間に対する有効期間


```
Not Before                                     Not After
    |                                              |
    |<--------------- validityDays --------------->|
                              <--- renewalDays --->|
```

Kafkaクラスターの作成後に有効期間と更新期間の変更を行うには、Kafkaカスタムリソースの設定と適用、およびmanually renew the CA certificatesを行います。証明書を手動で更新しないと、証明

書が次回自動更新される際に新しい期間が使用されます。

証明書の有効および更新期間の Kafka 設定例

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
# ...
  clusterCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
  clientsCa:
    renewalDays: 30
    validityDays: 365
    generateCertificateAuthority: true
# ...
```

更新期間中のクラスターオペレーターの動作は、証明書生成のプロパティであるgenerateCertificateAuthorityおよびgenerateCertificateAuthorityの設定に依存します。

true

プロパティーが true に設定されている場合、CA 証明書は Cluster Operator によって自動的に生成され、更新期間内に自動的に更新されます。

false

プロパティーが false に設定されている場合、CA 証明書は Cluster Operator によって生成されません。独自の証明書をインストールする場合は、このオプションを使用します。

12.3.1. 自動生成された CA 証明書での更新プロセス

Cluster Operator は以下のプロセスを実行して CA 証明書を更新します。

1. 新しい CA 証明書を生成しますが、既存の鍵は保持します。該当する Secret 内の ca.crt という名前の古い証明書が新しい証明書に置き換えられます。

2.

新しいクライアント証明書を生成します (ZooKeeper ノード、Kafka ブローカー、および Entity Operator 用)。署名鍵は変わっておらず、CA 証明書と同期してクライアント証明書の有効期間を維持するため、これは必須ではありません。

3. ZooKeeper ノードを再起動して、ZooKeeper ノードが新しい CA 証明書を信頼し、新しいクライアント証明書を使用するようにします。

4. Kafka ブローカーを再起動して、Kafka ブローカーが新しい CA 証明書を信頼し、新しいクライアント証明書を使用するようにします。

5. Topic Operator および User Operator を再起動して、それらの Operator が新しい CA 証明書を信頼し、新しいクライアント証明書を使用するようにします。

12.3.2. クライアント証明書の更新

Cluster Operator は、Kafka クラスターを使用するクライアントアプリケーションを認識しません。

クラスターに接続し、クライアントアプリケーションが正しく機能するように確認するには、クライアントアプリケーションは以下を行う必要があります。

- <cluster>-cluster-ca-cert Secret でパブリッシュされるクラスター CA 証明書を信頼する必要があります。

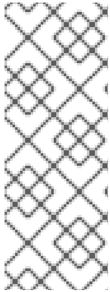
- <user-name> Secret でパブリッシュされたクレデンシャルを使用してクラスターに接続します。

  User Secret は PEM および PKCS #12 形式のクレデンシャルを提供し、SCRAM-SHA 認証を使用する場合はパスワードを提供できます。ユーザーの作成時に User Operator によってユーザークレデンシャルが生成されます。

証明書の更新後もクライアントが動作するようにする必要があります。更新プロセスは、クライアントの設定によって異なります。

クライアント証明書と鍵のプロビジョニングを手動で行う場合、新しいクライアント証明書を生成し、更新期間内に新しい証明書がクライアントによって使用されるようにする必要があります。更新期

間の終了までにこれが行われないと、クライアントアプリケーションがクラスターに接続できなくなる可能性があります。



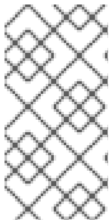
注記

同じ OpenShift クラスターおよび namespace 内で実行中のワークロードの場合、Secrets はボリュームとしてマウントできるので、クライアント Pod はそれらのキーストアとトラストストアを現在の状態の Secrets から構築できます。この手順の詳細は、「クラスター CA を信頼する内部クライアントの設定」を参照してください。

12.3.3. Cluster Operator によって生成される CA 証明書の手動更新

Cluster Operator によって生成されるクラスターおよびクライアント CA 証明書は、各証明書の更新期間の開始時に自動更新されます。ただし、strimzi.io/force-renew アノテーションを使用して、証明書の更新期間が始まる前に、これらの証明書の一方または両方を手動で更新することができます。セキュリティー上の理由や、証明書の更新または有効期間を変更した 場合などに、自動更新を行うことがあります。

更新された証明書は、更新前の証明書と同じ秘密鍵を使用します。



注記

独自の CA 証明書を使用している場合は、force-renew アノテーションは使用できません。代わりに、独自の CA 証明書を更新する手順に従ってください。

前提条件

- Cluster Operator が稼働している必要があります。
- CA 証明書と秘密鍵がインストールされている Kafka クラスターが必要です。

手順

1. strimzi.io/force-renew アノテーションを、更新対象の CA 証明書が含まれる Secret に適用します。

表12.8 証明書の更新を強制する Secret のアノテーション。

証明書	Secret	annotate コマンド
クラスター CA	KAFKA-CLUSTER-NAME-cluster-ca-cert	**oc annotate secret KAFKA-CLUSTER-NAME-cluster-ca-cert strimzi.io/force-renew=true**
クライアント CA	KAFKA-CLUSTER-NAME-clients-ca-cert	**oc annotate secret KAFKA-CLUSTER-NAME-clients-ca-cert strimzi.io/force-renew=true**

次回の調整で、アノテーションを付けた Secret の新規 CA 証明書が Cluster Operator によって生成されます。メンテナンス時間枠が設定されている場合、Cluster Operator によって、最初の調整時に次のメンテナンス時間枠内で新規 CA 証明書が生成されます。

Cluster Operator によって更新されたクラスターおよびクライアント CA 証明書をクライアントアプリケーションがリロードする必要があります。

2. CA 証明書が有効である期間を確認します。

   たとえば、openssl コマンドを使用します。

```
oc get secret CA-CERTIFICATE-SECRET -o 'jsonpath={.data.CA-CERTIFICATE}' |
base64 -d | openssl x509 -subject -issuer -startdate -enddate -noout
```

   CA-CERTIFICATE-SECRETはSecretの名前で、クラスタCA証明書の場合はKAFKA-CLUSTER-NAME-cluster-ca-certであり、クライアントCA証明書の場合はKAFKA-CLUSTER-NAME-clients-ca-certとなります。

   CA-CERTIFICATEは、jsonpath={.data.ca\.crt}のように、CA証明書の名前です。

   このコマンドは、CA 証明書の有効期間である notBefore および notAfter の日付を返します。

   たとえば、クラスター CA 証明書の場合は以下のようになります。

```
subject=O = io.strimzi, CN = cluster-ca v0
issuer=O = io.strimzi, CN = cluster-ca v0
notBefore=Jun 30 09:43:54 2020 GMT
notAfter=Jun 30 09:43:54 2021 GMT
```

3. Secret から古い証明書を削除します。

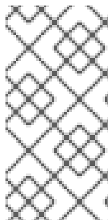
   コンポーネントで新しい証明書が使用される場合でも、古い証明書がアクティブであることがあります。古い証明書を削除して、潜在的なセキュリティーリスクを取り除きます。

関連情報

- 「Secret」
- 「ローリングアップデートのメンテナンス時間枠」
- 「CertificateAuthority スキーマ参照」

12.3.4. Cluster Operator によって生成された CA 証明書によって使用される秘密鍵の置き換え

Cluster Operator によって生成されるクラスター CA およびクライアント CA 証明書によって使用される秘密鍵を置換できます。秘密鍵が交換されると、Cluster Operator によって新しい秘密鍵の新しい CA 証明書が生成されます。



注記

独自の CA 証明書を使用している場合は、force-replace アノテーションは使用できません。代わりに、独自の CA 証明書を更新する手順に従ってください。

前提条件

- Cluster Operator が稼働している必要があります。
- CA 証明書と秘密鍵がインストールされている Kafka クラスターが必要です。

手順

- 更新対象の秘密鍵が含まれる Secret に strimzi.io/force-replace アノテーションを適用します。

表12.9 秘密鍵を置き換えるコマンド

秘密鍵	Secret	annotate コマンド
クラスター CA	CLUSTER-NAME-cluster-ca	**oc annotate secret CLUSTER-NAME-cluster-ca strimzi.io/force-replace=true**
クライアント CA	CLUSTER-NAME-clients-ca	**oc annotate secret CLUSTER-NAME-clients-ca strimzi.io/force-replace=true**

次回の調整時に、Cluster Operator は以下を生成します。

- アノテーションを付けた Secret の新しい秘密鍵
- 新規 CA 証明書

メンテナンス時間枠が設定されている場合、Cluster Operator によって、最初の調整時に次のメンテナンス時間枠内で新しい秘密鍵と CA 証明書が生成されます。

Cluster Operator によって更新されたクラスターおよびクライアント CA 証明書をクライアントアプリケーションがリロードする必要があります。

関連情報

- 「Secret」
- 「ローリングアップデートのメンテナンス時間枠」

12.3.5. 独自の CA 証明書の更新

この手順では、Cluster Operator によって生成される証明書を使用せずに、独自にインストールした CA 証明書および鍵を更新する方法を説明します。

独自の証明書を使用している場合、Cluster Operator は自動的に更新されません。したがって、期限切れ間近の CA 証明書を交換するために、証明書の更新期間中にこの手順を実行することが重要になります。

この手順では、PEM 形式の CA 証明書の更新を説明します。

前提条件

- Cluster Operator が稼働している必要があります。
- 独自の CA 証明書と秘密鍵がインストールされている必要があります。
- クラスターおよびクライアントの PEM 形式による新しい X.509 証明書と鍵が必要です。

これらは、openssl コマンドを使用して生成できます。以下に例を示します。

```
openssl req -x509 -new -days NUMBER-OF-DAYS-VALID --nodes -out ca.crt -keyout ca.key
```

手順

1. Secret で現在の CA 証明書の詳細を確認します。

   ```
   oc describe secret CA-CERTIFICATE-SECRET
   ```

   CA-CERTIFICATE-SECRETはSecretの名前で、クラスタCA証明書の場合はKAFKA-CLUSTER-NAME-cluster-ca-certであり、クライアントCA証明書の場合はKAFKA-CLUSTER-NAME-clients-ca-certとなります。

2. シークレットに既存の CA 証明書が含まれるディレクトリーを作成します。

   ```
   mkdir new-ca-cert-secret
   cd new-ca-cert-secret
   ```

3. 更新する各 CA 証明書のシークレットを取得します。

```
oc get secret CA-CERTIFICATE-SECRET -o 'jsonpath={.data.CA-CERTIFICATE}' |
base64 -d > CA-CERTIFICATE
```

CA-CERTIFICATE を各 CA 証明書の名前に置き換えます。

4. 古いca.crtファイルの名前をca-DATE.crt に変更します。ここで DATEは、YEAR-MONTH-DAYTHOUR-MINUTE-SECONDZ形式の証明書の有効期限です。

例: ca-2018-09-27T17-32-00Z.crt

```
mv ca.crt ca-$(date -u -d$(openssl x509 -enddate -noout -in ca.crt | sed 's/.*=//') +'%Y-
%m-%dT%H-%M-%SZ').crt
```

5. 新規 CA 証明書をディレクトリーにコピーし、ca.crt という名前を付けます。

```
cp PATH-TO-NEW-CERTIFICATE ca.crt
```

6. CA 証明書を対応する Secret に配置します。

a. 既存のシークレットを削除します。

```
oc delete secret CA-CERTIFICATE-SECRET
```

CA-CERTIFICATE-SECRET は最初のステップで返される Secret の名前です。

「Not Exists」エラーを無視します。

b. シークレットを再作成します。

```
oc create secret generic CA-CERTIFICATE-SECRET --from-file=.
```

–

7. 作成したディレクトリーを削除します。

```
cd ..
rm -r new-ca-cert-secret
```

8. CA キーを対応する Secret に配置します。

   a. 既存のシークレットを削除します。

   ```
   oc delete secret CA-KEY-SECRET
   ```

   CA-KEY-SECRET は CA キーの名前です。これは、クラスター CA キーの場合は KAFKA-CLUSTER-NAME-cluster-ca、クライアント CA キーの場合は KAFKA-CLUSTER-NAME-clients-ca です。

   b. 新しい CA 鍵でシークレットを再作成します。

   ```
   oc create secret generic CA-KEY-SECRET --from-file=ca.key=CA-KEY-SECRET-FILENAME
   ```

9. シークレットに strimzi.io/kind=Kafka および strimzi.io/cluster=KAFKA-CLUSTER-NAME ラベルを付けます。

```
oc label secret CA-CERTIFICATE-SECRET strimzi.io/kind=Kafka
strimzi.io/cluster=KAFKA-CLUSTER-NAME
oc label secret CA-KEY-SECRET strimzi.io/kind=Kafka strimzi.io/cluster=KAFKA-CLUSTER-NAME
```

12.4. TLS 接続

12.4.1. ZooKeeper の通信

すべてのポート上の ZooKeeper ノード間の通信と、クライアントと ZooKeeper 間の通信は TLS を使用して暗号化されます。

Kafka ブローカーと ZooKeeper ノード間の通信も暗号化されます。

12.4.2. Kafka のブローカー間の通信

Kafka ブローカー間の通信は常に TLS を使用して暗号化されます。

ControlPlaneListener feature gate が有効になっていない限り、ブローカー間の通信はすべてポート 9091 の内部リスナーを通過します。フィーチャーゲートを有効にすると、コントロールプレーンからのトラフィックはポート 9090 の内部コントロールプレーンリスナーを経由します。データプレーンからのトラフィックは引き続き、ポート 9091 で既存の内部リスナーを使用します。

これらの内部リスナーは Kafka クライアントでは利用できません。

12.4.3. Topic Operator および User Operator

すべての Operator は、Kafka と ZooKeeper 両方との通信に暗号化を使用します。Topic Operator および User Operator では、ZooKeeper との通信時に TLS サイドカーが使用されます。

12.4.4. Cruise Control

Cruise Control は、Kafka と ZooKeeper 両方との通信に暗号化を使用します。TLS サイドカーは、ZooKeeper との通信時に使用されます。

12.4.5. Kafka クライアント接続

Kafka ブローカーとクライアント間の暗号化または暗号化されていない通信は、spec.kafka.listeners の tls プロパティーを使用して設定されます。

12.5. クラスター CA を信頼する内部クライアントの設定

この手順では、TLS リスナーに接続する OpenShift クラスター内部に存在する Kafka クライアントがクラスター CA 証明書を信頼するように設定する方法を説明します。

これを内部クライアントで実現するには、ボリュームマウントを使用して、必要な証明書および鍵が含まれる Secrets にアクセスするのが最も簡単な方法です。

以下の手順に従い、クラスター CA によって署名された信頼できる証明書を Java ベースの Kafka Producer、Consumer、および Streams API に設定します。

クラスター CA の証明書の形式が PKCS #12 (.p12) または PEM (.crt) であるかに応じて、手順を選択します。

この手順では、Kafka クラスターの ID を検証する Cluster Secret をクライアント Pod にマウントする方法を説明します。

前提条件

- Cluster Operator が稼働している必要があります。
- OpenShift クラスター内に Kafka リソースが必要です。
- TLS を使用して接続し、クラスター CA 証明書を必ず信頼する Kafka クライアントアプリケーションが、OpenShift クラスター内部に必要です。
- クライアントアプリケーションが Kafka リソースと同じ namespace で実行している必要があります。

PKCS #12 形式 (.p12) の使用

1. クライアント Pod の定義時に、Cluster Secret をボリュームとしてマウントします。

   以下に例を示します。

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
  - name: client-name
    image: client-name
    volumeMounts:
    - name: secret-volume
      mountPath: /data/p12
    env:
    - name: SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: my-password
```

```
volumes:
- name: secret-volume
  secret:
    secretName: my-cluster-cluster-ca-cert
```

ここでは、以下をマウントしています。

- PKCS #12 ファイルを設定可能な正確なパスにマウント。
- パスワードを Java 設定に使用できる環境変数にマウント。

2. Kafka クライアントを以下のプロパティーで設定します。
   - セキュリティープロトコルのオプション:
     - security.protocol: SSL (TLS 認証ありまたはなしで、暗号化に TLS を使用する場合)。
     - security.protocol: SASL_SSL (TLS 経由で SCRAM-SHA 認証を使用する場合)。
   - ssl.truststore.location (証明書がインポートされたトラストストアを指定)。
   - ssl.truststore.password (トラストストアにアクセスするためのパスワードを指定)。
   - ssl.truststore.type=PKCS12 (トラストストアのタイプを識別)。

PEM 形式の使用 (.crt)

1. クライアント Pod の定義時に、Cluster Secret をボリュームとしてマウントします。

   以下は例になります。

```
kind: Pod
apiVersion: v1
metadata:
  name: client-pod
spec:
  containers:
  - name: client-name
    image: client-name
    volumeMounts:
    - name: secret-volume
      mountPath: /data/crt
  volumes:
  - name: secret-volume
    secret:
      secretName: my-cluster-cluster-ca-cert
```

2. X.509 形式の証明書を使用するクライアントでこの証明書を使用します。

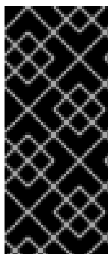
12.6. クラスター CA を信頼する外部クライアントの設定

この手順では、external に接続する OpenShift クラスター外部に存在する Kafka クライアントを設定し、クラスター CA 証明書を信頼する方法を説明します。クライアントのセットアップ時および更新期間中に、古いクライアント CA 証明書を交換する場合は、以下の手順に従います。

以下の手順に従い、クラスター CA によって署名された信頼できる証明書を Java ベースの Kafka Producer、Consumer、および Streams API に設定します。

クラスター CA の証明書の形式が PKCS #12 (.p12) または PEM (.crt) であるかに応じて、手順を選択します。

この手順では、Kafka クラスターの ID を検証する Cluster Secret から証明書を取得する方法を説明します。



重要

CA 証明書の更新期間中に、<cluster-name>-cluster-ca-cert Secret に複数の CA 証明書が含まれます。クライアントは、それらを すべて をクライアントのトラストストアに追加する必要があります。

前提条件

- Cluster Operator が稼働している必要があります。

- OpenShift クラスター内に Kafka リソースが必要です。

- TLS を使用して接続し、クラスター CA 証明書を必ず信頼する Kafka クライアントアプリケーションが、OpenShift クラスター外部に必要です。

PKCS #12 形式 (.p12) の使用

1. Kafka クラスタのCLUSTER-NAME-cluster-ca-cert Secret からクラスタ CA 証明書とパスワードを抽出します。

```
oc get secret CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.p12}' | base64 -d
> ca.p12
```

```
oc get secret CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.password}' |
base64 -d > ca.password
```

   CLUSTER-NAME は、Kafka クラスターの名前に置き換えます。

2. Kafka クライアントを以下のプロパティーで設定します。

   - セキュリティープロトコルのオプション:

     - security.protocol: SSL (TLS 認証ありまたはなしで、暗号化に TLS を使用する場合)。

     - security.protocol: SASL_SSL (TLS 経由で SCRAM-SHA 認証を使用する場合)。

   - ssl.truststore.location (証明書がインポートされたトラストストアを指定)。

   - ssl.truststore.password (トラストストアにアクセスするためのパスワードを指定)。このプロパティーは、トラストストアで必要なければ省略できます。

- ssl.truststore.type=PKCS12 (トラストストアのタイプを識別)。

PEM 形式の使用 (.crt)

1. Kafka クラスターの CLUSTER-NAME-cluster-ca-cert Secret からクラスター CA 証明書を抽出します。

```
oc get secret CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64 -d >
ca.crt
```

2. X.509 形式の証明書を使用するクライアントでこの証明書を使用します。

12.7. KAFKA リスナー証明書

以下のタイプのリスナーに、独自のサーバー証明書と秘密鍵を指定できます。

- OpenShift クラスター内で通信するための内部 TLS リスナー
- KafkaクライアントとKafkaブローカー間の通信用にTLS暗号化を有効にした外部リスナーroute型、loadbalancer型、ingress型、nodeport型）。

これらのユーザー提供による証明書は、Kafka リスナー証明書 と呼ばれます。

外部リスナーに Kafka リスナー証明書を提供すると、既存のセキュリティーインフラストラクチャー (所属組織のプライベート CA やパブリック CA など) を利用できます。Kafka クライアントは Kafka ブローカーに接続する際に、クラスター CA またはクライアント CA によって署名された証明書ではなく、Kafka リスナー証明書を使用します。

Kafka リスナー証明書の更新が必要な場合は、手作業で更新する必要があります。

12.7.1. 独自の Kafka リスナー証明書の指定

この手順では、独自の秘密鍵と Kafka リスナー証明書と呼ばれるサーバー証明書を使用するようにリスナーを設定する方法について説明します。

Kafka ブローカーの ID を検証するため、クライアントアプリケーションは CA 公開鍵を信頼できる証明書として使用する必要があります。

前提条件

- OpenShift クラスターが必要です。
- Cluster Operator が稼働している必要があります。
- リスナーごとに、外部 CA によって署名された互換性のあるサーバー証明書が必要です。
  - X.509 証明書を PEM 形式で提供します。
  - リスナーごとに正しい SAN (サブジェクト代替名) を指定します。詳細は、「Kafka リスナーのサーバー証明書の SAN」 を参照してください。
  - 証明書ファイルに CA チェーン全体が含まれる証明書を提供できます。

手順

1. 秘密鍵およびサーバー証明書が含まれる Secret を作成します。

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

2. クラスターの Kafka リソースを編集します。Secret、証明書ファイル、および秘密鍵ファイルを使用するように、リスナーを configuration.brokerCertChainAndKey プロパティーで設定します。

TLS 暗号化が有効な loadbalancer 外部リスナーの設定例

```
# ...
listeners:
  - name: plain
    port: 9092
```

```
    type: internal
    tls: false
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
# ...
```

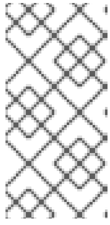
TLS リスナーの設定例

```
# ...
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication:
      type: tls
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
# ...
```

3. 新しい設定を適用してリソースを作成または更新します。

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

Cluster Operator は、Kafka クラスターのローリングアップデートを開始し、これによりリスナーの設定が更新されます。



注記

TLS または外部リスナーによってすでに使用されている Secret の Kafka リスナー証明書を更新した場合でも、ローリングアップデートが開始されます。

関連情報

- Kafka リスナーのサーバー証明書の SAN
- GenericKafkaListener schema reference
- Kafka リスナー証明書

12.7.2. Kafka リスナーのサーバー証明書の SAN

独自の Kafka リスナー証明書で TLS ホスト名検証を使用するには、リスナーごとに SAN (サブジェクト代替名) を使用する必要があります。証明書の SAN は、以下のホスト名を指定する必要があります。

- クラスターのすべての Kafka ブローカー
- Kafka クラスターブートストラップサービス

ワイルドカード証明書は、CA でサポートされれば使用できます。

12.7.2.1. TLS リスナー SAN の例

以下の例を利用して、TLS リスナーの証明書で SAN のホスト名を指定できます。

ワイルドカードの例

```
//Kafka brokers
```

```
*.<cluster-name>-kafka-brokers
*.<cluster-name>-kafka-brokers.<namespace>.svc

// Bootstrap service
<cluster-name>-kafka-bootstrap
<cluster-name>-kafka-bootstrap.<namespace>.svc
```

ワイルドカードのない例

```
// Kafka brokers
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers
<cluster-name>-kafka-0.<cluster-name>-kafka-brokers.<namespace>.svc
<cluster-name>-kafka-1.<cluster-name>-kafka-brokers
<cluster-name>-kafka-1.<cluster-name>-kafka-brokers.<namespace>.svc
# ...

// Bootstrap service
<cluster-name>-kafka-bootstrap
<cluster-name>-kafka-bootstrap.<namespace>.svc
```

12.7.2.2. 外部リスナー SAN の例

TLS 暗号化が有効になっている外部リスナーの場合、証明書に指定する必要があるホスト名は、外部リスナーの **type** によって異なります。

表12.10 外部リスナー各タイプの SAN

外部リスナータイプ	SAN で指定する内容
Route	すべての Kafka ブローカー **Routes** のアドレス、およびブートストラップ **Route** のアドレス。 一致するワイルドカード名を使用できます。
loadbalancer	すべての Kafka ブローカー **loadbalancers** のアドレス、およびブートストラップ **loadbalancer** のアドレス。 一致するワイルドカード名を使用できます。

外部リスナータイプ	SAN で指定する内容
NodePort	Kafka ブローカー Pod がスケジュールされるすべての OpenShift ワーカーノードのアドレス。 一致するワイルドカード名を使用できます。

関連情報

- 「独自の Kafka リスナー証明書の指定」

第13章 AMQ STREAMS の管理

本章では、AMQ Streams のデプロイメントを維持するタスクについて説明します。

13.1. カスタムリソースの使用

oc コマンドを使用して、AMQ Streams カスタムリソースで情報を取得し、他の操作を実行できます。

カスタムリソースの status サブリソースで oc を使用すると、リソースに関する情報を取得できます。

13.1.1. カスタムリソースでの oc 操作の実施

リソースタイプに対して操作を行うには、get、describe、edit、deleteなどのocコマンドを使用します。たとえば、oc get kafkatopics はすべての Kafka トピックのリストを取得し、oc get kafkas はデプロイされたすべての Kafka クラスターを取得します。

リソースタイプを参照する際には、単数形と複数形の両方の名前を使うことができます。oc get kafkasはoc get kafka と同じ結果になります。

リソースの 短縮名 を使用することもできます。短縮名を理解すると、AMQ Streams を管理する時間を節約できます。Kafkaのショートネームはkなので、oc get kを実行してすべてのKafkaクラスターをリストアップすることもできます。

```
oc get k

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

表13.1 各 AMQ Streams リソースの正式名および短縮名

AMQ Streams リソース	正式名	短縮名
Kafka	kafka	k
Kafka Topic	kafkatopic	kt
Kafka User	kafkauser	ku

AMQ Streams リソース	正式名	短縮名
Kafka Connect	kafkaconnect	kc
Kafka Connector	kafkaconnector	kctr
Kafka Mirror Maker	kafkamirrormaker	kmm
Kafka Mirror Maker 2	kafkamirrormaker2	kmm2
Kafka Bridge	kafkabridge	kb
Kafka Rebalance	kafkarebalance	kr

13.1.1.1. リソースカテゴリー

カスタムリソースのカテゴリーは、ocコマンドでも使用できます。

すべての AMQ Streams カスタムリソースはカテゴリー strimzi に属するため、strimziを使用してすべての AMQ Streams リソースを 1 つのコマンドで取得できます。

例えば、oc get strimziを実行すると、指定された名前空間のすべてのAMQ Streamsカスタムリソースが一覧表示されます。

```
oc get strimzi

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

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

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

oc get strimzi -o nameコマンドは、すべてのリソースタイプとリソース名を返します。-o name オプションは type/name 形式で出力を取得します。

```
oc get strimzi -o name

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

この strimzi コマンドを他のコマンドと組み合わせることができます。たとえば、これを oc delete コマンドに渡して、単一のコマンドですべてのリソースを削除できます。

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

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

1 つの操作ですべてのリソースを削除することは、AMQ Streams の新機能をテストする場合などに役立ちます。

13.1.1.2. サブリソースのステータスのクエリー

-o オプションに渡すことのできる他の値もあります。たとえば、-o yaml を使用すると、YAML 形式で出力されます。usng -o json は JSON として返します。

oc get --help のすべてのオプションが表示されます。

最も便利なオプションの 1 つは JSONPath サポート で、JSONPath 式を渡して Kubernetes API にクエリーを実行できます。JSONPath 式は、リソースの特定部分を抽出または操作できます。

たとえば、JSONPath 式 {.status.listeners[?(@.type=="tls")].bootstrapServers} を使用して、Kafka カスタムリソースのステータスからブートストラップアドレスを取得し、Kafka クライアントで使用できます。

ここで、コマンドは tls リスナーの bootstrapServers 値を見つけます。

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?(@.type=="tls")].bootstrapServers}
{"\n"}'

my-cluster-kafka-bootstrap.myproject.svc:9093
```

タイプ条件を @.type=="external" または @.type=="plain" に変更すると、他の Kafka リスナーのアドレスを取得することもできます。

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.type=="external")].bootstrapServers}{"\n"}'
```

192.168.1.247:9094

jsonpath を使用して、カスタムリソースから他のプロパティーまたはプロパティーのグループを抽出できます。

13.1.2. AMQ Streams カスタムリソースのステータス情報

下記の表のとおり、複数のリソースに status プロパティーがあります。

表13.2 カスタムリソースの status プロパティー

AMQ Streams リソース	スキーマ参照	ステータス情報がパブリッシュされる場所
Kafka	「**KafkaStatus** スキーマ参照」	Kafka クラスター。
KafkaConnect	「**KafkaConnectStatus** スキーマ参照」	デプロイされている場合は Kafka Connect クラスター。
KafkaConnector	「**KafkaConnectorStatus** スキーマ参照」	デプロイされている場合は KafkaConnector リソース。
KafkaMirrorMaker	「**KafkaMirrorMakerStatus** スキーマ参照」	デプロイされている場合は Kafka MirrorMakerツール。
KafkaTopic	「**KafkaTopicStatus** スキーマ参照」	Kafka クラスターの Kafka トピック
KafkaUser	「**KafkaUserStatus** スキーマ参照」	Kafka クラスターの Kafka ユーザー。
KafkaBridge	「**KafkaBridgeStatus** スキーマ参照」	デプロイされている場合は AMQ Streams の Kafka Bridge。

リソースの status プロパティーによって、リソースの下記項目の情報が提供されます。

- status.conditions プロパティーの Current state (現在の状態)。
- status.observedGeneration プロパティーの Last observed generation (最後に確認された生成)。

status プロパティーによって、リソース固有の情報も提供されます。以下は例になります。

- KafkaStatus によって、リスナーアドレスに関する情報と Kafka クラスターの ID が提供されます。
- KafkaConnectStatus によって、Kafka Connect コネクターの REST API エンドポイントが提供されます。
- KafkaUserStatus によって、Kafka ユーザーの名前と、ユーザーのクレデンシャルが保存される Secret が提供されます。
- KafkaBridgeStatus によって、外部クライアントアプリケーションが Bridge サービスにアクセスできる HTTP アドレスが提供されます。

リソースの Current state (現在の状態) は、spec プロパティーによって定義される Desired state (望ましい状態) を実現するリソースに関する進捗を追跡するのに便利です。ステータス条件によって、リソースの状態が変更された時間および理由が提供され、Operator によるリソースの望ましい状態の実現を妨げたり遅らせたりしたイベントの詳細が提供されます。

Last observed generation (最後に確認された生成) は、Cluster Operator によって最後に照合されたリソースの生成です。observedGeneration の値が metadata.generation の値と異なる場合、リソースの最新の更新が Operator によって処理されていません。これらの値が同じである場合、リソースの最新の変更がステータス情報に反映されます。

AMQ Streams によってカスタムリソースのステータスが作成および維持されます。定期的にカスタムリソースの現在の状態が評価され、その結果に応じてステータスが更新されます。くださいーたとえば、oc edit を使用してカスタムリソースで更新を行う場合、その status は編集不可能です。さらに、status の変更は Kafka クラスターステータスの設定に影響しません。

以下では、Kafka カスタムリソースに status プロパティーが指定されています。

Kafka カスタムリソースとステータス

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
```

```
spec:
  # ...
status:
  conditions: 1
  - lastTransitionTime: 2021-07-23T23:46:57+0000
    status: "True"
    type: Ready 2
  observedGeneration: 4 3
  listeners: 4
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9092
    type: plain
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9093
    certificates:
    - |
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    type: tls
  - addresses:
    - host: 172.29.49.180
      port: 9094
    certificates:
    - |
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    type: external
  clusterId: CLUSTER-ID 5
# ...
```

1

status の conditions は、既存のリソース情報から推測できないステータスに関連する基準や、リソースのインスタンスに固有する基準を記述します。

2

Ready 条件は、Cluster Operator が現在 Kafka クラスターでトラフィックの処理が可能であると判断するかどうかを示しています。

3

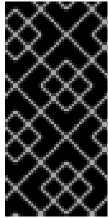
observedGeneration は、最後に Cluster Operator によって照合された Kafka カスタムリソースの生成を示しています。

4

listeners は、現在の Kafka ブートストラップアドレスをタイプ別に示しています。

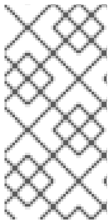
5

Kafka クラスター ID。



重要

タイプが nodeport の外部リスナーのカスタムリソースステータスにおけるアドレスは、現在サポートされていません。



注記

Kafka ブートストラップアドレスがステータスに一覧表示されても、それらのエンドポイントまたは Kafka クラスターが準備状態であるとは限りません。

ステータス情報のアクセス

リソースのステータス情報はコマンドラインから取得できます。詳細は、「カスタムリソースのステータスの検出」 を参照してください。

13.1.3. カスタムリソースのステータスの検出

この手順では、カスタムリソースのステータスを検出する方法を説明します。

前提条件

- OpenShift クラスターが必要です。
- Cluster Operator が稼働している必要があります。

手順

- カスタムリソースを指定し、-o jsonpath オプションを使用して標準の JSONPath 式を適用して status プロパティーを選択します。

```
oc get kafka <kafka_resource_name> -o jsonpath='{.status}'
```

この式は、指定されたカスタムリソースのすべてのステータス情報を返します。status.listeners または status.observedGeneration などのドット表記を使用すると、表示するステータス情報を微調整できます。

関連情報

- 「AMQ Streams カスタムリソースのステータス情報」
- JSONPath の使用に関する詳細は、「JSONPath support」を参照してください。

13.2. カスタムリソースの調整の一時停止

修正や更新を実行するために、AMQ Streams Operator によって管理されるカスタムリソースの調整を一時停止すると便利な場合があります。調整が一時停止されると、カスタムリソースに加えられた変更は一時停止が終了するまで Operator によって無視されます。

カスタムリソースの調整を一時停止するには、configure で strimzi.io/pause-reconciliation アノテーションを true に設定します。これにより、適切な Operator がカスタムリソースの調整を一時停止するよう指示されます。たとえば、Cluster Operator による調整が一時停止されるように、アノテーションを KafkaConnect リソースに適用できます。

pause アノテーションを有効にしてカスタムリソースを作成することもできます。カスタムリソースは作成されますが、無視されます。

前提条件

- カスタムリソースを管理する AMQ Streams Operator が稼働している必要があります。

手順

1. pause-reconciliation を true に設定して、OpenShift のカスタムリソースにアノテーションを付けます。

```
oc annotate KIND-OF-CUSTOM-RESOURCE NAME-OF-CUSTOM-RESOURCE
strimzi.io/pause-reconciliation="true"
```

たとえば、KafkaConnect カスタムリソースの場合は以下のようになります。

```
oc annotate KafkaConnect my-connect strimzi.io/pause-reconciliation="true"
```

2. カスタムリソースの status 条件で、ReconciliationPaused への変更が表示されることを確認し ます。

```
oc describe KIND-OF-CUSTOM-RESOURCE NAME-OF-CUSTOM-RESOURCE
```

type 条件は、lastTransitionTime で ReconciliationPaused に変わります。

一時停止された調整条件タイプを持つカスタムリソースの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  annotations:
    strimzi.io/pause-reconciliation: "true"
    strimzi.io/use-connector-resources: "true"
  creationTimestamp: 2021-03-12T10:47:11Z
  #...
spec:
  # ...
status:
  conditions:
  - lastTransitionTime: 2021-03-12T10:47:41.689249Z
    status: "True"
    type: ReconciliationPaused
```

一時停止からの再開

- 調整を再開するには、アノテーションを false に設定するか、アノテーションを削除します。

関連情報

- OpenShift リソースのカスタマイズ

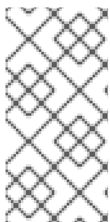
- カスタムリソースのステータスの検出

13.3. AMQ STREAMS DRAIN CLEANER での POD のエビクト

Kafka および ZooKeeper Pod は、OpenShift のアップグレード、メンテナンス、または Pod の再スケジュール時にエビクトされる可能性があります。Kafka ブローカーおよび ZooKeeper Pod が AMQ Streams によってデプロイされた場合、AMQ Streams の Drain Cleaner ツールを使用して Pod のエビクションを処理できます。AMQ Streams の Drain クリーニングが機能するには、Kafka デプロイメントの podDisruptionBudget を 0 (ゼロ) に設定する必要があります。

AMQ Streams Drain Cleaner をデプロイすると、Cluster Operator を使用して OpenShift ではなく Kafka Pod を移動できます。Cluster Operator は、トピックが複製の数が最低数未満になるようにします。エビクションプロセス中に Kafka は動作し続けます。OpenShift ワーカーノードが連続してドレイン (解放) されるため、Cluster Operator はトピックの同期を待機します。

受付 Webhook は AMQ Streams に Pod のエビクション要求のクリーニングを Kubernetes API に通知します。その後、AMQ Streams Drain Cleaner は、ドレイン (解放) される Pod にローリングアップデートアノテーションを追加します。これにより、Cluster Operator にエビクトされた Pod のローリングアップデートを実行するように指示します。



注記

AMQ Streams の Drain クリーニングを使用しない場合は、Pod アノテーションを追加して手動でローリングアップデートを実行できます。

Webhook の設定

AMQ Streams の Drain クリーニングデプロイメントファイルには、Validating WebhookConfiguration リソースファイルが含まれます。リソースは、Kubernetes API で Webhook を登録するための設定を提供します。

この設定は、Pod のエビクション要求時に実行する必要のある Kubernetes API の ルールを定義します。ルールは、pods /eviction サブリソースに関連する CREATE 操作のみがインターセプトされることを指定します。これらのルールが満たされると、API は通知を転送します。

clientConfig は、Webhook を公開する AMQ Streams Drain Cleaner サービスおよび /drainer エンドポイントを参照します。Webhook は認証が必要なセキュアな TLS 接続を使用します。The caBundle プロパティーは、HTTPS 通信を検証する証明書チェーンを指定します。証明書は Base64 でエンコードされます。

Pod エビクション通知の Webhook 設定

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
# ...
webhooks:
  - name: strimzi-drain-cleaner.strimzi.io
    rules:
      - apiGroups:   [""]
        apiVersions: ["v1"]
        operations:  ["CREATE"]
        resources:   ["pods/eviction"]
        scope:       "Namespaced"
    clientConfig:
      service:
        namespace: "strimzi-drain-cleaner"
        name: "strimzi-drain-cleaner"
        path: /drainer
        port: 443
        caBundle: Cg==
    # ...
```

13.3.1. 前提条件

AMQ Streams の Drain クリーニングをデプロイおよび使用するには、デプロイメントファイルをダウンロードする必要があります。

AMQ Streams Drain Cleaner デプロイメントファイルは、ダウンロード可能なインストールと、AMQ Streams のダウンロードサイト からサンプルファイルで提供されます。

13.3.2. AMQ Streams の Drain クリーニングのデプロイ

AMQ Streams の Drain クリーニングを、Cluster Operator および Kafka クラスターが稼働している OpenShift クラスターにデプロイします。

前提条件

- AMQ Streams の Drain クリーニングデプロイメントファイルをダウンロードしている。
-

更新する OpenShift ワーカーノードと共に、高可用性 Kafka クラスターデプロイメントが実行されている必要があります。

- 高可用性のためにトピックがレプリケートされます。

  トピック設定は、3 以上のレプリケーション係数を指定し、最小数の In-Sync レプリカをレプリケーション係数より 1 小さく指定します。

  高可用性のためにレプリケートされた Kafka トピック

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

ZooKeeper の除外

ZooKeeper を追加しない場合は、AMQ Streams Drain Cleaner Deployment 設定ファイルから --zookeeper コマンドオプションを削除できます。

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-drain-cleaner
      containers:
        - name: strimzi-drain-cleaner
          # ...
          command:
            - "/application"
            - "-Dquarkus.http.host=0.0.0.0"
```

```
    - "--kafka"
    - "--zookeeper" (1)
  # ...
```

(1) このオプションを削除して、ZooKeeper を AMQ Streams の Drain クリーニング操作から除外します。

手順

1. Kafka リソースのテンプレート 設定を使用して、Kafka デプロイメントの Pod の Disruption Budget （ ゼロ）を設定します 。

   Pod の Disruption Budget（停止状態の予算）の指定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    template:
      podDisruptionBudget:
        maxUnavailable: 0

  # ...
  zookeeper:
    template:
      podDisruptionBudget:
        maxUnavailable: 0
  # ...
```

   Pod の Disruption Budget（停止状態の予算）をゼロに減らすと、不要な中断が発生した場合に OpenShift が自動的に Pod をエビクトできなくなるため、Pod は AMQ Streams の Drain クリーニングer によってエビクトされる必要があります。

   AMQ Streams の Drain クリーニングを使用して ZooKeeper ノードをドレイン（解放）する場合は、ZooKeeper に同じ設定を追加します。

2. Kafka リソースを更新します。

```
oc apply -f <kafka-configuration-file>
```

3. AMQ Streams Drain Cleaner をデプロイします。

```
oc apply -f ./install/drain-cleaner/openshift
```

13.3.3. AMQ Streams の Drain クリーニングの使用

Cluster Operator と組み合わせて AMQ Streams の Drain Cleaner を使用し、ドレイン（解放）されたノードから Kafka ブローカーまたは ZooKeeper Pod を移動します。AMQ Streams Drain Cleaner を実行すると、Pod にローリングアップデート Pod アノテーションが付けられます。Cluster Operator はアノテーションに基づいてローリングアップデートを実行します。

前提条件

- AMQ Streams の Drain クリーニングがデプロイ済みである必要があります。

手順

1. Kafka ブローカーまたは ZooKeeper Pod をホストする指定された OpenShift ノードをドレイン（解放）します。

```
oc get nodes
oc drain <name-of-node> --delete-emptydir-data --ignore-daemonsets --timeout=6000s
--force
```

2. AMQ Streams Drain Cleaner ログでエビクションイベントを確認し、Pod に再起動のアノテーションが付けられていることを確認します。

AMQ Streams Drain Cleaner ログが Pod のアノテーションを表示する

```
INFO ... Received eviction webhook for Pod my-cluster-zookeeper-2 in namespace my-
project
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project will be annotated for
restart
INFO ... Pod my-cluster-zookeeper-2 in namespace my-project found and annotated
for restart
```

```
INFO ... Received eviction webhook for Pod my-cluster-kafka-0 in namespace my-project
INFO ... Pod my-cluster-kafka-0 in namespace my-project will be annotated for restart
INFO ... Pod my-cluster-kafka-0 in namespace my-project found and annotated for restart
```

3. Cluster Operator ログで調整イベントを確認し、ローリングアップデートを確認します。

Cluster Operator ログにローリングアップデートが表示される

```
INFO  PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster): Rolling Pod my-cluster-zookeeper-2
INFO  PodOperator:68 - Reconciliation #13(timer) Kafka(my-project/my-cluster): Rolling Pod my-cluster-kafka-0
INFO  AbstractOperator:500 - Reconciliation #13(timer) Kafka(my-project/my-cluster): reconciled
```

13.4. KAFKA および ZOOKEEPER クラスターの手動によるローリングアップデートの開始

AMQ Streams は、Cluster Operator 経由で Kafka および ZooKeeper クラスターのローリングアップデートを手動でトリガーするために、StatefulSet および Pod リソースでのアノテーションの使用をサポートします。ローリングアップデートにより、新しい Pod でリソースの Pod が再起動されます。

通常、例外的な状況でのみ、特定の Pod または同じ StatefulSet からの Pod のセットを手動で実行する必要があります。ただし、Pod を直接削除せずに、Cluster Operator 経由でローリングアップデートを実行すると、以下を確実に行うことができます。

- Pod を手動で削除しても、他の Pod を並行して削除するなどの、同時に行われる Cluster Operator の操作とは競合しません。
- Cluster Operator ロジックによって、In-Sync レプリカの数などの Kafka 設定で指定された内容が処理されます。

13.4.1. 前提条件

手動でローリングアップデートを実行するには、稼働中の Cluster Operator および Kafka クラスターが必要です。

以下を実行する方法については、『OpenShift での AMQ Streams のデプロイおよびアップグレード』を参照してください。

- Cluster Operator
- Kafka クラスター

13.4.2. StatefulSet アノテーションを使用したローリングアップデートの実行

この手順では、OpenShift StatefulSet アノテーションを使用して、既存の Kafka クラスターまたは ZooKeeper クラスターのローリングアップデートを手動でトリガーする方法を説明します。

手順

1. 手動で更新する Kafka または ZooKeeper Pod を制御する StatefulSet の名前を見つけます。

   たとえば、Kafka クラスターの名前が my-cluster の場合、対応する StatefulSet 名は my-cluster-kafka と my-cluster-zookeeper になります。

2. OpenShift で StatefulSet リソースにアノテーションを付けます。

   oc annotate を使用します。

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

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

3. 次の調整が発生するまで待ちます (デフォルトでは 2 分ごとです)。アノテーションが調整プロセスで検出されれば、アノテーションが付いた StatefulSet 内のすべての Pod でローリングアップデートがトリガーされます。すべての Pod のローリングアップデートが完了すると、アノテーションは StatefulSet から削除されます。

13.4.3. Pod アノテーションを使用したローリングアップデートの実行

この手順では、OpenShift Pod アノテーションを使用して、既存の Kafka クラスターまたは ZooKeeper クラスターのローリングアップデートを手動でトリガーする方法を説明します。同じ StatefulSet の複数の Pod にアノテーションが付けられると、連続したローリングアップデートは同じ調整実行内で実行されます。

前提条件

使用するトピックレプリケーション係数に関係なく、Kafka クラスターでローリングアップデートを実行できます。ただし、更新中 Kafka は引き続き稼働しているようにするには、以下が必要になります。

- 更新するノードで実行している高可用性 Kafka クラスターデプロイメント。
- 高可用性のためにレプリケートされたトピック。

  トピック設定は、3 以上のレプリケーション係数を指定し、最小数の In-Sync レプリカをレプリケーション係数より 1 小さく指定します。

  高可用性のためにレプリケートされた Kafka トピック

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

手順

1. 手動で更新する Kafka または ZooKeeper Pod の名前を見つけます。

   たとえば、Kafka クラスターの名前が my-cluster の場合、対応する Pod 名は my-cluster-kafka-index と my-cluster-zookeeper-index になります。インデックスは ゼロで始まり、レプリカの合計数から 1 を引いた値になります。

2. OpenShift で Pod リソースにアノテーションを付けます。

   oc annotate を使用します。

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

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

3. 次の調整が発生するまで待ちます (デフォルトでは 2 分ごとです)。アノテーションが調整プロセスで検出されれば、アノテーションが付けられた Pod のローリングアップデートがトリガーされます。Pod のローリングアップデートが完了すると、アノテーションは Pod から削除されます。

13.5. ラベルおよびアノテーションを使用したサービスの検出

サービスディスカバリーは、AMQ Streams と同じ OpenShift クラスターで稼働しているクライアントアプリケーションの Kafka クラスターとの対話を容易にします。

サービスディスカバリー ラベルおよびアノテーションは、Kafka クラスターにアクセスするために使用されるサービスに対して生成されます。

- 内部 Kafka ブートストラップサービス
- HTTP Bridge サービス

ラベルは、サービスの検出を可能にします。アノテーションは、クライアントアプリケーションが接続を確立するために使用できる接続詳細を提供します。

サービスディスカバリーラベル strimzi.io/discovery は、Service リソースに対して true に設定されています。サービスディスカバリーアノテーションには同じキーがあり、各サービスの接続詳細を

JSON 形式で提供します。

内部 Kafka ブートストラップサービスの例

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-
      [ {
        "port" : 9092,
        "tls" : false,
        "protocol" : "kafka",
        "auth" : "scram-sha-512"
      }, {
        "port" : 9093,
        "tls" : true,
        "protocol" : "kafka",
        "auth" : "tls"
      } ]
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/discovery: "true"
    strimzi.io/kind: Kafka
    strimzi.io/name: my-cluster-kafka-bootstrap
  name: my-cluster-kafka-bootstrap
spec:
  #...
```

HTTP Bridge サービスの例

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    strimzi.io/discovery: |-
      [ {
        "port" : 8080,
        "tls" : false,
        "auth" : "none",
        "protocol" : "http"
      } ]
  labels:
    strimzi.io/cluster: my-bridge
    strimzi.io/discovery: "true"
    strimzi.io/kind: KafkaBridge
    strimzi.io/name: my-bridge-bridge-service
```

13.5.1. サービスの接続詳細の返信

サービスを検出するには、コマンドラインまたは対応する API 呼び出しでサービスを取得するときに、ディスカバリーラベルを指定します。

```
oc get service -l strimzi.io/discovery=true
```

サービスディスカバリーラベルの取得時に接続詳細が返されます。

13.6. 永続ボリュームからのクラスターの復元

Kafka クラスターは、永続ボリューム (PV) が存在していれば、そこから復元できます。

たとえば、以下の場合に行います。

- namespace が意図せずに削除された後。
- OpenShift クラスター全体が失われた後でも PV がインフラストラクチャーに残っている場合。

13.6.1. namespace が削除された場合の復元

永続ボリュームと namespace の関係により、namespace の削除から復元することが可能です。PersistentVolume (PV) は、namespace の外部に存在するストレージリソースです。PV は、namespace 内部に存在する PersistentVolumeClaim (PVC) を使用して Kafka Pod にマウントされます。

PV の回収 (reclaim) ポリシーは、namespace が削除されるときにクラスターに動作方法を指示します。以下に、回収 (reclaim) ポリシーの設定とその結果を示します。

- Delete (デフォルト) に設定すると、PVC が namespace 内で削除されるときに PV が削除されます。
- Retain に設定すると、namespace の削除時に PV は削除されません。

namespace が意図せず削除された場合に PV から復旧できるようにするには、PV 仕様で persistentVolumeReclaimPolicy プロパティーを使用してポリシーを Delete から Retain にリセットする必要があります。

```
apiVersion: v1
```

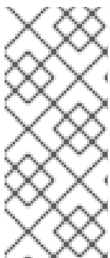
```
kind: PersistentVolume
# ...
spec:
  # ...
  persistentVolumeReclaimPolicy: Retain
```

または、PV は、関連付けられたストレージクラスの回収 (reclaim) ポリシーを継承できます。ストレージクラスは、動的ボリュームの割り当てに使用されます。

ストレージクラスの reclaimPolicy プロパティーを設定することで、ストレージクラスを使用する PV が適切な回収 (reclaim) ポリシー で作成されます。ストレージクラスは、storageClassName プロパティーを使用して PV に対して設定されます。

```
apiVersion: v1
kind: StorageClass
metadata:
  name: gp2-retain
parameters:
  # ...
# ...
reclaimPolicy: Retain
```

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
  # ...
  storageClassName: gp2-retain
```



注記

Retain を回収 (reclaim) ポリシーとして使用しながら、クラスター全体を削除する場合は、PV を手動で削除する必要があります。そうしないと、PV は削除されず、リソースに不要な経費がかかる原因になります。

13.6.2. OpenShift クラスター喪失からの復旧

クラスターが失われた場合、ディスク/ボリュームのデータがインフラストラクチャー内に保持されていれば、それらのデータを使用してクラスターを復旧できます。PV が復旧可能でそれらが手動で作成されていれば、復旧の手順は namespace の削除と同じです。

13.6.3. 削除したクラスターの永続ボリュームからの復元

この手順では、削除されたクラスターを永続ボリューム (PV) から復元する方法を説明します。

この状況では、Topic Operator はトピックが Kafka に存在することを認識しますが、KafkaTopic リソースは存在しません。

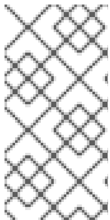
クラスター再作成の手順を行うには、2 つの方法があります。

1. すべての KafkaTopic リソースを復旧できる場合は、オプション 1 を使用します。

   これにより、クラスターが起動する前に KafkaTopic リソースを復旧することで、該当するトピックが Topic Operator によって削除されないようにする必要があります。

2. すべての KafkaTopic リソースを復旧できない場合は、オプション 2 を使用します。

   この場合、Topic Operator なしでクラスターをデプロイし、Topic Operator のトピックストアメタデータを削除してから、Topic Operator で Kafka クラスターを再デプロイすることで、該当するトピックから KafkaTopic リソースを再作成できるようにします。



注記

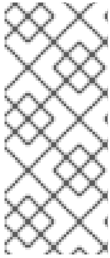
Topic Operator がデプロイされていない場合は、PersistentVolumeClaim (PVC) リソースのみを復旧する必要があります。

作業を始める前に

この手順では、データの破損を防ぐために PV を正しい PVC にマウントする必要があります。volumeName が PVC に指定されており、それが PV の名前に一致する必要があります。

詳細は以下を参照してください。

- Persistent Volume Claim (永続ボリューム要求、PVC) の命名
- JBOD および 永続ボリューム要求 (PVC)



注記

この手順には、手動での再作成が必要な KafkaUser リソースの復旧は含まれません。パスワードと証明書を保持する必要がある場合は、KafkaUser リソースの作成前にシークレットを再作成する必要があります。

手順

1. クラスターの PV についての情報を確認します。

```
oc get pv
```

PV の情報がデータとともに表示されます。

この手順で重要な列を示す出力例:

```
NAME                                        RECLAIMPOLICY CLAIM
pvc-5e9c5c7f-3317-11ea-a650-06e1eadd9a4c ... Retain ...    myproject/data-my-cluster-zookeeper-1
pvc-5e9cc72d-3317-11ea-97b0-0aef8816c7ea ... Retain ...    myproject/data-my-cluster-zookeeper-0
pvc-5ead43d1-3317-11ea-97b0-0aef8816c7ea ... Retain ...    myproject/data-my-cluster-zookeeper-2
pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c ... Retain ...    myproject/data-0-my-cluster-kafka-0
pvc-7e21042e-3317-11ea-9786-02deaf9aa87e ... Retain ...    myproject/data-0-my-cluster-kafka-1
pvc-7e226978-3317-11ea-97b0-0aef8816c7ea ... Retain ...    myproject/data-0-my-cluster-kafka-2
```

- NAME は各 PV の名前を示します。
- RECLAIM POLICY は PV が 保持される ことを示します。
- CLAIM は元の PVC へのリンクを示します。

2. 元の namespace を再作成します。

```
oc create namespace myproject
```

3. 元の PVC リソース仕様を再作成し、PVC を該当する PV にリンクします。

   以下は例になります。

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-0-my-cluster-kafka-0
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
  storageClassName: gp2-retain
  volumeMode: Filesystem
  volumeName: pvc-7e1f67f9-3317-11ea-a650-06e1eadd9a4c
```

4. PV 仕様を編集して、元の PVC にバインドされた claimRef プロパティーを削除します。

   以下は例になります。

```
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    kubernetes.io/createdby: aws-ebs-dynamic-provisioner
    pv.kubernetes.io/bound-by-controller: "yes"
    pv.kubernetes.io/provisioned-by: kubernetes.io/aws-ebs
  creationTimestamp: "<date>"
  finalizers:
  - kubernetes.io/pv-protection
  labels:
    failure-domain.beta.kubernetes.io/region: eu-west-1
    failure-domain.beta.kubernetes.io/zone: eu-west-1c
  name: pvc-7e226978-3317-11ea-97b0-0aef8816c7ea
  resourceVersion: "39431"
  selfLink: /api/v1/persistentvolumes/pvc-7e226978-3317-11ea-97b0-0aef8816c7ea
  uid: 7efe6b0d-3317-11ea-a650-06e1eadd9a4c
spec:
  accessModes:
  - ReadWriteOnce
  awsElasticBlockStore:
    fsType: xfs
    volumeID: aws://eu-west-1c/vol-09db3141656d1c258
  capacity:
    storage: 100Gi
  claimRef:
    apiVersion: v1
```

```
    kind: PersistentVolumeClaim
    name: data-0-my-cluster-kafka-2
    namespace: myproject
    resourceVersion: "39113"
    uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: failure-domain.beta.kubernetes.io/zone
          operator: In
          values:
          - eu-west-1c
        - key: failure-domain.beta.kubernetes.io/region
          operator: In
          values:
          - eu-west-1
  persistentVolumeReclaimPolicy: Retain
  storageClassName: gp2-retain
  volumeMode: Filesystem
```

この例では、以下のプロパティーが削除されます。

```
claimRef:
  apiVersion: v1
  kind: PersistentVolumeClaim
  name: data-0-my-cluster-kafka-2
  namespace: myproject
  resourceVersion: "39113"
  uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
```

5. Cluster Operator をデプロイします。

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

6. クラスターを再作成します。

クラスターの再作成に必要なすべての KafkaTopic リソースがあるかどうかに応じて、以下の手順を実行します。

オプション 1: クラスターを失う前に存在した KafkaTopic リソースが すべて ある場合 (__consumer_offsets からコミットされたオフセットなどの内部トピックを含む)。

1. すべての KafkaTopic リソースを再作成します。

クラスターをデプロイする前にリソースを再作成する必要があります。そうでないと、Topic Operator によってトピックが削除されます。

2. Kafka クラスターをデプロイします。

   以下は例になります。

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

オプション 2: クラスターを失う前に存在したすべての KafkaTopic リソースがない場合。

1. オプション 1 と同様に Kafka クラスターをデプロイしますが、デプロイ前に Kafka リソースから topicOperator プロパティーを削除して、Topic Operator がない状態でデプロイします。

   デプロイメントに Topic Operator が含まれると、Topic Operator によってすべてのトピックが削除されます。

2. Kafka クラスターから内部トピックストアのトピックを削除します。

```
oc run kafka-admin -ti --image=registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

   このコマンドは、Kafka クラスターへのアクセスに使用されるリスナーおよび認証のタイプに対応している必要があります。

3. Kafka クラスターを topicOperator プロパティーで再デプロイして TopicOperator を有効にし、KafkaTopic リソースを再作成します。

   以下は例になります。

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

```
spec:
  #...
  entityOperator:
    topicOperator: {} 1
    #...
```

1

ここで示すデフォルト設定には、追加のプロパティーはありません。「EntityTopicOperatorSpec スキーマ参照」に説明されているプロパティーを使用して、必要な設定を指定します。

7. KafkaTopic リソースのリストを表示して、復旧を確認します。

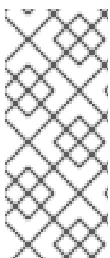
```
oc get KafkaTopic
```

13.7. KAFKA STATIC QUOTA プラグインを使用したブローカーへの制限の設定

Kafka Static Quota プラグインを使用して、Kafka クラスターのブローカーにスループットおよびストレージの制限を設定します。Kafka リソースを設定して、プラグインを有効にし、制限を設定します。バイトレートのしきい値およびストレージクォータを設定して、ブローカーと対話するクライアントに制限を設けることができます。

プロデューサーおよびコンシューマー帯域幅にバイトレートのしきい値を設定できます。制限の合計は、ブローカーにアクセスするすべてのクライアントに分散されます。たとえば、バイトレートのしきい値として 40 MBps をプロデューサーに設定できます。2 つのプロデューサーが実行されている場合、それぞれのスループットは 20MBps に制限されます。

ストレージクォータは、Kafka ディスクストレージの制限をソフト制限とハード制限間で調整します。この制限は、利用可能なすべてのディスク容量に適用されます。プロデューサーは、ソフト制限とハード制限の間で徐々に遅くなります。制限により、ディスクの使用量が急激に増加しないようにし、容量を超えないようにします。ディスクがいっぱいになると、修正が難しい問題が発生する可能性があります。ハード制限は、ストレージの上限です。



注記

JBOD ストレージの場合、制限はすべてのディスクに適用されます。ブローカーが 2 つの 1 TB ディスクを使用し、クォータが 1.1 TB の場合は、1 つのディスクにいっぱいになり、別のディスクがほぼ空になることがあります。

前提条件

- Kafka クラスターを管理する Cluster Operator が稼働している。

手順

1. Kafkaリソースのconfigにプラグインのプロパティを追加します。

   プラグインプロパティーは、この設定例のとおりです。

Kafka Static Quota プラグインの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    config:
      client.quota.callback.class: io.strimzi.kafka.quotas.StaticQuotaCallback (1)
      client.quota.callback.static.produce: 1000000 (2)
      client.quota.callback.static.fetch: 1000000 (3)
      client.quota.callback.static.storage.soft: 400000000000 (4)
      client.quota.callback.static.storage.hard: 500000000000 (5)
      client.quota.callback.static.storage.check-interval: 5 (6)
```

(1) Kafka Static Quota プラグインを読み込みます。

(2) プロデューサーのバイトレートしきい値を設定します。この例では 1 MBps です。

(3) コンシューマーのバイトレートしきい値を設定します。この例では 1 MBps です。

(4)

ストレージのソフト制限の下限を設定します。この例では 400 GB です。

5

ストレージのハード制限の上限を設定します。この例では 500 GB です。

6

ストレージのチェックの間隔 (秒単位) を設定します。この例では 5 秒です。これを 0 に設定するとチェックを無効にできます。

2. リソースを更新します。

```
oc apply -f KAFKA-CONFIG-FILE
```

その他のリソース

- Kafka ブローカー設定のチューニング
- ユーザークォータの設定

13.8. KAFKA 設定のチューニング

設定プロパティーを使用して、Kafka ブローカー、プロデューサー、およびコンシューマーのパフォーマンスを最適化します。

最小セットの設定プロパティーが必要ですが、プロパティーを追加または調整して、プロデューサーとコンシューマーが Kafka ブローカーと対話する方法を変更できます。たとえば、クライアントがリアルタイムでデータに応答できるように、メッセージのレイテンシーおよびスループットをチューニングできます。

メトリックを分析して初期設定を行う場所を判断することから始め、必要な設定になるまで段階的に変更を加え、さらにメトリクスの比較を行うことができます。

Apache Kafka 設定プロパティーの詳細は、Apache Kafka のドキュメントを参照してください。

13.8.1. Kafka ブローカー設定のチューニング

設定プロパティーを使用して、Kafka ブローカーのパフォーマンスを最適化します。AMQ Streams によって直接管理されるプロパティーを除き、標準の Kafka ブローカー設定オプションを使用できます。

13.8.1.1. 基本的なブローカー設定

特定のブローカー設定オプションは AMQ Streams によって直接管理されます。これは、Kafka カスタムリソース仕様によって実行されます。

- broker.id は Kafka ブローカーの ID です。
- log.dirs はログデータのディレクトリーです。
- zookeeper.connect は、ZooKeeper と Kafka に接続するための設定です。
- listenerは Kafka クラスターをクライアントに公開します。
- authorization がユーザーが実行するアクションを許可または拒否する
- authenticationは、Kafka へのアクセスを必要とするユーザーのアイデンティティーを証明します。

ブローカー ID は 0 (ゼロ) から開始し、ブローカーレプリカの数に対応します。ログディレクトリは、Kafkaカスタムリソースのspec.kafka.storage設定に基づき、/var/lib/kafka/data/kafka-logIDXにマウントされます。IDX は Kafka ブローカー Pod インデックスです。

そのため、Kafka カスタムリソースの config プロパティーを使用して、これらのオプションを設定することはできません。除外項目の一覧については、KafkaClusterSpec schema referenceを参照してください。

ただし、通常のブローカー設定には、トピック、スレッド、およびログに関連するプロパティーの設定が含まれます。

基本的なブローカープロパティー

```
# ...
num.partitions=1
default.replication.factor=3
offsets.topic.replication.factor=3
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
num.network.threads=3
num.io.threads=8
num.recovery.threads.per.data.dir=1
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
group.initial.rebalance.delay.ms=0
zookeeper.connection.timeout.ms=6000
# ...
```

13.8.1.2. 高可用性のためのトピックの複製

基本的なトピックプロパティーは、トピックのデフォルト数のパーティションおよびレプリケーション係数を設定します。これは、トピックが自動的に作成される場合を含め、これらのプロパティーを明示的に設定せずに作成されたトピックに適用されます。

```
# ...
num.partitions=1
auto.create.topics.enable=false
default.replication.factor=3
min.insync.replicas=2
replica.fetch.max.bytes=1048576
# ...
```

auto.create.topics.enable プロパティーはデフォルトで有効になっており、存在しないトピックがプロデューサーおよびコンシューマーによって必要になると自動的に作成されます。トピックの自動作成を使用する場合は、**num.partitions** を使用してトピックのデフォルトのパーティション数を設定できます。しかし、一般的には、このプロパティは無効にして、明示的なトピック作成によってトピックをより制御できるようにします。例えば、AMQ StreamsKafkaTopicリソースやアプリケーションを使ってトピックを作成することができます。

高可用性環境の場合は、トピックに対してレプリケーション係数を 3 以上に引き上げ、必要な同期

レプリカの最小数をレプリケーション係数より 1 少なく設定することをお勧めします。KafkaTopicリソースを使用して作成されたトピックの場合、レプリケーションファクターはspec.replicasで設定されます。

また、データの耐久性を確保するために、トピックの設定でmin.insync.replicasを設定し、プロデューサーの設定でacks=allを使用してメッセージ配信の確認を行う必要があります。

replica.fetch.max.bytes を使用して、リーダーパーティションを複製する各フォロワーが取得したメッセージの最大サイズ（バイト単位）を設定します。この値は、平均のメッセージサイズおよびスループットに応じて変更します。読み取り/書き込みバッファーに必要なメモリー割り当ての合計を考慮する際に、利用可能なメモリーも、すべてのフォロワーで乗算したレプリケートされたメッセージの最大サイズに対応できる必要があります。

delete.topic.enable プロパティーはデフォルトで有効になっており、トピックの削除を許可します。実稼働環境では、誤ってトピックが削除され、データが失われるのを防ぐために、このプロパティーを無効にする必要があります。ただし、トピックを一時的に有効にして、トピックを削除してから再度無効にできます。delete.topic.enable が有効になっている場合は、KafkaTopic リソースを使用してトピックを削除できます。

```
# ...
auto.create.topics.enable=false
delete.topic.enable=true
# ...
```

13.8.1.3. トランザクションおよびコミットの内部トピック設定

トランザクションを使用してプロデューサーからのパーティションへのアトミック書き込みを有効にする場合、トランザクションの状態は内部 __transaction_state トピックに保存されます。デフォルトでは、ブローカーはレプリケーション係数が 3 で設定され、このトピックでは少なくとも 2 つの同期レプリカが設定されます。つまり、Kafka クラスターには少なくとも 3 つのブローカーが必要になります。

```
# ...
transaction.state.log.replication.factor=3
transaction.state.log.min.isr=2
# ...
```

同様に、コンシューマーの状態を格納する内部 __consumer_offsets トピックには、パーティションおよびレプリケーション係数のデフォルト設定があります。

```
# ...
offsets.topic.num.partitions=50
offsets.topic.replication.factor=3
# ...
```

-

実稼働ではこれらの設定を下げないでください。実稼働環境で設定を大きくすることができます。例外として、単一ブローカーのテスト環境の設定を下げる必要がある場合があります。

13.8.1.4. I/O スレッドの増加によるリクエスト処理スループットの向上

ネットワークスレッドは、クライアントアプリケーションからのリクエストの生成や取得など、Kafka クラスターへのリクエストを処理します。生成リクエストはリクエストキューに配置されます。応答は応答キューに配置されます。

ネットワークスレッドの数は、レプリケーション係数と、Kafkaクラスターと、対話するクライアントプロデューサーおよびコンシューマーからのアクティビティーのレベルを反映する必要があります。リクエストが多い場合は、スレッドがアイドル状態である時間を使用してスレッドの数を増やし、スレッドを追加するタイミングを決定できます。

輻輳を軽減し、要求トラフィックを規制するには、ネットワークスレッドがブロックされる前に、要求キューで許可されるリクエスト数を制限できます。

I/O スレッドはリクエストキューからリクエストを選択して処理します。スレッド数を増やすとスループットが向上しますが、CPU のコアの数とおよびディスク帯域幅により、実用的な上限が決まります。最低でも、I/O スレッドの数はストレージボリュームの数と同じでなければなりません。

```
# ...
num.network.threads=3 (1)
queued.max.requests=500 (2)
num.io.threads=8 (3)
num.recovery.threads.per.data.dir=1 (4)
# ...
```

(1) Kafka クラスターのネットワークスレッドの数。

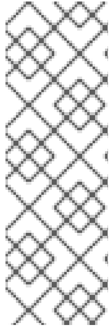
(2) リクエストキューで許可されるリクエストの数。

(3) Kafka ブローカーの I/O スレッドの数。

(4)

起動時のログの読み込みおよびシャットダウン時のフラッシュに使用されるスレッドの数。

すべてのブローカーのスレッドプールへの設定の更新は、クラスターレベルで動的に発生する可能性があります。これらの更新は、現在のサイズの半分から現在のサイズの 2 倍までに制限されます。



注記

Kafka ブローカーメトリクスは、必要なスレッドの数を計算するのに役立ちます。たとえば、平均のネットワークスレッドのメトリクスはアイドル状態 (kafka.network:type=SocketServer,name=NetworkProcessorAvgIdlePercent)の場合は、使用されるリソースのパーセンテージを示します。0% のアイドル時間がある場合、すべてのリソースが使用中であり、スレッドの追加は有益になります。

ディスクの数によりスレッドが遅くなり、制限される場合は、ネットワーク要求のバッファーのサイズを増やしてスループットを向上させることができます。

```
# ...
replica.socket.receive.buffer.bytes=65536
# ...
```

また、Kafka が受信可能な最大バイト数も増やします。

```
# ...
socket.request.max.bytes=104857600
# ...
```

13.8.1.5. レイテンシーの高い接続に対する帯域幅の引き上げ

Kafka はデータをバッチ処理して、データセンター間の接続など、Kafka からクライアントへのレイテンシーの高い接続で妥当なスループットを実現します。ただし、レイテンシーの高さが問題である場合、メッセージを送受信するためのバッファーのサイズを増やすことができます。

```
# ...
socket.send.buffer.bytes=1048576
socket.receive.buffer.bytes=1048576
# ...
```

帯域幅遅延積の計算を使用して、バッファーの最適なサイズを見積もることができます。これは、リンクの最大帯域幅 (バイト/秒) にラウンドトリップ遅延 (秒) を掛けて、最大スループットを維持する

ために必要なバッファーの大きさを見積もります。

13.8.1.6. データ保持ポリシーでのログの管理

Kafka はログを使用してメッセージデータを保存します。ログは、さまざまなインデックスに関連付けられた一連のセグメントです。新しいメッセージはアクティブなセグメントに書き込まれ、その後変更されません。セグメントは、コンシューマーからのフェッチ要求に対応するときに読み取られます。定期的に、アクティブセグメントがロールされて読み取り専用になり、それを置き換えるために新しいアクティブセグメントが作成されます。一度にアクティブにできるセグメントは 1 つだけです。古いセグメントは、削除対象となるまで保持されます。

ブローカーレベルでの設定では、ログセグメントの最大サイズをバイト単位で設定し、アクティブなセグメントがロールされるまでの時間をミリ秒単位で設定します。

```
# ...
log.segment.bytes=1073741824
log.roll.ms=604800000
# ...
```

これらの設定は、segment.bytes および segment.ms を使用してトピックレベルで上書きできます。これらの値を下げるまたは上げる必要があるかどうかは、セグメント削除のポリシーによって異なります。サイズが大きいほど、アクティブセグメントに含まれるメッセージが多くなり、ロールされる頻度が少なくなります。セグメントも削除の対象となる頻度が少なくなります。

時間ベースまたはサイズベースのログの保持およびクリーンアップポリシーを設定して、ログを管理しやすくすることができます。要件によっては、ログ保持の設定を使用して古いセグメントを削除できます。ログ保持ポリシーが使用される場合、保持制限に達すると、アクティブではないログセグメントが削除されます。古いセグメントを削除すると、ディスク領域が超過しないように、ログに必要なストレージ領域がバインドされます。

期間ベースのログの保持には、時間、分、およびミリ秒に基づいて保持期間を設定します。保持期間は、メッセージがセグメントに追加された時間に基づいています。

ミリ秒設定は分設定よりも優先され、分設定は時間設定おりも優先されます。分とミリ秒の設定はデフォルトで null ですが、3つのオプションにより、保持するデータを実質的に制御できます。動的に更新できるのは 3 つのプロパティーの 1 つだけであるため、ミリ秒設定を優先する必要があります。

```
# ...
log.retention.ms=1680000
# ...
```

log.retention.ms が -1 に設定されている場合には、ログ保持には時間制限が適用されないため、すべてのログが保持されます。ディスクの使用状況は常に監視する必要がありますが、-1 の設定は、ディスクがいっぱいになると問題が発生する可能性があり、修正が難しいため、一般的にはお勧めしません。

サイズベースのログの保持には、最大ログサイズ (ログのすべてのセグメント) をバイト単位で設定します。

```
# ...
log.retention.bytes=1073741824
# ...
```

つまり、通常、ログが定常状態に達すると、およそ log.retention.bytes /log.segment.bytes の数のセグメントを持ちます。最大ログサイズに達すると、古いセグメントが削除されます。

最大ログサイズの使用に関する潜在的な問題は、メッセージがセグメントに追加された時刻が考慮されていないことです。クリーンアップポリシーに時間ベースおよびサイズベースのログ保持を使用して、必要なバランスをとることができます。どちらのしきい値に最初に到達しても、クリーンアップがトリガーされます。

セグメントファイルがシステムから削除される前に遅延を追加する場合は、トピック設定の特定のトピックについて、ブローカーレベルまたは file.delete.delay.ms のトピックで log.segment.delete.delay.ms を使用して遅延を追加できます。

```
# ...
log.segment.delete.delay.ms=60000
# ...
```

13.8.1.7. クリーンアップポリシーによるログデータの削除

古いログデータを削除する方法は、ログクリーナー設定によって決定されます。

ログクリーナーは、ブローカーに対してデフォルトで有効になっています。

```
# ...
log.cleaner.enable=true
# ...
```

クリーンアップポリシーは、トピックまたはブローカーレベルで設定できます。ブローカーレベルの設定は、ポリシーが設定されていないトピックのデフォルトです。

ログの削除、ログの圧縮、その両方を行うためにポリシーを設定できます。

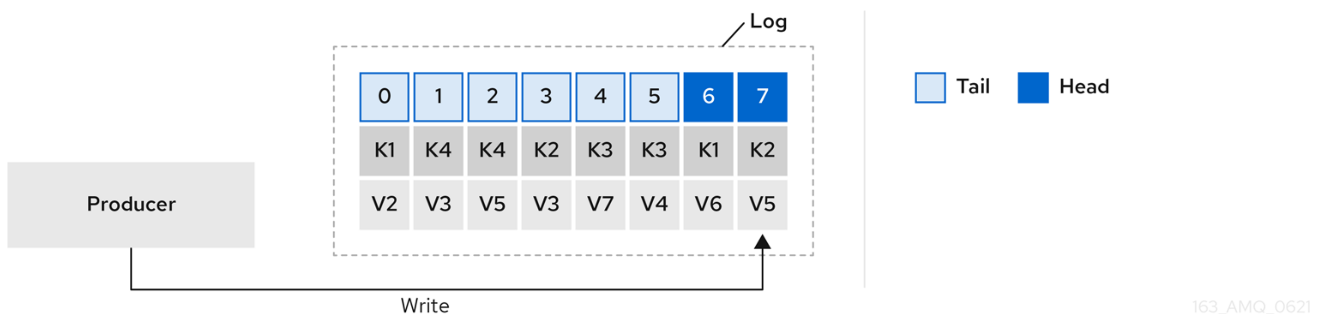
```
# ...
log.cleanup.policy=compact,delete
# ...
```

delete ポリシーは、データ保持ポリシーを使用したログの管理に対応します。データを永久に保持する必要がない場合に適しています。compact ポリシーは、各メッセージキーの最新のメッセージを維持することを保証します。ログコンパクションは、メッセージ値の変更が可能で、最新の更新を保持する場合に適しています。

ログを削除するようにクリーンアップポリシーが設定されている場合、ログの保持制限に基づいて古いセグメントが削除されます。それ以外の場合、ログクリーナーが有効になっておらず、ログの保持制限がないと、ログは増え続けます。

ログコンパクションにクリーンアップポリシーが設定されている場合、ログの先頭は標準の Kafka ログとして機能し、新しいメッセージへの書き込みが順番に追加されます。ログクリーナーが動作する圧縮ログの末尾で、同じキーを持つ別のレコードがログの後半で発生した場合、レコードは削除されます。null 値を持つメッセージも削除されます。キーを使用していない場合、関連するメッセージを識別するためにキーが必要になるため、コンパクションを使用することはできません。Kafka は、各キーの最新のメッセージが保持されることを保証しますが、圧縮されたログ全体に重複が含まれないことを保証するものではありません。

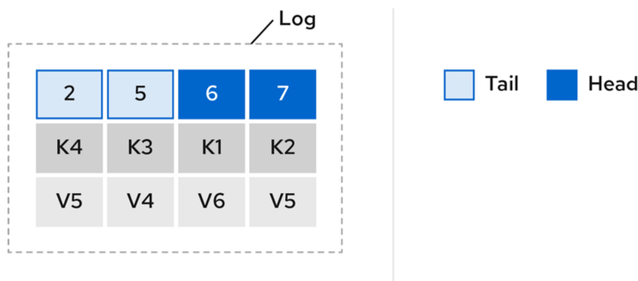
図13.1 コンパクション前のオフセットの位置によるキー値の書き込みを示すログ



鍵を使用してメッセージを特定することで、Kafka のコンパクションは特定のメッセージキーの最新メッセージ (オフセットが最大) を維持し、最終的に同じキーを持つ以前のメッセージを破棄します。つまり、最新状態のメッセージは常に利用可能であり、その特定のメッセージの古いレコードは、ログクリーナーの実行時に最終的に削除されます。メッセージを以前の状態に復元できます。

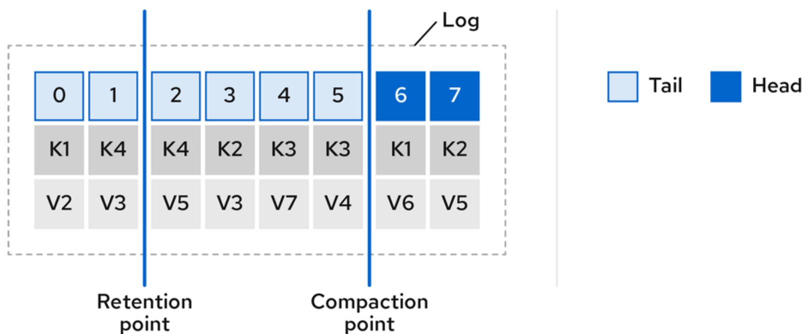
周囲のレコードが削除されても、レコードは元のオフセットを保持します。その結果、末尾は連続しないオフセットを持つ可能性があります。末尾で使用できなくなったオフセットを消費すると、次に高いオフセットを持つレコードが見つかります。

図13.2 コンパクション後のログ



圧縮ポリシーのみを選択すると、ログが任意に大きくなる可能性があります。この場合、ログの圧縮および削除を行うためにポリシーを設定します。コンパクションおよび削除を選択した場合、まずログデータが圧縮され、ログの先頭にあるキーでレコードが削除されます。その後、ログ保持しきい値より前のデータは削除されます。

図13.3 ログ保持ポイントおよびコンパクションポイント



ログのクリーンアップがチェックされる頻度をミリ秒単位で設定します。

```
# ...
log.retention.check.interval.ms=300000
# ...
```

ログ保持設定に関連して、ログ保持チェックの間隔を調整します。保持サイズが小さいほど、より頻繁なチェックが必要になる場合があります。

クリーンアップの頻度は、ディスクスペースを管理するのに十分な頻度である必要がありますが、トピックのパフォーマンスに影響を与えるほど頻度を上げてはなりません。

クリーニングするログがない場合にクリーナーをスタンバイにする時間をミリ秒単位で設定することもできます。

```
# ...
log.cleaner.backoff.ms=15000
# ...
```

古いログデータの削除を選択した場合、パージする前に削除されたデータを保持する期間をミリ秒単位で設定できます。

```
# ...
log.cleaner.delete.retention.ms=86400000
# ...
```

削除されたデータの保持期間は、データが完全に削除される前に、データが削除されたことに気付く時間を確保します。

特定のキーに関連するすべてのメッセージを削除するために、プロデューサーは廃棄 (tombstone) メッセージを送信できます。廃棄 (tombstone) には null 値があり、値が削除されることを示すマーカーとして機能します。コンパクション後に廃棄 (tombstone) のみが保持されます。これは、コンシューマーがメッセージが削除されたことを認識するのに十分な期間である必要があります。古いメッセージが削除され、値がないと、tombstone キーもパーティションから削除されます。

13.8.1.8. ディスク使用率の管理

ログクリーンアップに関する他の設定には数多くありますが、特に重要なのはメモリー割り当てです。

重複排除 (deduplication) プロパティーは、すべてのログクリーナースレッド全体でクリーンアップの合計メモリーを指定します。バッファー負荷係数で使用されるメモリーの割合の上限を設定できます。

```
# ...
log.cleaner.dedupe.buffer.size=134217728
log.cleaner.io.buffer.load.factor=0.9
# ...
```

各ログエントリーは正確に 24 バイトを使用するため、バッファーが 1 回の実行で処理できるログエントリーの数を計算し、それに応じて設定を調整できます。

可能であれば、ログのクリーニング時間を短縮する場合は、ログクリーナースレッドの数を増やすことを検討してください。

```
# ...
log.cleaner.threads=8
# ...
```

ディスク帯域幅の使用率が100%で問題が発生している場合は、読み書き操作の合計が、操作を実行するディスクの機能に基づいて指定された値の 2 倍未満になるように、ログクリーナーの I/O を調整できます。

```
# ...
log.cleaner.io.max.bytes.per.second=1.7976931348623157E308
# ...
```

13.8.1.9. 大きなメッセージサイズの処理

メッセージのデフォルトのバッチサイズは 1MB で、ほとんどのユースケースで最大のスループットを得るのに最適です。Kafkaは、十分なディスク容量があれば、スループットを下げてより大きなバッチに対応できます。

大きなメッセージサイズは、以下の 4 つの方法で処理されます。

1. プロデューサー側のメッセージ圧縮 は、圧縮メッセージをログに書き込みます。

2. 参照ベースのメッセージングは、メッセージの値で他のシステムに格納されているデータへの参照のみを送信します。

3. インラインメッセージングは、メッセージを同じキーを使用するチャンクに分割し、Kafka Streams などのストリームプロセッサーを使用して出力に組み合わされます。

4. より大きなメッセージサイズを処理するように構築されたブローカーおよびプロデューサー/コンシューマークライアントアプリケーション。

リファレンスベースのメッセージングおよびメッセージ圧縮オプションが推奨されます。これはほとんどの状況に対応します。これらのオプションのいずれかを使用する場合は、パフォーマンスの問題が発生しないように注意する必要があります。

プロデューサー側の圧縮

プロデューサー設定の場合は、Gzip などの compression.type を指定します。これは、プロ

デューサーによって生成されたデータのバッチに適用されます。ブローカー設定の compression.type=producer を使用すると、ブローカーは使用されるプロデューサーを圧縮します。プロデューサーとトピックの圧縮が一致しない場合は常に、ブローカーはバッチをログに追加する前に再度圧縮する必要があります。これはブローカーのパフォーマンスに影響を与えます。

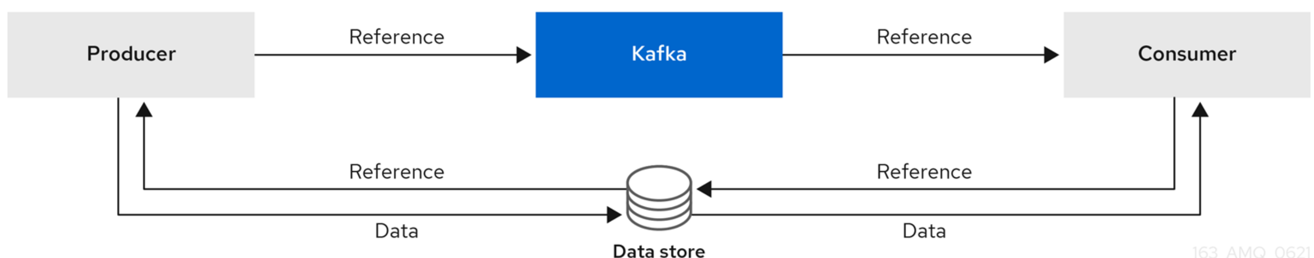
圧縮はまた、プロデューサーに追加の処理オーバーヘッドを追加し、コンシューマーに解凍オーバーヘッドを追加しますが、バッチにより多くのデータが含まれるため、メッセージデータが適切に圧縮される場合、スループットに役立つことがよくあります。

プロデューサー側の圧縮とバッチサイズの微調整を組み合わせて、最適なスループットを促進します。メトリクスを使用すると、必要な平均バッチサイズの測定に役立ちます。

参照ベースのメッセージング

参照ベースのメッセージングは、メッセージの大きさがわからない場合のデータ複製に役立ちます。この設定が機能するには、外部データストアは高速で永続性があり、高可用性である必要があります。データはデータストアに書き込まれ、データへの参照が返されます。プロデューサーは、Kafka への参照が含まれるメッセージを送信します。コンシューマーはメッセージから参照を取得し、これを使用してデータストアからデータを取得します。

図13.4 参照ベースのメッセージングフロー



メッセージを渡すにはより多くの通信が必要なため、エンドツーエンドのレイテンシーが増加します。このアプローチのもう 1 つの重大な欠点は、Kafka メッセージがクリーンアップされたときに、外部システムのデータが自動的にクリーンアップされないことです。ハイブリッドアプローチは、大きなメッセージのみをデータストアに送信し、標準サイズのメッセージを直接処理することです。

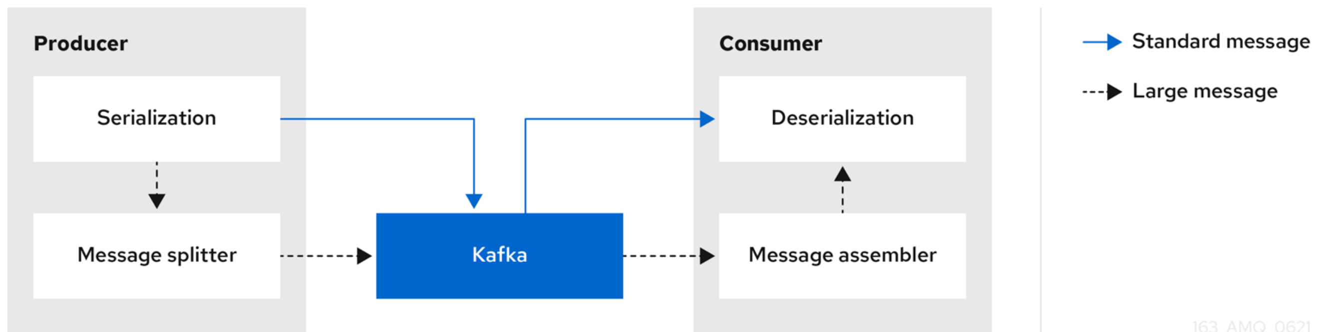
インラインメッセージング

インラインメッセージングは複雑ですが、参照ベースのメッセージングのように外部システムに依存するオーバーヘッドはありません。

メッセージが大きすぎる場合、生成するクライアントアプリケーションは、データをシリアライズしてからチャンクにする必要があります。その後、プロデューサーは Kafka ByteArraySerializer を使用し、送信前に各チャンクを再度シリアライズするのと同様のものを使用します。コンシューマーはメッセージを追跡し、完全なメッセージが得られるまでチャンクをバッファリングします。消費側のク

ライアントアプリケーションは、デシリアライズの前にアセンブルされたチャンクを受け取ります。完全なメッセージは、チャンクになったメッセージの各セットの最初または最後のチャンクのオフセットに従って、消費する残りのアプリケーションに配信されます。リバランス中の重複を避けるために、完全なメッセージの正常な配信がオフセットメタデータと照合されます。

図13.5 インラインメッセージングフロー



インラインメッセージングは、特に一連の大きなメッセージを並行して処理する場合に必要なバッファリングのために、コンシューマー側でパフォーマンスのオーバーヘッドが発生します。大きなメッセージのチャンクはインターリーブされる可能性があるため、バッファー内の別の大きなメッセージのチャンクが不完全な場合、メッセージのすべてのチャンクが消費されたときにコミットできるとは限りません。このため、バッファリングは通常、メッセージチャンクを永続化するか、コミットロジックを実装することでサポートされます。

より大きなメッセージを処理するための設定

より大きなメッセージを回避できない場合、およびメッセージフローの任意の時点でブロックを回避するために、メッセージ制限を増やすことができます。これを行うには、トピックレベルで **message.max.bytes** を設定し、個別のトピックの最大レコードバッチサイズを設定します。ブローカーレベルで **message.max.bytes** を設定すると、すべてのトピックに大きなメッセージが許可されます。

ブローカーは、**message.max.bytes** で設定された制限よりも大きなメッセージを拒否します。プロデューサー (**max.request.size**) およびコンシューマー (**message.max.bytes**) のバッファーサイズは、より大きなメッセージに対応できなければなりません。

13.8.1.10. メッセージデータのログフラッシュの制御

ログフラッシュプロパティーは、キャッシュされたメッセージデータのディスクへの定期的な書き込みを制御します。スケジューラーは、ログキャッシュのチェック頻度をミリ秒単位で指定します。

```
# ...
log.flush.scheduler.interval.ms=2000
# ...
```

メッセージがメモリに保持される最大時間と、ディスクに書き込む前にログに記録されるメッセージの最大数に基づいて、フラッシュの頻度を制御できます。

```
# ...
log.flush.interval.ms=50000
log.flush.interval.messages=100000
# ...
```

フラッシュ間の待機時間には、チェックを行う時間と、フラッシュが実行される前の指定された間隔が含まれます。フラッシュの頻度を増やすと、スループットに影響を及ぼす可能性があります。

一般に、明示的なフラッシュしきい値を設定せず、オペレーティングシステムにデフォルト設定を使用してバックグラウンドフラッシュを実行させることをお勧めします。パーティションレプリケーションは、障害が発生したブローカーが同期レプリカから回復できるため、単一のディスクへの書き込みよりも優れたデータ耐久性を提供します。

アプリケーションフラッシュ管理を使用している場合、より高速なディスクを使用していると、フラッシュしきい値を低く設定するのが適切であることがあります。

13.8.1.11. 可用性のためのパーティションリバランス

フォールトトレランスのために、パーティションはブローカー間全体で複製できます。指定したパーティションでは、1 つのブローカーがリーダーに選出され、すべての生成リクエストを処理します (ログへの書き込み)。他のブローカーのパーティションフォロワーは、リーダーに障害が発生した場合のデータの信頼性のために、パーティションリーダーのパーティションデータを複製します。

通常、フォロワーはクライアントを提供しませんが、rack 設定は、Kafka クラスターが複数のデータセンターにまたがる場合に最も近いレプリカからメッセージを消費できます。フォロワーは、パーティションリーダーからのメッセージを複製して、リーダーに障害が発生した場合に回復できるようにするためにのみ動作します。リカバリーには、同期のフォロワーが必要です。フォロワーは、フェッチ要求をリーダーに送信することで同期を維持します。リーダーは、メッセージを順番にフォロワーに返します。フォロワーは、リーダーで最後にコミットされたメッセージに追いついた場合に、同期していると見なされます。リーダーは、フォロワーによってリクエストされた最後のオフセットを確認してこれをチェックします。クリーンでないリーダーの選出 (unclean leader election) が許可されない限り、非同期のフォロワーは通常、現在のリーダーが失敗した場合にリーダーとしての資格がありません。

フォロワーが同期していないと見なされるまでのラグタイムを調整できます。

```
# ...
replica.lag.time.max.ms=30000
# ...
```

ラグタイムは、メッセージをすべての同期レプリカにレプリケートする時間と、プロデューサーが確認応答を待機する必要がある時間に上限を設定します。フォロワーがフェッチリクエストの作成に失敗し、指定されたラグタイム内に最新のメッセージに追いつくと、同期レプリカから削除されます失敗したレプリカをより早く検出するためにラグタイムを短縮することができますが、そうすることで、不必要に同期しなくなるフォロワーの数が増えます。適切なラグタイムの値は、ネットワークレイテンシーとブローカーのディスク帯域幅の両方に依存します。

リーダーパーティションが利用できなくなると、同期レプリカの 1 つが新しいリーダーとして選択されます。パーティションにあるレプリカの一覧の最初のブローカーは、優先リーダーと呼ばれます。デフォルトでは、Kafka はリーダー分散の定期的なチェックに基づいて自動パーティションリーダーリバランスに対して有効になっています。つまり、Kafka は優先リーダーが現在のリーダーであるかどうかを確認します。リバランスにより、リーダーがブローカー間で均等に分散され、ブローカーがオーバーロードされないようにします。

AMQ Streams の Cruise Control を使用すると、クラスター全体で負荷を均等に分散するブローカーへのレプリカの割り当てを把握できます。その計算では、リーダーとフォロワーで発生するさまざまな負荷が考慮されています。リーダーが失敗すると、残りのブローカーが追加のパーティションをリードするという余分な作業が発生するため、Kafka クラスターのバランスに影響を与えます。

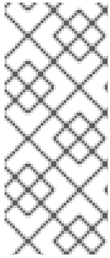
Cruise Control によって検出される割り当てがバランスを取るには、パーティションのリーダーが優先リーダーである必要があります。Kafkaは、優先リーダーが使用されていることを自動的に確認し (可能な場合)、必要に応じて現在のリーダーを変更します。これにより、クラスターは CruiseControl によって検出されたバランスの取れた状態に保たれます。

リバランスチェックの頻度 (秒単位) と、リバランスがトリガーされる前にブローカーに許可される非バランスの最大率を制御できます。

```
#...
auto.leader.rebalance.enable=true
leader.imbalance.check.interval.seconds=300
leader.imbalance.per.broker.percentage=10
#...
```

ブローカーのリーダーの非バランスの割合は、ブローカーが現在のリーダーであるパーティションの現在の数と、ブローカーが優先リーダーであるパーティションの数との比率です。優先リーダーが同期状態にあることを前提として、割合をゼロにして、優先リーダーが常に選択されるようにすることができます。

リバランスのチェックでさらに制御が必要な場合は、自動リバランスを無効にすることができます。次に、kafka-leader-election.sh コマンドラインツールを使用してリバランスをトリガーするタイミングを選択できます。



注記

AMQ Streams で提供される Grafana ダッシュボードでは、複製の数が最低数未満のパーティションや、アクティブなリーダーを持たないパーティションのメトリクスが表示されます。

13.8.1.12. クリーンでないリーダーの選出 (unclean leader election)

同期レプリカへのリーダーの選出は、データの損失がないことを保証するため、クリーンであると見なされます。これは、デフォルトで行われます。しかし、リーダーに選出する同期レプリカがない場合はどうなるのでしょうか。おそらく、ISR (同期レプリカ) には、リーダーのディスクが停止したときにのみリーダーが含まれていました。同期レプリカの最小数が設定されておらず、ハードドライブに取り返しのつかない障害が発生したときにパーティションリーダーと同期しているフォロワーがない場合、データはすでに失われています。それだけでなく、同期しているフォロワーがいないため、新しいリーダーを選出することはできません。

Kafka がリーダーの失敗を処理する方法を設定できます。

```
# ...
unclean.leader.election.enable=false
# ...
```

クリーンでないリーダーの選択はデフォルトでは無効になっており、同期されていないレプリカはリーダーになれません。クリーンリーダーの選出では、古いリーダーが失われたときに ISR に他のブローカーがない場合に Kafkaはそのリーダーがオンラインに戻るまで待機してから、メッセージの読み書きが行われます。クリーンでないリーダーの選出は、同期していないレプリカがリーダーになる可能性があることを意味しますが、メッセージが失われるリスクがあります。どちらを選択するかは、要件が可用性と耐久性のどちらを優先するかによって異なります。

トピックレベルで特定のトピックのデフォルト設定を上書きできます。データ損失のリスクを許容できない場合は、デフォルト設定のままにします。

13.8.1.13. 不要なコンシューマーグループリバランスの回避

新しいコンシューマーグループに参加するコンシューマーの場合、ブローカーへの不要なリバランスを回避するために遅延を追加できます。

```
# ...
group.initial.rebalance.delay.ms=3000
# ...
```

この遅延は、コーディネーターがメンバーの参加を待つ期間です。遅延が長いほど、すべてのメン

バーが時間内に参加し、リバランスを回避できる可能性が高くなります。ただし、遅延により、期間が終了するまでグループは消費できなくなります。

その他のリソース

- Kafka Static Quota プラグインを使用したブローカーへの制限の設定

13.8.2. Kafka プロデューサー設定のチューニング

特定のユースケースに合わせて調整されたオプションのプロパティーとともに、基本的なプロデューサー設定を使用します。

設定を調整してスループットを最大化すると、レイテンシーが増加する可能性があり、その逆も同様です。必要なバランスを取得するために、プロデューサー設定を実験して調整する必要があります。

13.8.2.1. 基本のプロデューサー設定

接続およびシリアライザープロパティーはすべてのプロデューサーに必要です。通常、追跡用のクライアント ID を追加し、プロデューサーで圧縮してリクエストのバッチサイズを減らすことが推奨されます。

基本的なプロデューサー設定には以下が含まれます。

- パーティション内のメッセージの順序は保証されません。
- ブローカーに到達するメッセージの完了通知は持続性を保証しません。

基本的なプロデューサー設定プロパティー

```
# ...
bootstrap.servers=localhost:9092 (1)
key.serializer=org.apache.kafka.common.serialization.StringSerializer (2)
value.serializer=org.apache.kafka.common.serialization.StringSerializer (3)
client.id=my-client (4)
compression.type=gzip (5)
# ...
```

1
(必須) Kafka ブローカーの host:port ブートストラップサーバーアドレスを使用して Kafka クラスターに接続するようプロデューサーを指示します。プロデューサーはアドレスを使用して、クラスター内のすべてのブローカーを検出し、接続します。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して 2 つまたは 3 つのアドレスを指定しますが、クラスター内のすべてのブローカーのリストを提供する必要はありません。

2
(必須) メッセージがブローカーに送信される前に、各メッセージの鍵をバイトに変換するシリアライザー。

3
(必須) メッセージがブローカーに送信される前に、各メッセージの値をバイトに変換するシリアライザー。

4
(任意) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリクスで使用されます。

5
(任意) メッセージを圧縮するコーデック。これは、送信され、圧縮された形式で格納された後、コンシューマーへの到達時に圧縮解除される可能性があります。圧縮はスループットを改善し、ストレージの負荷を減らすのに役立ちますが、圧縮や圧縮解除のコストが異常に高い低レイテンシーのアプリケーションには不適切である場合があります。

13.8.2.2. データの持続性

メッセージ配信の完了通知を使用して、データの持続性を適用し、メッセージが失われる可能性を最小限に抑えることができます。

```
# ...
acks=all 1
# ...
```

1
acks=all を指定すると、パーティションリーダーは、メッセージリクエストが正常に受信されたことを確認する前に、一定数のフォロワーにメッセージを複製するように強制します。追加のチェックにより、acks=all はプロデューサーがメッセージを送信して確認応答を受信する間のレ

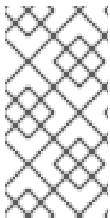
イテンシーを長くします。

完了通知がプロデューサーに送信される前にメッセージをログに追加する必要のあるブローカーの数は、トピックの **min.insync.replicas** 設定によって決定されます。最初に、トピックレプリケーション係数を 3 にし、他のブローカーの **In-Sync** レプリカを 2 にするのが一般的です。この設定では、単一のブローカーが利用できない場合でもプロデューサーは影響を受けません。2 番目のブローカーが利用できなくなると、プロデューサーは完了通知を受信せず、それ以上のメッセージを生成できなくなります。

acks=allをサポートするトピック設定

```
# ...
min.insync.replicas=2 (1)
# ...
```

(1) Sync レプリカでは2を使用します。デフォルトは 1 です。



注記

システムに障害が発生すると、バッファーの未送信データが失われる可能性があります。

13.8.2.3. 順序付き配信

メッセージは 1 度だけ配信されるため、べき等プロデューサーは重複を回避します。障害発生時でも配信の順序が維持されるように、ID とシーケンス番号がメッセージに割り当てられます。データの一貫性を保つために **acks=all** を使用している場合は、順序付けられた配信に冪等性を有効にすることが妥当です。

べき等を使った順序付き配信

```
# ...
enable.idempotence=true (1)
```

```
max.in.flight.requests.per.connection=5 (2)
acks=all (3)
retries=2147483647 (4)
# ...
```

1. べき等プロデューサーを有効にするには true に設定します。

2. べき等配信では、インフライトリクエストの数が 1 を越えることがありますがメッセージの順序は維持されます。デフォルトのインフライトリクエストの数は 5 です。

3. acks を all に設定します。

4. 失敗したメッセージリクエストを再送信する試行回数を設定します。

パフォーマンスコストが原因で acks=all と idempotency を使用していない場合には、インフライト（承認されていない）リクエストの数を 1 に設定して、順序を保持します。そうしないと、Message-A が失敗し、Message-B がブローカーに書き込まれた後にのみ成功する可能性があります。

べき等を使用しない順序付け配信

```
# ...
enable.idempotence=false (1)
max.in.flight.requests.per.connection=1 (2)
retries=2147483647
# ...
```

1.

false に設定すると、べき等プロデューサーを無効にします。

2

インフライトリクエストの数のみを 1 に設定します。

13.8.2.4. 信頼性の保証

べき等は、1 つのパーティションへの書き込みを 1 回だけ行う場合に便利です。トランザクションをべき等と使用すると、複数のパーティション全体で 1 度だけ書き込みを行うことができます。

トランザクションは、同じトランザクション ID を使用するメッセージが 1 度作成され、すべてがそれぞれのログに書き込まれるか、何も書き込まれないかのどちらかになることを保証します。

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=UNIQUE-ID (1)
transaction.timeout.ms=900000 (2)
# ...
```

1

一意のトランザクション ID を指定します。

2

タイムアウトエラーが返されるまでのトランザクションの最大許容時間 (ミリ秒単位) を設定します。デフォルトは900000または 15 分です。

トランザクション保証を維持するには、transactional.id の選択が重要です。トランザクション ID は、一意なトピックパーティションセットに使用する必要があります。たとえば、トピックパーティション名からトランザクション ID への外部マッピングを使用したり、競合を回避する関数を使用してトピックパーティション名からトランザクション IDを算出したりすると、これを実現できます。

13.8.2.5. スループットおよびレイテンシーの最適化

通常、システムの要件は、指定のレイテンシー内であるメッセージの割合に対して、特定のスループットのターゲットを達成することです。たとえば、95 % のメッセージが 2 秒以内に完了確認され

る、1 秒あたり 500,000 個のメッセージをターゲットとします。

プロデューサーのメッセージングセマンティック (メッセージの順序付けと持続性) は、アプリケーションの要件によって定義される可能性があります。たとえば、アプリケーションが提供する重要なプロパティーや保証を壊さずに acks=0 または acks=1 を使用するオプションはありません。

ブローカーの再起動は、パーセンタイルの高いの統計に大きく影響します。たとえば、長期間では、99% のレイテンシーはブローカーの再起動に関する動作によるものです。これは、ベンチマークを設計したり、本番環境のパフォーマンスで得られた数字を使ってベンチマークを行い、そのパフォーマンスの数字を比較したりする場合に検討する価値があります。

目的に応じて、Kafka はスループットとレイテンシーのプロデューサーパフォーマンスを調整するために多くの設定パラメーターと設定方法を提供します。

メッセージのバッチ処理（linger.ms および batch.size）

メッセージのバッチ処理では、同じブローカー宛のメッセージをより多く送信するために、メッセージの送信を遅らせ、単一の生成リクエストでバッチ処理できるようにします。バッチ処理では、スループットを増やすためにレイテンシーを長くして妥協します。時間ベースのバッチ処理は linger.ms を使用して設定され、サイズベースのバッチ処理は batch.size を使用して設定されます。

Compression(compression.type)

メッセージ圧縮処理により、プロデューサー (メッセージの圧縮に費やされた CPU 時間) のレイテンシーが追加されますが、リクエスト (および場合によってはディスクの書き込み) を小さくするため、スループットが増加します。圧縮に価値があるかどうか、および使用に最適な圧縮は、送信されるメッセージによって異なります。圧縮は KafkaProducer.send() を呼び出すスレッドで発生するため、アプリケーションでこのメソッドのレイテンシーが重要となる場合は、より多くのスレッドの使用を検討する必要があります。

パイプライン処理(max.in.flight.requests.per.connection)

パイプライン処理は、以前のリクエストへの応答を受け取る前により多くのリクエストを送信します。通常、パイプライン処理を増やすと、バッチ処理の悪化などの別の問題がスループットに悪影響を与え始めるしきい値まではスループットが増加します。

レイテンシーの短縮

アプリケーションが KafkaProducer.send() を呼び出す場合、メッセージは以下のようになります。

- インターセプターによる処理。

- シリアライズ。
- パーティションへの割り当て。
- 圧縮処理。
- パーティションごとのキューでメッセージのバッチに追加。

ここでの send() メソッドが返されます。そのため、time send() は以下によって決定されます。

- インターセプター、シリアライザー、およびパーティションヤーで費やされた時間。
- 使用される圧縮アルゴリズム。
- 圧縮に使用するバッファーの待機に費やされた時間。

バッチは、以下のいずれかが行われるまでキューに残ります。

- バッチが満杯になる（batch.sizeによる）。
- linger.ms によって導入される遅延が渡される。
- 送信者は他のパーティションのメッセージバッチを同じブローカーに送信しようとし、このバッチの追加も可能。
- プロデューサーがフラッシュまたは閉じられる。

バッチ処理とバッファーの設定を参照して、レイテンシーをブロックする send() の影響を軽減します。

```
# ...
linger.ms=100 (1)
batch.size=16384 (2)
buffer.memory=33554432 (3)
# ...
```

(1) lingerプロパティは、より大きなメッセージのバッチが蓄積され、リクエストで送信されるように、ミリ秒単位の遅延を追加します。デフォルトは0'.です。

(2) batch.sizeの最大値をバイト単位で指定した場合、最大値に達したとき、またはメッセージがlinger.msを超えてキューに入っていたとき（いずれか早いほう）にリクエストが送信されます。遅延を追加すると、メッセージをバッチサイズまで累積できます。

(3) バッファーサイズは、少なくともバッチサイズと同じ大きさである必要があり、バッファー、圧縮、およびインフライトリクエストに対応できる必要があります

スループットの増加

メッセージの配信および送信リクエストの完了までの最大待機時間を調整して、メッセージリクエストのスループットを向上します。

また、カスタムパーティションを作成してデフォルトを置き換えることで、メッセージを指定のパーティションに転送することもできます。

```
# ...
delivery.timeout.ms=120000 (1)
partitioner.class=my-custom-partitioner (2)

# ...
```

(1) 送信リクエストの完了まで待機する最大時間 (ミリ秒単位)。この値を MAX_LONG に設定すると、Kafka に無限の再試行を委譲できます。デフォルトは 120000 または 2 分です。

(2) カスタムパーティショナーのクラス名を指定します。

13.8.3. Kafka コンシューマー設定の調整

特定のユースケースに合わせて調整されたオプションのプロパティーとともに、基本的なコンシューマー設定を使用します。

コンシューマーを調整する場合、最も重要なことは、取得するデータ量に効率的に対処できるようにすることです。プロデューサーのチューニングと同様に、コンシューマーが想定どおりに動作するまで、段階的に変更を加える必要があります。

13.8.3.1. 基本的なコンシューマー設定

接続およびデシリアライザープロパティーはすべてのコンシューマーに必要です。通常、追跡用にクライアント ID を追加することが推奨されます。

コンシューマー設定では、後続の設定に関係なく、以下を行います。

- メッセージをスキップまたは再読み取りするようオフセットを変更しない限り、コンシューマーはメッセージを指定のオフセットから取得し、順番に消費します。
- オフセットはクラスターの別のブローカーに送信される可能性があるため、オフセットを Kafka にコミットした場合でも、ブローカーはコンシューマーが応答を処理したかどうかを認識しません。

基本的なコンシューマー設定プロパティー

```
# ...
bootstrap.servers=localhost:9092 (1)
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer (2)
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer (3)
client.id=my-client (4)
group.id=my-group-id (5)
# ...
```

(1)

2

(必須) Kafka ブローカーから取得されたバイトをメッセージキーに変換するデシリアライザー。

3

(必須) Kafka ブローカーから取得されたバイトをメッセージ値に変換するデシリアライザー。

4

(任意) クライアントの論理名。リクエストのソースを特定するためにログおよびメトリクスで使用されます。ID は、時間クォータの処理に基づいてコンシューマーにスロットリングを適用するために使用することもできます。

5

(条件) コンシューマーがコンシューマーグループに参加するには、グループ ID が 必要 です。

13.8.3.2. コンシューマーグループを使用したデータ消費のスケーリング

コンシューマーグループは、特定のトピックから 1 つまたは複数のプロデューサーによって生成される、典型的な大量のデータストリームを共有します。コンシューマーは group.id プロパティーを使用してグループ化されるため、メッセージをメンバー全体に分散できます。グループ内のコンシューマーの 1 つがリーダーを選択し、パーティションをグループのコンシューマーにどのように割り当てるかを決定します。各パーティションは 1 つのコンシューマーにのみ割り当てることができます。

コンシューマーの数がパーティション数だけない場合には、同じ group.id を持つコンシューマーインスタンスを追加して、データ消費をスケーリングできます。コンシューマーをグループに追加して、パーティションの数より多くしても、スループットは改善されませんが、コンシューマーが機能しなくなったときに予備のコンシューマーを使用できます。より少ないコンシューマーでスループットの目標を達成できれば、リソースを節約できます。

同じコンシューマーグループのコンシューマーは、オフセットコミットとハートビートを同じブローカーに送信します。グループのコンシューマーの数が多いほど、ブローカーのリクエスト負荷が高くなります。

```
# ...
group.id=my-group-id 1
# ...
```

1

13.8.3.3. メッセージの順序の保証

Kafka ブローカーは、トピック、パーティション、およびオフセット位置のリストからメッセージを送信するようブローカーに要求するコンシューマーからフェッチリクエストを受け取ります。

コンシューマーは、ブローカーにコミットされたのと同じ順序でメッセージを単一のパーティションで監視します。つまり、Kafka は単一パーティションのメッセージ のみ 順序付けを保証します。逆に、コンシューマーが複数のパーティションからメッセージを消費している場合、コンシューマーによって監視される異なるパーティションのメッセージの順序は、必ずしも送信順序を反映しません。

1 つのトピックからメッセージを厳格に順序付ける場合は、コンシューマーごとに 1 つのパーティションを使用します。

13.8.3.4. スループットおよびレイテンシーの最適化

クライアントアプリケーションが KafkaConsumer.poll() を呼び出すときに返されるメッセージの数を制御します。

fetch.max.wait.ms および fetch.min.bytes プロパティーを使用して、Kafka ブローカーからコンシューマーによって取得される最小データ量を増やします。時間ベースのバッチ処理は fetch.max.wait.ms を使用して設定され、サイズベースのバッチ処理は fetch.min.bytes を使用して設定されます。

コンシューマーまたはブローカーの CPU 使用率が高い場合、コンシューマーからのリクエストが多すぎる可能性があります。fetch.max.wait.ms プロパティーおよび fetch.min.bytes プロパティーを調整して、より大きなバッチで要求とメッセージが配信されるようにすることができます。より高い値に調整することでスループットが改善されますが、レイテンシーのコストが発生します。生成されるデータ量が少ない場合、より高い値に調整することもできます。

たとえば、fetch.max.wait.ms を 500ms に設定し、fetch.min.bytes を 16384 バイトに設定した場合、Kafka がコンシューマーからフェッチリクエストを受信すると、いずれかのしきい値に最初に到達した時点で応答されます。

逆に、fetch.max.wait.ms プロパティーおよび fetch.min.bytes プロパティーを調整して、エンドツーエンドのレイテンシーを改善できます。

```
# ...
```

```
fetch.max.wait.ms=500 (1)
fetch.min.bytes=16384 (2)
# ...
```

(1) ブローカーがフェッチリクエストを完了するまで待機する最大時間 (ミリ秒単位)。デフォルトは 500 ミリ秒です。

(2) 最小バッチサイズ（バイト単位）が使用された場合、最低限到達時にリクエストが送信されます。または、メッセージが fetch.max.wait.ms よりも長くキューに入れられると、リクエストが送信されます。遅延を追加すると、メッセージをバッチサイズまで累積できます。

フェッチリクエストサイズの増加によるレイテンシーの短縮

fetch.max.bytes プロパティーおよび max.partition.fetch.bytes プロパティーを使用して、Kafka ブローカーからコンシューマーによって取得されるデータの最大量を増やします。

fetch.max.bytes プロパティーは、一度にブローカーから取得されるデータ量の上限をバイト単位で設定します。

max.partition.fetch.bytes は、各パーティションで返されるデータ量の上限をバイト単位で設定します。これは、max.message.bytes のブローカーまたはトピック設定に設定されたバイト数よりも大きくする必要があります。

クライアントが消費できるメモリーの最大量は、以下のように概算されます。

```
NUMBER-OF-BROKERS * fetch.max.bytes and NUMBER-OF-PARTITIONS *
max.partition.fetch.bytes
```

メモリー使用量がこれに対応できる場合は、これら 2 つのプロパティーの値を増やすことができます。各リクエストでより多くのデータを許可すると、フェッチリクエストが少なくなるため、レイテンシーが向上されます。

```
# ...
fetch.max.bytes=52428800 (1)
max.partition.fetch.bytes=1048576 (2)
# ...
```

(1)

2

各パーティションに対して返されるデータの最大量 (バイト単位)。

13.8.3.5. オフセットをコミットする際のデータ損失または重複の回避

Kafka の 自動コミットメカニズム により、コンシューマーはメッセージのオフセットを自動的にコミットできます。有効にすると、コンシューマーはブローカーをポーリングして受信したオフセットを 5000ms 間隔でコミットします。

自動コミットのメカニズムは便利ですが、データ損失と重複のリスクが発生します。コンシューマーが多くのメッセージを取得および変換し、自動コミットの実行時にコンシューマーバッファーに処理されたメッセージがある状態でシステムがクラッシュすると、そのデータは失われます。メッセージの処理後、自動コミットの実行前にシステムがクラッシュした場合、リバランス後に別のコンシューマーインスタンスでデータが複製されます。

ブローカーへの次のポーリングの前またはコンシューマーが閉じられる前に、すべてのメッセージが処理された場合は、自動コミットによるデータの損失を回避できます。

データの損失や重複の可能性を最小限に抑えるには、enable.auto.commit を false に設定し、クライアントアプリケーションを開発して、オフセットのコミットをより詳細に制御できるようにします。または、auto.commit.interval.ms を使用してコミットの間隔を減らすことができます。

```
# ...
enable.auto.commit=false 1
# ...
```

1

enable.auto.commit を false に設定すると、すべての 処理が実行され、メッセージが消費された後にオフセットをコミットできます。たとえば、Kafka commitSync および commitAsync コミット API を呼び出すようにアプリケーションを設定できます。

commitSync API は、ポーリングから返されるメッセージバッチのオフセットをコミットします。バッチのメッセージすべての処理が完了したら API を呼び出します。commitSync API を使用する場合、アプリケーションはバッチの最後のオフセットがコミットされるまで新しいメッセージをポーリングしません。これがスループットに悪影響する場合は、コミットする頻度が低いか、commitAsync API を使用できます。commitAsync API はブローカーがコミットリクエストに応答するまで待機しませんが、リバランス時にさらに重複が発生するリスクがあります。一般的な方法として、両方のコミッ

ト API をアプリケーションで組み合わせ、コンシューマーをシャットダウンまたはリバランスの直前に commitSync API を使用し、最終コミットが正常に実行されるようにします。

13.8.3.5.1. トランザクションメッセージの制御

プロデューサー側でトランザクション ID を使用し、べき等性(enable.idempotence=true)を有効にすることを検討してください。これにより、1 回限りの配信を保証します。コンシューマー側で、isolation.level プロパティーを使用して、コンシューマーによってトランザクションメッセージが読み取られる方法を制御できます。

isolation.level プロパティーには、有効な 2 つの値があります。

- read_committed
- read_uncommitted (デフォルト)

read_committed を使用して、コミットされたトランザクションメッセージのみがコンシューマーによって読み取られるようにします。ただし、これによりトランザクションの結果を記録するトランザクションマーカー (committed または aborted) がブローカーによって書き込まれるまで、コンシューマーはメッセージを返すことができないため、エンドツーエンドのレイテンシーが長くなります。

```
# ...
enable.auto.commit=false
isolation.level=read_committed (1)
# ...
```

(1)

13.8.3.6. データ損失を回避するための障害からの復旧

session.timeout.ms および heartbeat.interval.ms プロパティーを使用して、コンシューマーグループ内のコンシューマー障害をチェックし、復旧するのにかかった時間を設定します。

session.timeout.msプロパティは、コンシューマーグループ内のコンシューマーがブローカーとのコンタクトを絶った場合に、非アクティブとみなされ、グループ内のアクティブなコンシューマー間でリバランスが発生するまでの最大時間をミリ秒単位で指定します。グループのリバランス時に、パーティションはグループのメンバーに再割り当てされます。

heartbeat.interval.ms プロパティーは、コンシューマーがアクティブで接続されていることを示す、コンシューマーグループコーディネーターへの ハートビート チェックの間隔をミリ秒単位で指定します。通常、ハートビートの間隔はセッションタイムアウトの間隔の 3 分の 2 にする必要があります。

session.timeout.ms プロパティーを低く設定すると、失敗したコンシューマーが先に検出され、リバランスがより迅速に実行されます。ただし、タイムアウトの値を低くしすぎて、ブローカーがハートビートを時間内に受信できず、不必要なリバランスがトリガーされることがないように気を付けてください。

ハートビートの間隔が短くなると、誤ってリバランスを行う可能性が低くなりますが、ハートビートを頻繁に行うとブローカーリソースのオーバーヘッドが増えます。

13.8.3.7. オフセットポリシーの管理

auto.offset.reset プロパティーを使用して、オフセットがコミットされていない場合にコンシューマーの動作を制御するか、コミットされたオフセットが有効でなくなったりします。

コンシューマーアプリケーションを初めてデプロイし、既存のトピックからメッセージを読み取る場合について考えてみましょう。これは group.id が初めて使用されるため、__consumer_offsets トピックには、このアプリケーションのオフセット情報は含まれません。新しいアプリケーションは、ログの始めからすべての既存メッセージの処理を開始するか、新しいメッセージのみ処理を開始できます。デフォルトのリセット値は、パーティションの最後に開始する latest で、一部のメッセージは見逃されることを意味します。データの損失は避けたいが、処理量を増やしたい場合は、auto.offset.resetをearliestに設定して、パーティションの先頭から開始します。

また、ブローカーに設定されたオフセットの保持期間（offsets.retention.minutes）が終了したときにメッセージが失われるのを防ぐために、earliestオプションの使用も検討してください。コンシューマーグループまたはスタンドアロンコンシューマーが非アクティブで、保持期間中にオフセットをコミットしない場合、以前にコミットされたオフセットは __consumer_offsets から削除されます。

```
# ...
heartbeat.interval.ms=3000 1
session.timeout.ms=10000 2
auto.offset.reset=earliest 3
# ...
```

1

予想されるリバランスに応じて、ハートビートの間隔を短くして調整します。

2

3

パーティションの最初に戻り、オフセットがコミットされなかった場合にデータの損失を回避するために、最も早い値に設定します。

1 つのフェッチリクエストで返されるデータ量が大きい場合、コンシューマーが処理する前にタイムアウトが発生することがあります。この場合、**max.partition.fetch.bytes** を減らしたり、**session.timeout.ms** を増やすこともできます。

13.8.3.8. リバランスの影響を最小限にする

グループのアクティブなコンシューマー間で行うパーティションのリバランスは、以下にかかる時間です。

- コンシューマーによるオフセットのコミット
- 作成される新しいコンシューマーグループ
- グループリーダーによるグループメンバーへのパーティションの割り当て。
- 割り当てを受け取り、取得を開始するグループのコンシューマー

明らかに、このプロセスは特にコンシューマーグループクラスターのローリング再起動時に繰り返し発生するサービスのダウンタイムを増やします。

このような場合、静的メンバーシップ の概念を使用してリバランスの数を減らすことができます。リバランスによって、コンシューマーグループメンバー全体でトピックパーティションが割り当てられます。静的メンバーシップは永続性を使用し、セッションタイムアウト後の再起動時にコンシューマーインスタンスが認識されるようにします。

コンシューマーグループコーディネーターは、**group.instance.id** プロパティーを使用して指定される一意の ID を使用して新しいコンシューマーインスタンスを特定できます。再起動時には、コンシューマーには新しいメンバー ID が割り当てられますが、静的メンバーとして、同じインスタンス ID を使用し、同じトピックパーティションの割り当てが行われます。

コンシューマーアプリケーションが少なくとも max.poll.interval.ms ミリ秒毎にポーリングへの呼び出しを行わない場合、コンシューマーが失敗したと見なされ、リバランスが発生します。アプリケーションがポーリングから返されたすべてのレコードを時間内に処理できない場合は、max.poll.interval.ms プロパティーを使用してコンシューマーから新しいメッセージのポーリングの間隔をミリ秒単位で指定して、リバランスを回避することができます。または、max.poll.records プロパティーを使用して、コンシューマーバッファーから返されるレコードの数の上限を設定できます。これにより、アプリケーションは max.poll.interval.ms の制限内のより少ないレコードを処理できます。

```
# ...
group.instance.id=UNIQUE-ID (1)
max.poll.interval.ms=300000 (2)
max.poll.records=500 (3)
# ...
```

(1) 一意のインスタンス ID により、新しいコンシューマーインスタンスに同じトピックパーティションが割り当てられます。

(2) コンシューマーがメッセージの処理を継続していることを確認する間隔を設定します。

(3) コンシューマーから返される処理済のレコードの数を設定します。

13.9. AMQ STREAMS のアンインストール

この手順では、AMQ Streams をアンインストールし、デプロイメントに関連するリソースを削除する方法を説明します。

前提条件

この手順を実行するには、デプロイメント用に特別に作成され、AMQ Streams リソースから参照されるリソースを特定します。

このようなリソースには以下があります。

- シークレット (カスタム CA および証明書、Kafka Connect Secrets、その他の Kafka シークレット)

- ロギング ConfigMap (external タイプ)

Kafka、KafkaConnect、KafkaMirrorMaker、KafkaBridgeのいずれかの設定で参照されるリソースです。

手順

1. Cluster Operator Deployment、関連する CustomResourceDefinitions、および RBAC リソースを削除します。

```
oc delete -f install/cluster-operator
```

警告

CustomResourceDefinitions を削除すると、対応するカスタムリソース（Kafka、KafkaConnect、KafkaMirrorMaker、または KafkaBridge）、およびそれらに依存するリソース（Deployments、StatefulSets、およびその他の依存リソース）のガベージコレクションが実行されます。

2. 前提条件で特定したリソースを削除します。

13.10. よくある質問

13.10.1. Cluster Operator に関する質問

13.10.1.1. AMQ Streams のインストールに、クラスター管理者の権限が必要なのはなぜですか?

AMQ Streams をインストールするには、以下のクラスタースコープのリソースの作成が可能でなければなりません。

- Kafka や KafkaConnectなどの AMQ Streams 固有のリソースについて OpenShift に指示するカスタムリソース定義 (CRD)

- ClusterRoles および ClusterRoleBindings

特定の OpenShift namespace にスコープ指定されていないクラスタースコープのリソースをインストールするには、通常は クラスター管理者 の権限が必要です。

クラスター管理者として、(/install/ ディレクトリーに) インストールされているすべてのリソースを検査し、ClusterRole が不要な権限を付与しないようにします。

インストール後に、Cluster Operator は通常の Deployment として実行されるため、Deployment へのアクセス権限を持つ OpenShift の標準ユーザー (管理者以外) であれば誰でもこれを設定できます。クラスター管理者は、Kafka カスタムリソースの管理に必要な権限を標準ユーザーに付与できます。

以下も参照してください。

- Cluster Operator が ClusterRoleBindings を作成する必要があるのはなぜですか?
- OpenShift の標準ユーザーは Kafka カスタムリソースを作成できますか?

13.10.1.2. Cluster Operator が ClusterRoleBindings を作成する必要があるのはなぜですか?

OpenShift には組み込みの 権限昇格防止機能 があります。これは、Cluster Operator は付与されていない権限を付与できず、特にアクセスできない namespace にこのような権限を付与できないことを意味します。そのため、Cluster Operator は、オーケストレーションする すべて のコンポーネントに必要な権限を持つ必要があります。

以下を実行するため、Cluster Operator によるアクセス権の付与が必要です。

- Topic Operator は、operator が実行される namespace に Roles および RoleBindings を作成することで、KafkaTopics を管理できます。
- User Operator は、operator が実行される namespace に Roles および RoleBindings を作成することで、KafkaUsers を管理できます。

- Node の障害ドメインは、ClusterRoleBinding を作成することで、AMQ Streams によって検出されます。

ラック認識のパーティション割り当てを使用する場合、ブローカー Pod は、Amazon AWS のアベイラビリティーゾーンなどの実行中の Node についての情報を取得できなければなりません。Node はクラスタースコープのリソースであるため、アクセスは namespace スコープの RoleBinding ではなく、ClusterRoleBinding を介してのみ許可できます。

13.10.1.3. OpenShift の標準ユーザーは Kafka カスタムリソースを作成できますか?

デフォルトでは、OpenShift の標準ユーザーには、Cluster Operator で処理されるカスタムリソースの管理に必要な権限がありません。クラスター管理者は、OpenShift RBAC リソースを使用してユーザーに必要な権限を付与できます。

詳細は、『OpenShift での AMQ Streams のデプロイおよびアップグレード』の「AMQ Streams の管理者 の指名」を参照してください。

13.10.1.4. ログの Failed to acquire lock 警告の意味

Cluster Operator は各クラスターに対して一度に単一の操作のみを実行します。Cluster Operator はロックを使用して、同じクラスターに対して並列操作が実行されないようにします。その他の操作は、ロックがリリースされる前に現在の操作が完了するまで待機する必要があります。

INFO

クラスター操作の例には、クラスターの作成、ローリングアップデート、スケールダウン、スケールアップ が含まれます。

ロックの待機時間が長すぎると、操作はタイムアウトになり、以下の警告メッセージがログに出力されます。

```
2018-03-04 17:09:24 WARNING AbstractClusterOperations:290 - Failed to acquire lock for kafka cluster lock::kafka::myproject::my-cluster
```

STRIMZI_FULL_RECONCILIATION_INTERVAL_MS と STRIMZI_OPERATION_TIMEOUT_MS の正確な設定によっては、根本的な問題を示すことなく、この警告メッセージが時々表示される場合があります。タイムアウトする操作が次の定期的な調整で検出され、これにより操作がロックを取得し、再度実行することができます。

指定のクラスターに他の操作が実行されていないはずの状況においても、このメッセージが定期的に表示される場合は、エラーが原因でロックが適切にリリースされなかったことを示している可能性があります。この場合、Cluster Operator の再起動を試行します。

13.10.1.5. TLS を使用して NodePort に接続するとホスト名の検証に失敗するのはなぜですか?

現時点では、TLS 暗号化が有効になっている NodePorts を使用したクラスター外のアクセスは、TLS ホスト名の検証をサポートしていません。その結果、ホスト名を確認するクライアントが接続に失敗します。たとえば、Java クライアントは以下の例外によって失敗します。

```
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
```

接続するには、ホスト名の検証を無効にする必要があります。Java クライアントでは、設定オプション ssl.endpoint.identification.algorithm を空の文字列に設定することでこれを実行できます。

プロパティーファイルを使用してクライアントを設定する場合は、以下のように実行できます。

```
ssl.endpoint.identification.algorithm=
```

Java でクライアントを直接設定する場合は、設定オプションを空の文字列に設定します。

```
props.put("ssl.endpoint.identification.algorithm", "");
```

第14章 カスタムリソース API のリファレンス

14.1. 共通の設定プロパティー

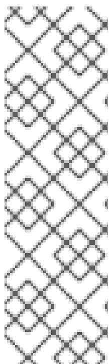
共通設定プロパティーは複数のリソースに適用されます。

14.1.1. replicas

replicas プロパティーを使用してレプリカを設定します。

レプリケーションのタイプはリソースによって異なります。

- KafkaTopic はレプリケーション係数を使用して、Kafka クラスター内で各パーティションのレプリカ数を設定します。
- Kafka コンポーネントはレプリカを使用してデプロイメントの Pod 数を設定し、可用性とスケーラビリティーを向上します。



注記

OpenShift で Kafka コンポーネントを実行している場合、高可用性のために複数のレプリカを実行する必要がない場合があります。コンポーネントがデプロイされたノードがクラッシュすると、OpenShift によって自動的に Kafka コンポーネント Pod が別のノードに再スケジュールされます。ただし、複数のレプリカで Kafka コンポーネントを実行すると、他のノードが稼働しているため、フェイルオーバー時間が短縮されます。

14.1.2. bootstrapServers

bootstrapServers プロパティーを使用してブートストラップサーバーのリストを設定します。

ブートストラップサーバーリストは、同じ OpenShift クラスターにデプロイされていない Kafka クラスターを参照できます。AMQ Streams によってデプロイされた Kafka クラスターを参照することもできます。

同じ OpenShift クラスターである場合、各リストに CLUSTER-NAME-kafka-bootstrap という名前の Kafka クラスターブートストラップサービスとポート番号が含まれる必要があります。AMQ

Streams によって異なる OpenShift クラスターにデプロイされた場合、リストの内容はクラスターを公開するために使用された方法によって異なります (route、ingress、nodeport、または loadbalancer)。

AMQ Streams によって管理されない Kafka クラスターで Kafka を使用する場合は、指定のクラスターの設定に応じてブートストラップサーバーのリストを指定できます。

14.1.3. ssl

TLSバージョンの特定のcipher suiteを使用するクライアント接続には、3つの許可されたssl設定オプションを使用します。暗号スイートは、セキュアな接続とデータ転送のためのアルゴリズムを組み合わせます。

ssl.endpoint.identification.algorithm プロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

SSL の設定例

```
# ...
spec:
  config:
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" 1
    ssl.enabled.protocols: "TLSv1.2" 2
    ssl.protocol: "TLSv1.2" 3
    ssl.endpoint.identification.algorithm: HTTPS 4
# ...
```

1. TLS の暗号スイートは、ECDHE 鍵交換メカニズム、RSA 認証アルゴリズム、AES 一括暗号化アルゴリズム、および SHA384 MAC アルゴリズムの組み合わせを使用します。

2. SSl プロトコル TLSv1.2 は有効になります。

3.

TLSv1.2 プロトコルを指定し、SSL コンテキスト生成します。許可される値は TLSv1.1 および TLSv1.2 です。

4

ホスト名の検証は、HTTPS に設定して有効化されます。空の文字列を指定すると検証が無効になります。

14.1.4. trustedCertificates

tls を設定して TLS 暗号化を設定する場合は、trustedCertificates プロパティーを使用して、証明書が X.509 形式で保存されるキー名にシークレットの一覧を提供します。

Kafka クラスターの Cluster Operator によって作成されるシークレットを使用するか、独自の TLS 証明書ファイルを作成してから、ファイルから Secret を作成できます。

```
oc create secret generic MY-SECRET \
--from-file=MY-TLS-CERTIFICATE-FILE.crt
```

TLS による暗号化の設定例

```
tls:
  trustedCertificates:
    - secretName: my-cluster-cluster-cert
      certificate: ca.crt
    - secretName: my-cluster-cluster-cert
      certificate: ca2.crt
```

複数の証明書が同じシークレットに保存されている場合は、複数回リストできます。

TLS を有効にし、Java に同梱されるデフォルトの公開認証局のセットを使用する場合は、trustedCertificates を空の配列として指定できます。

デフォルトの Java 証明書で TLS を有効にする例

```
tls:
  trustedCertificates: []
```

TLS クライアント認証の設定に関する詳細は、KafkaClientAuthenticationTls schema reference を参照してください。

14.1.5. resources

コンポーネントの CPU およびメモリーリソースを要求します。制限によって、指定のコンテナーが消費可能な最大リソースが指定されます。

Topic Operator および User Operator のリソース要求および制限は Kafka リソースに設定されます。

reources.requests および resources.limits プロパティーを使用して、リソース要求および制限を設定します。

AMQ Streams では、デプロイされたコンテナーごとに特定のリソースを要求し、これらのリソースの最大消費を定義できます。

AMQ Streams では、以下のリソースタイプの要求および制限がサポートされます。

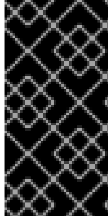
- cpu
- memory

AMQ Streams では、このようなリソースの指定に OpenShift の構文が使用されます。

OpenShift におけるコンピュートリソースの管理に関する詳細は、「Managing Compute Resources for Containers」を参照してください。

リソース要求

要求によって、指定のコンテナーに対して予約するリソースが指定されます。リソースを予約すると、リソースが常に利用できるようになります。



重要

リソース要求が OpenShift クラスターで利用可能な空きリソースを超える場合、Pod はスケジュールされません。

1 つまたは複数のサポートされるリソースに対してリクエストを設定できます。

リソース要求の設定例

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

リソース制限

制限によって、指定のコンテナーが消費可能な最大リソースが指定されます。制限は予約されず、常に利用できるとは限りません。コンテナーは、リソースが利用できる場合のみ、制限以下のリソースを使用できます。リソース制限は、常にリソース要求よりも高くする必要があります。

1 つまたは複数のサポートされる制限に対してリソースを設定できます。

リソース制限の設定例

```
# ...
```

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

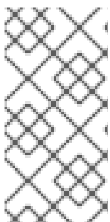
サポートされる CPU 形式

CPU の要求および制限は以下の形式でサポートされます。

- 整数値 (5) または少数 (2.5) の CPU コアの数。
- 数値または ミリ CPU / ミリコア (100m)。1000 ミリコア は CPU コア 1 つと同じです。

CPU ユニットの例

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



注記

1 つの CPU コアのコンピューティング能力は、OpenShift がデプロイされたプラットフォームによって異なることがあります。

CPU 仕様の詳細は、「Meaning of CPU」を参照してください。

サポートされるメモリー形式

メモリー要求および制限は、メガバイト、ギガバイト、メビバイト、およびギビバイトで指定されます。

- メモリーをメガバイトで指定するには、M 接尾辞を使用します。たとえば、1000M のように指定します。
- メモリーをギガバイトで指定するには、G 接尾辞を使用します。たとえば、1G のように指定します。
- メモリーをメビバイトで指定するには、Mi 接尾辞を使用します。たとえば、1000Mi のように指定します。
- メモリーをギビバイトで指定するには、Gi 接尾辞を使用します。たとえば、1Gi のように指定します。

異なるメモリー単位を使用するリソースの例

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

メモリーの指定およびサポートされるその他の単位に関する詳細は、「Meaning of memory」を参照してください。

14.1.6. image

image プロパティーを使用して、コンポーネントによって使用されるコンテナーイメージを設定します。

コンテナーイメージのオーバーライドは、別のコンテナーレジストリーやカスタマイズされたイメージを使用する必要がある特別な状況でのみ推奨されます。

たとえば、ネットワークで AMQ Streams によって使用されるコンテナーリポジトリーへのアクセスが許可されない場合、AMQ Streams イメージのコピーまたはソースからのビルドを行うことができます。しかし、設定したイメージが AMQ Streams イメージと互換性のない場合は、適切に機能しない可能性があります。

コンテナーイメージのコピーはカスタマイズでき、デバッグに使用されることもあります。

以下のリソースの image プロパティーを使用すると、コンポーネントに使用するコンテナーイメージを指定できます。

- Kafka.spec.kafka
- Kafka.spec.zookeeper
- Kafka.spec.entityOperator.topicOperator
- Kafka.spec.entityOperator.userOperator
- Kafka.spec.entityOperator.tlsSidecar
- KafkaConnect.spec
- KafkaMirrorMaker.spec
- KafkaMirrorMaker2.spec

- KafkaBridge.spec

Kafka、Kafka Connect、および Kafka MirrorMaker の image プロパティーの設定

Kafka、Kafka Connect、および Kafka MirrorMaker では、複数の Kafka バージョンがサポートされます。各コンポーネントには独自のイメージが必要です。異なる Kafka バージョンのデフォルトイメージは、以下の環境変数で設定されます。

- STRIMZI_KAFKA_IMAGES
- STRIMZI_KAFKA_CONNECT_IMAGES
- STRIMZI_KAFKA_MIRROR_MAKER_IMAGES

これらの環境変数には、Kafka バージョンと対応するイメージ間のマッピングが含まれます。マッピングは、image および version プロパティーとともに使用されます。

- image と version のどちらもカスタムリソースに指定されていない場合、version は Cluster Operator のデフォルトの Kafka バージョンに設定され、環境変数のこのバージョンに対応するイメージが指定されます。
- image が指定されていても version が指定されていない場合、指定されたイメージが使用され、Cluster Operator のデフォルトの Kafka バージョンが version であると想定されます。
- version が指定されていても image が指定されていない場合、環境変数の指定されたバージョンに対応するイメージが使用されます。
- version と image の両方を指定すると、指定されたイメージが使用されます。このイメージには、指定のバージョンの Kafka イメージが含まれると想定されます。

異なるコンポーネントの image および version は、以下のプロパティーで設定できます。

- Kafka の場合は spec.kafka.image および spec.kafka.version。
- spec. image および spec. version の Kafka Connect および Kafka MirrorMaker の場合。

警告

version のみを提供し、image プロパティーを未指定のままにしておくことが推奨されます。これにより、カスタムリソースの設定時に間違いが発生する可能性が低減されます。異なるバージョンの Kafka に使用されるイメージを変更する必要がある場合は、Cluster Operator の環境変数を設定することが推奨されます。

他のリソースでの image プロパティーの設定

他のカスタムリソースの image プロパティーでは、デプロイメント中に指定の値が使用されます。image プロパティーがない場合、Cluster Operator 設定に指定された image が使用されます。image 名が Cluster Operator 設定に定義されていない場合、デフォルト値が使用されます。

- Topic Operator の場合:
  1. Cluster Operator 設定から STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE 環境変数に指定されたコンテナーイメージ。
  2. registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.1 container image.
- User Operator の場合:
  1. Cluster Operator 設定から STRIMZI_DEFAULT_USER_OPERATOR_IMAGE 環境変数に指定されたコンテナーイメージ。

2. registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.1 container image.

- Entity Operator TLS サイドカーの場合:

1. Cluster Operator 設定から STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE 環境変数に指定されたコンテナーイメージ。

2. registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1 container image.

- Kafka Exporter の場合:

1. Cluster Operator 設定から STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE 環境変数に指定されたコンテナーイメージ。

2. registry.redhat.io/amq7/amq-streams-kafka-30-rhel8:2.0.1 container image.

- Kafka Bridge の場合:

1. Cluster Operator 設定から STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE 環境変数に指定されたコンテナーイメージ。

2. registry.redhat.io/amq7/amq-streams-bridge-rhel8:2.0.1 container image.

- Kafka ブローカーイニシャライザーの場合:

1. Cluster Operator 設定から STRIMZI_DEFAULT_KAFKA_INIT_IMAGE 環境変数に指定されたコンテナーイメージ。

2. registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.1 container image.

コンテナーイメージ設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    image: my-org/my-image:latest
    # ...
  zookeeper:
    # ...
```

14.1.7. livenessProbe および readinessProbe healthcheck

livenessProbe および readinessProbe プロパティーを使用して、AMQ Streams でサポートされる healthcheck プローブを設定します。

Healthcheck は、アプリケーションの健全性を検証する定期的なテストです。ヘルスチェックプローブが失敗すると、OpenShift によってアプリケーションが正常でないと見なされ、その修正が試行されます。

プローブの詳細は、「Configure Liveness and Readiness Probes」を参照してください。

livenessProbe および readinessProbe の両方で以下のオプションがサポートされます。

- initialDelaySeconds
- timeoutSeconds
- periodSeconds
- successThreshold

- failureThreshold

Liveness および Readiness プローブの設定例

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

livenessProbe および readinessProbe のオプションに関する詳細は、「Probe スキーマ参照」を参照してください。

14.1.8. metricsConfig

metricsConfig プロパティーを使用して、Prometheus メトリクスを有効化および設定します。

metricsConfig プロパティーには、Prometheus JMX Exporter の追加設定が含まれる ConfigMap への参照が含まれます。AMQ Streams では、Apache Kafka および ZooKeeper によってサポートされる JMX メトリクスを Prometheus メトリクスに変換するために、Prometheus JMX エクスポーターを使用した Prometheus メトリクスがサポートされます。

追加設定なしで Prometheus メトリクスのエクスポートを有効にするには、metricsConfig.valueFrom.configMapKeyRef.key 配下に空のファイルが含まれる ConfigMap を参照します。空のファイルを参照する場合、名前が変更されていない限り、すべてのメトリクスが公開されます。

Kafka のメトリクス設定が含まれる ConfigMap の例

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: my-configmap
```

```
data:
  my-key: |
    lowercaseOutputName: true
    rules:
    # Special cases and very specific rules
    - pattern: kafka.server<type=(.+), name=(.+), clientId=(.+), topic=(.+), partition=(.*)><>Value
      name: kafka_server_$1_$2
      type: GAUGE
      labels:
       clientId: "$3"
       topic: "$4"
       partition: "$5"
    # further configuration
```

Kafka のメトリクス設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: my-key
    # ...
  zookeeper:
    # ...
```

有効になったメトリクスは、9404 番ポートで公開されます。

metricsConfig（または非推奨の metrics）プロパティーがリソースに定義されていない場合、Prometheus メトリクスは無効になります。

Prometheus および Grafana の設定およびデプロイに関する詳細は、『OpenShift での AMQ Streams のデプロイおよびアップグレード』の「Kafka へのメトリクスの導入」を参照してくださ

い。

14.1.9. jvmOptions

以下の AMQ Streams コンポーネントは、Java 仮想マシン (JVM) 内で実行されます。

- Apache Kafka
- Apache ZooKeeper
- Apache Kafka Connect
- Apache Kafka MirrorMaker
- AMQ Streams Kafka Bridge

異なるプラットフォームやアーキテクチャーでパフォーマンスを最適化するには、以下のリソースに jvmOptions プロパティーを設定します。

- Kafka.spec.kafka
- Kafka.spec.zookeeper
- KafkaConnect.spec
- KafkaMirrorMaker.spec
- KafkaMirrorMaker2.spec

- **KafkaBridge.spec**

設定では、以下のオプションを指定できます。

-Xms

JVM の起動時に最初に割り当てられる最小ヒープサイズ。

-Xmx

最大ヒープサイズ。

-XX

JVM の高度なランタイムオプション。

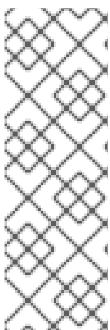
javaSystemProperties

追加のシステムプロパティー。

gcLoggingEnabled

ガベッジコレクターのロギングを有効にします。

jvmOptions の完全なスキーマは、JvmOptions schema referenceに記載されています。



注記

-Xmx や -Xms などの JVM 設定で使用できる単位は、対応するイメージの JDK java バイナリーで使用できる単位と同じです。そのため、1g または 1G は 1,073,741,824 バイトを意味し、Gi はサフィックスとして有効な単位ではありません。これは、メモリの要求や制限に使用される単位とは異なります。OpenShiftの規約では、1Gは 1,000,000,000バイト、1Giは1,073,741,824バイトを意味します。

-Xms および -Xmx オプション

-Xms および -Xmx に使用されるデフォルト値は、コンテナーに メモリー要求 の制限が設定されているかどうかによって異なります。

- メモリーの制限がある場合は、JVM の最小および最大メモリーは制限に対応する値に設定されます。

- メモリーの制限がない場合、JVM の最小メモリーは 128M に設定されます。JVM の最大メモリーは、必要に応じてメモリーを拡張するようには定義されていません。これは、テストおよび開発での単一ノード環境に適しています。

-Xmx を明示的に設定する前に、以下を考慮してください。

- JVM メモリー使用量の合計は、最大ヒープサイズよりも大きくなる可能性があります。**-Xmx** の値を見つけようとし、コンテナーのメモリー要求を超過せずに最適に使用できるようにします。
- 適切な OpenShift メモリー要求の設定
  - OpenShift は、ノードで実行されている他の Pod からのメモリーが不足すると、コンテナーを強制終了する可能性があります。
  - OpenShift は、メモリーが十分にないノードにコンテナーをスケジュールする可能性があります。**-Xms** が **-Xmx** に設定されている場合、コンテナーはすぐにクラッシュします。そうでない場合には、コンテナーは後でクラッシュします。

以下を行うことが推奨されます。

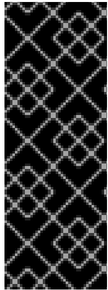
- メモリー要求とメモリー制限を同じ値に設定します。
- **-Xmx**の 4.5 倍以上のメモリー要求を使用します。
- **-Xms** を **-Xmx**と同じ値に設定することを検討してください。

この例では、JVM のヒープに 2 GiB (2,147,483,648 バイト) が使用されます。メモリー使用量の合計は約 8GiB です。

-Xmx および -Xms の設定例

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

最初のヒープサイズ (-Xms) および最大ヒープサイズ (-Xmx) に同じ値を設定すると、JVM が必要以上のヒープを割り当てて起動後にメモリーを割り当てないようにすることができます。



重要

Kafka ブローカーコンテナーなど、多数のディスク I/O を実行するコンテナーには、オペレーティングシステムのページキャッシュとして使用できるメモリーが必要です。このようなコンテナーでは、要求されるメモリーは JVM によって使用されるメモリーよりもはるかに多くなります。

-XX オプション

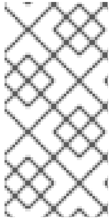
-XX オプションは、Apache Kafka の KAFKA_JVM_PERFORMANCE_OPTS オプションの設定に使用されます。

例 -XX 設定

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

-XX 設定から生成される JVM オプション

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



注記

-XX オプションを指定しないと、Apache Kafka の **KAFKA_JVM_PERFORMANCE_OPTS** のデフォルト設定が使用されます。

javaSystemProperties

javaSystemProperties は、デバッグユーティリティーなどの追加の Java システムプロパティーの設定に使用されます。

javaSystemProperties の設定例

```
jvmOptions:
  javaSystemProperties:
    - name: javax.net.debug
      value: ssl
```

14.1.10. ガベッジコレクターのロギング

jvmOptions プロパティーでは、ガベージコレクター(GC)のロギングを有効または無効にすることもできます。GC ロギングはデフォルトで無効になっています。これを有効にするには、以下のように **gcLoggingEnabled** プロパティーを設定します。

GC ロギングの設定例

```
# ...
jvmOptions:
  gcLoggingEnabled: true
# ...
```

14.2. スキーマプロパティー

14.2.1. Kafka スキーマ参照

プロパティー	説明
spec	Kafka および ZooKeeper クラスター、Topic Operator の仕様。
KafkaSpec	
status	Kafka および ZooKeeper クラスター、Topic Operator のステータス。
KafkaStatus	

14.2.2. KafkaSpec スキーマ参照

Kafka で使用

プロパティー	説明
kafka	Kafka クラスターの設定。
KafkaClusterSpec	
zookeeper	ZooKeeper クラスターの設定。
ZookeeperClusterSpec	
entityOperator	Entity Operator の設定。
EntityOperatorSpec	
clusterCa	クラスター認証局の設定。
CertificateAuthority	
clientsCa	クライアント認証局の設定。
CertificateAuthority	

プロパティー	説明
cruiseControl **CruiseControlSpec**	Cruise Control デプロイメントの設定。指定時に Cruise Control インスタンスをデプロイします。
kafkaExporter **KafkaExporterSpec**	Kafka Exporter の設定。Kafka Exporter は追加のメトリクスを提供できます (例: トピック/パーティションでのコンシューマーグループのラグなど)。
maintenanceTimeWindows string array	メンテナンスタスク (証明書の更新) 用の時間枠の一覧。それぞれの時間枠は、cron 式で定義されます。

14.2.3. KafkaClusterSpec スキーマ参照

KafkaSpec で使用

KafkaClusterSpec スキーマプロパティーの完全リスト

Kafka クラスターを設定します。

14.2.3.1. listeners

listeners プロパティーを使用して、Kafka ブローカーへのアクセスを提供するようにリスナーを設定します。

認証のないプレーン (暗号化されていない) リスナーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
```

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

14.2.3.2. 設定

config プロパティーを使用して、Kafka ブローカーオプションをキーとして設定します。

標準の Apache Kafka 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。

以下に関連する設定オプションは設定できません。

- セキュリティー (暗号化、認証、および承認)
- リスナーの設定
- Broker ID の設定
- ログデータディレクトリーの設定
- ブローカー間の通信
- ZooKeeper の接続

値は以下の JSON タイプのいずれかになります。

- 文字列

- 数値
- ブール値

AMQ Streams で直接管理されるオプションを除き、Apache Kafka ドキュメント に記載されているオプションを指定および設定できます。以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはすべて禁止されています。

- 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

禁止されているオプションが config プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。サポートされるその他すべてのオプションは Kafka に渡されます。

禁止されているオプションには例外があります。TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。zookeeper.connection.timeout.ms プロパティーを設定して、ZooKeeper 接続の確立に許可される最大時間を設定することもできます。

Kafka ブローカーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
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
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    zookeeper.connection.timeout.ms: 6000
  # ...
```

14.2.3.3. brokerRackInitImage

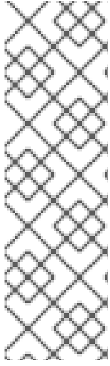
ラックアウェアネス (Rack Awareness) が有効である場合、Kafka ブローカー Pod は init コンテナーを使用して OpenShift クラスターノードからラベルを収集します。このコンテナーに使用されるコンテナーイメージは、brokerRackInitImage プロパティーを使用して設定できます。brokerRackInitImage フィールドがない場合、以下のイメージが優先度順に使用されます。

1. Cluster Operator 設定の STRIMZI_DEFAULT_KAFKA_INIT_IMAGE 環境変数に指定されたコンテナーイメージ。

2. registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.1 container image.

brokerRackInitImage の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    brokerRackInitImage: my-org/my-image:latest
    # ...
```



注記

コンテナーイメージのオーバーライドは、別のコンテナーレジストリーを使用する必要がある特別な状況でのみ推奨されます。たとえば、AMQ Streams によって使用されるコンテナーレジストリーにネットワークがアクセスできない場合などがこれに該当します。この場合は、AMQ Streams イメージをコピーするか、ソースからビルドする必要があります。設定したイメージが AMQ Streams イメージと互換性のない場合は、適切に機能しない可能性があります。

14.2.3.4. ログ

Kafka には独自の設定可能なロガーがあります。

- 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

Kafka では Apache log4j ロガー実装が使用されます。

logging プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は log4j.properties を使用して記述されます。logging.valueFrom.configMapKeyRef.name および logging.valueFrom.configMapKeyRef.key プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、「Apache logging services」を参照してください。

ここで、inline および external ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    # ...
    logging:
      type: inline
      loggers:
        kafka.root.logger.level: "INFO"
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: kafka-log4j.properties
  # ...
```

設定されていない利用可能なロガーのレベルは OFF に設定されています。

Cluster Operator を使用して Kafka がデプロイされた場合、Kafka のロギングレベルの変更は動的に適用されます。

外部ロギングを使用する場合は、ロギングアペンダーが変更されるとローリングアップデートがトリガーされます。

ガベッジコレクター (GC)

ガベッジコレクターのロギングは jvmOptions プロパティーを使用して 有効（または無効）にすることもできます。

14.2.3.5. KafkaClusterSpec スキーマプロパティー

プロパティー	説明
version string	Kafka ブローカーのバージョン。デフォルトは 3.0.0 です。バージョンのアップグレードまたはダウングレードに必要なプロセスを理解するには、ユーザードキュメントを参照してください。
replicas integer	クラスター内の Pod 数。

プロパティー	説明
image	Pod の Docker イメージ。デフォルト値は、設定した **Kafka.spec.kafka.version** によって異なります。
string	
listeners	Kafka ブローカーのリスナーを設定します。
GenericKafkaListener 配列	
設定	次の接頭辞のある Kafka ブローカーの config プロパティーは設定できません: 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、zookeeper.ssl、zookeeper.clientCnxnSocket、authorizer.、super.user、cruise.control.metrics.topic、cruise.control.metrics.reporter.bootstrap.servers (次の例外を除く: zookeeper.connection.timeout.ms、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols,cruise.control.metrics.topic.num.partitions、cruise.control.metrics.topic.replication.factor、cruise.control.metrics.topic.retention.ms、cruise.control.metrics.topic.auto.create.retries、cruise.control.metrics.topic.auto.create.timeout.ms、cruise.control.metrics.topic.min.insync.replicas)
map	
storage	ストレージの設定 (ディスク)。更新はできません。タイプは、指定のオブジェクト内の **storage.type** プロパティーの値によって異なり、[ephemeral、persistent-claim、jbod] のいずれかでなければなりません。
EphemeralStorage、**PersistentClaimStorage**、**JbodStorage**	
認可	Kafka ブローカーの承認設定。タイプは、指定のオブジェクト内の **authorization.type** プロパティーの値によって異なり、[simple、opa、keycloak、custom] のいずれかでなければなりません。
KafkaAuthorizationSimple、**KafkaAuthorizationOpa**、**KafkaAuthorizationKeycloak**、**KafkaAuthorizationCustom**	
rack	**broker.rack** ブローカー設定の設定
Rack	
brokerRackInitImage	**broker.rack** の初期化に使用される init コンテナーのイメージ。
string	

プロパティー	説明
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	
jvmOptions	Pod の JVM オプション。
JvmOptions	
jmxOptions	Kafka ブローカーの JMX オプション。
KafkaJmxOptions	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
metricsConfig	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
JmxPrometheusExporterMetrics	
ログ	Kafka のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
template	Kafka クラスターリソースのテンプレート。ユーザーはテンプレートにより、**StatefulSet**、**Pod**、および **Service** の生成方法を指定できます。
KafkaClusterTemplate	

14.2.4. Generic KafkaListener スキーマ参照

KafkaClusterSpec で使用

GenericKafkaListener スキーマプロパティーの完全リスト

OpenShift 内外の Kafka ブローカーに接続するようにリスナーを設定します。

Kafka リソースでリスナーを設定します。

リスナー設定を示す Kafka リソースの例

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    #...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
      - name: external1
        port: 9094
        type: route
        tls: true
      - name: external2
        port: 9095
        type: ingress
        tls: true
        authentication:
          type: tls
        configuration:
          bootstrap:
            host: bootstrap.myingress.com
          brokers:
          - broker: 0
            host: broker-0.myingress.com
          - broker: 1
            host: broker-1.myingress.com
          - broker: 2
            host: broker-2.myingress.com
    #...
```

14.2.4.1. listeners

Kafka リソースの listeners プロパティーを使用して Kafka ブローカーリスナーを設定します。リスナーは配列として定義されます。

リスナーの設定例

```
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
```

名前およびポートは Kafka クラスター内で一意である必要があります。名前は最大 25 文字で、小文字と数字で構成されます。許可されるポート番号は 9092 以上ですが、すでに Prometheus および JMX によって使用されているポート 9404 および 9999 以外になります。

各リスナーに一意の名前とポートを指定することで、複数のリスナーを設定できます。

14.2.4.2. type

タイプはinternal、外部リスナーの場合はroute、loadbalancer、nodeport、ingressのいずれかに設定されます。

internal

tls プロパティーを使用して、暗号化の有無に関わらず内部リスナーを設定できます。

internalリスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: plain
        port: 9092
        type: internal
        tls: false
```

```
    - name: tls
      port: 9093
      type: internal
      tls: true
      authentication:
        type: tls
  #...
```

route

OpenShift Routes および HAProxy ルーターを使用して Kafka を公開するように外部リスナーを設定します。

Kafkaブローカーポッドごとに専用のRouteが作成されます。追加の Route が作成され、Kafka ブートストラップアドレスとして提供されます。これらの Routes を使用すると、Kafka クライアントを 443 番ポートで Kafka に接続することができます。クライアントはデフォルトのルーターポートであるポート 443 に接続しますが、トラフィックは設定するポート（この例では 9094 ）にルーティングされます。

route リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external1
        port: 9094
        type: route
        tls: true
    #...
```

ingress

Kubernetes Ingress および NGINX Ingress Controller for Kubernetes を使用して Kafka を公開するように外部リスナーを設定します。

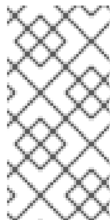
各 Kafka ブローカー Pod に専用の Ingress リソースが作成されます。追加の Ingress リソースが作成され、Kafka ブートストラップアドレスとして提供されます。これらの Ingress リソースを使用すると、Kafka クライアントを 443 番ポートで Kafka に接続することができます。クライア

ントはデフォルトのコントローラーポートであるポート 443 に接続しますが、トラフィックは設定するポート（以下の例では 9095 にルーティングされます）。

GenericKafkaListenerConfigurationBootstrap および GenericKafkaListenerConfigurationBroker プロパティーを使用して、ブートストラップおよびブローカーごとのサービスによって使用されるホスト名を指定する必要があります。

Ingress リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external2
        port: 9095
        type: ingress
        tls: true
        authentication:
          type: tls
        configuration:
          bootstrap:
            host: bootstrap.myingress.com
          brokers:
          - broker: 0
            host: broker-0.myingress.com
          - broker: 1
            host: broker-1.myingress.com
          - broker: 2
            host: broker-2.myingress.com
  #...
```



注記

Ingress を使用する外部リスナーは、現在 NGINX Ingress Controller for Kubernetes でのみテストされています。

loadbalancer

Kafka Loadbalancer タイプの Services を公開するように外部リスナーを設定します。

Kafka ブローカー Pod ごとに新しいロードバランサーサービスが作成されます。追加のロードバランサーが作成され、Kafka の ブートストラップ アドレスとして提供されます。ロードバランサーは指定のポート番号をリッスンします。以下の例ではポート 9094 です。

loadBalancerSourceRanges プロパティーを使用して、指定された IP アドレスへのアクセスを制限する ソース範囲 を設定できます。

loadbalancer リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        configuration:
          loadBalancerSourceRanges:
            - 10.0.0.0/8
            - 88.208.76.87/32
    #...
```

nodeport

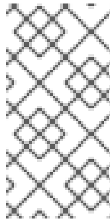
NodePort タイプの Services を使用して Kafka を公開するように外部リスナーを設定します。

Kafka クライアントは OpenShift のノードに直接接続します。追加の NodePort タイプのサービスが作成され、Kafka ブートストラップアドレスとして提供されます。

Kafka ブローカー Pod にアドバタイズされたアドレスを設定する場合、AMQ Stremas では該当の Pod が稼働しているノードのアドレスが使用されます。preferredNodePortAddressTypeプロパティを使用して、チェックした最初のアドレスタイプをノードアドレスとして設定することができます。

nodeport リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external4
        port: 9095
        type: nodeport
        tls: false
        configuration:
          preferredNodePortAddressType: InternalDNS
    #...
```



注記

ノードポートを使用して Kafka クラスターを公開する場合、現在 TLS ホスト名の検証はサポートされません。

14.2.4.3. port

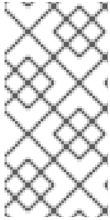
ポート番号は Kafka クラスターで使用されるポートで、クライアントによるアクセスに使用されるポートとは異なる場合があります。

- loadbalancer リスナーは internal リスナーのように、指定されたポート番号を使用します。
- ingress および route リスナーはアクセスにポート 443 を使用します。
- nodeport リスナーは OpenShift によって割り当てられたポート番号を使用します。

クライアント接続の場合は、リスナーのブートストラップサービスのアドレスおよびポートを使用します。これは、Kafka リソースのステータス から取得できます。

クライアント接続のアドレスおよびポートを取得するコマンドの例

```
oc get kafka KAFKA-CLUSTER-NAME -o=jsonpath='{.status.listeners[?(@.type=="external")].bootstrapServers}{"\n"}'
```



注記

ブローカー間通信 (9091) およびメトリクス (9404) 用に確保されたポートを使用するようにリスナーを設定することはできません。

14.2.4.4. tls

TLS プロパティーが必要です。

デフォルトでは、TLS による暗号化は有効になっていません。これを有効にするには、tlsプロパティをtrueに設定します。

routeリスナーでは、常にTLSによる暗号化が行われます。

14.2.4.5. 認証

リスナーの認証は以下のように指定できます。

- 相互のTLS(tls)
- SCRAM-SHA-512 (scram-sha-512)
- トークンベースのOAuth 2.0 (oauth)です。

14.2.4.6. networkPolicyPeers

ネットワークレベルでリスナーへのアクセスを制限するネットワークポリシーを設定するには、networkPolicyPeersを使用します。次の例では、plainとtlsリスナーのnetworkPolicyPeersの設定を示しています。

```yaml
listeners:
  #...
  - name: plain
    port: 9092
    type: internal
    tls: true
    authentication:
      type: scram-sha-512
    networkPolicyPeers:
      - podSelector:
          matchLabels:
            app: kafka-sasl-consumer
      - podSelector:
          matchLabels:
            app: kafka-sasl-producer
  - name: tls
    port: 9093
    type: internal
    tls: true
    authentication:
      type: tls
    networkPolicyPeers:
      - namespaceSelector:
          matchLabels:
            project: myproject
      - namespaceSelector:
          matchLabels:
            project: myproject2
# ...
```

この例では以下が設定されています。

- ラベル app: kafka-sasl-consumer および app: kafka-sasl-producer と一致するアプリケーション Pod のみが plain リスナーに接続できます。アプリケーション Pod は Kafka ブローカーと同じ namespace で実行されている必要があります。
- ラベル project: myproject および project: myproject2 と一致する namespace で稼働しているアプリケーション Pod のみ、tls リスナーに接続できます。

networkPolicyPeers フィールドの構文は、NetworkPolicy リソースの from フィールドと同じです。

14.2.4.7. GenericKafkaListenerスキーマプロパティ

プロパティー	説明
name	リスナーの名前。名前は、リスナーおよび関連する OpenShift オブジェクトの識別に使用されます。指定の Kafka クラスター内で一意となる必要があります。この名前には、小文字と数字を使用でき、最大 11 文字まで使用できます。
文字列	
port	Kafka 内でリスナーによって使用されるポート番号。ポート番号は指定の Kafka クラスター内で一意である必要があります。許可されるポート番号は 9092 以上ですが、すでに Prometheus および JMX によって使用されているポート 9404 および 9999 以外になります。リスナーのタイプによっては、ポート番号は Kafka クライアントに接続するポート番号と同じではない場合があります。
integer	
type	リスナーのタイプ。現在サポートされているタイプは、**internal**、**route**、**loadbalancer**、**nodeport**、**ingress**です。 • **internal**タイプは、OpenShiftクラスタ内でのみKafkaを内部的に公開します。 • **route**タイプは、OpenShift Routesを使用してKafkaを公開します。 • **loadbalancer**タイプは、LoadBalancerタイプのサービスを使用してKafkaを公開します。 • **nodeport**タイプは、NodePortタイプのサービスを使ってKafkaを公開します。 • **ingress**タイプは、OpenShift Nginx Ingressを使用してKafkaを公開します。
string ([ingress、internal、route、loadbalancer、nodeport] のいずれか)	
tls	リスナーで TLS による暗号化を有効にします。これは必須プロパティーです。
boolean	
authentication	このリスナーの認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls、scram-sha-512、oauth] のいずれかでなければなりません。
KafkaListenerAuthenticationTls、**KafkaListenerAuthenticationScramSha512**、**KafkaListenerAuthenticationOAuth**	
設定	追加のリスナー設定。
GenericKafkaListenerConfiguration	

プロパティー	説明
networkPolicyPeers	このリスナーに接続できるピアの一覧。この一覧のピアは、論理演算子 OR を使用して組み合わせます。このフィールドが空であるか、または存在しない場合、このリスナーのすべてのコネクションが許可されます。このフィールドが存在し、1 つ以上の項目が含まれる場合、リスナーはこの一覧の少なくとも 1 つの項目と一致するトラフィックのみを許可します。詳細は、networking.k8s.io/v1 networkpolicypeer の外部ドキュメントを参照してください。
NetworkPolicyPeer array	

14.2.5. KafkaListenerAuthenticationTls スキーマ参照

GenericKafkaListener で使用されています。

typeプロパティは、KafkaListenerAuthenticationTlsタイプの使用を KafkaListenerAuthenticationScramSha512、KafkaListenerAuthenticationOAuth と区別するための識別子です。KafkaListenerAuthenticationTls タイプには tls の値が必要です。

プロパティー	説明
type	**tls** でなければなりません。
文字列	

14.2.6. KafkaListenerAuthenticationScramSha512 スキーマ参照

GenericKafkaListener で使用されています。

typeプロパティは、KafkaListenerAuthenticationScramSha512タイプの使用を KafkaListenerAuthenticationTls、KafkaListenerAuthenticationOAuth と区別するための識別子です。KafkaListenerAuthenticationScramSha512 タイプには scram-sha-512 の値が必要です。

プロパティー	説明
type	**scram-sha-512** でなければなりません。
文字列	

14.2.7. KafkaListenerAuthenticationOAuth スキーマ参照

GenericKafkaListener で使用されています。

typeプロパティは、KafkaListenerAuthenticationOAuthタイプの使用を KafkaListenerAuthenticationTls、KafkaListenerAuthenticationScramSha512 と区別するための識別子です。KafkaListenerAuthenticationOAuth タイプには oauth の値が必要です。

プロパティー	説明
accessTokenIsJwt boolean	アクセストークンを JWT として処理するかどうかを設定します。承認サーバーが不透明なトークンを返す場合は、**false** に設定する必要があります。デフォルトは **true** です。
checkAccessTokenType boolean	アクセストークンタイプのチェックを行うかどうかを設定します。承認サーバーの JWT トークンに 'typ' 要求が含まれない場合は、**false** に設定する必要があります。デフォルトは **true** です。
checkAudience boolean	オーディエンスのチェックを有効または無効にします。オーディエンスのチェックによって、トークンの受信者が特定されます。オーディエンスチェックが有効な場合、OAuthクライアントIDも**clientId**プロパティで設定する必要があります。Kafkaブローカは、**aud**（オーディエンス）クレームに**clientId**がないトークンを拒否します。デフォルト値は**false**です。
checkIssuer boolean	発行元のチェックを有効または無効にします。デフォルトでは、**validIssuerUri** によって設定された値を使用して発行元がチェックされます。デフォルト値は **true** です。
clientAudience 文字列	承認サーバーのトークンエンドポイントにリクエストを送信するときに使用するオーディエンス。ブローカー間の認証や、**clientId**と**secret**メソッドを用いたPLAIN上のOAuth 2.0の設定に使用されます。
clientId 文字列	Kafka ブローカーは、OAuth クライアント ID を使用して承認サーバーに対して認証し、イントロスペクションエンドポイント URI を使用することができます。
clientScope 文字列	承認サーバーのトークンエンドポイントにリクエストを送信するときに使用するスコープ。ブローカー間の認証や、**clientId**と**secret**メソッドを用いたPLAIN上のOAuth 2.0の設定に使用されます。

プロパティー	説明
clientSecret **GenericSecretSource**	OAuth クライアントシークレットが含まれる OpenShift シークレットへのリンク。Kafka ブローカーは、OAuth クライアントシークレットを使用して承認サーバーに対して認証し、イントロスペクションエンドポイント URI を使用することができます。
customClaimCheck string	JWT トークンに適用される JSONPath フィルタークエリー、または追加のトークン検証のイントロスペクションエンドポイントの応答に適用される JSONPath フィルタークエリー。デフォルトでは設定されません。
disableTlsHostnameVerification boolean	TLS ホスト名の検証を有効または無効にします。デフォルト値は **false** です。
enableECDSA boolean	**enableECDSAプロパティは非推奨となりました。**BouncyCastle 暗号プロバイダーをインストールして、ECDSA サポートを有効または無効にします。ECDSA サポートが常に有効になります。BouncyCastle ライブラリーは、AMQ Streams とパッケージ化されなくなりました。値は無視されます。
enableOauthBearer boolean	SASL_OAUTHBEARER での OAuth 認証を有効または無効にします。デフォルト値は **true** です。
enablePlain boolean	SASL_PLAIN で OAuth 認証を有効または無効にします。このメカニズムが使用される場合、再認証はサポートされません。デフォルト値は **false** です。
fallbackUserNameClaim string	**userNameClaim** によって指定された要求が存在しない場合に、ユーザー ID に使用するフォールバックユーザー名要求。これは、**client_credentials** 認証によってクライアント ID が別の要求のみに提供される場合に便利です。**userNameClaim** が設定されている場合のみ有効です。
fallbackUserNamePrefix string	ユーザー ID を構成するために **fallbackUserNameClaim** の値と使用される接頭辞。**fallbackUserNameClaim** が true で、要求の値が存在する場合のみ有効です。ユーザー名とクライアント ID を同じユーザー ID 領域にマッピングすると、名前の競合を防ぐことができ便利です。

プロパティー	説明
introspectionEndpointUri	不透明な JWT 以外のトークンの検証に使用できるトークンイントロスペクションエンドポイントの URI。
string	
jwksEndpointUri	ローカルの JWT 検証に使用できる JWKS 証明書エンドポイントの URI。
string	
jwksExpirySeconds	JWKS 証明書が有効とみなされる頻度を設定します。期限切れの間隔は、**jwksRefreshSeconds** で指定される更新間隔よりも 60 秒以上長くする必要があります。デフォルトは 360 秒です。
integer	
jwksMinRefreshPauseSeconds	連続する 2 回の更新の間に適用される最小の一時停止期間。不明な署名鍵が検出されると、更新は即座にスケジュールされますが、この最小一時停止の期間は待機します。デフォルトは 1 秒です。
integer	
jwksRefreshSeconds	JWKS 証明書が更新される頻度を設定します。更新間隔は、**jwksExpirySeconds** で指定される期限切れの間隔よりも 60 秒以上短くする必要があります。デフォルトは 300 秒です。
integer	
maxSecondsWithoutReauthentication	再認証せずに認証されたセッションが有効な状態でいられる最大期間 (秒単位)。これにより、Apache Kafka の再認証機能が有効になり、アクセストークンの有効期限が切れるとセッションが期限切れになります。最大期間の前または最大期間の到達時にアクセストークンが期限切れになると、クライアントは再認証する必要があります。そうでないと、サーバーは接続を切断します。デフォルトでは設定されません。アクセストークンが期限切れになっても認証されたセッションは期限切れになりません。このオプションは、SASL_OAUTHBEARER認証メカニズムにのみ適用されます（**enableOauthBearer**が**true**の場合）。
integer	
tlsTrustedCertificates	OAuth サーバーへの TLS 接続の信頼済み証明書。
CertSecretSource array	

プロパティー	説明
tokenEndpointUri	クライアントが**clientId**と**secret**で認証する際に、SASL_PLAINメカニズムで使用するToken EndpointのURIです。設定されている場合、クライアントはSASL_PLAIN で認証を行うことができます。**username**を**clientId** に、**password**をclient**secret** に設定するか、または**username**をアカウントのユーザー名に、**password**を**$accessToken:** のプレフィックスが付いたアクセストークンに設定します。このオプションが設定されていない場合、**password**は常にアクセストークンとして（プレフィックスなしで）解釈され、**username**はアカウントのユーザー名として解釈されます（いわゆる no-client-credentials モードです）。
string	
type	**oauth** でなければなりません。
string	
userInfoEndpointUri	Introspection Endpoint がユーザー ID に使用できる情報を返さない場合に、ユーザー ID 取得のフォールバックとして使用する User Info Endpoint の URL。
string	
userNameClaim	ユーザー ID の取得に使用される JWT 認証トークン、Introspection Endpoint の応答、または User Info Endpoint の応答からの要求の名前。デフォルトは **sub** です。
string	
validIssuerUri	認証に使用されるトークン発行者の URI。
string	
validTokenType	Introspection Endpoint によって返される **token_type** 属性の有効な値。デフォルト値はなく、デフォルトではチェックされません。
string	

14.2.8. GenericSecretSource スキーマ参照

KafkaClientAuthenticationOAuth, **KafkaListenerAuthenticationOAuth** で使用

プロパティー	説明
key	OpenShift シークレットでシークレット値が保存されるキー。

プロパティー	説明
文字列	
secretName	シークレット値が含まれる OpenShift シークレットの名前。
文字列	

14.2.9. CertSecretSource スキーマ参照

ClientTls、KafkaAuthorizationKeycloak、KafkaClientAuthenticationOAuth、KafkaListenerAuthenticationOAuthで使用

プロパティー	説明
certificate	Secret のファイル証明書の名前。
文字列	
secretName	証明書が含まれる Secret の名前。
文字列	

14.2.10. GenericKafkaListenerConfiguration スキーマ参照

GenericKafkaListener で使用されています。

GenericKafkaListenerConfigurationスキーマプロパティの全リスト

Kafka リスナーの設定。

14.2.10.1. brokerCertChainAndKey

brokerCertChainAndKeyプロパティは、TLS暗号化が有効になっているリスナーでのみ使用されます。独自の Kafka リスナー証明書を提供してこのプロパティーを使用できます。

TLS 暗号化が有効な loadbalancer 外部リスナーの設定例

```
listeners:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    configuration:
      brokerCertChainAndKey:
        secretName: my-secret
        certificate: my-listener-certificate.crt
        key: my-listener-key.key
# ...
```

14.2.10.2. externalTrafficPolicy

externalTrafficPolicyプロパティは、loadbalancerやnodeportのリスナーで使用されます。OpenShiftの外でKafkaを公開する場合は、LocalまたはClusterを選択できます。Localは他のノードへのホップを避け、クライアントのIPを保持しますが、Clusterはそのどちらでもありません。デフォルトはClusterです。

14.2.10.3. loadBalancerSourceRanges

loadBalancerSourceRangesプロパティは、loadbalancerでのみ使用されます。OpenShift 外部で Kafka を公開する場合、ラベルやアノテーションの他にソースの範囲を使用して、サービスの作成方法をカスタマイズします。

ロードバランサーリスナー向けに設定されたソース範囲の例

```
listeners:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: false
    configuration:
      externalTrafficPolicy: Local
      loadBalancerSourceRanges:
        - 10.0.0.0/8
```

```
        - 88.208.76.87/32
      # ...
# ...
```

14.2.10.4. class

classプロパティは、ingressリスナーでのみ使用されます。Ingressクラスの設定はclassプロパティで行います。

Ingress クラスの nginx-internalを使用した ingress タイプの外部リスナーの例

```
listeners:
  #...
  - name: external
    port: 9094
    type: ingress
    tls: true
    configuration:
      class: nginx-internal
    # ...
# ...
```

14.2.10.5. preferredNodePortAddressType

preferredNodePortAddressTypeプロパティは、nodeportリスナーでのみ使用されます。

リスナーの設定でpreferredNodePortAddressTypeプロパティを使用して、ノードアドレスとしてチェックされる最初のアドレスタイプを指定します。たとえば、デプロイメントに DNS サポートがない場合や、内部 DNS または IP アドレスを介してブローカーを内部でのみ公開する場合、このプロパティーは便利です。該当タイプのアドレスが見つかった場合はそのアドレスが使用されます。アドレスタイプが見つからなかった場合、AMQ Streams は標準の優先順位でタイプの検索を続行します。

1. ExternalDNS

2. **ExternalIP**

3. **Hostname**

4. **InternalDNS**

5. **InternalIP**

優先ノードポートアドレスタイプで設定された外部リスナーの例

```
listeners:
  #...
  - name: external
    port: 9094
    type: nodeport
    tls: false
    configuration:
      preferredNodePortAddressType: InternalDNS
      # ...
# ...
```

14.2.10.6. useServiceDnsDomain

useServiceDnsDomainプロパティは、internalリスナーでのみ使用されます。クラスタサービスの接尾辞（通常は.cluster.local）を含む完全修飾DNS名を使用するかどうかを定義します。useServiceDnsDomainをfalseに設定すると、アドバタイズされるアドレスはサービスサフィックスなしで生成されます。（例:my-cluster-kafka-0.my-cluster-kafka-brokers.myproject.svc)useServiceDnsDomainをtrueに設定すると、アドバタイズされたアドレスはサービスのサフィックスで生成されます。（例:my-cluster-kafka-0.my-cluster-kafka-brokers.myproject.svc.cluster.local)デフォルトは false です。

サービス DNS ドメインを使用するよう設定された内部リスナーの例

```
listeners:
  #...
  - name: plain
```

```
      port: 9092
      type: internal
      tls: false
      configuration:
        useServiceDnsDomain: true
        # ...
  # ...
```

OpenShiftクラスターが.cluster.localとは異なるサービスサフィックスを使用している場合は、Cluster Operatorの設定でKUBERNETES_SERVICE_DNS_DOMAIN環境変数を使用してサフィックスを設定することができます。詳細は「Cluster Operator の設定」を参照してください。

14.2.10.7. GenericKafkaListenerConfigurationスキーマプロパティ

プロパティー	説明
brokerCertChainAndKey CertAndKeySecretSource	このリスナーに使用される証明書とプライベートキーのペアを保持する**Secret**への参照。証明書には、任意でチェーン全体を含めることができます。このフィールドは、TLS による暗号化が有効なリスナーでのみ使用できます。
externalTrafficPolicy string ([Local、Cluster] のいずれか)	サービスによって外部トラフィックがローカルノードのエンドポイントまたはクラスター全体のエンドポイントにルーティングされるかどうかを指定します。**Cluster** を指定すると、別のノードへの 2 回目のホップが発生し、クライアントソースの IP が特定しにくくなる可能性があります。**Local** を指定すると、LoadBalancer および Nodeport タイプのサービスに対して 2 回目のホップが発生しないようにし、クライアントソースの IP を維持します (インフラストラクチャーでサポートされる場合)。指定のない場合、OpenShift は **Cluster** をデフォルトとして使用します。このフィールドは、**loadbalancer** または **nodeport** タイプリスナーとのみ使用できます。

プロパティー	説明
loadBalancerSourceRanges	クライアントがロードバランサータイプのリスナーに接続できる CIDR 形式による範囲 (例: **10.0.0.0/8**、**130.211.204.1/32**) の一覧。プラットフォームでサポートされる場合、ロードバランサー経由のトラフィックは指定された CIDR 範囲に制限されます。このフィールドは、ロードバランサータイプのサービスのみに適用され、クラウドプロバイダーがこの機能をサポートしない場合は無視されます。詳細は「https://v1-17.docs.kubernetes.io/docs/tasks/access-application-cluster/configure-cloud-provider-firewall/」を参照してください。このフィールドは、**loadbalancer**のリスナーでのみ使用できます。
string array	
bootstrap	ブートストラップの設定。
GenericKafkaListenerConfigurationBootstrap	
brokers	ブローカーごとの設定。
GenericKafkaListenerConfigurationBroker array	
ipFamilyPolicy	サービスによって使用される IP Family Policy を指定します。利用可能なオプションは、**SingleStack**、**PreferDualStack**、**RequireDualStack**です。**SingleStack**は単一のIPファミリー用です。**PreferDualStack**は、デュアルスタック構成のクラスターでは2つのIPファミリーを、シングルスタック構成のクラスターでは1つのIPファミリーを対象としています。**RequireDualStack**は、デュアルスタック構成のクラスターに2つのIPファミリーがないと失敗します。指定されていない場合、OpenShift はサービスタイプに基づいてデフォルト値を選択します。OpenShift 1.20 以降で利用できます。
string ([RequireDualStack、SingleStack、PreferDualStack] のいずれか)	
ipFamilies	サービスによって使用される IP Families を指定します。利用可能なオプションは、**IPv4**と**IPv6です。指定されていない場合、OpenShift は `ipFamilyPolicy** の設定に基づいてデフォルト値を選択します。OpenShift 1.20 以降で利用できます。
string ([IPv6, IPv4] の 1 つ以上) array	

プロパティー	説明
class	使用する **Ingress** コントローラーを定義する **Ingress** クラスを設定します。このフィールドは、**ingress** タイプのリスナーでのみ使用できます。指定されていない場合、デフォルトの Ingress コントローラーが使用されます。
string	
finalizers	このリスナーに作成された**LoadBalancer**タイプのServicesに設定されるファイナライザのリストです。プラットフォームでサポートされている場合は、ファイナライザー**service.kubernetes.io/load-balancer-cleanup**を使用して、外部ロードバランサーがサービスと一緒に削除されるようにします。詳細は、https://kubernetes.io/docs/tasks/access-application-cluster/create-external-load-balancer/#garbage-collecting-load-balancers を参照してください。このフィールドは、**loadbalancer**タイプのリスナーでのみ使用できます。
string array	
maxConnectionCreationRate	このリスナーでいつでも許可される最大接続作成率。制限に達すると、新しい接続はスロットリングされます。
integer	
maxConnections	ブローカーのこのリスナーでいつでも許可される最大接続数。制限に達すると、新しい接続はブロックされます。
integer	

プロパティー	説明
preferredNodePortAddressType	ノードアドレスとして使用するアドレスタイプを定義します。使用できるタイプ: **ExternalDNS**、**ExternalIP**、**InternalDNS**、**InternalIP**、**Hostname**デフォルトでは、アドレスは以下の順序で使用されます (最初に見つかったアドレスが使用されます): • **ExternalDNS** • **ExternalIP** • **InternalDNS** • **InternalIP** • **Hostname** このフィールドは、最初にチェックされる優先アドレスタイプの選択に使用されます。このアドレスタイプのアドレスが見つからない場合、他のタイプがデフォルトの順序でチェックされます。このフィールドは、**nodeport**タイプのリスナーでのみ使用できます。
string ([ExternalDNS、ExternalIP、Hostname、InternalIP、InternalDNS] のいずれか)	
useServiceDnsDomain	OpenShift サービス DNS ドメインを使用するべきかどうかを設定します。**true**に設定すると、生成されるアドレスにはサービスのDNSドメインのサフィックスが含まれます（デフォルトでは**.cluster.local**、環境変数**KUBERNETES_SERVICE_DNS_DOMAIN**で設定可能です）。デフォルトは**false**です。このフィールドは、**internal**タイプのリスナーでのみ使用できます。
boolean	

14.2.11. CertAndKeySecretSource スキーマ参照

GenericKafkaListenerConfiguration、**KafkaClientAuthenticationTls** で使用されます。

プロパティー	説明
certificate	Secret のファイル証明書の名前。
string	
key	Secret の秘密鍵の名前。
string	

プロパティー	説明
secretName	証明書が含まれる Secret の名前。
string	

14.2.12. GenericKafkaListenerConfigurationBootstrap schema reference

GenericKafkaListenerConfiguration で使用されます。

GenericKafkaListenerConfigurationBootstrapスキーマプロパティの全リスト

nodePort、host、loadBalancerIP、annotationsプロパティに相当するブローカーサービスは、GenericKafkaListenerConfigurationBroker schema で構成されます。

14.2.12.1. alternativeNames

ブートストラップサービスの代替名を指定できます。名前はブローカー証明書に追加され、TLS ホスト名の検証に使用できます。alternativeNamesプロパティは、すべてのタイプのリスナーに適用されます。

ブートストラップアドレスを追加設定した外部routeリスナーの例です。

```
listeners:
  #...
  - name: external
    port: 9094
    type: route
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        alternativeNames:
          - example.hostname1
          - example.hostname2
# ...
```

14.2.12.2. host

hostプロパティは、routeリスナーとingressリスナーで使用され、ブートストラップサービスとブローカーサービスで使用されるホスト名を指定します。

Ingressコントローラが自動的にホスト名を割り当てることはないため、ingressリスナーの設定にはhostのプロパティ値が必須となります。確実にホスト名が Ingress エンドポイントに解決されるようにしてください。AMQ Streams では、要求されたホストが利用可能で、適切に Ingress エンドポイントにルーティングされることを検証しません。

Ingress リスナーのホスト設定例

```
listeners:
  #...
  - name: external
    port: 9094
    type: ingress
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        host: bootstrap.myingress.com
      brokers:
      - broker: 0
        host: broker-0.myingress.com
      - broker: 1
        host: broker-1.myingress.com
      - broker: 2
        host: broker-2.myingress.com
# ...
```

デフォルトでは、routeリスナーのホストはOpenShiftによって自動的に割り当てられます。ただし、ホストを指定して、割り当てられたルートをオーバーライドすることができます。

AMQ Streams では、要求されたホストが利用可能であることを検証しません。ホストが使用可能であることを確認する必要があります。

route リスナーのホスト設定例

```
# ...
listeners:
  #...
  - name: external
    port: 9094
    type: route
    tls: true
    authentication:
      type: tls
    configuration:
      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
# ...
```

14.2.12.3. nodePort

デフォルトでは、ブートストラップおよびブローカーサービスに使用されるポート番号は OpenShift によって自動的に割り当てられます。nodeportリスナーに割り当てられたノードポートを上書きするには、要求されたポート番号を指定します。

AMQ Streams は要求されたポートの検証を行いません。ポートが使用できることを確認する必要があります。

ノードポートのオーバーライドが設定された外部リスナーの例

```
# ...
listeners:
  #...
  - name: external
    port: 9094
    type: nodeport
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        nodePort: 32100
```

```
    brokers:
    - broker: 0
      nodePort: 32000
    - broker: 1
      nodePort: 32001
    - broker: 2
      nodePort: 32002
# ...
```

14.2.12.4. loadBalancerIP

ロードバランサーの作成時に特定のIPアドレスを要求するには、loadBalancerIPプロパティを使用します。特定の IP アドレスでロードバランサーを使用する必要がある場合は、このプロパティーを使用します。クラウドプロバイダーがこの機能に対応していない場合、loadBalancerIP フィールドは無視されます。

特定のロードバランサー IP アドレスリクエストのある loadbalancer タイプの外部リスナーの例

```
# ...
listeners:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        loadBalancerIP: 172.29.3.10
      brokers:
      - broker: 0
        loadBalancerIP: 172.29.3.1
      - broker: 1
        loadBalancerIP: 172.29.3.2
      - broker: 2
        loadBalancerIP: 172.29.3.3
# ...
```

14.2.12.5. annotations

annotationsを使用して、リスナーに関連するOpenShiftリソースにアノテーションを追加します。これらのアノテーションを使用すると、自動的に DNS 名をロードバランサーサービスに割り当てる 外部 DNS などの DNS ツールをインストルメント化できます。

annotationsを使用したloadbalancer型の外部リスナーの例

```
# ...
listeners:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-bootstrap.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      brokers:
      - broker: 0
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-0.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 1
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-1.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
      - broker: 2
        annotations:
          external-dns.alpha.kubernetes.io/hostname: kafka-broker-2.mydomain.com.
          external-dns.alpha.kubernetes.io/ttl: "60"
# ...
```

14.2.12.6. GenericKafkaListenerConfigurationBootstrapスキーマのプロパティ

プロパティー	説明
alternativeNames	ブートストラップサービスの追加の代替名。代替名は、TLS 証明書のサブジェクト代替名のリストに追加されます。
string array	

プロパティー	説明
host	ブートストラップホスト。このフィールドは、ホスト名を指定するために Ingress リソースまたは Route リソースで使用されます。このフィールドは、**route**（オプション）または**ingress**（必須）タイプのリスナーでのみ使用できます。
string	
nodePort	ブートストラップサービスのノードポート。このフィールドは、**nodeport**タイプリスナーでのみ使用できます。
integer	
loadBalancerIP	ロードバランサーは、このフィールドに指定された IP アドレスで要求されます。この機能は、ロードバランサーの作成時に、基礎となるクラウドプロバイダーが **loadBalancerIP** の指定をサポートするかどうかによって異なります。このフィールドは、クラウドプロバイダーがこの機能をサポートしていない場合は無視されます。このフィールドは、**loadbalancer**タイプのリスナーでのみ使用できます。
string	
annotations	**Ingress**、**Route**、**Service**のいずれかのリソースに追加されるアノテーション。このフィールドを使用して、外部 DNS などの DNS プロバイダーを設定できます。このフィールドは、**loadbalancer**、**nodeport**、**route**、**ingress** タイプのリスナーでのみ使用できます。
map	
labels	**Ingress**、**Route**、**Service**のいずれかのリソースに追加されるラベル。このフィールドは、**loadbalancer**、**nodeport**、**route**、**ingress** タイプのリスナーでのみ使用できます。
map	

14.2.13. GenericKafkaListenerConfigurationBroker schema reference

GenericKafkaListenerConfiguration で使用されます。

GenericKafkaListenerConfigurationBrokerスキーマプロパティの全リスト

ブートストラップサービスのオーバーライドを設定するGenericKafkaListenerConfigurationBootstrap schemaでは、nodePort、host、loadBalancerIP、annotationsプロパティの構成例を見ることができます。

ブローカーのアドバタイズされたアドレス

デフォルトでは、AMQ Streams は Kafka クラスターがそのクライアントにアドバタイズするホスト名とポートを自動的に決定しようとします。AMQ Streams が稼働しているインフラストラクチャーでは Kafka にアクセスできる正しいホスト名やポートを提供しない可能性があるため、デフォルトの動作はすべての状況に適しているわけではありません。

ブローカーIDを指定し、リスナーのconfigurationプロパティでアドバタイズされたホスト名とポートをカスタマイズすることができます。その後、AMQ Streams では Kafka ブローカーでアドバタイズされたアドレスが自動設定され、ブローカー証明書に追加されるため、TLS ホスト名の検証が使用できるようになります。アドバタイズされたホストおよびポートのオーバーライドは、すべてのタイプのリスナーで利用できます。

アドバタイズされたアドレスのオーバーライドを設定した外部routeリスナーの例

```
listeners:
  #...
  - name: external
    port: 9094
    type: route
    tls: true
    authentication:
      type: tls
    configuration:
      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
# ...
```

14.2.13.1. GenericKafkaListenerConfigurationBrokerスキーマプロパティ

プロパティー	説明
broker integer	Kafka ブローカーの ID (ブローカー識別子)。ブローカー ID は 0 から始まり、ブローカーレプリカの数に対応します。

プロパティー	説明
advertisedHost	ブローカーの **advertised.brokers** で使用されるホスト名。
string	
advertisedPort	ブローカーの **advertised.brokers** で使用されるポート番号。
integer	
host	ブローカーホスト。このフィールドは、ホスト名を指定するために Ingress リソースまたは Route リソースで使用されます。このフィールドは、**route**（オプション）または**ingress**（必須）タイプのリスナーでのみ使用できます。
string	
nodePort	ブローカーごとのサービスのノードポート。このフィールドは、**nodeport**タイプリスナーでのみ使用できます。
integer	
loadBalancerIP	ロードバランサーは、このフィールドに指定された IP アドレスで要求されます。この機能は、ロードバランサーの作成時に、基礎となるクラウドプロバイダーが **loadBalancerIP** の指定をサポートするかどうかによって異なります。このフィールドは、クラウドプロバイダーがこの機能をサポートしていない場合は無視されます。このフィールドは、**loadbalancer**タイプのリスナーでのみ使用できます。
string	
annotations	**Ingress**または**Service**リソースに追加されるアノテーション。このフィールドを使用して、外部 DNS などの DNS プロバイダーを設定できます。このフィールドは、**loadbalancer**、**nodeport**、**ingress**タイプのリスナーでのみ使用できます。
map	
labels	**Ingress**、**Route**、**Service**のいずれかのリソースに追加されるラベル。このフィールドは、**loadbalancer**、**nodeport**、**route**、**ingress** タイプのリスナーでのみ使用できます。
map	

14.2.14. EphemeralStorage スキーマ参照

JbodStorage、**KafkaClusterSpec**、**ZookeeperClusterSpec** で使用

typeプロパティは、**EphemeralStorage**タイプの使用を、**PersistentClaimStorage**から区別する識別子です。**EphemeralStorage** タイプには **ephemeral** の値が必要です。

プロパティー	説明
id	ストレージ ID 番号。これは、'jbod' タイプのストレージで定義されるストレージボリュームのみで必須です。
integer	
sizeLimit	type=ephemeral の場合、この EmptyDir ボリュームに必要なローカルストレージの合計容量を定義します (例: 1Gi)。
string	
type	**ephemeral** でなければなりません。
string	

14.2.15. PersistentClaimStorage スキーマ参照

JbodStorage、KafkaClusterSpec、ZookeeperClusterSpec で使用

typeプロパティは、PersistentClaimStorageタイプの使用を、EphemeralStorageから区別する識別子です。PersistentClaimStorage タイプには persistent-claim の値が必要です。

プロパティー	説明
type	**persistent-claim** でなければなりません。
string	
size	type=persistent-claim の場合、永続ボリューム要求のサイズを定義します (例: 1Gi)。type=persistent-claim の場合には必須です。
string	
selector	使用する特定の永続ボリュームを指定します。このようなボリュームを選択するラベルを表す key:value ペアが含まれます。
map	
deleteClaim	クラスターのアンデプロイ時に永続ボリューム要求を削除する必要があるかどうかを指定します。
boolean	
class	動的ボリュームの割り当てに使用するストレージクラス。

プロパティー	説明
string	
id	ストレージ ID 番号。これは、'jbod' タイプのストレージで定義されるストレージボリュームのみで必須です。
integer	
overrides	個々のブローカーを上書きします。**overrides** フィールドでは、異なるブローカーに異なる設定を指定できます。
PersistentClaimStorageOverride array	

14.2.16. PersistentClaimStorageOverride スキーマ参照

PersistentClaimStorage で使用

プロパティー	説明
class	このブローカーの動的ボリュームの割り当てに使用するストレージクラス。
文字列	
broker	Kafka ブローカーの ID (ブローカー ID)。
integer	

14.2.17. JbodStorage スキーマ参照

KafkaClusterSpec で使用

typeプロパティは、JbodStorageタイプの使用を EphemeralStorageと PersistentClaimStorage から区別する識別子です。JbodStorage タイプには jbod の値が必要です。

プロパティー	説明
type	**jbod** でなければなりません。
文字列	
volumes	JBOD ディスクアレイを表すストレージオブジェクトとしてのボリュームの一覧。

プロパティー	説明
EphemeralStorage、PersistentClaimStorage array	

14.2.18. KafkaAuthorizationSimple スキーマ参照

KafkaClusterSpec で使用

KafkaAuthorizationSimpleスキーマプロパティの全リスト

AMQ Streamsでのシンプルな認証は、Apache Kafkaで提供されているデフォルトのACL（Access Control Lists）認証プラグインであるAclAuthorizerプラグインを使用します。ACL を使用すると、ユーザーがアクセスできるリソースを細かく定義できます。

Kafkaのカスタムリソースに簡易認証を使用するように設定します。authorizationセクションのtypeプロパティにsimpleという値を設定し、スーパーユーザーのリストを設定します。

アクセスルールは、ACLRule schema referenceで説明されているように、KafkaUserに対して設定されます。

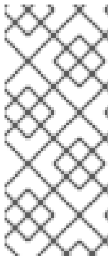
14.2.18.1. superUsers

スーパーユーザーとして扱われるユーザープリンシパルのリスト。このリストのユーザープリンシパルは、ACL ルールをクエリーしなくても常に許可されます。詳細は「Kafka の承認」を参照してください。

簡易承認の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
```

```
    type: simple
    superUsers:
      - CN=client_1
      - user_2
      - CN=client_3
  # ...
```



注記

Kafka.spec.kafka の config プロパティーにある super.user 設定オプションは無視されます。この代わりに、authorization プロパティーでスーパーユーザーを指定します。詳細は「Kafka ブローカーの設定」を参照してください。

14.2.18.2. KafkaAuthorizationSimpleスキーマのプロパティ

typeプロパティは、KafkaAuthorizationSimpleタイプの使用をKafkaAuthorizationOpaおよびKafkaAuthorizationKeycloak、KafkaAuthorizationCustomと区別するための識別子です。KafkaAuthorizationSimple タイプには simple の値が必要です。

プロパティー	説明
type string	**simple** でなければなりません。
superUsers 文字列の配列	スーパーユーザーの一覧。無制限のアクセス権を取得する必要のあるユーザープリンシパルの一覧が含まれなければなりません。

14.2.19. KafkaAuthorizationOpa スキーマ参照

KafkaClusterSpec で使用

KafkaAuthorizationOpaスキーマプロパティの全リスト

Open Policy Agentの認証を使用するには、authorizationセクションのtypeプロパティにopaという値を設定し、必要に応じてOPAのプロパティを構成します。AMQ Streamsは、BisnodeのKafka認可プラグインをOpen Policy Agentのオーソライザーとして使用しています。入力データのフォーマッ

トやポリシーの例については、Open Policy Agent plugin for Kafka authorizationを参照してください。

14.2.19.1. url

Open Policy Agent サーバーへの接続に使用される URL。URL には、オーソライザーによってクエリーされるポリシーが含まれる必要があります。必須。

14.2.19.2. allowOnError

一時的に利用できない場合など、オーソライザーによる Open Policy Agent へのクエリーが失敗した場合に、デフォルトで Kafka クライアントを許可または拒否するかどうかを定義します。デフォルトは false で、すべてのアクションが拒否されます。

14.2.19.3. initialCacheCapacity

すべてのリクエストに対して Open Policy Agent をクエリーしないようにするために、オーソライザーによって使用されるローカルキャッシュの初期容量。デフォルトは 5000 です。

14.2.19.4. maximumCacheSize

すべてのリクエストに対して Open Policy Agent をクエリーしないようにするために、オーソライザーによって使用されるローカルキャッシュの最大容量。デフォルトは 50000 です。

14.2.19.5. expireAfterMs

すべてのリクエストに対して Open Policy Agent をクエリーしないようにするために、ローカルキャッシュに保持されるレコードの有効期限。キャッシュされた承認決定が Open Policy Agent サーバーからリロードされる頻度を定義します。ミリ秒単位です。デフォルトは 3600000 ミリ秒 (1 時間) です。

14.2.19.6. superUsers

スーパーユーザーとして扱われるユーザープリンシパルのリスト。このリストのユーザープリンシパルは、Open Policy Agent ポリシーをクエリーしなくても常に許可されます。詳細は「Kafka の承認」を参照してください。

Open Policy Agent オーソライザーの設定例

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: opa
      url: http://opa:8181/v1/data/kafka/allow
      allowOnError: false
      initialCacheCapacity: 1000
      maximumCacheSize: 10000
      expireAfterMs: 60000
      superUsers:
        - CN=fred
        - sam
        - CN=edward
    # ...
```

14.2.19.7. KafkaAuthorizationOpaスキーマのプロパティ

typeプロパティは、KafkaAuthorizationOpaタイプの使用をKafkaAuthorizationSimple、KafkaAuthorizationKeycloak、KafkaAuthorizationCustomと区別するための識別子です。KafkaAuthorizationOpa タイプには opa の値が必要です。

プロパティー	説明
type 文字列	**opa** でなければなりません。
url 文字列	Open Policy Agent サーバーへの接続に使用される URL。URL には、オーソライザーによってクエリーされるポリシーが含まれる必要があります。このオプションは必須です。
allowOnError boolean	一時的に利用できない場合など、オーソライザーによる Open Policy Agent へのクエリーが失敗した場合に、デフォルトで Kafka クライアントを許可または拒否するかどうかを定義します。デフォルトは **false** で、すべてのアクションが拒否されます。

プロパティー	説明
initialCacheCapacity	すべてのリクエストに対して Open Policy Agent をクエリーしないようにするために、オーソライザーによって使用されるローカルキャッシュの初期容量。デフォルトは **5000** です。
integer	
maximumCacheSize	すべてのリクエストに対して Open Policy Agent をクエリーしないようにするために、オーソライザーによって使用されるローカルキャッシュの最大容量。デフォルトは **50000** です。
integer	
expireAfterMs	すべてのリクエストに対して Open Policy Agent をクエリーしないようにするために、ローカルキャッシュに保持されるレコードの有効期限。キャッシュされた承認決定が Open Policy Agent サーバーからリロードされる頻度を定義します。ミリ秒単位です。デフォルトは **3600000** です。
integer	
superUsers	スーパーユーザーのリスト。これは、無制限のアクセス権限を持つユーザープリンシパルのリストです。
文字列の配列	

14.2.20. KafkaAuthorizationKeycloak スキーマ参照

KafkaClusterSpec で使用

typeプロパティは、KafkaAuthorizationKeycloakタイプの使用をKafkaAuthorizationSimple、KafkaAuthorizationOpa、KafkaAuthorizationCustomと区別するための識別子です。KafkaAuthorizationKeycloak タイプには keycloak の値が必要です。

プロパティー	説明
type	**keycloak** でなければなりません。
string	
clientId	Kafka クライアントが OAuth サーバーに対する認証に使用し、トークンエンドポイント URI を使用することができる OAuth クライアント ID。
string	
tokenEndpointUri	承認サーバートークンエンドポイント URI。
string	

プロパティー	説明
tlsTrustedCertificates	OAuth サーバーへの TLS 接続の信頼済み証明書。
CertSecretSource array	
disableTlsHostnameVerification	TLS ホスト名の検証を有効または無効にします。デフォルト値は **false** です。
boolean	
delegateToKafkaAcls	Red Hat Single Sign-On の Authorization Services ポリシーにより DENIED となった場合に、承認の決定を 'Simple' オーソライザーに委譲すべきかどうか。デフォルト値は **false** です。
boolean	
grantsRefreshPeriodSeconds	連続する付与 (Grants) 更新実行の間隔 (秒単位)。デフォルト値は 60 です。
integer	
grantsRefreshPoolSize	アクティブなセッションの付与(Grants) の更新に使用するスレッドの数。スレッドが多いほど並列処理多くなるため、ジョブがより早く完了します。ただし、使用するスレッドが多いほど、承認サーバーの負荷が大きくなります。デフォルト値は 5 です。
integer	
superUsers	スーパーユーザーの一覧。無制限のアクセス権を取得する必要のあるユーザープリンシパルの一覧が含まれなければなりません。
文字列の配列	

14.2.21. KafkaAuthorizationCustomスキーマリファレンス

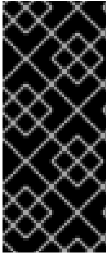
KafkaClusterSpec で使用

KafkaAuthorizationCustomスキーマプロパティの全リスト

AMQ Streamsでカスタム認証を使用するには、独自のAuthorizerプラグインを設定して、アクセスコントロールリスト（ACLs）を定義します。

ACL を使用すると、ユーザーがアクセスできるリソースを細かく定義できます。

Kafkaのカスタムリソースにカスタム認証を使用するように設定します。authorizationセクションのtypeプロパティに値customを設定し、以下のプロパティを設定します。



重要

カスタムオーソライザーは、org.apache.kafka.server.authorizer.Authorizerインターフェースを実装し、super.users 設定プロパティを使用してsuper.usersの設定をサポートする必要があります。

14.2.21.1. authorizerClass

(必須) カスタム ACL をサポートするためのorg.apache.kafka.server.authorizer.Authorizerインターフェースを実装した Java クラスです。

14.2.21.2. superUsers

スーパーユーザーとして扱われるユーザープリンシパルのリスト。このリストのユーザープリンシパルは、ACL ルールをクエリーしなくても常に許可されます。詳細は「Kafka の承認」を参照してください。

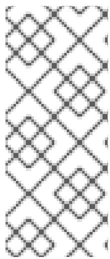
Kafka.spec.kafka.configを使って、カスタムオーサライザーを初期化するための設定を追加することができます。

Kafka.specでのカスタム認証設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: custom
      authorizerClass: io.mycompany.CustomAuthorizer
      superUsers:
        - CN=client_1
        - user_2
        - CN=client_3
    # ...
    config:
      authorization.custom.property1=value1
      authorization.custom.property2=value2
    # ...
```

Kafkaカスタムリソースの設定に加えて、カスタムオーソライザークラスとその依存関係を含むJAR ファイルが Kafka ブローカーのクラスパス上で利用可能である必要があります。

AMQ StreamsのMavenビルドプロセスでは、docker-images/kafka/kafka-thirdparty-libsディレクトリの下にあるpom.xmlファイルに依存関係として追加することで、生成されたKafkaブローカーコンテナイメージにカスタムサードパーティライブラリを追加する仕組みがあります。ディレクトリーには、Kafka のバージョンごとに異なるフォルダーが含まれています。適切なフォルダーを選択します。pom.xmlファイルを修正する前に、サードパーティのライブラリがMavenリポジトリで利用可能であり、そのMavenリポジトリがAMQ Streamsのビルドプロセスからアクセス可能である必要があります。



注記

Kafka.spec.kafka の config プロパティーにある super.user 設定オプションは無視されます。この代わりに、authorization プロパティーでスーパーユーザーを指定します。詳細は「Kafka ブローカーの設定」を参照してください。

14.2.21.3. KafkaAuthorizationCustomスキーマのプロパティ

typeプロパティは、KafkaAuthorizationCustomタイプの使用をKafkaAuthorizationSimple、KafkaAuthorizationOpa、KafkaAuthorizationKeycloakと区別する識別子です。タイプKafkaAuthorizationCustomの値がcustomである必要があります。

プロパティー	説明
type 文字列	**custom**である必要があります。
authorizerClass 文字列	認証実装クラス。クラスパスで使用できる必要があります。
superUsers 文字列の配列	スーパーユーザーのリスト。これは、無制限のアクセス権限を持つユーザープリンシパルです。
supportsAdminApi	カスタムオーソライザーが、Kafka Admin APIを使用してACLを管理するためのAPIをサポートしているかどうかを示します。デフォルトは **false** です。

プロパティー	説明
boolean	

14.2.22. Rack スキーマ参照

KafkaClusterSpec、KafkaConnectSpecで使用されます。

Rackスキーマプロパティの全リスト

rackオプションは、ラックの認識を設定します。ラックは、アベイラビリティーゾーン、データセンター、またはデータセンターの実際のラックを表すことができます。rackの設定は、topologyKeyで行います。topologyKeyは、OpenShift ノード上のラベルを識別するもので、その値にはトポロジーの名前が含まれています。このようなラベルの例としては、topology.kubernetes.io/zone（古いOpenShiftバージョンではfailure-domain.beta.kubernetes.io/zone）があり、これにはOpenShiftノードが実行されているアベイラビリティゾーンの名前が含まれています。Kafka クラスターが実行するラックを認識するように設定し、パーティションレプリカを異なるラックに分散したり、最も近いレプリカからのメッセージの消費したりするなどの追加機能を有効にできます。

OpenShift ノードラベルの詳細は、「Well-Known Labels, Annotations and Taints」を参照してください。ノードがデプロイされたゾーンやラックを表すノードラベルについては、OpenShift 管理者に相談します。

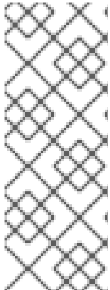
14.2.22.1. ラック間でのパーティションレプリカの分散

ラックアウェアネスを設定すると、AMQ Streamsは各Kafkaブローカーのbroker.rack設定を行います。broker.rackの設定では、各ブローカにラックIDを割り当てます。broker.rackを設定すると、Kafkaブローカーはパーティションレプリカをできるだけ多くの異なるラックに分散して配置します。レプリカが複数のラックに分散されている場合、複数のレプリカが同時に失敗する可能性は、同じラックにある場合よりも低くなります。レプリカを分散すると回復性が向上し、可用性と信頼性にとっても重要です。Kafkaでラックアウェアネスを有効にするには、以下の例のように、Kafkaのカスタムリソースの.spec.kafkaセクションにrackオプションを追加します。

Kafkaのrack設定例

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

```
# ...
rack:
  topologyKey: topology.kubernetes.io/zone
# ...
```



注記

Pod が削除または再起動すると、ブローカーが実行されているラックは、変更されることがあります。その結果、異なるラックで実行しているレプリカが、同じラックを共有する可能性があります。RackAwareGoalでCruise ControlとKafkaRebalanceリソースを使用して、レプリカが異なるラックに分散していることを確認します。

Kafkaカスタムリソースでラックアウェアネスが有効になっている場合、AMQ Streamsは自動的にOpenShiftのpreferredDuringSchedulingIgnoredDuringExecutionアフィニティルールを追加して、Kafkaブローカーを異なるラックに分散させます。ただし、優先 ルールは、ブローカーが分散されることを保証しません。OpenShiftとKafkaの構成に応じて、affinityルールを追加したり、ZooKeeperとKafkaの両方にtopologySpreadConstraintsを設定したりして、できるだけ多くのラックにノードが適切に分散されるようにしてください。詳細は、「Pod スケジューリングの設定」 を参照してください。

14.2.22.2. 最も近いレプリカからのメッセージの消費

ラックアウェアネスをコンシューマーで使用して、最も近いレプリカからデータを取得することもできます。これは、Kafka クラスターが複数のデータセンターにまたがる場合に、ネットワークの負荷を軽減するのに役立ちます。また、パブリッククラウドで Kafka を実行する場合にコストを削減することもできます。ただし、レイテンシーが増加する可能性があります。

最も近いレプリカから利用するためには、Kafkaクラスターでラックアウェアが設定されており、RackAwareReplicaSelectorが有効になっている必要があります。レプリカセレクタープラグインは、クライアントが最も近いレプリカから消費できるようにするロジックを提供します。デフォルトの実装では、LeaderSelectorを使って、常にクライアントのリーダーレプリカを選択します。replica.selector.classにRackAwareReplicaSelectorを指定すると、デフォルトの実装から切り替わります。

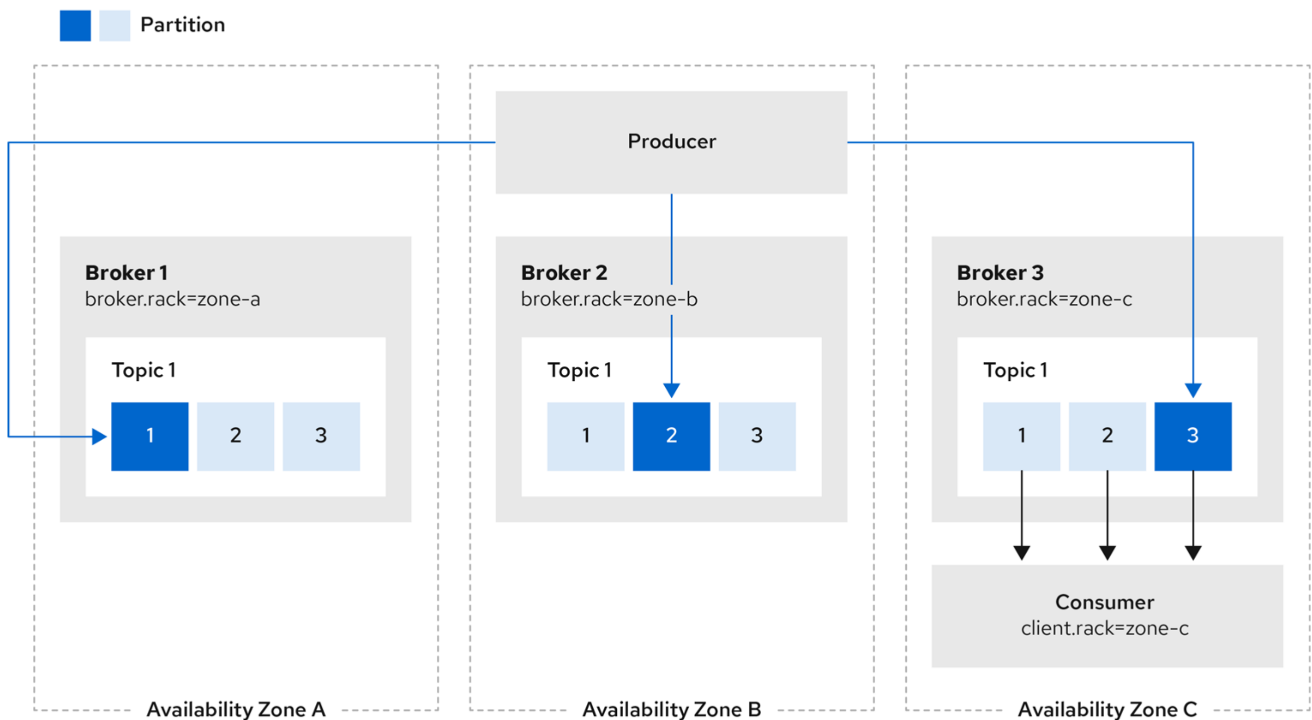
レプリカ対応セレクタを有効にしたrack構成例

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

```
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    config:
      # ...
      replica.selector.class: org.apache.kafka.common.replica.RackAwareReplicaSelector
    # ...
```

Kafkaブローカーの設定に加えて、コンシューマーにclient.rackオプションを指定する必要があります。client.rackオプションには、コンシューマーが稼動しているrack IDを指定する必要があります。RackAwareReplicaSelectorは、マッチングしたbroker.rackとclient.rackIDを関連付けて、最も近いレプリカを見つけ、そこからデータを取得します。同じラック内に複数のレプリカがある場合、RackAwareReplicaSelectorは常に最新のレプリカを選択します。ラック ID が指定されていない場合や、同じラック ID を持つレプリカが見つからない場合は、リーダーレプリカにフォールバックします。

図14.1 同じアベイラビリティーゾーンのレプリカから消費するクライアントの例



最も近いレプリカからメッセージを消費することは、メッセージを消費しているシンクコネクターの Kafka Connect でも行えます。AMQ Streamsを使用してKafka Connectを展開する場合、KafkaConnectカスタムリソースのrackセクションを使用して、client.rackオプションを自動的に設定することができます。

Kafka Connectのrack設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
# ...
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    # ...
```

KafkaConnectのカスタムリソースでラックアウェアネスを有効にすると、アフィニティルールが設定されませんが、affinityやtopologySpreadConstraintsを設定することもできます。詳細は、「Pod スケジューリングの設定」 を参照してください。

14.2.22.3. Rackスキーマのプロパティ

プロパティー	説明
topologyKey 文字列	OpenShift クラスターノードに割り当てられたラベルに一致するキー。このラベルの値は、Kafka Connectのブローカーの**broker.rack**設定および**client.rack**に使用されます。

14.2.23. Probe スキーマ参照

CruiseControlSpec、EntityTopicOperatorSpec、EntityUserOperatorSpec、KafkaBridgeSpec、KafkaClusterSpec、KafkaConnectSpec、KafkaExporterSpec、KafkaMirrorMaker2Spec、KafkaMirrorMakerSpec、TlsSidecar、ZookeeperClusterSpec で使用されています。

プロパティー	説明
failureThreshold integer	正常に実行された後に失敗とみなされるプローブの連続失敗回数の最小値。デフォルトは 3 です。最小値は 1 です。
initialDelaySeconds	最初に健全性をチェックするまでの初期の遅延。デフォルトは 15 秒です。最小値は 0 です。

プロパティー	説明
integer	
periodSeconds	プローブを実行する頻度 (秒単位)。デフォルトは 10 秒です。最小値は 1 です。
integer	
successThreshold	失敗後に、プローブが正常とみなされるための最小の連続成功回数。デフォルトは 1 です。liveness は 1 でなければなりません。最小値は 1 です。
integer	
timeoutSeconds	ヘルスチェック試行のタイムアウト。デフォルトは 5 秒です。最小値は 1 です。
integer	

14.2.24. JvmOptions スキーマ参照

CruiseControlSpec、EntityTopicOperatorSpec、EntityUserOperatorSpec、KafkaBridgeSpec、KafkaClusterSpec、KafkaConnectSpec、KafkaMirrorMaker2Spec、KafkaMirrorMakerSpec、ZookeeperClusterSpec で使用されています。

プロパティー	説明
-XX	JVM への -XX オプションのマップ。
map	
-Xms	JVM への -Xms オプション。
文字列	
-Xmx	JVM への -Xmx オプション。
文字列	
gcLoggingEnabled	ガベージコレクションのロギングが有効かどうかを指定します。デフォルトは false です。
boolean	
javaSystemProperties	**-D** オプションを使用して、JVM に渡される追加のシステムプロパティーのマップ。
SystemProperty array	

14.2.25. SystemProperty スキーマ参照

JvmOptions で使用

プロパティー	説明
name	システムプロパティー名。
string	
value	システムプロパティーの値。
string	

14.2.26. KafkaJmxOptions スキーマ参照

KafkaClusterSpec、KafkaConnectSpec、KafkaMirrorMaker2Spec、ZookeeperClusterSpec で使用されます。

KafkaJmxOptionsスキーマプロパティの全リスト

JMX 接続オプションを設定します。

Kafka ブローカー、Zookeeper ノード、Kafka Connect、MirrorMaker 2.0 からは、9999 の JMX ポートを開くことで JMX メトリクスを取得できます。パスワードで保護されたJMXポート、または保護されていないJMXポートを設定するには、jmxOptionsプロパティを使用します。パスワードで保護すると、未許可の Pod によるポートへの不正アクセスを防ぐことができます。

その後、コンポーネントに関するメトリクスを取得できます。

たとえば、Kafka ブローカーごとに、クライアントからのバイト/秒の使用度データや、ブローカーのネットワークの要求レートを取得することができます。

JMXポートのセキュリティを有効にするには、authenticationフィールドのtypeパラメータをpasswordに設定します。

Kafka ブローカーと Zookeeper ノード用のパスワードで保護された JMX 設定の例

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    jmxOptions:
      authentication:
        type: "password"
    # ...
  zookeeper:
    # ...
    jmxOptions:
      authentication:
        type: "password"
    #...
```

次に、対応するブローカーを指定して、Pod をクラスターにデプロイし、ヘッドレスサービスを使用して JMX メトリクスを取得できます。

たとえば、ブローカー 0 から JMX メトリクスを取得するには、以下を指定します。

```
"CLUSTER-NAME-kafka-0.CLUSTER-NAME-kafka-brokers"
```

CLUSTER-NAME-kafka-0 はブローカーポッドの名前、CLUSTER-NAME-kafka-brokers はブローカーポッドのIPを返すヘッドレスサービスの名前です。

JMX ポートがセキュアである場合、Pod のデプロイメントで JMX Secret からユーザー名とパスワードを参照すると、そのユーザー名とパスワードを取得できます。

保護されていないJMXポートの場合は、空のオブジェクト{}を使用して、ヘッドレスサービスのJMXポートを開きます。保護されたポートと同じ方法で Pod をデプロイし、メトリクスを取得できますが、この場合はどの Pod も JMX ポートから読み取ることができます。

Kafka ブローカーと Zookeeper ノードのオープンポート JMX 構成例

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

関連情報

- JMX を使用して公開される Kafka コンポーネントメトリクスの詳細は、Apache Kafka のドキュメントを参照してください。

14.2.26.1. KafkaJmxOptionsスキーマプロパティ

プロパティー	説明
authentication	JMX ポートに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[password] の 1 つでなければなりません。
KafkaJmxAuthenticationPassword	

14.2.27. KafkaJmxAuthenticationPassword スキーマ参照

KafkaJmxOptions で使用

typeプロパティは、KafkaJmxAuthenticationPasswordタイプの使用と、今後追加される可能性のある他のサブタイプとを区別するための識別情報です。KafkaJmxAuthenticationPassword タイプには password の値が必要です。

プロパティー	説明
type	**password** でなければなりません。
string	

14.2.28. JmxPrometheusExporterMetrics schema reference

CruiseControlSpec、**KafkaClusterSpec**、**KafkaConnectSpec**、**KafkaMirrorMaker2Spec**、**KafkaMirrorMakerSpec**、**ZookeeperClusterSpec** で使用されています。

typeプロパティは、JmxPrometheusExporterMetricsタイプの使用と、将来追加される可能性のある他のサブタイプとを区別する識別子です。JmxPrometheusExporterMetricsタイプの値jmxPrometheusExporterを持つ必要があります。

プロパティー	説明
type string	**jmxPrometheusExporter**でなければなりません。
valueFrom **ExternalConfigurationReference**	Prometheus JMX Exporter 設定が保存される ConfigMap エントリー。この設定の構造に関する詳細は、「Prometheus JMX Exporter」を参照してください。

14.2.29. ExternalConfigurationReferenceのスキーマ参照

ExternalLoggingJmxPrometheusExporterMetrics で使用されています。

プロパティー	説明
configMapKeyRef ConfigMapKeySelector	設定が含まれる ConfigMap のキーへの参照。詳細は、core/v1 configmapkeyselector の外部ドキュメントを参照してください。

14.2.30. InlineLogging スキーマ参照

CruiseControlSpec、**EntityTopicOperatorSpec**、**EntityUserOperatorSpec**、**KafkaBridgeSpec**、**KafkaClusterSpec**、**KafkaConnectSpec**、**KafkaMirrorMaker2Spec**、**KafkaMirrorMakerSpec**、**ZookeeperClusterSpec** で使用されています。

typeプロパティは、InlineLoggingタイプの使用と、ExternalLogging.を区別するための識別子です。InlineLogging タイプには inline の値が必要です。

プロパティー	説明
type	**inline** でなければなりません。
string	
loggers	ロガー名からロガーレベルへのマップ。
map	

14.2.31. ExternalLogging スキーマ参照

CruiseControlSpec、EntityTopicOperatorSpec、EntityUserOperatorSpec、KafkaBridgeSpec、KafkaClusterSpec、KafkaConnectSpec、KafkaMirrorMaker2Spec、KafkaMirrorMakerSpec、ZookeeperClusterSpec で使用されています。

typeプロパティは、ExternalLoggingタイプの使用を、InlineLoggingと区別するための識別子です。ExternalLogging タイプには external の値が必要です。

プロパティー	説明
type	**external** でなければなりません。
string	
valueFrom	ロギング設定が保存される**ConfigMap**エントリです。
ExternalConfigurationReference	

14.2.32. KafkaClusterTemplate スキーマ参照

KafkaClusterSpec で使用

プロパティー	説明
statefulset	Kafka **StatefulSet** のテンプレート。
StatefulSetTemplate	
Pod	Kafka **Pod** のテンプレート。

プロパティー	説明
PodTemplate	
bootstrapService	Kafka ブートストラップ **Service** のテンプレート。
InternalServiceTemplate	
brokersService	Kafka ブローカー **Service** のテンプレート。
InternalServiceTemplate	
externalBootstrapService	Kafka 外部ブートストラップ **Service** のテンプレート。
ResourceTemplate	
perPodService	OpenShift の外部からアクセスするために使用される Pod ごとの Kafka **Services** のテンプレート。
ResourceTemplate	
externalBootstrapRoute	Kafka 外部ブートストラップ **Route** のテンプレート。
ResourceTemplate	
perPodRoute	OpenShift の外部からアクセスするために使用される Kafka の Pod ごとの **Routes** のテンプレート。
ResourceTemplate	
externalBootstrapIngress	Kafka 外部ブートストラップ **Ingress** のテンプレート。
ResourceTemplate	
perPodIngress	OpenShift の外部からアクセスするために使用される Kafka の Pod ごとの **Ingress** のテンプレート。
ResourceTemplate	
persistentVolumeClaim	すべての Kafka **PersistentVolumeClaims** のテンプレート。
ResourceTemplate	
podDisruptionBudget	Kafka **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	
kafkaContainer	Kafka ブローカーコンテナーのテンプレート。

プロパティー	説明
ContainerTemplate	
initContainer	Kafka init コンテナーのテンプレート。
ContainerTemplate	
clusterCaCert	Kafka Cluster 証明書の公開鍵が含まれる Secret のテンプレート。
ResourceTemplate	
serviceAccount	Kafka サービスアカウントのテンプレート。
ResourceTemplate	
jmxSecret	Kafka Cluster JMX認証のSecretのテンプレートです。
ResourceTemplate	
clusterRoleBinding	Kafka ClusterRoleBinding のテンプレート。
ResourceTemplate	

14.2.33. StatefulSetTemplate スキーマ参照

KafkaClusterTemplate、**ZookeeperClusterTemplate** で使用

プロパティー	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	
podManagementPolicy	この StatefulSet に使用される PodManagementPolicy。有効な値は **Parallel** および **OrderedReady** です。デフォルトは **Parallel** です。
string ([OrderedReady、Parallel] のいずれか)	

14.2.34. MetadataTemplate スキーマ参照

BuildConfigTemplate、**DeploymentTemplate**、**InternalServiceTemplate**、**Pod DisruptionBudgetTemplate**、**PodTemplate**、**ResourceTemplate**、**StatefulSetTemplate**で使用

MetadataTemplateスキーマプロパティの全リスト

Labels および Annotations は、リソースの識別および整理に使用され、metadata プロパティーで設定されます。

以下は例になります。

```
# ...
template:
  statefulset:
    metadata:
      labels:
        label1: value1
        label2: value2
      annotations:
        annotation1: value1
        annotation2: value2
# ...
```

labels および annotations フィールドには、予約された文字列 strimzi.io が含まれないすべてのラベルやアノテーションを含めることができます。strimzi.io が含まれるラベルやアノテーションは、内部で AMQ Streams によって使用され、設定することはできません。

14.2.34.1. MetadataTemplateスキーマのプロパティ

プロパティー	説明
labels	リソーステンプレートに追加されたラベル。**StatefulSets**、**Deployments**、**Pods**、**Services** などの異なるリソースに適用できます。
map	
annotations	リソーステンプレートに追加されたアノテーション。**StatefulSets**、**Deployments**、**Pods**、**Services** などの異なるリソースに適用できます。
map	

14.2.35. PodTemplate スキーマ参照

CruiseControlTemplate、EntityOperatorTemplate、KafkaBridgeTemplate、KafkaClusterTemplate、KafkaConnectTemplate、KafkaExporterTemplate、KafkaMirrorMakerTemplate、ZookeeperClusterTemplate で使用されます。

PodTemplateスキーマプロパティの全リスト

Kafka Pod のテンプレートを設定します。

PodTemplateの構成例

```
# ...
template:
  pod:
    metadata:
      labels:
        label1: value1
      annotations:
        anno1: value1
    imagePullSecrets:
      - name: my-docker-credentials
    securityContext:
      runAsUser: 1000001
      fsGroup: 0
    terminationGracePeriodSeconds: 120
# ...
```

14.2.35.1. hostAliases

hostAliasesプロパティを使用して、ポッドの/etc/hostsファイルに注入されるホストとIPアドレスのリストを指定します。

この設定は特に、クラスター外部の接続がユーザーによっても要求される場合に Kafka Connect または MirrorMaker で役立ちます。

hostAliasesの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
#...
spec:
  # ...
  template:
```

```
pod:
  hostAliases:
  - ip: "192.168.1.86"
    hostnames:
    - "my-host-1"
    - "my-host-2"
  #...
```

14.2.35.2. PodTemplateスキーマのプロパティ

プロパティー	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	
imagePullSecrets	この Pod で使用されるイメージのプルに使用する同じ namespace のシークレットへの参照の一覧です。Cluster Operatorの環境変数**STRIMZI_IMAGE_PULL_SECRETS**と**imagePullSecrets**オプションが指定されている場合、**imagePullSecrets**変数のみが使用され、**STRIMZI_IMAGE_PULL_SECRETS**変数は無視されます。詳細は、core/v1 localobjectreference の外部ドキュメント を参照してください。
LocalObjectReference array	
securityContext	Pod レベルのセキュリティー属性と共通のコンテナー設定を設定します。詳細は、core/v1 podsecuritycontext の外部ドキュメント を参照してください。
PodSecurityContext	
terminationGracePeriodSeconds	猶予期間とは、Pod で実行されているプロセスに終了シグナルが送信されてから、kill シグナルでプロセスを強制的に終了するまでの期間 (秒単位) です。この値は、プロセスの予想されるクリーンアップ時間よりも長く設定します。値は負の値ではない整数にする必要があります。値をゼロにすると、即座に削除されます。非常に大型な Kafka クラスターの場合は、正常終了期間を延長し、Kafka ブローカーの終了前に作業を別のブローカーに転送する時間を十分確保する必要があることがあります。デフォルトは 30 秒です。
integer	
affinity	Pod のアフィニティールール。詳細は、core/v1 affinity の外部ドキュメント を参照してください。
Affinity	

プロパティー	説明
tolerations Toleration array	Pod の許容 (Toleration)。詳細は、core/v1 toleration の外部ドキュメント を参照してください。
priorityClassName string	優先順位を Pod に割り当てるために使用される優先順位クラス (Priority Class) の名前。Priority Class (優先順位クラス) の詳細は、「Pod Priority and Preemption」を参照してください。
schedulerName string	この **Pod** のディスパッチに使用されるスケジューラーの名前。指定されていない場合、デフォルトのスケジューラーが使用されます。
hostAliases HostAlias array	Pod の HostAliases。HostAliases は、指定された場合に Pod の hosts ファイルに注入されるホストおよび IP のオプションのリストです。詳細は、core/v1 HostAlias の外部ドキュメント を参照してください。
tmpDirSizeLimit string	一時的な EmptyDir ボリューム(/**tmp**)に必要なローカルストレージの合計量（例： **1Gi**）。デフォルト値は **1Mi** です。
enableServiceLinks boolean	サービスについての情報を Pod の環境変数に注入するかどうかを示します。
topologySpreadConstraints TopologySpreadConstraint array	Pod のトポロジー分散制約。詳細は、core/v1 topologyspreadconstraint の外部ドキュメント を参照してください。

14.2.36. InternalServiceTemplateのスキーマ参照

CruiseControlTemplate、KafkaBridgeTemplate、KafkaClusterTemplate、KafkaConnectTemplate、ZookeeperClusterTemplate で使用されています。

プロパティー	説明
metadata **MetadataTemplate**	リソースに適用済みのメタデータ。

プロパティー	説明
ipFamilyPolicy	サービスによって使用される IP Family Policy を指定します。利用可能なオプションは、**SingleStack**、**PreferDualStack**、**RequireDualStack**です。**SingleStack**は単一のIPファミリー用です。**PreferDualStack**は、デュアルスタック構成のクラスターでは2つのIPファミリーを、シングルスタック構成のクラスターでは1つのIPファミリーを対象としています。**RequireDualStack**は、デュアルスタック構成のクラスターに2つのIPファミリーがないと失敗します。指定されていない場合、OpenShift はサービスタイプに基づいてデフォルト値を選択します。OpenShift 1.20 以降で利用できます。
string ([RequireDualStack、SingleStack、PreferDualStack] のいずれか)	
ipFamilies	サービスによって使用される IP Families を指定します。利用可能なオプションは、**IPv4**と**IPv6**です。**指定されていない場合、OpenShift は `ipFamilyPolicy** の設定に基づいてデフォルト値を選択します。OpenShift 1.20 以降で利用できます。
string ([IPv6, IPv4] の 1 つ以上) array	

14.2.37. ResourceTemplate スキーマ参照

CruiseControlTemplate, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **KafkaUserTemplate**, **ZookeeperClusterTemplate** で使用されています。

プロパティー	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	

14.2.38. PodDisruptionBudgetTemplate スキーマ参照

CruiseControlTemplate、**KafkaBridgeTemplate**、**KafkaClusterTemplate**、**KafkaConnectTemplate**、**KafkaMirrorMakerTemplate**、**ZookeeperClusterTemplate** で使用されます。

PodDisruptionBudgetTemplateスキーマプロパティの全リスト

AMQ Streamsは、新しいStatefulSetやDeploymentごとにPodDisruptionBudgetを作成します。デフォルトでは、Pod の Disruption Budget (停止状態の予算) は単一の Pod を指定時に利用不可能にすることのみ許可します。maxUnavailable プロパティーのデフォルト値を変更することで、許容される利用不可能な Pod の数を増やすことができます。

PodDisruptionBudgetのテンプレートの一例です。

```
# ...
template:
  podDisruptionBudget:
    metadata:
      labels:
        key1: label1
        key2: label2
      annotations:
        key1: label1
        key2: label2
    maxUnavailable: 1
# ...
```

14.2.38.1. PodDisruptionBudgetTemplate schema properties

プロパティー	説明
metadata	**PodDisruptionBudgetTemplate** リソースに適用するメタデータ。
MetadataTemplate	
maxUnavailable	自動 Pod エビクションを許可するための利用不可能な Pod の最大数。Pod エビクションは、**maxUnavailable** の Pod 数またはそれより少ない Pod 数がエビクション後に利用できない場合に許可されます。この値を 0 に設定するとすべての自発的なエビクションを阻止するため、Pod を手動でエビクトする必要があります。デフォルトは 1 です。
integer	

14.2.39. ContainerTemplate スキーマ参照

CruiseControlTemplate、EntityOperatorTemplate、KafkaBridgeTemplate、KafkaClusterTemplate、KafkaConnectTemplate、KafkaExporterTemplate、KafkaMirrorMakerTemplate、ZookeeperClusterTemplate で使用されます。

ContainerTemplateスキーマプロパティの全リスト

コンテナーのカスタムのセキュリティーコンテキストおよび環境変数を設定できます。

環境変数は、env プロパティーで name および value フィールドのあるオブジェクトのリストとして定義されます。以下の例は、Kafka ブローカーコンテナーに設定された 2 つのカスタム環境変数と 1 つのセキュリティーコンテキストを示しています。

```
# ...
template:
  kafkaContainer:
    env:
    - name: EXAMPLE_ENV_1
      value: example.env.one
    - name: EXAMPLE_ENV_2
      value: example.env.two
    securityContext:
      runAsUser: 2000
# ...
```

KAFKA_ で始まる環境変数は AMQ Streams 内部となるため、使用しないようにしてください。AMQ Streams によってすでに使用されているカスタム環境変数を設定すると、その環境変数は無視され、警告がログに記録されます。

14.2.39.1. ContainerTemplateスキーマのプロパティ

プロパティー	説明
env ContainerEnvVar array	コンテナーに適用する必要のある環境変数。
securityContext SecurityContext	コンテナーのセキュリティーコンテキスト。詳細は、core/v1 securitycontext の外部ドキュメント を参照してください。

14.2.40. ContainerEnvVar スキーマ参照

ContainerTemplate で使用

プロパティー	説明
name	環境変数のキー。
string	
value	環境変数の値。
string	

14.2.41. ZookeeperClusterSpec スキーマ参照

KafkaSpec で使用

ZookeeperClusterSpecスキーマプロパティの全リスト

ZooKeeper クラスターを設定します。

14.2.41.1. 設定

configプロパティを使用して、ZooKeeperのオプションをキーとして設定します。

標準の Apache ZooKeeper 設定が提供されることがあり、AMQ Streams によって直接管理されないプロパティーに限定されます。

以下に関連する設定オプションは設定できません。

- セキュリティー (暗号化、認証、および承認)
- リスナーの設定
- データディレクトリーの設定

- ZooKeeper クラスターの構成

値は以下の JSON タイプのいずれかになります。

- 文字列
- 数値
- ブール値

AMQ Streams で直接管理されるオプション以外の、ZooKeeper ドキュメント に記載されているオプションを指定および設定できます。以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはすべて禁止されています。

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

禁止されているオプションが config プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。サポートされるその他すべてのオプ

ションは ZooKeeper に渡されます。

禁止されているオプションには例外があります。TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。

ZooKeeper の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
  zookeeper:
    # ...
    config:
      autopurge.snapRetainCount: 3
      autopurge.purgeInterval: 1
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
    # ...
```

14.2.41.2. ログ

ZooKeeper には設定可能なロガーがあります。

- zookeeper.root.logger

ZooKeeperは、Apachelog4jのロガー実装を使用しています。

logging プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は log4j.properties を使用して記述されます。logging.valueFrom.configMapKeyRef.name および

logging.valueFrom.configMapKeyRef.key プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、「Apache logging services」を参照してください。

ここで、inline および external ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  zookeeper:
    # ...
    logging:
      type: inline
      loggers:
        zookeeper.root.logger: "INFO"
    # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  zookeeper:
    # ...
    logging:
      type: external
      valueFrom:
        configMapKeyRef:
          name: customConfigMap
          key: zookeeper-log4j.properties
  # ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは jvmOptions プロパティーを使用して 有効（または無効）にすることもできます。

14.2.41.3. ZookeeperClusterSpec schema properties

プロパティー	説明
replicas	クラスター内の Pod 数。
integer	
image	Pod の Docker イメージ。
string	
storage	ストレージの設定 (ディスク)。更新はできません。タイプは、指定のオブジェクト内の **storage.type** プロパティーの値によって異なり、[ephemeral、persistent-claim] のいずれかでなければなりません。
EphemeralStorage、PersistentClaimStorage	
config	ZooKeeper ブローカーの設定。次の接頭辞のあるプロパティーは設定できません: server.、 dataDir、dataLogDir、clientPort、authProvider、quorum.auth、requireClientAuthScheme、snapshot.trust.empty、standaloneEnabled、reconfigEnabled、4lw.commands.whitelist、secureClientPort、ssl、serverCnxnFactory、sslQuorum (次の例外を除く: ssl.protocol、ssl.quorum.protocol、ssl.enabledProtocols、ssl.quorum.enabledProtocols、ssl.ciphersuites、ssl.quorum.ciphersuites、ssl.hostnameVerification、ssl.quorum.hostnameVerification)
map	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	
jvmOptions	Pod の JVM オプション。
JvmOptions	
jmxOptions	Zookeeper ノードの JMX オプション。

プロパティー	説明
KafkaJmxOptions	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
metricsConfig	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
JmxPrometheusExporterMetrics	
logging	ZooKeeper のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
template	ZooKeeper クラスターリソースのテンプレート。ユーザーはテンプレートにより、**StatefulSet**、**Pod**、および **Service** の生成方法を指定できます。
ZookeeperClusterTemplate	

14.2.42. ZookeeperClusterTemplate スキーマ参照

ZookeeperClusterSpec で使用

プロパティー	説明
statefulset	ZooKeeper **StatefulSet** のテンプレート。
StatefulSetTemplate	
Pod	ZooKeeper **Pod** のテンプレート。
PodTemplate	
clientService	ZooKeeper クライアント **Service** のテンプレート。
InternalServiceTemplate	
nodesService	ZooKeeper ノード **Service** のテンプレート。
InternalServiceTemplate	

プロパティー	説明
persistentVolumeClaim	すべての ZooKeeper **PersistentVolumeClaims** のテンプレート。
ResourceTemplate	
podDisruptionBudget	ZooKeeper **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	
zookeeperContainer	ZooKeeper コンテナーのテンプレート。
ContainerTemplate	
serviceAccount	ZooKeeper サービスアカウントのテンプレート。
ResourceTemplate	
jmxSecret	Zookeeper Cluster JMX 認証の Secret のテンプレート。
ResourceTemplate	

14.2.43. EntityOperatorSpec スキーマ参照

KafkaSpec で使用

プロパティー	説明
topicOperator	Topic Operator の設定。
EntityTopicOperatorSpec	
userOperator	User Operator の設定。
EntityUserOperatorSpec	
tlsSidecar	TLS サイドカーの設定。
TlsSidecar	
template	Entity Operator リソースのテンプレート。ユーザーはテンプレートにより、**Deployment** および **Pod** の生成方法を指定できます。
EntityOperatorTemplate	

14.2.44. EntityTopicOperatorSpec スキーマ参照

EntityOperatorSpec で使用

EntityTopicOperatorSpecスキーマプロパティの全リスト

Topic Operator を設定します。

14.2.44.1. ログ

Topic Operator には設定可能なロガーがあります。

- rootLogger.level

Topic Operatorでは、Apachelog4j2のロガー実装を使用しています。

KafkaリソースKafkaリソースのentityOperator.topicOperatorフィールドのloggingプロパティを使用して、ロガーとロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は log4j2.properties を使用して記述されます。logging.valueFrom.configMapKeyRef.name および logging.valueFrom.configMapKeyRef.key プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、「Apache logging services」を参照してください。

ここで、inline および external ロギングの例を示します。

inline ロギング

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

```
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: external
        valueFrom:
          configMapKeyRef:
            name: customConfigMap
            key: topic-operator-log4j2.properties
  # ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは jvmOptions プロパティーを使用して 有効（または無効）にすることもできます。

14.2.44.2. EntityTopicOperatorSpec schema properties

プロパティー	説明
watchedNamespace	Topic Operator が監視する必要のある namespace。
string	
image	Topic Operator に使用するイメージ。
string	
reconciliationIntervalSeconds	定期的な調整の間隔。
integer	
zookeeperSessionTimeoutSeconds	ZooKeeper セッションのタイムアウト。
integer	
startupProbe	Pod の起動チェック。
Probe	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
topicMetadataMaxAttempts	トピックメタデータの取得を試行する回数。
integer	

プロパティー	説明
ログ	ロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、ExternalLogging	
jvmOptions	Pod の JVM オプション。
JvmOptions	

14.2.45. EntityUserOperatorSpec スキーマ参照

EntityOperatorSpec で使用

EntityUserOperatorSpecスキーマプロパティの全リスト

User Operator を設定します。

14.2.45.1. ログ

User Operator には設定可能なロガーがあります。

- rootLogger.level

User Operatorでは、Apachelog4j2のロガー実装を使用しています。

KafkaリソースのentityOperator.userOperatorフィールドのloggingプロパティを使用して、ロガーとロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は log4j2.properties を使用して記述されます。logging.valueFrom.configMapKeyRef.name および logging.valueFrom.configMapKeyRef.key プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロ

ギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、「Apache logging services」を参照してください。

ここで、inline および external ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
```

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: customConfigMap
      key: user-operator-log4j2.properties
# ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは jvmOptions プロパティーを使用して 有効（または無効）にすることもできます。

14.2.45.2. EntityUserOperatorSpec schema properties

プロパティー	説明
watchedNamespace	User Operator が監視する必要のある namespace。
string	
image	User Operator に使用するイメージ。
string	
reconciliationIntervalSeconds	定期的な調整の間隔。
integer	
zookeeperSessionTimeoutSeconds	**zookeeperSessionTimeoutSecondsプロパティは非推奨となりました。** ZooKeeperは、User Operatorで使用されなくなったため、このプロパティは非推奨となりました。ZooKeeper セッションのタイムアウト。
integer	
secretPrefix	KafkaUser 名に追加され、Secret 名として使用されるプレフィックス。
string	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。

プロパティー	説明
Probe	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
ログ	ロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
jvmOptions	Pod の JVM オプション。
JvmOptions	

14.2.46. TlsSidecar スキーマ参照

CruiseControlSpec、**EntityOperatorSpec** で使用されています。

TlsSidecarスキーマプロパティの全リスト

Pod で実行されるコンテナーである TLS サイドカーを設定しますが、サポートの目的で提供されます。。AMQ Streams では、TLS サイドカーは TLS を使用して、コンポーネントと ZooKeeper との間の通信を暗号化および復号化します。

TLS サイドカーは以下で使用されます。

- **Entitiy Operator**
- **Cruise Control**

TLSサイドカーの設定には、tlsSidecarプロパティが使用されます。

- **Kafka.spec.entityOperator**

- Kafka.spec.cruiseControl

TLS サイドカーは、以下の追加オプションをサポートします。

- image
- resources
- logLevel
- readinessProbe
- livenessProbe

resourcesプロパティは、TLSサイドカーに割り当てられたメモリとCPUのリソースを指定します。

imageプロパティは、使用されるコンテナイメージを設定します。

readinessProbeプロパティとlivenessProbeプロパティは、TLSサイドカーのhealthcheck プローブを構成します。

logLevelプロパティは、ロギングレベルを指定します。以下のログレベルがサポートされます。

- emerg
- alert
- crit

- err
- warning
- notice
- info
- debug

デフォルト値は notice です。

TLS サイドカーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  entityOperator:
    # ...
    tlsSidecar:
      resources:
        requests:
          cpu: 200m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
    # ...
  cruiseControl:
    # ...
    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
  # ...
```

14.2.46.1. TlsSidecarスキーマのプロパティ

プロパティー	説明
image string	コンテナーの Docker イメージ。
livenessProbe **Probe**	Pod の liveness チェック。
logLevel string ([emerg、debug、crit、err、alert、warning、notice、info] のいずれか)	TLS サイドカーのログレベル。デフォルト値は **notice** です。
readinessProbe **Probe**	Pod の readiness チェック。
resources ResourceRequirements	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。

14.2.47. EntityOperatorTemplate スキーマ参照

EntityOperatorSpec で使用

プロパティー	説明
デプロイメント	Entity Operator **Deployment** のテンプレート。

プロパティー	説明
ResourceTemplate	
pod	Entity Operator **Pod** のテンプレート。
PodTemplate	
topicOperatorContainer	Entity Topic Operator コンテナーのテンプレート。
ContainerTemplate	
userOperatorContainer	Entity User Operator コンテナーのテンプレート。
ContainerTemplate	
tlsSidecarContainer	Entity Operator TLS サイドカーコンテナーのテンプレート。
ContainerTemplate	
serviceAccount	Entity Operator サービスアカウントのテンプレート。
ResourceTemplate	

14.2.48. CertificateAuthority スキーマ参照

KafkaSpec で使用されます。

TLS 証明書のクラスター内での使用方法の設定。これは、クラスター内の内部通信に使用される証明書および Kafka.spec.kafka.listeners.tls を介したクライアントアクセスに使用される証明書の両方に適用されます。

プロパティー	説明
generateCertificateAuthority	true の場合、認証局の証明書が自動的に生成されます。それ以外の場合は、ユーザーは CA 証明書で Secret を提供する必要があります。デフォルトは true です。
boolean	

プロパティー	説明
generateSecretOwnerReference	**true**の場合、クラスターとクライアントのCAシークレットは、**Kafka**リソースに設定された**ownerReference**で構成されます。**true**の場合に**Kafka**リソースが削除されると、CAシークレットも削除されます。**false**の場合は、**ownerReference**が無効になります。**false** のときに**Kafka**リソースが削除されても、CAシークレットは保持され、再利用可能となります。デフォルトは **true** です。
boolean	
validityDays	生成される証明書の有効日数。デフォルトは 365 です。
integer	
renewalDays	証明書更新期間の日数。これは、証明書の期限が切れるまでの日数です。この間に、更新アクションを実行することができます。**generateCertificateAuthority** が true の場合、新しい証明書が生成されます。**generateCertificateAuthority** が true の場合、保留中の証明書の有効期限に関する追加のロギングが WARN レベルで実行されます。デフォルトは 30 です。
integer	
certificateExpirationPolicy	**generateCertificateAuthority=true** の場合に CA 証明書の有効期限を処理する方法。デフォルトでは、既存の秘密鍵を再度使用して新規の CA 証明書が生成されます。
string ([replace-key、renew-certificate] のいずれか)	

14.2.49. CruiseControlSpec スキーマ参照

KafkaSpec で使用されます。

プロパティー	説明
image	Pod の Docker イメージ。
string	
tlsSidecar	TLS サイドカーの設定。
TlsSidecar	

プロパティー	説明
resources ResourceRequirements	Cruise Control コンテナー用に予約された CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
livenessProbe **Probe**	Cruise Control コンテナーの Pod liveness チェック
readinessProbe **Probe**	Cruise Control コンテナーの Pod readiness チェック
jvmOptions **JvmOptions**	Cruise Control コンテナーの JVM オプション
logging **InlineLogging**、**ExternalLogging**	Cruise Control のロギング設定 (Log4j 2)。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
template **CruiseControlTemplate**	Cruise Control のリソースである **Deployments** および **Pods**の生成方法を指定するテンプレート。
brokerCapacity **BrokerCapacity**	Cruise Control の **brokerCapacity** の設定。

プロパティー	説明
設定	Cruise Control の設定。設定オプションの完全リストは、https://github.com/linkedin/cruise-control/wiki/Configurations を参照してください。bootstrap.servers, client.id, zookeeper., network., security., failed.brokers.zk.path,webserver.api.urlprefix, webserver.session.path, webserver.accesslog., two.step., request.reason.required,metric.reporter.sampler.bootstrap.servers, webserver.session.path, webserver.accesslog.sampler.bootstrap.servers, metric.reporter.topic、partition.metric.sample.store.topic、broker.metric.sample.store.topic、capacity.config.file、self.healing.、ssl.（ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols、webserver.http.cors.enabled、webserver.http.cors.origin を除く）webserver.http.cors.exposeheaders、webserver.security.enable、webserver.ssl.enable）。
map	
metricsConfig	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
JmxPrometheusExporterMetrics	

14.2.50. CruiseControlTemplate スキーマ参照

CruiseControlSpec で使用されます。

プロパティー	説明
deployment	Cruise Control **Deployment** のテンプレート。
ResourceTemplate	
pod	Cruise Control **Pods** のテンプレート。
PodTemplate	
apiService	Cruise Control API **Service** のテンプレート。
InternalServiceTemplate	

プロパティー	説明
podDisruptionBudget	Cruise Control **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	
cruiseControlContainer	Cruise Control コンテナーのテンプレート。
ContainerTemplate	
tlsSidecarContainer	Cruise Control TLS サイドカーコンテナーのテンプレート。
ContainerTemplate	
serviceAccount	Cruise Control サービスアカウントのテンプレート。
ResourceTemplate	

14.2.51. BrokerCapacity スキーマー参照

CruiseControlSpec で使用されます。

プロパティー	説明
disk	ディスクのバイト単位のブローカー容量 (例: 100Gi)
string	
cpuUtilization	パーセントで表された CPU リソース使用率のブローカー容量 (0 - 100)。
integer	
inboundNetwork	バイト毎秒単位のインバウンドネットワークスループットのブローカー容量 (例: 10000KB/s)。
string	
outboundNetwork	バイト毎秒単位のアウトバウンドネットワークスループットのブローカー容量 (例: 10000KB/s)。
string	

14.2.52. KafkaExporterSpec スキーマ参照

KafkaSpec で使用されます。

プロパティー	説明
image	Pod の Docker イメージ。
string	
groupRegex	収集するコンシューマーグループを指定する正規表現。デフォルト値は .* です。
string	
topicRegex	収集するトピックを指定する正規表現。デフォルト値は .* です。
string	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
logging	指定の重大度以上のログメッセージのみ。有効なレベル：[**info**、**debug**、**trace**]デフォルトのログレベルは **info** です。
string	
enableSaramaLogging	Kafka Exporter によって使用される Go クライアントライブラリーである Sarama ロギングを有効にします。
boolean	
template	デプロイメントテンプレートおよび Pod のカスタマイズ。
KafkaExporterTemplate	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	

14.2.53. KafkaExporterTemplate スキーマ参照

KafkaExporterSpec で使用

プロパティー	説明
deployment	Kafka Exporter **Deployment** のテンプレート。
ResourceTemplate	
pod	Kafka Exporter **Pod** のテンプレート。
PodTemplate	
サービス	**サービスプロパティは廃止されました。** Kafka Exporter サービスは削除されました。Kafka Exporter **Service** のテンプレート。
ResourceTemplate	
container	Kafka Exporter コンテナーのテンプレート。
ContainerTemplate	
serviceAccount	Kafka Exporter サービスアカウントのテンプレート。
ResourceTemplate	

14.2.54. KafkaStatus スキーマ参照

Kafka で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
listeners	内部リスナーおよび外部リスナーのアドレス。
ListenerStatus array	
clusterId	Kafka クラスター ID。
string	

14.2.55. Condition スキーマ参照

で使用されます。 KafkaBridgeStatus, KafkaConnectorStatus, KafkaConnectStatus, KafkaMirrorMaker2Status, KafkaMirrorMakerStatus, KafkaRebalanceStatus, KafkaStatus, KafkaTopicStatus, KafkaUserStatus

プロパティー	説明
type	リソース内の他の条件と区別するために使用される条件の固有識別子。
string	
status	条件のステータス (True、False、または Unknown のいずれか)。
string	
lastTransitionTime	タイプの条件がある状態から別の状態へと最後に変更した時間。必須形式は、UTC タイムゾーンの 'yyyy-MM-ddTHH:mm:ssZ' です。
string	
reason	条件の最後の遷移の理由 (CamelCase の単一の単語)。
string	
message	条件の最後の遷移の詳細を示す、人間が判読できるメッセージ。
string	

14.2.56. ListenerStatus スキーマ参照

KafkaStatus で使用

プロパティー	説明
type	リスナーのタイプ。次の 3 つのタイプのいずれかになります: **plain**、**tls**、**external**
string	
addresses	このリスナーのアドレス一覧。
ListenerAddress array	
bootstrapServers	このリスナーを使用して Kafka クラスターに接続するための **host:port** ペアのコンマ区切りリスト。

プロパティー	説明
string	
certificates	指定のリスナーへの接続時に、サーバーのアイデンティティーを検証するために使用できる TLS 証明書の一覧。**tls** リスナーと **external** リスナーに対してのみ設定します。
string array	

14.2.57. ListenerAddress スキーマ参照

ListenerStatus で使用

プロパティー	説明
host	Kafka ブートストラップサービスの DNS 名または IP アドレス。
string	
port	Kafka ブートストラップサービスのポート。
integer	

14.2.58. KafkaConnect スキーマ参照

プロパティー	説明
spec	Kafka Connect クラスターの仕様。
KafkaConnectSpec	
status	Kafka Connect クラスターのステータス。
KafkaConnectStatus	

14.2.59. KafkaConnectSpec スキーマ参照

KafkaConnect で使用

KafkaConnectSpecスキーマプロパティの全リスト

Kafka Connect クラスターを設定します。

14.2.59.1. 設定

Kafkaのオプションをキーとして設定するには、configプロパティを使用します。

標準の Apache Kafka Connect 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。

以下に関連する設定オプションは設定できません。

- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- リスナー / REST インターフェースの設定
- プラグインパスの設定

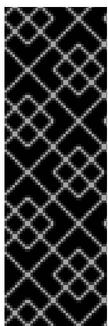
値は以下の JSON タイプのいずれかになります。

- 文字列
- 数値
- ブール値

AMQ Streams で直接管理されるオプションを除き、Apache Kafka ドキュメント に記載されているオプションを指定および設定できます。以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションは禁止されています。

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

禁止されているオプションが config プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka Connect に渡されます。



重要

提供された config オブジェクトのキーまたは値は Cluster Operator によって検証されません。無効な設定を指定すると、Kafka Connect クラスターが起動しなかったり、不安定になる可能性があります。この場合、KafkaConnect.spec.configオブジェクトで構成を修正すると、クラスター運用者は新しい構成をすべてのKafka Connectノードに展開することができます。

以下のオプションにはデフォルト値があります。

- **group.id、デフォルト値 connect-cluster**
- **offset.storage.topic、デフォルト値 connect-cluster-offsets**

- config.storage.topic、デフォルト値 connect-cluster-configs
- status.storage.topic、デフォルト値 connect-cluster-status
- key.converter、デフォルト値 org.apache.kafka.connect.json.JsonConverter
- value.converter、デフォルト値 org.apache.kafka.connect.json.JsonConverter

これらのオプションは、KafkaConnect.spec.configのプロパティに存在しない場合、自動的に設定されます。

禁止されているオプションには例外があります。TLSバージョンの特定の暗号スイートを使用したクライアント接続には、3つの許可されたssl設定オプションを使用できます。暗号スイートは、セキュアな接続とデータ転送のためのアルゴリズムを組み合わせます。ssl.endpoint.identification.algorithmプロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

Kafka Connect の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
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
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    ssl.endpoint.identification.algorithm: HTTPS
  # ...
```

TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。また、 ssl.endpoint.identification.algorithmプロパティを設定して、ホスト名の検証を有効または無効にすることもできます。

14.2.59.2. ログ

Kafka Connect には独自の設定可能なロガーがあります。

- connect.root.logger.level
- log4j.logger.org.reflections

実行中の Kafka Connect プラグインに応じて、さらにロガーが追加されます。

curl リクエストを使用して、Kafka ブローカー Pod から稼働している Kafka Connect ロガーの完全リストを取得します。

```
curl -s http://<connect-cluster-name>-connect-api:8083/admin/loggers/
```

Kafka Connect では Apache log4j ロガー実装が使用されます。

logging プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は log4j.properties を使用して記述されます。logging.valueFrom.configMapKeyRef.name および logging.valueFrom.configMapKeyRef.key プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、「Apache logging services」を参照してください。

ここで、inline および external ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
spec:
  # ...
  logging:
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: connect-logging.log4j
  # ...
```

設定されていない利用可能なロガーのレベルは OFF に設定されています。

Cluster Operator を使用して Kafka Connect がデプロイされた場合、Kafka Connect のロギングレベルの変更は動的に適用されます。

外部ロギングを使用する場合は、ロギングアペンダーが変更されるとローリングアップデートがトリガーされます。

ガベッジコレクター (GC)

ガベッジコレクターのロギングは jvmOptions プロパティーを使用して 有効（または無効）にすることもできます。

14.2.59.3. KafkaConnectSpecスキーマプロパティ

プロパティー	説明
version string	Kafka Connect のバージョン。デフォルトは 3.0.0 です。バージョンのアップグレードまたはダウングレードに必要なプロセスを理解するには、ユーザードキュメントを参照してください。
replicas integer	Kafka Connect グループの Pod 数。
image string	Pod の Docker イメージ。
bootstrapServers string	接続するブートストラップサーバー。これは **<hostname>:<port>** ペアのコンマ区切りリストとして指定する必要があります。
tls ClientTls	TLS 設定。
authentication KafkaClientAuthenticationTls、KafkaClientAuthenticationScramSha512、KafkaClientAuthenticationPlain、KafkaClientAuthenticationOAuth	Kafka Connect の認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls、scram-sha-512、plain、oauth] のいずれかでなければなりません。
config map	Kafka Connect の設定。次の接頭辞を持つプロパティーは設定できません: ssl.、sasl.、security.、listeners、plugin.path、rest.、bootstrap.servers、consumer.interceptor.classes、producer.interceptor.classes (ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols を除く)

プロパティー	説明
resources	CPU とメモリーリソースおよび要求された初期リソースの上限。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	
jvmOptions	Pod の JVM オプション。
JvmOptions	
jmxOptions	JMX オプション。
KafkaJmxOptions	
ログ	Kafka Connect のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
tracing	Kafka Connect でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger] の 1 つでなければなりません。
JaegerTracing	
template	Kafka ConnectとKafka Mirror Makerの2つのリソースのテンプレートです。ユーザーはテンプレートにより、**Deployment**、**Pod** および **Service** の生成方法を指定できます。
KafkaConnectTemplate	
externalConfiguration	Secret または ConfigMap から Kafka Connect Pod にデータを渡し、これを使用してコネクターを設定します。
ExternalConfiguration	
build	Connect コンテナーイメージを構築する方法を設定します。オプション。
ビルド	

プロパティー	説明
clientRackInitImage	**client.rack**の初期化に使用されるinitコンテナのイメージです。
string	
metricsConfig	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
JmxPrometheusExporterMetrics	
rack	client.rack コンシューマー設定として使用されるノードラベルの設定。
Rack	

14.2.60. ClientTls スキーマ参照

で使用されます。 KafkaBridgeSpec（カフカブリッジスペック, KafkaConnectSpec（カフカコネクトスペック, Kafkaミラーメーカー2クラスタ仕様, Kafkaミラーメーカーコンシュマー仕様, KafkaMirrorMakerProducerSpec(カフカミラーメーカープロデューサースペック)

ClientTls スキーマプロパティーの完全リスト

KafkaConnect、KafkaBridge、KafkaMirror、KafkaMirrorMaker2 をクラスターに接続するための TLS で信頼される証明書を設定します。

14.2.60.1. trustedCertificates

trustedCertificatesプロパティを使ってシークレットのリストを提供する。

14.2.60.2. ClientTls スキーマプロパティー

プロパティー	説明
trustedCertificates	TLS 接続の信頼済み証明書。
CertSecretSource array	

14.2.61. KafkaClientAuthenticationTlsスキーマ参照

で使用されます。 KafkaBridgeSpec（カフカブリッジスペック, KafkaConnectSpec（カフカコネクトスペック, Kafkaミラーメーカー2クラスタ仕様, Kafkaミラーメーカーコンシュマー仕様, KafkaMirrorMakerProducerSpec(カフカミラーメーカープロデューサースペック)

KafkaClientAuthenticationTlsスキーマプロパティの全リスト

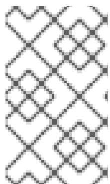
TLSクライアント認証を構成するには、typeプロパティに値tlsを設定します。TLS クライアント認証は TLS 証明書を使用して認証します。

14.2.61.1. certificateAndKey

証明書は certificateAndKey プロパティーで指定され、常に OpenShift シークレットからロードされます。シークレットでは、公開鍵と秘密鍵の 2 つの鍵を使用して証明書を X509 形式で保存する必要があります。

ユーザーオペレーターが作成したシークレットを使用することもできますが、認証に使用する鍵を含む独自のTLS証明書ファイルを作成し、そのファイルからシークレットを作成することもできます。

```
oc create secret generic MY-SECRET \
--from-file=MY-PUBLIC-TLS-CERTIFICATE-FILE.crt \
--from-file=MY-PRIVATE.key
```



注記

TLS クライアント認証は TLS 接続でのみ使用できます。

TLS クライアント認証の設定例

```
authentication:
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: my-public-tls-certificate-file.crt
    key: private.key
```

14.2.61.2. KafkaClientAuthenticationTlsスキーマプロパティ

typeプロパティは、KafkaClientAuthenticationTlsタイプの使用を以下のように区別するための識別情報です。 KafkaClientAuthenticationScramSha512, KafkaClientAuthenticationPlain, KafkaClientAuthenticationOAuth.KafkaClientAuthenticationTls タイプには tls の値が必要です。

プロパティー	説明
certificateAndKey	証明書と秘密鍵のペアを保持する Secret への参照。
CertAndKeySecretSource	
type	tls でなければなりません。
string	

14.2.62. KafkaClientAuthenticationScramSha512 スキーマ参照

で使用されます。 KafkaBridgeSpec（カフカブリッジスペック, KafkaConnectSpec（カフカコネクトスペック, Kafkaミラーメーカー2クラスタ仕様, Kafkaミラーメーカーコンシュマー仕様, KafkaMirrorMakerProducerSpec(カフカミラーメーカープロデューサースペック)

KafkaClientAuthenticationScramSha512スキーマプロパティの全リスト

SASL ベースの SCRAM-SHA-512 認証を設定するには、type プロパティーを scram-sha-512 に設定します。SCRAM-SHA-512 認証メカニズムには、ユーザー名とパスワードが必要です。

14.2.62.1. username

username プロパティーでユーザー名を指定します。

14.2.62.2. passwordSecret

passwordSecret プロパティーで、パスワードが含まれる Secret へのリンクを指定します。

User Operator によって作成されたシークレットを使用できます。

必要に応じて、認証に使用するクリアテキストのパスワードが含まれるテキストファイルを作成できます。

```
echo -n PASSWORD > MY-PASSWORD.txt
```

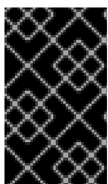
そして、テキストファイルから、パスワードに独自のフィールド名（キー）を設定して、Secretを作成することができます。

```
oc create secret generic MY-CONNECT-SECRET-NAME --from-file=MY-PASSWORD-FIELD-NAME=./MY-PASSWORD.txt
```

Kafka Connect の SCRAM-SHA-512 クライアント認証の Secret 例

```
apiVersion: v1
kind: Secret
metadata:
  name: my-connect-secret-name
type: Opaque
data:
  my-connect-password-field: LFTIyFRFIMmU2N2Tm
```

secretNameプロパティにはSecretの名前が、passwordプロパティにはSecretの中にパスワードが格納されているキーの名前が入ります。



重要

passwordプロパティに実際のパスワードを指定しないでください。

Kafka Connect の SASL ベース SCRAM-SHA-512 クライアント認証の設定例

```
authentication:
  type: scram-sha-512
  username: my-connect-username
  passwordSecret:
    secretName: my-connect-secret-name
    password: my-connect-password-field
```

14.2.62.3. KafkaClientAuthenticationScramSha512 schema properties

typeプロパティは、KafkaClientAuthenticationScramSha512タイプの使用を以下のように区別するための識別情報です。 KafkaClientAuthenticationTls, KafkaClientAuthenticationPlain, KafkaClientAuthenticationOAuth.KafkaClientAuthenticationScramSha512 タイプには scram-sha-512 の値が必要です。

プロパティー	説明
passwordSecret	パスワードを保持する **Secret** への参照。
PasswordSecretSource	
type	**scram-sha-512** でなければなりません。
文字列	
username	認証に使用されるユーザー名。
文字列	

14.2.63. PasswordSecretSource スキーマ参照

KafkaClientAuthenticationPlain、KafkaClientAuthenticationScramSha512 で使用

プロパティー	説明
password	パスワードが保存される Secret のキーの名前。
文字列	
secretName	パスワードを含むシークレットの名前。
文字列	

14.2.64. KafkaClientAuthenticationPlain スキーマ参照

で使用されます。 KafkaBridgeSpec（カフカブリッジスペック, KafkaConnectSpec（カフカコネクトスペック, Kafkaミラーメーカー2クラスタ仕様, Kafkaミラーメーカーコンシュマー仕様,

KafkaMirrorMakerProducerSpec(カフカミラーメーカープロデューサースペック)

KafkaClientAuthenticationPlainスキーマプロパティの全リスト

SASL ベースの PLAIN 認証を設定するには、type プロパティーを plain に設定します。SASL PLAIN 認証メカニズムには、ユーザー名とパスワードが必要です。

警告

SASL PLAIN メカニズムは、クリアテキストでユーザー名とパスワードをネットワーク全体に転送します。TLS による暗号化が有効になっている場合にのみ SASL PLAIN 認証を使用します。

14.2.64.1. username

username プロパティーでユーザー名を指定します。

14.2.64.2. passwordSecret

passwordSecret プロパティーで、パスワードが含まれる Secret へのリンクを指定します。

User Operator によって作成されたシークレットを使用できます。

必要に応じて、認証に使用するクリアテキストのパスワードが含まれるテキストファイルを作成します。

```
echo -n PASSWORD > MY-PASSWORD.txt
```

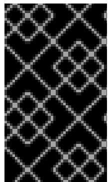
そして、テキストファイルから、パスワードに独自のフィールド名（キー）を設定して、Secretを作成することができます。

```
oc create secret generic MY-CONNECT-SECRET-NAME --from-file=MY-PASSWORD-FIELD-NAME=./MY-PASSWORD.txt
```

Kafka Connect の PLAIN クライアント認証の Secret 例

```
apiVersion: v1
kind: Secret
metadata:
  name: my-connect-secret-name
type: Opaque
data:
  my-password-field-name: LFTIyFRFlMmU2N2Tm
```

secretName プロパティーには Secret の名前が含まれ、password プロパティーには Secret 内にパスワードが格納されるキーの名前が含まれます。



重要

passwordプロパティに実際のパスワードを指定しないでください。

SASL ベースの PLAIN クライアント認証の設定例

```
authentication:
  type: plain
  username: my-connect-username
  passwordSecret:
    secretName: my-connect-secret-name
    password: my-password-field-name
```

14.2.64.3. KafkaClientAuthenticationPlainスキーマプロパティ

typeプロパティは、KafkaClientAuthenticationPlainタイプの使用を以下のように区別するための識別情報です。 KafkaClientAuthenticationTls, KafkaClientAuthenticationScramSha512, KafkaClientAuthenticationOAuth.KafkaClientAuthenticationPlain タイプには plain の値が必要です。

プロパティー	説明
passwordSecret	パスワードを保持する **Secret** への参照。
PasswordSecretSource	
type	**plain** でなければなりません。
string	
username	認証に使用されるユーザー名。
string	

14.2.65. KafkaClientAuthenticationOAuth スキーマ参照

で使用されます。 KafkaBridgeSpec（カフカブリッジスペック, KafkaConnectSpec（カフカコネクトスペック, Kafkaミラーメーカー2クラスタ仕様, Kafkaミラーメーカーコンシュマー仕様, KafkaMirrorMakerProducerSpec(カフカミラーメーカープロデューサースペック)

KafkaClientAuthenticationOAuthスキーマプロパティの全リスト

OAuthクライアント認証を構成するには、typeプロパティをoauthに設定します。

OAuth 認証は、以下のオプションのいずれかを使用して設定できます。

- クライアント ID およびシークレット
- クライアント ID および更新トークン
- アクセストークン
- TLS

クライアント ID およびシークレット

認証で使用されるクライアント ID およびクライアントシークレットとともに、tokenEndpointUri プロパティーで承認サーバーのアドレスを設定できます。OAuth クライアントは OAuth サーバーに接続し、クライアント ID およびシークレットを使用して認証し、Kafka ブローカーとの認証に使用するアクセストークンを取得します。clientSecret プロパティーで、クライアントシークレットを含む Secret へのリンクを指定します。

クライアント ID およびクライアントシークレットを使用した OAuth クライアント認証の例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  clientSecret:
    secretName: my-client-oauth-secret
    key: client-secret
```

必要に応じて、スコープとオーディエンスを指定することもできます。

クライアント ID および更新トークン

OAuth クライアント ID および更新トークンとともに、tokenEndpointUri プロパティーで OAuth サーバーのアドレスを設定できます。OAuth クライアントは OAuth サーバーに接続し、クライアント ID と更新トークンを使用して認証し、Kafka ブローカーとの認証に使用するアクセストークンを取得します。refreshToken プロパティーで、更新トークンが含まれる Secret へのリンクを指定します。

クライアント ID と更新トークンを使用した OAuth クライアント認証の例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
```

アクセストークン

Kafka ブローカーとの認証に使用されるアクセストークンを直接設定できます。この場合、tokenEndpointUri は指定しません。accessToken プロパティーで、アクセストークンが含まれる

Secret へのリンクを指定します。

アクセストークンのみを使用した OAuth クライアント認証の例

```
authentication:
  type: oauth
  accessToken:
    secretName: my-access-token-secret
    key: access-token
```

TLS

HTTPS プロトコルを使用して OAuth サーバーにアクセスする場合、信頼される認証局によって署名された証明書を使用し、そのホスト名が証明書に記載されている限り、追加の設定は必要ありません。

OAuth サーバーが自己署名証明書を使用している場合、または信頼されていない認証局によって署名されている場合は、カスタムリソースで信頼済み証明書の一覧を設定できます。tlsTrustedCertificates プロパティーには、証明書が保存されるキーの名前を持つシークレットの一覧が含まれます。証明書は X509 形式で保存する必要があります。

提供される TLS 証明書の例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
  tlsTrustedCertificates:
    - secretName: oauth-server-ca
      certificate: tls.crt
```

OAuth クライアントはデフォルトで、OAuth サーバーのホスト名が、証明書サブジェクトまたは別

の DNS 名のいずれかと一致することを確認します。必要でない場合は、ホスト名の検証を無効にできます。

無効にされた TLS ホスト名の検証例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
  disableTlsHostnameVerification: true
```

14.2.65.1. KafkaClientAuthenticationOAuthスキーマプロパティ

typeプロパティは、KafkaClientAuthenticationOAuthタイプの使用を次のように区別する識別記号です。 KafkaClientAuthenticationTls, KafkaClientAuthenticationScramSha512, KafkaClientAuthenticationPlain.KafkaClientAuthenticationOAuth タイプには oauth の値が必要です。

プロパティー	説明
accessToken GenericSecretSource	承認サーバーから取得したアクセストークンが含まれる OpenShift シークレットへのリンク。
accessTokenIsJwt boolean	アクセストークンを JWT として処理すべきかどうかを設定します。承認サーバーが不透明なトークンを返す場合は、**false** に設定する必要があります。デフォルトは **true** です。
audience string	承認サーバーに対して認証を行うときに使用する OAuth オーディエンス。一部の承認サーバーでは、オーディエンスを明示的に設定する必要があります。許可される値は、承認サーバーの設定によります。デフォルトでは、トークン・エンドポイント・リクエストの実行時に**オーディエンス**は指定されません。

プロパティー	説明
clientId	Kafka クライアントが OAuth サーバーに対する認証に使用し、トークンエンドポイント URI を使用することができる OAuth クライアント ID。
string	
clientSecret	Kafka クライアントが OAuth サーバーに対する認証に使用し、トークンエンドポイント URI を使用することができる OAuth クライアントシークレットが含まれる OpenShift シークレットへのリンク。
GenericSecretSource	
disableTlsHostnameVerification	TLS ホスト名の検証を有効または無効にします。デフォルト値は **false** です。
boolean	
maxTokenExpirySeconds	アクセストークンの有効期間を指定の秒数に設定または制限します。これは、承認サーバーが不透明なトークンを返す場合に設定する必要があります。
integer	
refreshToken	承認サーバーからアクセストークンを取得するために使用できる更新トークンが含まれる OpenShift シークレットへのリンク。
GenericSecretSource	
scope	承認サーバーに対して認証を行うときに使用する OAuth スコープ。一部の承認サーバーでこれを設定する必要があります。許可される値は、承認サーバーの設定によります。デフォルトでは、トークンエンドポイントリクエストを実行する場合は **scope** は指定されません。
string	
tlsTrustedCertificates	OAuth サーバーへの TLS 接続の信頼済み証明書。
CertSecretSource array	
tokenEndpointUri	承認サーバートークンエンドポイント URI。
string	
type	**oauth** でなければなりません。
string	

14.2.66. JaegerTracing スキーマ参照

で使用されます。 **KafkaBridgeSpec（カフカブリッジスペック**, **KafkaConnectSpec（カフカコネクトスペック**, **カフカミラーメーカー2仕様**, **KafkaMirrorMakerSpec（カフカミラーメーカースペック**

typeプロパティは、JaegerTracingタイプの使用と、今後追加される可能性のある他のサブタイプとを区別するための識別情報です。JaegerTracing タイプには jaeger の値が必要です。

プロパティー	説明
type	**jaeger** でなければなりません。
string	

14.2.67. KafkaConnectTemplate スキーマ参照

で使用されています。 KafkaConnectSpec（カフカコネクトスペック, KafkaMirrorMaker2Spec（カフカミラーメーカー2スペック

プロパティー	説明
deployment	Kafka Connect **Deployment** のテンプレート。
DeploymentTemplate	
pod	Kafka Connect **Pod** のテンプレート。
PodTemplate	
apiService	Kafka Connect API **Service** のテンプレート。
InternalServiceTemplate	
connectContainer	Kafka Connect コンテナーのテンプレート。
ContainerTemplate	
initContainer	Kafka init コンテナーのテンプレート。
ContainerTemplate	
podDisruptionBudget	Kafka Connect **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	
serviceAccount	Kafka Connect サービスアカウントのテンプレート。
ResourceTemplate	

プロパティー	説明
clusterRoleBinding	Kafka Connect ClusterRoleBinding のテンプレート。
ResourceTemplate	
buildPod	Kafka Connect Build**Pods**のテンプレートです。build Pod は OpenShift でのみ使用されます。
PodTemplate	
buildContainer	Kafka Connect Build コンテナーのテンプレート。build コンテナーは OpenShift でのみ使用されます。
ContainerTemplate	
buildConfig	新しいコンテナーイメージをビルドするために使用される Kafka Connect BuildConfig のテンプレート。BuildConfig は OpenShift でのみ使用されます。
BuildConfigTemplate	
buildServiceAccount	Kafka Connect Build サービスアカウントのテンプレート。
ResourceTemplate	
jmxSecret	Kafka Connect Cluster JMX認証のSecretのテンプレートです。
ResourceTemplate	

14.2.68. DeploymentTemplateスキーマ参照

で使用されています。 KafkaBridgeTemplate（カフカブリッジテンプレート, KafkaConnectTemplate（カフカコネクトテンプレート, KafkaMirrorMakerTemplate

プロパティー	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	
deploymentStrategy	このデプロイメントに使用される DeploymentStrategy。有効な値は**RollingUpdate**と**Recreate**です。デフォルトは**RollingUpdate**です。
string ([RollingUpdate、Recreate] のいずれか)	

14.2.69. BuildConfigTemplate schema reference

KafkaConnectTemplateで使用

プロパティー	説明
metadata	**PodDisruptionBudgetTemplate** リソースに適用するメタデータ。
MetadataTemplate	
pullSecret	ベースイメージをプルするためのクレデンシャルが含まれるコンテナーレジストリーシークレット。
string	

14.2.70. ExternalConfiguration スキーマ参照

で使用されています。 KafkaConnectSpec（カフカコネクトスペック, KafkaMirrorMaker2Spec（カフカミラーメーカー2スペック

ExternalConfigurationスキーマ・プロパティの全リスト

Kafka Connect コネクターの設定オプションを定義する外部ストレージプロパティーを設定します。

ConfigMap またはシークレットを環境変数またはボリュームとして Kafka Connect Pod にマウントできます。ボリュームと環境変数は、KafkaConnect.specのexternalConfigurationプロパティで設定します。

これが適用されると、コネクターの開発時に環境変数とボリュームを使用できます。

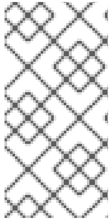
14.2.70.1. env

envプロパティを使用して、1つまたは複数の環境変数を指定します。これらの変数には ConfigMap または Secret からの値を含めることができます。

環境変数の値が含まれるシークレットの例

```
apiVersion: v1
```

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



注記

ユーザー定義の環境変数に、KAFKA_ または STRIMZI_ で始まる名前を付けることはできません。

Secretの値を環境変数にマウントするには、valueFromプロパティとsecretKeyRefを使用します。

Secret からの値に設定された環境変数の例

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

Secret をマウントする一般的なユースケースは、コネクターが Amazon AWS と通信するためのものです。コネクタは、AWS_ACCESS_KEY_IDとAWS_SECRET_ACCESS_KEYを読み取ることがで

きる必要があります。

ConfigMap から環境変数に値をマウントするには、以下の例のように valueFrom プロパティーで configMapKeyRef を使用します。

ConfigMap からの値に設定された環境変数の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

14.2.70.2. volumes

ボリュームを使用して ConfigMap またはシークレットを Kafka Connect Pod にマウントします。

以下の場合、環境変数の代わりにボリュームを使用すると便利です。

- Kafka Connect コネクターの設定に使用されるプロパティーファイルのマウント
- TLS 証明書でのトラストストアまたはキーストアのマウント

ボリュームは、Kafka Connectコンテナ内のパス/opt/kafka/external-configuration/<volume-name>にマウントされます。例えば、「connector-config」という名前のボリュームのファイルは、「/opt/kafka/external-configuration/connector-config」というディレクトリに表示されます。

設定プロバイダーは設定外から値を読み込みます。プロバイダーメカニズムを使用して、制限された情報がKafka ConnectREST インターフェースを介して渡されないようにします。

- FileConfigProviderは、ファイル内のプロパティから構成値をロードします。
- DirectoryConfigProviderは、ディレクトリ構造内の別々のファイルから構成値をロードします。

複数のプロバイダー (カスタムプロバイダーを含む) を追加する場合は、コンマ区切りリストを使用します。カスタムプロバイダーを使用して、他のファイルの場所から値をロードできます。

FileConfigProviderを使用してプロパティ値をロードする。

この例では、mysecretという名前の Secret に、データベース名とパスワードを指定するコネクタプロパティが含まれています。

データベースプロパティーのある Secret の例

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |- (1)
    dbUsername: my-username (2)
    dbPassword: my-password
```

(1) プロパティーファイル形式のコネクター設定。

(2) 設定で使用されるデータベースのユーザー名およびパスワードプロパティー。

Kafka Connectの設定では、SecretとFileConfigProviderの設定プロバイダを指定します。

- Secret はconnector-config という名前のボリュームにマウントされています。
- FileConfigProviderには、エイリアスファイルが与えられます。

Secret からの値に設定された外部ボリュームの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: file (1)
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider
(2)
  #...
  externalConfiguration:
    volumes:
      - name: connector-config (3)
        secret:
          secretName: mysecret (4)
```

(1) 設定プロバイダーのエイリアスは、他の設定パラメーターを定義するために使用されます。

(2) FileConfigProviderは、プロパティファイルから値を提供します。このパラメータは、config.providersからのエイリアスを使用し、config.providers.${alias}.classという形式をとります。

(3) Secret が含まれるボリュームの名前。各ボリュームは、nameプロパティに名前を指定し、ConfigMapまたはSecretへの参照を指定する必要があります。

4

Secret の名前。

Secret のプロパティー値のプレースホルダーは、コネクター設定で参照されます。プレースホルダーの構造は、file:PATH-AND-FILE-NAME:PROPERTYとなっています。FileConfigProviderは、コネクタ構成でマウントされたSecretからデータベースのユーザ名とパスワードのプロパティ値を読み取り、抽出します。

外部値のプレースホルダーを示すコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    database.hostname: 192.168.99.1
    database.port: "3306"
    database.user: "${file:/opt/kafka/external-configuration/connector-config/mysecret:dbUsername}"
    database.password: "${file:/opt/kafka/external-configuration/connector-config/mysecret:dbPassword}"
    database.server.id: "184054"
    #...
```

DirectoryConfigProviderを使用して、別々のファイルからプロパティ値をロードする。

この例では、SecretにはTLSのトラストストアとキーストアのユーザー認証情報が別々のファイルで含まれています。

ユーザークレデンシャルのある Secret の例

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
```

```
  labels:
    strimzi.io/kind: KafkaUser
    strimzi.io/cluster: my-cluster
type: Opaque
data: (1)
  ca.crt: # Public key of the client CA
  user.crt: # User certificate that contains the public key of the user
  user.key: # Private key of the user
  user.p12: # PKCS #12 archive file for storing certificates and keys
  user.password: # Password for protecting the PKCS #12 archive file
```

Kafka Connectの設定では、SecretとDirectoryConfigProviderの設定プロバイダーを指定します。

- Secret はconnector-config という名前のボリュームにマウントされています。
- DirectoryConfigProviderには、エイリアスのディレクトリが与えられます。

ユーザークレデンシャルファイルに設定された外部ボリュームの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    config.providers: directory
    config.providers.directory.class:
org.apache.kafka.common.config.provider.DirectoryConfigProvider (1)
  #...
  externalConfiguration:
    volumes:
      - name: connector-config
        secret:
          secretName: mysecret
```

(1) (1)

クレデンシャルのプレースホルダーはコネクター設定で参照されます。プレースホルダの構造は、directory:PATH:FILE-NAMEとなっています。DirectoryConfigProviderは、コネクタ構成でマウントされたSecretから認証情報を読み取り、抽出します。

外部値のプレースホルダーを示すコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    security.protocol: SSL
    ssl.truststore.type: PEM
    ssl.truststore.location: "${directory:/opt/kafka/external-configuration/connector-config:ca.crt}"
    ssl.keystore.type: PEM
    ssl.keystore.location: "${directory:/opt/kafka/external-configuration/connector-config:user.key}"
    #...
```

14.2.70.3. ExternalConfigurationスキーマのプロパティ

プロパティー	説明
env ExternalConfigurationEnv array	Secret または ConfigMap からのデータを環境変数として Kafka Connect Pod で利用できるようにします。
volumes ExternalConfigurationVolumeSource array	Secret または ConfigMap からのデータをボリュームとして Kafka Connect Pod で利用できるようにします。

14.2.71. ExternalConfigurationEnv スキーマ参照

ExternalConfiguration で使用

プロパティー	説明
name	Kafka Connect Pod に渡される環境変数の名前。環境変数に、**KAFKA_** または **STRIMZI_** で始まる名前を付けることはできません。
string	
valueFrom	Kafka Connect Pod に渡される環境変数の値。Secret または ConfigMap フィールドのいずれかへ参照として渡すことができます。このフィールドでは、Secret または ConfigMap を 1 つだけ指定する必要があります。
ExternalConfigurationEnvVarSource	

14.2.72. ExternalConfigurationEnvVarSource スキーマ参照

ExternalConfigurationEnv で使用

プロパティー	説明
configMapKeyRef	ConfigMap のキーへの参照。詳細は、core/v1 configmapkeyselector の外部ドキュメントを参照してください。
ConfigMapKeySelector	
secretKeyRef	Secret のキーへの参照。詳細は、core/v1 secretkeyselector の外部ドキュメント を参照してください。
SecretKeySelector	

14.2.73. ExternalConfigurationVolumeSource スキーマ参照

ExternalConfiguration で使用

プロパティー	説明
configMap	ConfigMap のキーへの参照。Secret または ConfigMap を 1 つだけ指定する必要があります。詳細は、core/v1 configmapvolumesource の外部ドキュメント を参照してください。
ConfigMapVolumeSource	
name	Kafka Connect Pod に追加されるボリュームの名前。
string	

プロパティー	説明
secret SecretVolumeSource	Secret のキーへの参照。Secret または ConfigMap を 1 つだけ指定する必要があります。詳細は、core/v1 secretvolumesource の外部ドキュメント を参照してください。

14.2.74. スキーマ・リファレンスの構築

で使用されています。 KafkaConnectSpec

ビルド・スキーマ・プロパティの全リスト

Kafka Connect デプロイメントの追加コネクターを設定します。

14.2.74.1. 出力

追加のコネクタープラグインで新しいコンテナーイメージをビルドするには、イメージをプッシュ、保存、およびプルできるコンテナーレジストリーが AMQ Streams に必要です。AMQ Streams は独自のコンテナーレジストリーを実行しないため、レジストリーを指定する必要があります。AMQ Streams は、プライベートコンテナーレジストリーだけでなく、Quay や Docker Hub などのパブリックレジストリーもサポートします。コンテナレジストリは、KafkaConnectカスタムリソースの.spec.build.outputセクションで設定されます。必須である出力設定は、dockerとimagestreamの2種類に対応しています。

Docker レジストリーの使用

Dockerレジストリを使用するには、タイプをdockerとし、イメージフィールドに新しいコンテナイメージのフルネームを指定する必要があります。フルネームには以下が含まれる必要があります。

- レジストリーのアドレス
- ポート番号 (標準以外のポートでリッスンしている場合)
- 新しいコンテナーイメージのタグ

有効なコンテナーイメージ名の例:

- docker.io/my-org/my-image/my-tag
- quay.io/my-org/my-image/my-tag
- image-registry.image-registry.svc:5000/myproject/kafka-connect-build:latest

Kafka Connect デプロイメントごとに個別のイメージを使用する必要があります。これは、最も基本的なレベルで異なるタグを使用する可能性があることを意味します。

レジストリに認証が必要な場合は、pushSecretを使ってレジストリの認証情報を持つSecretの名前を設定します。Secretには、kubernetes.io/dockerconfigjsonタイプを使用し、Dockerの認証情報を含む.dockerconfigjsonファイルを作成します。プライベートレジストリーからイメージをプルする方法の詳細は、「Create a Secret based on existing Docker credentials」を参照してください。

出力構成例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      type: docker (1)
      image: my-registry.io/my-org/my-connect-cluster:latest (2)
      pushSecret: my-registry-credentials (3)
  #...
```

(1) (必須) AMQ Streams によって使用される出力のタイプ。

(2)

3

(任意) コンテナーレジストリーのクレデンシャルが含まれるシークレットの名前。

OpenShift ImageStream の使用

Docker の代わりに OpenShift ImageStream を使用して、新しいコンテナーイメージを保存できます。Kafka Connect をデプロイする前に、ImageStream を手動で作成する必要があります。ImageStreamを使用するには、typeをimagestreamに設定し、imageプロパティでImageStreamの名前と使用するタグを指定します。例えば、my-connect-image-stream:latestのようになります。

出力構成例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      type: imagestream (1)
      image: my-connect-build:latest (2)
  #...
```

1

(必須) AMQ Streams によって使用される出力のタイプ。

2

(必須) ImageStream およびタグの名前。

14.2.74.2. plugins

コネクタープラグインは、特定タイプの外部システムへの接続に必要な実装を定義するファイルのセットです。コンテナイメージに必要なコネクタプラグインは、KafkaConnectカスタムリソースの.spec.build.pluginsプロパティを使用して設定する必要があります。各コネクタープラグインには、Kafka Connect デプロイメント内で一意となる名前が必要です。さらに、プラグインアーティファクト

もリストする必要があります。これらのアーティファクトは AMQ Streams によってダウンロードされ、新しいコンテナーイメージに追加され、Kafka Connect デプロイメントで使用されます。コネクタープラグインアーティファクトには、シリアライザーやデシリアライザーなどの追加のコンポーネントを含めることもできます。各コネクタープラグインは、異なるコネクターとそれらの依存関係が適切に サンドボックス化 されるように、個別のディレクトリーにダウンロードされます。各プラグインは、1 つ以上の artifact で設定する必要があります。

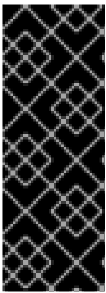
2つのコネクタプラグインを使用したプラグインの構成例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins: (1)
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb548045e1
15e33a9e69b1b2a352bee24df035a0447cb820077af00c03
      - name: camel-telegram
        artifacts:
          - type: tgz
            url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-
telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
            sha512sum:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e802
0bf4187870699819f54ef5859c7846ee4081507f48873479
  #...
```

(1) (必須) コネクタープラグインおよびそれらのアーティファクトの一覧。

AMQ Streamsは、以下の種類のアーティファクトをサポートしています。

- JARファイル（ダウンロードして直接使用するもの
- TGZアーカイブをダウンロードして解凍したもの
- ZIP アーカイブ（ダウンロードおよび展開される）
- Maven コーディネートを使用する Maven アーティファクト
- その他のアーティファクトで、ダウンロードして直接使用するもの



重要

AMQ Streams は、ダウンロードしたアーティファクトのセキュリティースキャンを実行しません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドと Kafka Connect デプロイメントで同じアーティファクトが使用されるようにする必要があります。

JAR アーティファクトの使用

JAR アーティファクトは、コンテナーイメージにダウンロードされ、追加された JAR ファイルを表します。JARアーティファクトを使用するには、typeプロパティをjarに設定し、urlプロパティでダウンロード先を指定します。

さらに、アーティファクトの SHA-512 チェックサムを指定することもできます。指定された場合、AMQ Streams は新しいコンテナーイメージのビルド中にアーティファクトのチェックサムを検証します。

JAR アーティファクトの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
```

```
  #...
  plugins:
    - name: my-plugin
      artifacts:
        - type: jar (1)
          url: https://my-domain.tld/my-jar.jar (2)
          sha512sum: 589...ab4 (3)
        - type: jar
          url: https://my-domain.tld/my-jar2.jar
#...
```

1. (必須) アーティファクトのタイプ。

2. (必須) アーティファクトのダウンロード元 URL。

3. (任意) アーティファクトを検証する SHA-512 チェックサム。

TGZ アーティファクトの使用

TGZ アーティファクトは、Gzip 圧縮を使用して圧縮された TAR アーカイブをダウンロードするために使用されます。複数の異なるファイルで構成される場合でも、TGZ アーティファクトに Kafka Connect コネクター全体を含めることができます。TGZ アーティファクトは、新しいコンテナーイメージのビルド時に AMQ Streams によって自動的にダウンロードおよび展開されます。TGZアーティファクトを使用するには、typeプロパティにtgzを設定し、urlプロパティでダウンロード先を指定します。

さらに、アーティファクトの SHA-512 チェックサムを指定することもできます。指定された場合、展開して新しいコンテナイメージをビルドする前に、チェックサムが AMQ Streams によって検証されます。

TGZ アーティファクトの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
```

```
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: tgz (1)
            url: https://my-domain.tld/my-connector-archive.tgz (2)
            sha512sum: 158...jg10 (3)
  #...
```

(1) (必須) アーティファクトのタイプ。

(2) (必須) アーカイブのダウンロード元 URL。

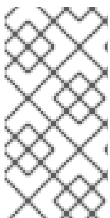
(3) (任意) アーティファクトを検証する SHA-512 チェックサム。

ZIP アーティファクトの使用

ZIP アーティファクトは、ZIP 圧縮アーカイブのダウンロードに使用されます。前のセクションで説明した TGZ アーティファクトと同じ方法で ZIP アーティファクトを使用します。唯一の違いは、type: tgz ではなく type: zip を指定することです。

Maven アーティファクトの使用

Maven アーティファクトは、コネクタープラグインアーティファクトを Maven コーディネートとして指定するために使用されます。Maven コーディネートはプラグインアーティファクトおよび依存関係を特定し、Maven リポジトリーから検索および取得できるようにします。



注記

アーティファクトをコンテナーイメージに追加するには、コネクタービルドプロセスで Maven リポジトリーにアクセスできる必要があります。

Maven アーティファクトの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: maven (1)
            repository: https://mvnrepository.com (2)
            group: org.apache.camel.kafkaconnector (3)
            artifact: camel-kafka-connector (4)
            version: 0.11.0 (5)
  #...
```

(1) (必須) アーティファクトのタイプ。

(2) （任意）アーティファクトのダウンロード元となる Maven リポジトリー。リポジトリーを指定しないと、デフォルトで Maven Central リポジトリー が使用されます。

(3) （必須）Maven グループ ID。

(4) （必須）Maven アーティファクトタイプ。

(5) （必須）Maven バージョン番号。

他のアーティファクトの使用

その他のアーティファクトは、ダウンロードされてコンテナイメージに追加されるあらゆる種類のファイルを表します。作成されるコンテナイメージのアーティファクトに特定の名前を使用したい場合は、fileNameフィールドを使用します。ファイル名が指定されていない場合、URL ハッシュを基にファイルの名前が付けられます。

さらに、アーティファクトの SHA-512 チェックサムを指定することもできます。指定された場合、AMQ Streams は新しいコンテナーイメージのビルド中にアーティファクトのチェックサムを検証します。

他のアーティファクトの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: other  1
            url: https://my-domain.tld/my-other-file.ext  2
            sha512sum: 589...ab4  3
            fileName: name-the-file.ext  4
  #...
```

1. (必須) アーティファクトのタイプ。
2. (必須) アーティファクトのダウンロード元 URL。
3. (任意) アーティファクトを検証する SHA-512 チェックサム。
4.

14.2.74.3. スキーマ・プロパティの構築

プロパティー	説明
output **DockerOutput**, **ImageStreamOutput**	新たにビルドされたイメージの保存先を設定します。必須。タイプは、与えられたオブジェクト内の**output.type**プロパティの値に依存し、[docker, imagestream]のいずれかでなければなりません。
resources ResourceRequirements	ビルド用に予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
plugins **プラグイン**配列	Kafka Connect に追加する必要のあるコネクタープラグインのリスト。必須。

14.2.75. DockerOutputスキーマリファレンス

で使用されています。 ビルド

typeプロパティは、DockerOutputタイプの使用を以下のように区別するための識別記号です。 イメージストリーム出力. DockerOutputのタイプにはdockerという値が必要です。

プロパティー	説明
image 文字列	新たにビルドされたイメージのタグ付けおよびプッシュに使用されるフルネーム。例えば、**quay.io/my-organization/my-custom-connect:latest**のように。必須。
pushSecret 文字列	新たにビルドされたイメージをプッシュするための、クレデンシャルが含まれる Container Registry Secret。

プロパティー	説明
additionalKanikoOptions	新しい Connect イメージをビルドする際に、Kaniko エグゼキューターに渡される追加オプションを設定します。指定できるオプションは --customPlatform、--insecure、--insecure-pull、--insecure-registry、--log-format、--log-timestamp、--registry-mirror、--reproducible、--single-snapshot、--skip-tls-verify、--skip-tls-verify-pull、--skip-tls-verify-registry、--verbosity、--snapshotMode、--use-new-run です。これらのオプションは、Kaniko エグゼキューターが使用される OpenShift でのみ使用されます。OpenShift では無視されます。オプションは、「Kaniko GitHub repository」に記載されています。このフィールドを変更しても、Kafka Connect イメージのビルドは新たにトリガーされません。
string array	
type	**docker**であること。
文字列	

14.2.76. ImageStreamOutputスキーマリファレンス

で使用されています。 ビルド

typeプロパティは、ImageStreamOutputタイプの使用を以下のように区別するための識別記号です。 DockerOutput. ImageStreamOutput型の場合は、imagestreamという値を持たなければなりません。

プロパティー	説明
image	新たにビルドされたイメージがプッシュされる ImageStream の名前およびタグ。例えば、**my-custom-connect:latest**のように。必須。
文字列	
type	**imagestream**に違いない。
文字列	

14.2.77. プラグインのスキーマ参照

で使用されています。 ビルド

プロパティー	説明
name	コネクタープラグインの一意名。コネクターアーティファクトが保存されるパスの生成に使用されます。名前は KafkaConnect リソース内で一意である必要があります。名前は、以下のパターンに従わなければなりません。**^[a-z][-_a-z0-9]*[a-z]$.**必須。
文字列	
artifacts	このコネクタープラグインに属するアーティファクトの一覧。必須。
Jarアーティファクト, **Tgzアーティファクト**, **Zipアーティファクト**, **Mavenアーティファクト**, **その他の成果物**配列	

14.2.78. JarArtifactスキーマリファレンス

で使用されています。 プラグイン

プロパティー	説明
url	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。 **jar**、**zip**、**tgz** **などの**アーティファクトに必要です。 **maven**アーティファクトタイプには適用されません。
文字列	
sha512sum	アーティファクトの SHA512 チェックサム。任意設定。指定すると、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。 **maven**アーティファクトタイプには適用されません。
文字列	
insecure	デフォルトでは、TLSを使用する接続は、安全であることを確認するために検証されます。使用するサーバー証明書は、有効かつ信頼できるもので、サーバー名が含まれている必要があります。このオプションを**true**に設定すると、すべてのTLS検証が無効になり、サーバーが安全でないと判断された場合でも、アーティファクトがダウンロードされます。
boolean	

プロパティー	説明
type	**ジャー**でなければならない。
文字列	

14.2.79. TgzArtifactスキーマリファレンス

で使用されています。 プラグイン

プロパティー	説明
url	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。 **jar**、**zip**、**tgz などのア**ーティファクトに必要です。 **maven**アーティファクトタイプには適用されません。
string	
sha512sum	アーティファクトの SHA512 チェックサム。任意設定。指定すると、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。 **maven**アーティファクトタイプには適用されません。
string	
insecure	デフォルトでは、TLSを使用する接続は、安全であることを確認するために検証されます。使用するサーバー証明書は、有効かつ信頼できるもので、サーバー名が含まれている必要があります。このオプションを**true**に設定すると、すべてのTLS検証が無効になり、サーバーが安全でないと判断された場合でも、アーティファクトがダウンロードされます。
boolean	
type	**tgz**でなければなりません。
string	

14.2.80. ZipArtifactスキーマリファレンス

で使用されています。 プラグイン

プロパティー	説明
url	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。 **jar**、**zip**、**tgz** などのアーティファクトに必要です。 **maven**アーティファクトタイプには適用されません。
文字列	
sha512sum	アーティファクトの SHA512 チェックサム。任意設定。指定すると、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。 **maven**アーティファクトタイプには適用されません。
文字列	
insecure	デフォルトでは、TLSを使用する接続は、安全であることを確認するために検証されます。使用するサーバー証明書は、有効かつ信頼できるもので、サーバー名が含まれている必要があります。このオプションを**true**に設定すると、すべてのTLS検証が無効になり、サーバーが安全でないと判断された場合でも、アーティファクトがダウンロードされます。
boolean	
type	**zip**でなければならない。
文字列	

14.2.81. MavenArtifactスキーマリファレンス

で使用されています。 プラグイン

typeプロパティは、MavenArtifactタイプの使用を次のように区別する識別記号です。 Jarアーティファクト, Tgzアーティファクト, Zipアーティファクト, その他のアーティファクト. MavenArtifact タイプの値mavenを持つ必要があります。

プロパティー	説明
repository	アーティファクトをダウンロードするMavenリポジトリ。 **maven**アーティファクトタイプにのみ適用されます。
string	

プロパティー	説明
group	Mavenグループの IDです。 **maven**アーティファクトタイプにのみ適用されます。
string	
アーティファクト	Mavenアーティファクト ID。 **maven**アーティファクトタイプにのみ適用されます。
string	
version	Mavenのバージョン番号。 **maven**アーティファクトタイプにのみ適用されます。
string	
type	**MAVEN**でなければならない。
string	

14.2.82. OtherArtifactスキーマ参照

で使用されています。 プラグイン

プロパティー	説明
url	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。 **jar**、**zip**、**tgz** **などのア**ーティファクトに必要です。 **maven**アーティファクトタイプには適用されません。
文字列	
sha512sum	アーティファクトの SHA512 チェックサム。任意設定。指定すると、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。 **maven**アーティファクトタイプには適用されません。
文字列	
fileName	保存されるアーティファクトの名前。
文字列	

プロパティー	説明
insecure	デフォルトでは、TLSを使用する接続は、安全であることを確認するために検証されます。使用するサーバー証明書は、有効かつ信頼できるもので、サーバー名が含まれている必要があります。このオプションを**true**に設定すると、すべてのTLS検証が無効になり、サーバーが安全でないと判断された場合でも、アーティファクトがダウンロードされます。
boolean	
type	**他**でなければならない。
文字列	

14.2.83. KafkaConnectStatus スキーマ参照

KafkaConnect で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
url	Kafka Connect コネクターの管理および監視用の REST API エンドポイントの URL。
文字列	
connectorPlugins	この Kafka Connect デプロイメントで使用できるコネクタープラグインの一覧。
ConnectorPlugin array	
labelSelector	このリソースを提供する Pod のラベルセレクター。
文字列	
replicas	このリソースを提供するために現在使用されている Pod の数。
integer	

14.2.84. ConnectorPlugin スキーマ参照

で使用されています。 KafkaConnectStatus, KafkaMirrorMaker2Status

プロパティー	説明
type 文字列	コネクタープラグインのタイプ。**sink** タイプと **source** タイプを利用できます。
version 文字列	コネクタープラグインのバージョン。
class 文字列	コネクタープラグインのクラス。

14.2.85. KafkaTopic スキーマ参照

プロパティー	説明
spec KafkaTopicSpec	トピックの仕様。
status KafkaTopicStatus	トピックのステータス。

14.2.86. KafkaTopicSpec スキーマ参照

KafkaTopic で使用

プロパティー	説明
partitions integer	トピックに存在するパーティション数。この数はトピック作成後に減らすことはできません。トピック作成後に増やすことはできますが、その影響について理解することが重要となります。特にセマンティックパーティションのあるトピックで重要となります。省略すると、デフォルトで**num.partitions** のブローカー構成になります。

プロパティー	説明
replicas	トピックのレプリカ数。省略すると、**default.replication.factor**のブローカー設定がデフォルトになります。
integer	
config	トピックの設定。
map	
topicName	トピックの名前。これがない場合、デフォルトではトピックの metadata.name に設定されます。トピック名が有効な OpenShift リソース名ではない場合を除き、これを設定しないことが推奨されます。
string	

14.2.87. KafkaTopicStatus スキーマ参照

KafkaTopic で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
topicName	トピック名。
string	

14.2.88. KafkaUser スキーマ参照

プロパティー	説明
spec	ユーザーの仕様。
KafkaUserSpec	
status	Kafka User のステータス。
KafkaUserStatus	

14.2.89. KafkaUserSpec スキーマ参照

KafkaUser で使用

プロパティー	説明
authentication	この Kafka ユーザーに対して有効になっている認証メカニズム。サポートされている認証メカニズムは、**scram-sha-512**、**tls**、および**tls-external**です。 • **scram-sha-512**は、SASL SCRAM-SHA-512の認証情報を持つ秘密を生成します。 • **tls**は、TLSの相互認証のために、ユーザー証明書と一緒に秘密を生成します。 • **tls-external**はユーザー証明書を生成しません。しかし、User Operatorの外部で生成されたユーザー証明書を用いた相互TLS認証を使用するための準備を行います。このユーザーに設定されたACLやクォータは、**CN=<ユーザー名>**の形式で設定されます。 認証は任意です。認証が設定されていない場合は、認証情報は生成されません。ユーザーに設定されるACLやクォータは、SASL認証に適した**<username>**形式で設定されます。このタイプは、指定されたオブジェクト内の**authentication.type**プロパティの値に依存し、[tls, tls-external, scram-sha-512] のいずれかでなければなりません。
KafkaUserTlsClientAuthentication（カフカユーザータイムズクライアント認証, **KafkaUserTlsExternalClientAuthentication（カフカユーザーTlsエクスターナルクライアント認証**, **KafkaUserScramSha512ClientAuthentication (カフカユーザースクラムシャ512クライアント認証)**	
認可	この Kafka ユーザーの承認ルール。タイプは、指定のオブジェクト内の **authorization.type** プロパティーの値によって異なり、[simple] の 1 つでなければなりません。
KafkaUserAuthorizationSimple	
quotas	クライアントによって使用されるブローカーリソースを制御する要求のクォータ。ネットワーク帯域幅および要求レートクォータの適用が可能です。Kafka ユーザークォータの Kafka ドキュメントは http://kafka.apache.org/documentation/#design_quotas を参照してください。
KafkaUserQuotas	
template	Kafka User**Secrets**の生成方法を指定するテンプレートです。
KafkaUserTemplate	

14.2.90. KafkaUserTlsClientAuthentication スキーマ参照

KafkaUserSpec で使用

typeプロパティは、KafkaUserTlsClientAuthenticationタイプの使用を次のように区別する識別記号です。 KafkaUserTlsExternalClientAuthentication（カフカユーザートラムエクスターナルクライアント認証, KafkaUserScramSha512ClientAuthentication（カフカユーザースクラムシャ512クライアント認証.KafkaUserTlsClientAuthentication タイプには tls の値が必要です。

プロパティー	説明
type	**tls** でなければなりません。
string	

14.2.91. KafkaUserTlsExternalClientAuthentication schema reference

KafkaUserSpec で使用

typeプロパティは、KafkaUserTlsExternalClientAuthenticationタイプの使用を次のものから区別する識別記号です。 KafkaUserTlsClientAuthentication（カフカユーザーTlsクライアント認証, KafkaUserScramSha512ClientAuthentication（カフカユーザースクラムシャ512クライアント認証. KafkaUserTlsExternalClientAuthenticationのタイプには、tls-externalという値が必要です。

プロパティー	説明
type	**tls-external**でなければなりません。
string	

14.2.92. KafkaUserScramSha512ClientAuthentication スキーマ参照

KafkaUserSpec で使用

typeプロパティは、KafkaUserScramSha512ClientAuthenticationタイプの使用を以下のように区別する識別記号です。 KafkaUserTlsClientAuthentication（カフカユーザータイムズクライアント認証, KafkaUserTlsExternalClientAuthentication（カフカユーザーTlsエクスターナルクライアント認証.KafkaUserScramSha512ClientAuthentication タイプには scram-sha-512 の値が必要です。

プロパティー	説明
password パスワード	ユーザーのパスワードを指定します。設定されていない場合、新しいパスワードはUser Operatorによって生成されます。
type string	**scram-sha-512** でなければなりません。

14.2.93. パスワードスキーマの参照

で使用されます。 **KafkaUserScramSha512ClientAuthentication**

プロパティー	説明
valueFrom **PasswordSource**	パスワードを読み取るための秘密。

14.2.94. PasswordSourceスキーマリファレンス

で使用しています。 パスワード

プロパティー	説明
secretKeyRef SecretKeySelector	リソースの名前空間にあるSecretのキーを選択します。詳細は、core/v1 secretkeyselector の外部ドキュメント を参照してください。

14.2.95. KafkaUserAuthorizationSimple スキーマ参照

KafkaUserSpec で使用

typeプロパティは、KafkaUserAuthorizationSimpleタイプの使用と、将来追加される可能性のある他のサブタイプとを区別する識別記号です。KafkaUserAuthorizationSimple タイプには simple の値が必要です。

プロパティー	説明
type	**simple** でなければなりません。
string	
ACL	このユーザーに適用される必要のある ACL ルールの一覧。
AclRule array	

14.2.96. AclRule スキーマ参照

KafkaUserAuthorizationSimple で使用

AclRuleスキーマプロパティの全リスト

ブローカーがAclAuthorizerを使用している場合に、KafkaUserのアクセスコントロールルールを設定します。

認証付きKafkaUserの設定例

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

14.2.96.1. resource

ルールが適用されるリソースを指定するには、resourceプロパティを使用します。

簡易承認は、type プロパティーに指定される、以下の 4 つのリソースタイプをサポートします。

- トピック (topic)
- コンシューマーグループ (group)
- クラスター (cluster)
- トランザクション ID (transactionalId)

Topic、Group、および Transactional ID リソースでは、name プロパティーでルールが適用されるリソースの名前を指定できます。

クラスタータイプのリソースには名前がありません。

名前は、patternType プロパティーを使用して literal または prefix として指定されます。

- リテラル (literal) 名には、name フィールドに指定された名前がそのまま使われます。
- プレフィックス名では、名前の値をプレフィックスとして使用し、その値で始まる名前を持つすべてのリソースにルールを適用します。

patternTypeがリテラルに設定されている場合、名前に*を設定することで、そのルールがすべてのリソースに適用されることを示すことができます。

すべてのトピックのメッセージを読むことを許可するACLルールの例

```
acls:
  - resource:
      type: topic
      name: "*"
      patternType: literal
    operation: Read
```

14.2.96.2. type

操作を許可するか拒否するか（現在はサポートされていません）というルールの種類。

type フィールドの設定は任意です。type の指定がない場合、ACL ルールは allow ルールとして処理されます。

14.2.96.3. operation

ルールで許可または拒否する操作を指定します。

以下の操作がサポートされます。

- Read
- Write
- Delete

- Alter
- Describe
- All
- IdempotentWrite
- ClusterAction
- Create
- AlterConfigs
- DescribeConfigs

特定の操作のみが各リソースで機能します。

AclAuthorizer、ACL、およびサポートされているリソースと操作の組み合わせの詳細については、「Authorization and ACL」を参照してください。

14.2.96.4. host

hostプロパティを使用して、ルールを許可または拒否するリモートホストを指定します。

アスタリスク (*) を使用して、すべてのホストからの操作を許可または拒否します。host フィールドの設定は任意です。host を指定しないと、値 * がデフォルトで使用されます。

14.2.96.5. AclRuleスキーマのプロパティ

プロパティー	説明
host	ACL ルールに記述されているアクションを許可または拒否するホスト。
string	
operation	許可または拒否される操作。サポートされる操作: Read、Write、Create、Delete、Alter、Describe、ClusterAction、AlterConfigs、DescribeConfigs、IdempotentWrite、All
string ([Read、Write、Delete、Alter、Describe、All、IdempotentWrite、ClusterAction、Create、AlterConfigs、DescribeConfigs] のいずれか)	
resource	指定の ACL ルールが適用されるリソースを示します。タイプは、指定のオブジェクト内の **resource.type** プロパティーの値によって異なり、[topic、group、cluster、transactionalId] のいずれかでなければなりません。
AclRuleTopicResource、**AclRuleGroupResource**、**AclRuleClusterResource**、**AclRuleTransactionalIdResource**	
type	ルールのタイプ。現在サポートされているタイプは **allow** のみです。**allow** タイプの ACL ルールを使用すると、ユーザーは指定した操作を実行できます。デフォルト値は **allow** です。
string ([allow、deny] のいずれか)	

14.2.97. AclRuleTopicResource スキーマ参照

AclRule で使用されます。

typeプロパティは、AclRuleTopicResourceタイプの使用を以下と区別する識別要素である。AclRuleGroupResource（アクリールグループリソース, AclRuleClusterResource（アクルーバルクラスターリソース, AclRuleTransactionalIdResource（アクルール トランザクション アイディール リソース.AclRuleTopicResource タイプには topic の値が必要です。

プロパティー	説明
type	**topic** でなければなりません。
string	
name	指定の ACL ルールが適用されるリソースの名前。**patternType** フィールドと組み合わせて、接頭辞のパターンを使用できます。
string	

プロパティー	説明
patternType	リソースフィールドで使用されるパターンを指定します。サポートされるタイプは **literal** と **prefix** です。**literal** パターンタイプでは、リソースフィールドは完全なトピック名の定義として使用されます。**prefix** パターンタイプでは、リソース名は接頭辞としてのみ使用されます。デフォルト値は **literal** です。
string ([prefix、literal] のいずれか)	

14.2.98. AclRuleGroupResource スキーマ参照

AclRule で使用されます。

typeプロパティは、AclRuleGroupResourceタイプの使用を以下と区別する識別要素である。AclRuleTopicResource（アクルールトピックリソース, AclRuleClusterResource（クラスターリソース, AclRuleTransactionalIdResource（アクルー トランザクション ID リソース.AclRuleGroupResource タイプには group の値が必要です。

プロパティー	説明
type	**group** でなければなりません。
string	
name	指定の ACL ルールが適用されるリソースの名前。**patternType** フィールドと組み合わせて、接頭辞のパターンを使用できます。
string	
patternType	リソースフィールドで使用されるパターンを指定します。サポートされるタイプは **literal** と **prefix** です。**literal** パターンタイプでは、リソースフィールドは完全なトピック名の定義として使用されます。**prefix** パターンタイプでは、リソース名は接頭辞としてのみ使用されます。デフォルト値は **literal** です。
string ([prefix、literal] のいずれか)	

14.2.99. AclRuleClusterResource スキーマ参照

AclRule で使用されます。

typeプロパティは、AclRuleClusterResourceタイプの使用を次のものから区別するための識別要素

である。 AclRuleTopicResource（アクルールトピックリソース, AclRuleGroupResource（規約グループリソース, AclRuleTransactionalIdResource (AclRuleClusterResource).AclRuleClusterResource タイプには cluster の値が必要です。

プロパティー	説明
type string	**cluster** でなければなりません。

14.2.100. AclRuleTransactionalIdResource スキーマ参照

AclRule で使用されます。

typeプロパティは、AclRuleTransactionalIdResourceタイプの使用を以下と区別する識別子である。 AclRuleTopicResource（アクルールトピックリソース, AclRuleGroupResource（ルールグループリソース, AclRuleClusterResource（アクルーレクラスターリソース.AclRuleTransactionalIdResource タイプには transactionalId の値が必要です。

プロパティー	説明
type string	**transactionalId** でなければなりません。
name string	指定の ACL ルールが適用されるリソースの名前。**patternType** フィールドと組み合わせて、接頭辞のパターンを使用できます。
patternType string ([prefix、literal] のいずれか)	リソースフィールドで使用されるパターンを指定します。サポートされるタイプは **literal** と **prefix** です。**literal** パターンタイプでは、リソースフィールドはフルネームの定義として使用されます。**prefix** パターンタイプでは、リソース名は接頭辞としてのみ使用されます。デフォルト値は **literal** です。

14.2.101. KafkaUserQuotas スキーマ参照

KafkaUserSpec で使用

KafkaUserQuotasスキーマプロパティの全リスト

Kafkaでは、ユーザーがクオータを設定して、クライアントによるリソースの使用を制御することができます。

14.2.101.1. quotas

クライアントを設定して、以下のタイプのクォータを使用できます。

- ネットワーク使用率 クォータは、クォータを共有するクライアントの各グループのバイトレートしきい値を指定します。

- CPU 使用率クォータは、クライアントからのブローカー要求のウィンドウを指定します。ウィンドウは、クライアントが要求を行う時間の割合 (パーセント) です。クライアントはブローカーの I/O スレッドおよびネットワークスレッドで要求を行います。

- パーティション変更クォータは、クライアントが 1 秒ごとに実行できるパーティション変更の数を制限します。

パーティション変更クォータにより、Kafka クラスターが同時にトピック操作に圧倒されないようにします。パーティション変更は、次のタイプのユーザー要求に応答して発生します。

- 新しいトピック用のパーティションの作成

- 既存のトピックへのパーティションの追加

- トピックからのパーティションの削除

パーティション変更クオータを設定して、ユーザー要求に対して変更が許可されるレートを制御できます。

Kafka クライアントにクォータを使用することは、さまざまな状況で役に立つ場合があります。レートが高すぎる要求を送信する Kafka プロデューサーを誤って設定したとします。このように設定が間違っていると、他のクライアントにサービス拒否を引き起こす可能性があるため、問題のあるクライ

アントはブロックする必要があります。ネットワーク制限クォータを使用すると、他のクライアントがこの状況の著しい影響を受けないようにすることが可能です。

AMQ Streams はユーザーレベルのクォータをサポートしますが、クライアントレベルのクォータはサポートしません。

Kafka ユーザークォータの設定例

```
spec:
  quotas:
    producerByteRate: 1048576
    consumerByteRate: 2097152
    requestPercentage: 55
    controllerMutationRate: 10
```

Kafka ユーザークォータの詳細は Apache Kafka ドキュメント を参照してください。

14.2.101.2. KafkaUserQuotasスキーマのプロパティ

プロパティー	説明
consumerByteRate integer	グループのクライアントにスロットリングが適用される前に、各クライアントグループがブローカーから取得できる最大 bps (ビット毎秒) のクオータ。ブローカーごとに定義されます。
controllerMutationRate number	トピックの作成リクエスト、パーティションの作成リクエスト、トピックの削除リクエストで変更が受け入れられるレートのクオータ。レートは、作成または削除されたパーティション数で累積されます。
producerByteRate integer	グループのクライアントにスロットリングが適用される前に、各クライアントグループがブローカーにパブリッシュできる最大 bps (ビット毎秒) のクオータ。ブローカーごとに定義されます。
requestPercentage integer	各クライアントグループの最大 CPU 使用率のクォータ。ネットワークと I/O スレッドの比率 (パーセント) として指定。

14.2.102. KafkaUserTemplateスキーマリファレンス

KafkaUserSpec で使用

KafkaUserTemplateスキーマプロパティの全リスト

User Operator によって作成されるシークレットの追加ラベルおよびアノテーションを指定します。

KafkaUserTemplateの例。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls
  template:
    secret:
      metadata:
        labels:
          label1: value1
        annotations:
          anno1: value1
  # ...
```

14.2.102.1. KafkaUserTemplateスキーマのプロパティ

プロパティー	説明
secret ResourceTemplate	KafkaUser リソースのテンプレート。テンプレートでは、パスワード付き**シークレット**やTLS証明書の生成方法をユーザーが指定できます。

14.2.103. KafkaUserStatus スキーマ参照

KafkaUser で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
username	ユーザー名。
string	
secret	認証情報が保存される **Secret** の名前。
string	

14.2.104. KafkaMirrorMaker スキーマ参照

type KafkaMirrorMaker が非推奨になりました。代わりに KafkaMirrorMaker2 を使用してください。

プロパティー	説明
spec	Kafka MirrorMaker の仕様。
KafkaMirrorMakerSpec	
status	Kafka MirrorMaker のステータス。
KafkaMirrorMakerStatus	

14.2.105. KafkaMirrorMakerSpec スキーマ参照

KafkaMirrorMaker で使用

KafkaMirrorMakerSpecスキーマプロパティの全リスト

Kafka MirrorMaker を設定します。

14.2.105.1. include

Kafka MirrorMakerがソースからターゲットのKafkaクラスターにミラーリングするトピックのリストを構成するには、includeプロパティを使用します。

このプロパティーでは、簡単な単一のトピック名から複雑なパターンまですべての正規表現が許可されます。例えば、A|Bを使ってトピックAとBをミラーリングしたり、*を使ってすべてのトピックをミラーリングすることができます。また、複数の正規表現をコンマで区切って Kafka MirrorMaker に渡すこともできます。

14.2.105.2. KafkaMirrorMakerConsumerSpec and KafkaMirrorMakerProducerSpec

KafkaMirrorMakerConsumerSpecとKafkaMirrorMakerProducerSpecを使って、ソース（コンシューマー）とターゲット（プロデューサー）のクラスターを構成します。

Kafka MirrorMaker は常に 2 つの Kafka クラスター (ソースおよびターゲット) と連携します。接続を確立するために、ソースとターゲットのKafkaクラスターのブートストラップサーバーを、コンマで区切ったリストの HOSTNAME:PORTペアで指定します。カンマで区切られた各リストには、1つ以上のKafkaブローカー、または1つのペアとして指定されたKafkaブローカーを指すサービスが含まれます。 HOSTNAME:PORTペアで指定されています。

14.2.105.3. ログ

Kafka MirrorMaker には、独自の設定可能なロガーがあります。

- mirrormaker.root.logger

MirrorMaker では Apache log4j ロガー実装が使用されます。

logging プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる

ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は log4j.properties を使用して記述されます。logging.valueFrom.configMapKeyRef.name および logging.valueFrom.configMapKeyRef.key プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、「Apache logging services」を参照してください。

inline および external ロギングの例は次のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: inline
    loggers:
      mirrormaker.root.logger: "INFO"
  # ...
```

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: mirror-maker-log4j.properties
  # ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは jvmOptions プロパティーを使用して 有効（または無効）にすることもできます。

14.2.105.4. KafkaMirrorMakerSpecスキーマのプロパティ

プロパティー	説明
version string	Kafka MirrorMaker のバージョン。デフォルトは 3.0.0 です。バージョンのアップグレードまたはダウングレードに必要なプロセスを理解するには、ドキュメントを参照してください。
replicas	**Deployment** の Pod 数。

プロパティー	説明
integer	
image	Pod の Docker イメージ。
string	
consumer	ソースクラスターの設定。
KafkaMirrorMakerConsumerSpec	
producer	ターゲットクラスターの設定。
KafkaMirrorMakerProducerSpec	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
whitelist	**ホワイトリストプロパティは廃止されましたので、spec.includeを使って設定してください。**ミラーリングに含まれるトピックの一覧。このオプションは、Java スタイルの正規表現を使用するあらゆる正規表現を許可します。AとBという2つのトピックをミラーリングするには、**A|B**という表現を使います。また、特殊なケースとして、正規表現の*を使ってすべてのトピックをミラーリングすることもできます。複数の正規表現をコンマで区切って指定することもできます。
string	
include	ミラーリングに含まれるトピックの一覧。このオプションは、Java スタイルの正規表現を使用するあらゆる正規表現を許可します。AとBという2つのトピックをミラーリングするには、**A|B**という表現を使います。また、特殊なケースとして、正規表現の*を使ってすべてのトピックをミラーリングすることもできます。複数の正規表現をコンマで区切って指定することもできます。
string	
jvmOptions	Pod の JVM オプション。
JvmOptions	
logging	MirrorMaker のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	

プロパティー	説明
metricsConfig **JmxPrometheusExporterMetrics**	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
tracing **JaegerTracing**	Kafka MirrorMaker でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger] の 1 つでなければなりません。
template **KafkaMirrorMakerTemplate**	Kafka MirrorMaker のリソースである **Deployments** および **Pods**の生成方法を指定するテンプレート。
livenessProbe **Probe**	Pod の liveness チェック。
readinessProbe **Probe**	Pod の readiness チェック。

14.2.106. KafkaMirrorMakerConsumerSpec スキーマ参照

KafkaMirrorMakerSpec で使用

KafkaMirrorMakerConsumerSpecスキーマプロパティの全リスト

MirrorMaker コンシューマーを設定します。

14.2.106.1. numStreams

consumer.numStreams プロパティーを使用して、コンシューマーのストリームの数を設定します。

コンシューマースレッドの数を増やすと、ミラーリングトピックのスループットを増やすことができます。コンシューマースレッドは、Kafka MirrorMaker に指定されたコンシューマーグループに属します。トピックパーティションはコンシューマースレッド全体に割り当てられ、メッセージが並行して消費されます。

14.2.106.2. offsetCommitInterval

consumer.offsetCommitInterval プロパティーを使用して、コンシューマーのオフセット自動コミット間隔を設定します。

Kafka MirrorMaker によってソース Kafka クラスターのデータが消費された後に、オフセットがコミットされる通常の間隔を指定できます。間隔はミリ秒単位で設定され、デフォルト値は 60,000 です。

14.2.106.3. 設定

consumer.configプロパティを使用して、コンシューマーのKafkaオプションを設定します。

configプロパティには、Kafka MirrorMaker コンシューマの設定オプションがキーとして含まれており、値は以下の JSON タイプのいずれかで設定されます。

- 文字列
- 数値
- ブール値

TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。また、 ssl.endpoint.identification.algorithmプロパティを設定して、ホスト名の検証を有効または無効にすることもできます。

例外

コンシューマー向けの Apache Kafka 設定ドキュメント に記載されているオプションを指定および設定できます。

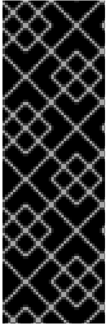
しかし、以下に関連する AMQ Streams によって自動的に設定され、直接管理されるオプションには例外があります。

- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- コンシューマーグループ識別子
- インターセプター

以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはすべて禁止されています。

- bootstrap.servers
- group.id
- interceptor.classes
- sslを使用しています。(特定の例外を含まない)
- sasl.
- security.

禁止されているオプションが config プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka MirrorMaker に渡されます。



重要

Cluster Operator では、提供された config オブジェクトのキーまたは値は検証されません。無効な設定が指定されると、Kafka MirrorMaker が起動しなかったり、不安定になったりする場合があります。このような場合、KafkaMirrorMaker.spec.consumer.configオブジェクトの構成を修正し、クラスター運用者がKafka MirrorMakerの新しい構成をロールアウトする必要があります。

14.2.106.4. groupId

consumer.groupId プロパティーを使用して、コンシューマーにコンシューマーグループ ID を設定します。

Kafka MirrorMaker は Kafka コンシューマーを使用してメッセージを消費し、他の Kafka コンシューマークライアントと同様に動作します。ソース Kafka クラスターから消費されるメッセージは、ターゲット Kafka クラスターにミラーリングされます。パーティションの割り当てには、コンシューマーがコンシューマーグループの一部である必要があるため、グループ ID が必要です。

14.2.106.5. KafkaMirrorMakerConsumerSpec schema properties

プロパティー	説明
numStreams integer	作成するコンシューマーストリームスレッドの数を指定します。
offsetCommitInterval integer	オフセットの自動コミット間隔をミリ秒単位で指定します。デフォルト値は 60000 です。
bootstrapServers string	Kafka クラスターへの最初の接続を確立するための host:port ペアの一覧。
groupId string	このコンシューマーが属するコンシューマーグループを識別する一意の文字列。
authentication **KafkaClientAuthenticationTls**、**KafkaClientAuthenticationScramSha512**、**KafkaClientAuthenticationPlain**、**KafkaClientAuthenticationOAuth**	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls、scram-sha-512、plain、oauth] のいずれかでなければなりません。

プロパティー	説明
config	MirrorMaker コンシューマーの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、bootstrap.servers、group.id、sasl.、security.、interceptor.classes (次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols)。
map	
tls	MirrorMaker をクラスターに接続するための TLS 設定。
ClientTls	

14.2.107. KafkaMirrorMakerProducerSpec スキーマ参照

KafkaMirrorMakerSpec で使用

KafkaMirrorMakerProducerSpecスキーマプロパティの全リスト

MirrorMaker プロデューサーを設定します。

14.2.107.1. abortOnSendFailure

producer.abortOnSendFailure プロパティーを使用して、プロデューサーからメッセージ送信の失敗を処理する方法を設定します。

デフォルトでは、メッセージを Kafka MirrorMaker から Kafka クラスターに送信する際にエラーが発生した場合、以下が行われます。

- Kafka MirrorMaker コンテナーが OpenShift で終了します。
- その後、コンテナーが再作成されます。

abortOnSendFailure オプションを false に設定した場合、メッセージ送信エラーは無視されます。

14.2.107.2. 設定

producer.configプロパティを使用して、producerのKafkaオプションを設定します。

configプロパティには、Kafka MirrorMaker プロデューサー設定オプションがキーとして含まれており、値は以下の JSON タイプのいずれかで設定されます。

- 文字列
- 数値
- ブール値

TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。また、 ssl.endpoint.identification.algorithmプロパティを設定して、ホスト名の検証を有効または無効にすることもできます。

例外

プロデューサー向けの Apache Kafka 設定ドキュメント に記載されているオプションを指定および設定できます。

しかし、以下に関連する AMQ Streams によって自動的に設定され、直接管理されるオプションには例外があります。

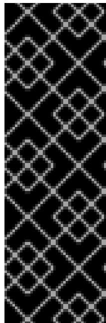
- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- インターセプター

以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはす

べて禁止されています。

- **bootstrap.servers**
- **interceptor.classes**
- **sslを使用しています。(特定の例外を含まない)**
- **sasl.**
- **security.**

禁止されているオプションが config プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka MirrorMaker に渡されます。



重要

Cluster Operator では、提供された config オブジェクトのキーまたは値は検証されません。無効な設定が指定されると、Kafka MirrorMaker が起動しなかったり、不安定になったりする場合があります。このような場合、KafkaMirrorMaker.spec.producer.configオブジェクトの構成を修正し、クラスター運用者がKafka MirrorMakerの新しい構成をロールアウトする必要があります。

14.2.107.3. KafkaMirrorMakerProducerSpec schema properties

プロパティー	説明
bootstrapServers	Kafka クラスターへの最初の接続を確立するための host:port ペアの一覧。
string	
abortOnSendFailure	送信失敗時に MirrorMaker が終了するように設定するフラグ。デフォルト値は **true** です。
boolean	

プロパティー	説明
authentication	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls、scram-sha-512、plain、oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls、KafkaClientAuthenticationScramSha512、KafkaClientAuthenticationPlain、KafkaClientAuthenticationOAuth	
config	MirrorMaker プロデューサーの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、bootstrap.servers、sasl.、security.、interceptor.classes (次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols)。
map	
tls	MirrorMaker をクラスターに接続するための TLS 設定。
ClientTls	

14.2.108. KafkaMirrorMakerTemplate スキーマ参照

KafkaMirrorMakerSpec で使用

プロパティー	説明
deployment	Kafka MirrorMaker **Deployment** のテンプレート。
DeploymentTemplate	
pod	Kafka MirrorMaker **Pods** のテンプレート。
PodTemplate	
podDisruptionBudget	Kafka MirrorMaker **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	
mirrorMakerContainer	Kafka MirrorMaker コンテナーのテンプレート。
ContainerTemplate	
serviceAccount	Kafka MirrorMaker サービスアカウントのテンプレート。
ResourceTemplate	

14.2.109. KafkaMirrorMakerStatus スキーマ参照

KafkaMirrorMaker で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
labelSelector	このリソースを提供する Pod のラベルセレクター。
string	
replicas	このリソースを提供するために現在使用されている Pod の数。
integer	

14.2.110. KafkaBridge スキーマ参照

プロパティー	説明
spec	Kafka Bridge の仕様。
KafkaBridgeSpec	
status	Kafka Bridge のステータス。
KafkaBridgeStatus	

14.2.111. KafkaBridgeSpec スキーマ参照

KafkaBridge で使用

KafkaBridgeSpecスキーマプロパティの全リスト

Kafka Bridge クラスターを設定します。

設定オプションは以下に関連しています。

- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- コンシューマー設定
- プロデューサーの設定
- HTTP の設定

14.2.111.1. ログ

Kafka Bridge には独自の設定可能なロガーがあります。

- logger.bridge
- logger.<operation-id>

を置き換えることができます。 <operation-id> <operation-id>logger で特定の操作に対するログレベルを設定します。

- createConsumer
- deleteConsumer

- **subscribe**
- **unsubscribe**
- **poll**
- **assign**
- **commit**
- **send**
- **sendToPartition**
- **seekToBeginning**
- **seekToEnd**
- **seek**
- **healthy**
- **ready**
- **openapi**

各操作は OpenAPI 仕様にしたがって定義されます。各操作にはブリッジが HTTP クライアントから要求を受信する対象の API エンドポイントがあります。各エンドポイントのログレベルを変更すると、送信および受信 HTTP リクエストに関する詳細なログ情報を作成できます。

各ロガーは、http.openapi.operation.<operation-id>のような名前を割り当てて設定しなければなりません。例えば、送信操作ロガーのロギングレベルを設定するとは、次のように定義することを意味します。

```
logger.send.name = http.openapi.operation.send
logger.send.level = DEBUG
```

Kafka Bridgeは、Apachelog4j2のロガー実装を使用しています。ロガーはlog4j2.propertiesファイルで定義され、健康で準備の整ったエンドポイントに対しては以下のようなデフォルトの設定になっています。

```
logger.healthy.name = http.openapi.operation.healthy
logger.healthy.level = WARN
logger.ready.name = http.openapi.operation.ready
logger.ready.level = WARN
```

その他すべての操作のログレベルは、デフォルトで INFO に設定されます。

logging プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、logging.valueFrom.configMapKeyRef.name プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。 logging.valueFrom.configMapKeyRef.nameとlogging.valueFrom.configMapKeyRef.keyのプロパティは必須です。name や key が設定されていない場合は、デフォルトのロギングが使用されます。ConfigMap 内では、ロギング設定は log4j.properties を使用して記述されます。ログレベルの詳細は、「Apache logging services」を参照してください。

ここで、inline および external ロギングの例を示します。

inline ロギング

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
spec:
  # ...
  logging:
    type: inline
    loggers:
      logger.bridge.level: "INFO"
```

```
    # enabling DEBUG just for send operation
    logger.send.name: "http.openapi.operation.send"
    logger.send.level: "DEBUG"
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: bridge-logj42.properties
  # ...
```

設定されていない利用可能なロガーのレベルは **OFF** に設定されています。

Cluster Operator を使用して Kafka Bridge がデプロイされた場合、Kafka Bridge のロギングレベルの変更は動的に適用されます。

外部ロギングを使用する場合は、ロギングアペンダーが変更されるとローリングアップデートがトリガーされます。

ガベッジコレクター (GC)

ガベッジコレクターのロギングは jvmOptions プロパティーを使用して 有効（または無効）にすることもできます。

14.2.111.2. KafkaBridgeSpecのスキーマプロパティ

プロパティー	説明
replicas	**Deployment** の Pod 数。
integer	
image	Pod の Docker イメージ。
string	
bootstrapServers	Kafka クラスターへの最初の接続を確立するための host:port ペアの一覧。
string	
tls	Kafka Bridge をクラスターに接続するための TLS 設定。
ClientTls	
authentication	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls、scram-sha-512、plain、oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls、**KafkaClientAuthenticationScramSha512**、**KafkaClientAuthenticationPlain**、**KafkaClientAuthenticationOAuth**	
http	HTTP 関連の設定。
KafkaBridgeHttpConfig	
adminClient	Kafka AdminClient 関連の設定。
KafkaBridgeAdminClientSpec	
コンシューマー	Kafka コンシューマーに関連する設定。
KafkaBridgeConsumerSpec	
producer	Kafka プロデューサーに関連する設定。
KafkaBridgeProducerSpec	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
jvmOptions	**現時点でサポートされていない** Pod の JVM オプション。

プロパティー	説明
JvmOptions	
logging **InlineLogging**、**ExternalLogging**	Kafka Bridge のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
enableMetrics boolean	Kafka Bridge のメトリクスを有効にします。デフォルトは false です。
livenessProbe **Probe**	Pod の liveness チェック。
readinessProbe **Probe**	Pod の readiness チェック。
template **KafkaBridgeTemplate**	Kafka Bridge リソースのテンプレート。ユーザーはテンプレートにより、**Deployment** および **Pod** の生成方法を指定できます。
tracing **JaegerTracing**	Kafka Bridge でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger] の 1 つでなければなりません。

14.2.112. KafkaBridgeHttpConfig スキーマ参照

KafkaBridgeSpec で使用

KafkaBridgeHttpConfigスキーマプロパティの全リスト

Kafka Bridge の Kafka クラスターへの HTTP アクセスを設定します。

デフォルトの HTTP 設定では、8080 番ポートで Kafka Bridge をリッスンします。

14.2.112.1. cors

HTTP プロパティーは、Kafka クラスターへの HTTP アクセスを有効にする他に、CPRS (Cross-Origin Resource Sharing) により Kafka Bridge のアクセス制御を有効化または定義する機能を提供します。CORS は、複数のオリジンから指定のリソースにブラウザーでアクセスできるようにする HTTP メカニズムです。CORS を設定するには、許可されるリソースオリジンのリストと、HTTP のアクセス方法を定義します。オリジンには、URL または Java 正規表現を使用できます。

Kafka Bridge HTTP の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  http:
    port: 8080
    cors:
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  # ...
```

14.2.112.2. KafkaBridgeHttpConfig schema properties

プロパティー	説明
port	サーバーがリッスンするポート。
integer	
cors	HTTP Bridge の CORS 設定。
KafkaBridgeHttpCors	

14.2.113. KafkaBridgeHttpCors スキーマ参照

KafkaBridgeHttpConfig で使用されます。

プロパティー	説明

プロパティー	説明
allowedOrigins	許可されるオリジンのリスト。Java の正規表現を使用できます。
文字列の配列	
allowedMethods	許可される HTTP メソッドのリスト。
文字列の配列	

14.2.114. KafkaBridgeAdminClientSpec schema reference

KafkaBridgeSpec で使用

プロパティー	説明
config	ブリッジによって作成された AdminClient インスタンスに使用される Kafka AdminClient 設定。
map	

14.2.115. KafkaBridgeConsumerSpec スキーマ参照

KafkaBridgeSpec で使用

KafkaBridgeConsumerSpecスキーマプロパティの全リスト

Kafka Bridge のコンシューマーオプションを鍵として設定します。

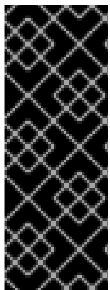
値は以下の JSON タイプのいずれかになります。

- **文字列**
- **数値**
- **ブール値**

AMQ Streams で直接管理されるオプションを除き、コンシューマー向けの Apache Kafka 設定ドキュメント に記載されているオプションを指定および設定できます。以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはすべて禁止されています。

- ssl.
- sasl.
- security.
- bootstrap.servers
- group.id

禁止されているオプションの1つがconfigプロパティに存在する場合、それは無視され、警告メッセージがCluster Operatorログファイルに出力されます。その他のオプションはすべて Kafka に渡されます。



重要

クラスタ・オペレータはconfigオブジェクト内のキーまたは値を検証しません。無効な設定を指定すると、Kafka Bridge クラスターが起動しなかったり、不安定になる可能性があります。Cluster Operator が新しい設定をすべての Kafka Bridge ノードにロールアウトできるように設定を修正します。

禁止されているオプションには例外があります。TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。

Kafka Bridge コンシューマーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
```

```
# ...
consumer:
 config:
   auto.offset.reset: earliest
   enable.auto.commit: true
   ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
   ssl.enabled.protocols: "TLSv1.2"
   ssl.protocol: "TLSv1.2"
   ssl.endpoint.identification.algorithm: HTTPS
 # ...
```

14.2.115.1. KafkaBridgeConsumerSpec schema properties

プロパティー	説明
config	ブリッジによって作成されたコンシューマーインスタンスに使用される Kafka コンシューマーの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、bootstrap.servers、group.id、sasl.、security. (次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols).
map	

14.2.116. KafkaBridgeProducerSpec スキーマ参照

KafkaBridgeSpec で使用

KafkaBridgeProducerSpecスキーマプロパティの全リスト

Kafka Bridge のプロデューサーオプションを鍵として設定します。

値は以下の JSON タイプのいずれかになります。

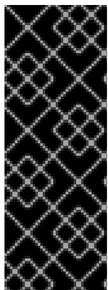
- 文字列
- 数値

- ブール値

AMQ Streams で直接管理されるオプションを除き、プロデューサー向けの Apache Kafka 設定ドキュメント に記載されているオプションを指定および設定できます。以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはすべて禁止されています。

- ssl.
- sasl.
- security.
- bootstrap.servers

禁止されているオプションの1つがconfigプロパティに存在する場合、それは無視され、警告メッセージがCluster Operatorログファイルに出力されます。その他のオプションはすべて Kafka に渡されます。



重要

クラスタ・オペレータはconfigオブジェクト内のキーまたは値を検証しません。無効な設定を指定すると、Kafka Bridge クラスターが起動しなかったり、不安定になる可能性があります。Cluster Operator が新しい設定をすべての Kafka Bridge ノードにロールアウトできるように設定を修正します。

禁止されているオプションには例外があります。TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。

Kafka Bridge プロデューサーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
```

```
# ...
producer:
  config:
    acks: 1
    delivery.timeout.ms: 300000
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    ssl.endpoint.identification.algorithm: HTTPS
  # ...
```

14.2.116.1. KafkaBridgeProducerSpec schema properties

プロパティー	説明
config	ブリッジによって作成されたプロデューサーインスタンスに使用される Kafka プロデューサーの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、bootstrap.servers、sasl.、security. (次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols).
map	

14.2.117. KafkaBridgeTemplate スキーマ参照

KafkaBridgeSpec で使用

プロパティー	説明
deployment	Kafka Bridge **Deployment** のテンプレート。
DeploymentTemplate	
pod	Kafka Bridge **Pod** のテンプレート。
PodTemplate	
apiService	Kafka Bridge API **Service** のテンプレート。
InternalServiceTemplate	
podDisruptionBudget	Kafka Bridge **PodDisruptionBudget** のテンプレート。

プロパティー	説明
PodDisruptionBudgetTemplate	
bridgeContainer	Kafka Bridge コンテナーのテンプレート。
ContainerTemplate	
serviceAccount	Kafka Bridge サービスアカウントのテンプレート。
ResourceTemplate	

14.2.118. KafkaBridgeStatus スキーマ参照

KafkaBridge で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
url	外部クライアントアプリケーションが Kafka Bridge にアクセスできる URL。
string	
labelSelector	このリソースを提供する Pod のラベルセレクター。
string	
replicas	このリソースを提供するために現在使用されている Pod の数。
integer	

14.2.119. KafkaConnector スキーマ参照

プロパティー	説明
spec	Kafka Connector の仕様。

プロパティー	説明
KafkaConnectorSpec	
status	Kafka Connector のステータス。
KafkaConnectorStatus	

14.2.120. KafkaConnectorSpec スキーマ参照

KafkaConnector で使用

プロパティー	説明
class	Kafka Connector のクラス。
string	
tasksMax	Kafka Connector のタスクの最大数。
integer	
設定	Kafka Connector の設定。次のプロパティーは設定できません: connector.class、 tasks.max
map	
pause	コネクターを一時停止すべきかどうか。デフォルトは false です。
boolean	

14.2.121. KafkaConnectorStatus スキーマ参照

KafkaConnector で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。

プロパティー	説明
integer	
connectorStatus	Kafka Connect REST API によって報告されるコネクターのステータス。
map	
tasksMax	Kafka Connector のタスクの最大数。
integer	
topics	Kafka Connector によって使用されるトピックのリスト。
文字列の配列	

14.2.122. KafkaMirrorMaker2 スキーマ参照

プロパティー	説明
spec	Kafka MirrorMaker 2.0 クラスターの仕様。
KafkaMirrorMaker2Spec	
status	Kafka MirrorMaker 2.0 クラスターのステータス。
KafkaMirrorMaker2Status	

14.2.123. KafkaMirrorMaker2Spec スキーマ参照

KafkaMirrorMaker2 で使用

プロパティー	説明
version	Kafka Connect のバージョン。デフォルトは 3.0.0 です。バージョンのアップグレードまたはダウングレードに必要なプロセスを理解するには、ユーザードキュメントを参照してください。
string	
replicas	Kafka Connect グループの Pod 数。
integer	
image	Pod の Docker イメージ。

プロパティー	説明
string	
connectCluster	Kafka Connect に使用されるクラスターエイリアス。エイリアスは **spec.clusters** にある一覧のクラスターと一致する必要があります。
string	
clusters	ミラーリング用の Kafka クラスター。
KafkaMirrorMaker2ClusterSpec array	
mirrors	MirrorMaker 2.0 コネクターの設定。
KafkaMirrorMaker2MirrorSpec array	
resources	CPU とメモリーリソースおよび要求された初期リソースの上限。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	
jvmOptions	Pod の JVM オプション。
JvmOptions	
jmxOptions	JMX オプション。
KafkaJmxOptions	
ログ	Kafka Connect のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
tracing	Kafka Connect でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger] の 1 つでなければなりません。
JaegerTracing	

プロパティー	説明
template **KafkaConnectTemplate**	Kafka ConnectとKafka Mirror Makerの2つのリソースのテンプレートです。ユーザーはテンプレートにより、**Deployment**、**Pod** および **Service** の生成方法を指定できます。
externalConfiguration **ExternalConfiguration**	Secret または ConfigMap から Kafka Connect Pod にデータを渡し、これを使用してコネクターを設定します。
metricsConfig **JmxPrometheusExporterMetrics**	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。

14.2.124. KafkaMirrorMaker2ClusterSpec スキーマ参照

KafkaMirrorMaker2Spec で使用

KafkaMirrorMaker2ClusterSpecスキーマプロパティの全リスト

ミラーリング用の Kafka クラスターを設定します。

14.2.124.1. 設定

Kafkaのオプションを設定するには、configプロパティを使用します。

標準の Apache Kafka 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。

TLSバージョンの特定の暗号スイートを使用するクライアント接続のために、許可されたsslプロパティを設定することができます。また、 ssl.endpoint.identification.algorithmプロパティを設定して、ホスト名の検証を有効または無効にすることもできます。

14.2.124.2. KafkaMirrorMaker2ClusterSpec schema properties

プロパティー	説明
alias	Kafka クラスターの参照に使用されるエイリアス。
string	
bootstrapServers	Kafka クラスターへの接続を確立するための **host:port** ペアのコンマ区切りリスト。
string	
tls	MirrorMaker 2.0 コネクターをクラスターに接続するための TLS 設定。
ClientTls	
authentication	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls、scram-sha-512、plain、oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls、**KafkaClientAuthenticationScramSha512**、**KafkaClientAuthenticationPlain**、**KafkaClientAuthenticationOAuth**	
config	MirrorMaker 2.0 クラスターの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、sasl.、security.、listeners、plugin.path、rest.、bootstrap.servers、consumer.interceptor.classes、producer.interceptor.classes (ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols を除く)
map	

14.2.125. KafkaMirrorMaker2MirrorSpec スキーマ参照

KafkaMirrorMaker2Spec で使用

プロパティー	説明
sourceCluster	Kafka MirrorMaker 2.0 コネクターによって使用されるソースクラスターのエイリアス。エイリアスは **spec.clusters** にある一覧のクラスターと一致する必要があります。
string	
targetCluster	Kafka MirrorMaker 2.0 コネクターによって使用されるターゲットクラスターのエイリアス。エイリアスは **spec.clusters** にある一覧のクラスターと一致する必要があります。
string	
sourceConnector	Kafka MirrorMaker 2.0 ソースコネクターの仕様。

プロパティー	説明
KafkaMirrorMaker2ConnectorSpec	
heartbeatConnector	Kafka MirrorMaker 2.0 ハートビートコネクターの仕様。
KafkaMirrorMaker2ConnectorSpec	
checkpointConnector	Kafka MirrorMaker 2.0 チェックポイントコネクターの仕様。
KafkaMirrorMaker2ConnectorSpec	
topicsPattern	ミラーリングするトピックに一致する正規表現 (例: "topic1|topic2|topic3")。コンマ区切りリストもサポートされます。
string	
topicsBlacklistPattern	**topicsBlacklistPatternプロパティは廃止され、.spec.mirror.topicsExcludePatternを使って設定するようになりました。**ミラーリングから除外するトピックに一致する正規表現。コンマ区切りリストもサポートされます。
string	
topicsExcludePattern	ミラーリングから除外するトピックに一致する正規表現。コンマ区切りリストもサポートされます。
string	
groupsPattern	ミラーリングされるコンシューマーグループに一致する正規表現。コンマ区切りリストもサポートされます。
string	
groupsBlacklistPattern	**groupsBlacklistPatternプロパティは非推奨となっており、.spec.mirror.groupsExcludePatternを使って設定してください。**ミラーリングから除外するコンシューマーグループに一致する正規表現。コンマ区切りリストもサポートされます。
string	
groupsExcludePattern	ミラーリングから除外するコンシューマーグループに一致する正規表現。コンマ区切りリストもサポートされます。
string	

14.2.126. KafkaMirrorMaker2ConnectorSpec スキーマ参照

KafkaMirrorMaker2MirrorSpec で使用

プロパティー	説明
tasksMax	Kafka Connector のタスクの最大数。
integer	
config	Kafka Connector の設定。次のプロパティーは設定できません: connector.class、 tasks.max
map	
pause	コネクターを一時停止すべきかどうか。デフォルトは false です。
boolean	

14.2.127. KafkaMirrorMaker2Status スキーマ参照

KafkaMirrorMaker2 で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
url	Kafka Connect コネクターの管理および監視用の REST API エンドポイントの URL。
string	
connectorPlugins	この Kafka Connect デプロイメントで使用できるコネクタープラグインの一覧。
ConnectorPlugin array	
connectors	Kafka Connect REST API によって報告される MirrorMaker 2.0 コネクターステータスの一覧。
map array	
labelSelector	このリソースを提供する Pod のラベルセレクター。
string	

プロパティー	説明
replicas integer	このリソースを提供するために現在使用されている Pod の数。

14.2.128. KafkaRebalance スキーマ参照

プロパティー	説明
spec **KafkaRebalanceSpec**	Kafka のリバランス (再分散) の仕様。
status **KafkaRebalanceStatus**	Kafka のリバランス (再分散) のステータス。

14.2.129. KafkaRebalanceSpec スキーマ参照

KafkaRebalance で使用されます。

プロパティー	説明
goals 文字列の配列	リバランスプロポーザルの生成および実行に使用されるゴールのリスト (優先度順)。サポートされるゴールは https://github.com/linkedin/cruise-control#goals を参照してください。空のゴールリストを指定すると、default.goals Cruise Control 設定パラメーターに宣言されたゴールが使用されます。
skipHardGoalCheck boolean	最適化プロポーザルの生成で、Kafka CR に指定されたハードゴールのスキップを許可するかどうか。これは、これらのハードゴールの一部が原因で分散ソリューションが検索できない場合に便利です。デフォルトは false です。
excludedTopics string	一致するトピックが最適化プロポーザルの計算から除外される正規表現。この正規表現は java.util.regex.Pattern クラスによって解析されます。サポートされる形式の詳細は、このクラスのドキュメントを参照してください。

プロパティー	説明
concurrentPartitionMovementsPerBroker	各ブローカーに出入りする継続中であるパーティションレプリカの移動の上限。デフォルトは 5 です。
integer	
concurrentIntraBrokerPartitionMovements	各ブローカー内のディスク間で継続中のパーティションレプリカ移動の上限。デフォルトは 2 です。
integer	
concurrentLeaderMovements	継続中のパーティションリーダーシップ移動の上限。デフォルトは 1000 です。
integer	
replicationThrottle	レプリカの移動に使用される帯域幅の上限 (バイト/秒単位)。デフォルトでは制限はありません。
integer	
replicaMovementStrategies	生成された最適化プロポーザルでのレプリカ移動の実行順序を決定するために使用されるストラテジークラス名のリスト。デフォルトでは、生成された順序でレプリカの移動が実行される BaseReplicaMovementStrategy が使用されます。
文字列の配列	

14.2.130. KafkaRebalanceStatus スキーマ参照

KafkaRebalance で使用されます。

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
sessionId	この KafkaRebalance リソースに関する Cruise Control へのリクエストのセッション識別子。これは、継続中のリバランス操作の状態を追跡するために、Kafka Rebalance operator によって使用されます。
string	
optimizationResult	最適化の結果を示す JSON オブジェクト。

プロパティー	説明
map	

付録A サブスクリプションの使用

AMQ Streams は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. サブスクリプション に移動します。
3. Activate a subscription に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合は、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの Product Downloads ページにログインします。
2. INTEGRATION AND AUTOMATION カテゴリーで Red Hat AMQ Streams エントリーを見つけます。
3. 必要な AMQ Streams 製品を選択します。Software Downloads ページが開きます。

4. **コンポーネントの Download リンクをクリックします。**

改訂日時： 2022-07-03 autoMember23 +1000