



Red Hat AMQ Streams 2.2

OpenShift での AMQ Streams の設定

OpenShift ContainerPlatform での AMQ Streams 2.2 のデプロイメントの設定および
管理

Red Hat AMQ Streams 2.2 OpenShift での AMQ Streams の設定

OpenShift ContainerPlatform での AMQ Streams 2.2 のデプロイメントの設定および管理

法律上の通知

Copyright © 2023 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

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構築します。

目次

多様性を受け入れるオープンソースの強化	4
第1章 設定の概要	5
1.1. カスタムリソースの設定	5
1.2. KAFKA ブローカーに接続するためのリスナー設定	5
1.3. 本書の表記慣例	6
1.4. 関連情報	6
第2章 OPENSIFT デプロイメントでの AMQ STREAMS の設定	7
2.1. 標準の KAFKA 設定プロパティの使用	7
2.2. KAFKA クラスターの設定	7
2.3. KAFKA CONNECT クラスターの設定	45
2.4. KAFKA MIRRORMAKER クラスターの設定	55
2.5. KAFKA MIRRORMAKER 2.0 クラスターの設定	59
2.6. KAFKA BRIDGE クラスターの設定	88
2.7. 大量のメッセージ処理	91
2.8. OPENSIFT リソースのカスタマイズ	96
2.9. POD スケジューリングの設定	98
2.10. ロギングの設定	105
第3章 外部ソースからの設定値の読み込み	111
3.1. 設定マップからの設定値の読み込み	111
3.2. 環境変数から設定値の読み込み	114
第4章 OPENSIFT クラスター外の KAFKA へのアクセス	116
4.1. ノードポートを使用した KAFKA へのアクセス	116
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. OPENSIFT ルートを使用した KAFKA へのアクセス	120
第5章 KAFKA へのセキュアなアクセスの管理	123
5.1. KAFKA のセキュリティーオプション	123
5.2. KAFKA クライアントのセキュリティーオプション	127
5.3. KAFKA ブローカーへのアクセスのセキュア化	133
5.4. OAUTH 2.0 トークンベース認証の使用	137
5.5. OAUTH 2.0 トークンベース承認の使用	159
第6章 STRIMZI OPERATOR の使用	181
6.1. AMQ STREAMS OPERATOR を使用した名前空間の監視	181
6.2. CLUSTER OPERATOR の使用	181
6.3. TOPIC OPERATOR の使用	200
6.4. USER OPERATOR の使用	208
6.5. フィーチャーゲートの設定	212
6.6. PROMETHEUS メトリクスを使用した OPERATOR の監視	216
第7章 CRUISE CONTROL によるクラスターのリバランス	217
7.1. CRUISE CONTROL とは	218
7.2. 最適化ゴールの概要	218
7.3. 最適化プロポーザルの概要	223
7.4. リバランスパフォーマンスチューニングの概要	228
7.5. CONFIGURING AND DEPLOYING CRUISE CONTROL WITH KAFKA	232
7.6. 最適化プロポーザルの生成	235
7.7. 最適化プロポーザルの承認	239

7.8. クラスターリバランスの停止	241
7.9. KAFKAREBALANCE リソースの問題の修正	242
第8章 SERVICE REGISTRY を使用したスキーマの検証	244
第9章 分散トレーシング	245
AMQ Streams によるトレーシングのサポート方法	245
手順の概要	245
9.1. OPENTRACING および JAEGER の概要	246
9.2. KAFKA クライアントのトレーシング設定	247
9.3. トレーサーでの KAFKA クライアントのインストルメント化	249
9.4. MIRRORMAKER、KAFKA CONNECT、および KAFKA BRIDGE のトレーシング設定	253
第10章 TLS 証明書の管理	257
10.1. 認証局	258
10.2. シークレット	261
10.3. 証明書の更新および有効期間	268
10.4. TLS 接続	279
10.5. クラスター CA を信頼する内部クライアントの設定	280
10.6. クラスター CA を信頼する外部クライアントの設定	282
10.7. KAFKA リスナー証明書	283
第11章 AMQ STREAMS の管理	287
11.1. カスタムリソースの使用	287
11.2. カスタムリソースの調整の一時停止	292
11.3. AMQ STREAMS DRAIN CLEANER での POD の退避	293
11.4. KAFKA および ZOOKEEPER クラスターの手動によるローリング更新の開始	297
11.5. 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	299
11.6. 永続ボリュームからのクラスターの復元	300
11.7. 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	306
11.8. よくある質問	307
第12章 カスタムリソース API のリファレンス	310
12.1. 共通の設定プロパティ	310
12.2. スキーマプロパティ	322
付録A サブスクリプションの使用	485
アカウントへのアクセス	485
サブスクリプションのアクティベート	485
Zip および Tar ファイルのダウンロード	485
DNF を使用したパッケージのインストール	485

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 設定の概要

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。

このガイドでは、AMQ Streams デプロイメントを設定および管理する方法について説明します。

1.1. カスタムリソースの設定

カスタムリソースを使用して、AMQ Streams デプロイメントを設定します。

カスタムリソースを使用して、次のコンポーネントのインスタンスを設定および作成できます。

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

カスタムリソース設定を使用してインスタンスを管理したり、デプロイメントを変更して追加機能を導入したりすることもできます。これには、以下をサポートする設定が含まれる場合があります。

- Kafka ブローカーへのクライアントアクセスの保護
- クラスター外からの Kafka ブローカーへのアクセス
- トピックの作成
- ユーザー (クライアント) の作成
- フィーチャーゲートの制御
- ロギングの頻度変更
- リソース制限とリクエストの割り当て
- AMQ Streams Drain Cleaner、Cruise Control、分散トレースなどの機能紹介

[カスタムリソース API リファレンス](#) では、設定で使用できるプロパティを説明しています。

1.2. KAFKA ブローカーに接続するためのリスナー設定

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設定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 schema reference](#) 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。外部クライアントを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。



注記

外部リスナーの使用時に Kafka クラスターをスケールアップする場合、すべての Kafka ブローカーのローリング更新がトリガーされる可能性があります。これは設定によって異なります。

1.3. 本書の表記慣例

ユーザー置換値

ユーザーが置き換える値は、**置き換え可能**な値とも呼ばれ、山かっこ (<>) を付けて **斜体** で表示されます。アンダースコア (_) は、複数単語の値に使用されます。値がコードまたはコマンドを参照する場合は **monospace** も使用されます。

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
```

1.4. 関連情報

- [AMQ Streams の概要](#)
- [AMQ Streams のデプロイおよびアップグレード](#)
- [AMQ Streams Kafka Bridge の使用](#)

第2章 OPENSIFT デプロイメントでの AMQ STREAMS の設定

カスタムリソースを使用して AMQ Streams の展開を設定します。AMQ Streams は、デプロイメント用の独自の Kafka コンポーネント設定を構築する際の開始点として役立つ [設定ファイルの例](#) を提供します。



注記

カスタムリソースに適用されるラベルは、クラスターを設定する 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簡単にラベルを付けることができます。

AMQ Streams デプロイメントのモニタリング

Prometheus および Grafana を使用して、AMQ Streams デプロイメントを監視できます。詳細は、[Kafka に追加されたメトリックの紹介](#) を参照してください。

2.1. 標準の KAFKA 設定プロパティの使用

標準の Kafka 設定プロパティを使用して Kafka コンポーネントを設定します。

プロパティは、以下の Kafka コンポーネントの設定を制御および調整するオプションを提供します。

- ブローカー
- Topics
- クライアント (プロデューサーとコンシューマー)
- 管理クライアント
- Kafka Connect
- Kafka Streams

ブローカーおよびクライアントパラメーターには、認可、認証、および暗号化を設定するオプションが含まれます。



注記

AMQ Streams on OpenShift では、一部の設定プロパティは AMQ Streams によって完全に管理されており、変更できません。

Kafka 設定プロパティの詳細と、プロパティを使用してデプロイメントを調整する方法は、以下のガイドを参照してください。

- [Kafka 設定プロパティ](#)
- [Kafka 設定のチューニング](#)

2.2. 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** リソースを使用して行います。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 で設定されます。

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 の詳細については、[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。

2.2.1. Kafka の設定

Kafka リソースのプロパティを使用して、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
- メトリック
- Cruise Control によるクラスターのリバランス

Kafka バージョン

Kafka **config** の **inter.broker.protocol.version** プロパティは、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 以降、**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るため、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は、**inter.broker.protocol.version** のアップグレードが必要です。詳細は、[Upgrading Kafka](#) 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** プロパティを編集します。
設定可能なプロパティ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: 3.2.3 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
  default.replication.factor: 3
  min.insync.replicas: 2
  inter.broker.protocol.version: "3.2"
  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  
# ...
```

- 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 **Kafka バージョン**。**アップグレード手順** に従うと、サポート対象のバージョンに変更できます。
- 3 ConfigMap を介して直接 (**inline**) または間接 (**external**) に追加された **Kafka ロガーとログレベル**。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 コンテナを再起動するタイミング (liveness) およびコンテナ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 TLS 暗号化は必要ありません。
- 13 クラスターサービス接尾辞 (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](#)を有効化します。簡易承認では、[AclAuthorizer](#) Kafka プラグインが使用されます。
- 18 ブローカー設定。AMQ Streams によって直接管理されないプロパティに限り、標準 [Apache Kafka 設定](#)の提供が可能です。
- 19 特定の [暗号スイート](#) または [TLS バージョン](#)を有効化するために TLS 暗号化が有効になっている [SSL プロパティ](#)。
- 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は拡張可能で、さらに [JBOD ストレージ](#)へのボリューム追加が可能です。
- 22 永続ストレージには、動的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 異なるラック、データセンター、または可用性ゾーンにレプリカを分散させるための [Rack awareness](#) 設定。[topologyKey](#) は、ラック ID を含むノードラベルと一致する必要があります。この設定で使用される例では、標準の [topology.kubernetes.io/zone](#) ラベルを使用するゾーンを指定します。
- 24 [Prometheus メトリック](#) は有効になっています。この例では、メトリックは Prometheus JMX Exporter (デフォルトのメトリックエクスポート) に対して設定されます。
- 25 Prometheus JMX Exporter 経由でメトリックを 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ックを有効にできます。
- 26 Kafka 設定と似たプロパティが含まれる、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失うと、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](#) ログ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のメトリクスデータを Kafka ブローカーから抽出する任意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。

- リソースを作成または更新します。

```
oc apply -f <kafka_configuration_file>
```

2.2.2. 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を管理します。

Entity Operator は以下で設定されます。

- 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。

これらの operator 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。Topic Operator と User Operator は、1 つの名前空間のみを監視できます。詳細は、「[AMQ Streams Operator を使用した名前空間の監視](#)」を参照してください。



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 pod に operator が含まれます。

2.2.2.1. Entity Operator の設定プロパティ

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

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

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

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

template プロパティには、ラベル、アノテーション、アフィニティ、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でデフォルト値が使用されます。

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

2.2.2.2. Topic Operator 設定プロパティ

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。以下のプロパティがサポートされます。

watchedNamespace

Topic Operator が **KafkaTopic** リソースを監視する OpenShift 名前空間。デフォルトは、Kafka クラスタがデプロイされた 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 プロパティを使用すると、使用されるコンテナイメージを設定できます。カスタムコンテナ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。

logging

logging プロパティは、Topic Operator のロギングを設定します。詳細は「[logging](#)」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
    # ...

```

2.2.2.3. User Operator 設定プロパティ

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。以下のプロパティがサポートされます。

watchedNamespace

User Operator が **KafkaUser** リソースを監視する OpenShift 名前空間。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 プロパティを使用すると、使用されるコンテナイメージを設定できます。カスタムコンテナ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。

logging

logging プロパティは、User Operator のロギングを設定します。詳細は「[logging](#)」を参照してください。

secretPrefix

secretPrefix プロパティは、KafkaUser リソースから作成されたすべての Secret の名前に接頭辞を追加します。たとえば、**secretPrefix: kafka-** は、すべてのシークレット名の前に **kafka-** を付けます。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:
    # ...

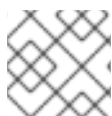
```

```
entityOperator:
# ...
userOperator:
  watchedNamespace: my-user-namespace
  reconciliationIntervalSeconds: 60
# ...
```

2.2.3. 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 でのみサポートされ、ZooKeeper ではサポートされません。

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 ノードで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の詳細は、スキーマリファレンスを参照してください。

- [EphemeralStorage](#) スキーマリファレンス
- [PersistentClaimStorage](#) スキーマリファレンス
- [JbodStorage](#) スキーマリファレンス



警告

Kafka クラスタ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。

2.2.3.1. 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 (ストレージエリアネットワーク) 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。

2.2.3.1.1. ファイルシステム

Kafka は、メッセージの保存にファイルシステムを使用します。AMQ Streams は、Kafka で一般的に使用される XFS および ext4 ファイルシステムと互換性があります。ファイルシステムを選択して設定するときは、デプロイメントの基盤となるアーキテクチャーと要件を考慮してください。

詳細については、Kafka ドキュメントの [Filesystem Selection](#) を参照してください。

2.2.3.1.2. 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 種類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、レプリケーション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。

2.2.3.2. 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
    # ...
```

2.2.3.2.1. 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 Pod 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。

2.2.3.3. 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
# ...

```

2.2.3.3.1. 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。
- ZooKeeper 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。

2.2.3.3.2. 永続ボリューム要求の命名

永続ストレージが使用されると、以下の名前永続ボリューム要求が作成されます。

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。

2.2.3.3.3. ログディレクトリー

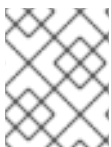
永続ボリュームは、以下のパスにマウントされるログディレクトリーとして Kafka ブローカーによって使用されます。

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

IDX は、Kafka ブローカー Pod 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。

2.2.3.4. 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 が稼働中である。
- 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_configuration_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。

関連情報

- 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。

2.2.3.5. 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。

2.2.3.5.1. 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** プロパティを使用すると、JBOD ストレージアレイまたは設定を設定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。

2.2.3.5.2. JBOD および 永続ボリューム要求

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の命名スキームは以下ようになります。

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。

2.2.3.5.3. 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。

2.2.3.6. 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_configuration_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ます。

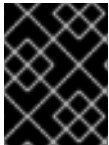
関連情報

トピックの再割り当てについて、詳しくは「[パーティション再割り当てツール](#)」を参照してください。

2.2.3.7. 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_configuration_file>
```

関連情報

トピックの再割り当てについて、詳しくは「[パーティション再割り当てツール](#)」を参照してください。

2.2.4. 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度割り当てる必要があります。

kafka-reassign-partitions.sh ツールを使用して、パーティションを再度割り当てます。このツールは、再割り当てを行う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。

2.2.4.1. 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。

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

StatefulSets を使用してブローカー Pod を管理する場合、クラスターから**任意の** Pod を削除することはできません。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。

2.2.4.2. パーティション再割り当てツール

現在、Topic Operator は別のブローカーへのレプリカの再割り当てをサポートしていないため、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度割り当てる必要があります。

ブローカー Pod 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度割り当てることができます。

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得し、**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る場合は使用できません。

--execute

再割り当て JSON ファイル を取得し、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 (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

--verify は、**--execute** ステップと同じ **再割り当て JSON ファイル** を使用して、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** は有効なトラフィックスロットル (**--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つの再割り当てのみを実行でき、実行中の再割り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。

2.2.4.2.1. 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。

2.2.4.2.2. 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の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リングの適用

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。

2.2.4.3. 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の再割り当てを実行します。

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。

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** が設定されています。

my-topic および my-cluster での操作を許可する ACL ルールを使用した Kafka ユーザーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication: ❶
    type: tls
  authorization:
    type: simple ❷
  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: "*"
  # ...
# ...
```

❶ 相互 **tls** として定義されたユーザー認証メカニズム。

❷ ACL ルールの承認および付随するリスト。



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** シークレットからクラスター 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。 **Kafka** リソースを使用して Kafka をデプロイすると、Kafka クラスター名 (**<cluster_name>-cluster-ca-cert**) でクラスター CA 証明書のシークレットが作成されます。例: **my-cluster-cluster-ca-cert**

2. AMQ Streams の Kafka イメージを使用してインタラクティブな Pod コンテナを新たに実行し、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-32-rhel8:2.2.2 <interactive_pod_name> -- /bin/sh -c "sleep 3600"
```

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

3. クラスター CA 証明書をインタラクティブな Pod コンテナにコピーします。

```
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 ユーザーのシークレットから、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> は 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 証明書のあるシークレットが 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、インタラクティブな Pod コンテナが 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。
前の手順でデプロイメント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 ❶
security.protocol=SSL ❷
ssl.truststore.location=/tmp/ca.p12 ❸
ssl.truststore.password=<truststore_password> ❹
ssl.keystore.location=/tmp/user.p12 ❺
ssl.keystore.password=<keystore_password> ❻
```

- ❶ 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。
- ❷ 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。
- ❸ 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。
- ❹ トラストストアにアクセスするためのパスワード (**ca.password**)。
- ❺ キースタアの場所には、Kafka ユーザーの公開鍵証明書 (**user.p12**) が含まれます。
- ❻ キースタアにアクセスするためのパスワード (**user.password**)。

7. **config.properties** ファイルをインタラクティブな Pod コンテナにコピーします。

```
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。トピック名をコンマ区切りのリストとして指定します。

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** ファイルをインタラクティブな Pod コンテナにコピーします。

```
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](#)

2.2.4.4. 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 クラスター内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 クラスター内のブローカーにパーティションを再度割り当てる方法を記述する必要があります。

この手順では、TLS を使用する安全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。
- **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 Pod コンテナを実行している。
- **KafkaUser** として接続されている。このユーザーは、Kafka クラスターとそのトピックの管理権限を指定する ACL ルールで設定されている。

[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. まだ確認していない場合には、[インタラクティブな Pod コンテナを実行](#) 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 コンテナにコピーします。

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

<interactive_pod_name> は 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。

- インタラクティブな Pod コンテナから **kafka-reassign-partitions.sh** スクリプトを使用して、パーティション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> は、独自の Kafka クラスターの名前に置き換えます。例: **my-cluster-kafka-bootstrap:9093**

レプリケーションにスロットリングを適用する場合、**--throttle** と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。以下に例を示します。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--reassignment-json-file /tmp/reassignment.json \
--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 ファイルに渡したターゲットの再割り当てです。

再割り当ての最中にスロットルを変更する必要がある場合は、同じコマンドに別のスロットル率を指定して実行します。以下に例を示します。

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

- 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かどうかを確認します。これは先ほどの手順と同じコマンドですが、**--verify** オプションの代わりに **--execute** 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 コマンドによって、移動した各パーティションが正常に完了したことが報告されると、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。

2.2.4.5. 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 クラスター内のブローカーの数を減らします。

再割り当てファイルでは、Kafka クラスターの残りのブローカーにパーティションを再割り当てする方法を記述する必要があります。最も番号の大きい Pod のブローカーが最初に削除されます。

この手順では、TLS を使用する安全なスケールアップ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。
- **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 Pod コンテナを実行している。
- **KafkaUser** として接続されている。このユーザーは、Kafka クラスターとそのトピックの管理権限を指定する ACL ルールで設定されている。

[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. まだ確認していない場合には、[インタラクティブな Pod コンテナを実行](#) 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 コンテナにコピーします。

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

<interactive_pod_name> は 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. インタラクティブな Pod コンテナから **kafka-reassign-partitions.sh** スクリプトを使用して、パーティション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>` は、独自の Kafka クラスターの名前に置き換えます。例: **my-cluster-kafka-bootstrap:9093**

レプリケーションにスロットリングを適用する場合、**--throttle** と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。以下に例を示します。

```
bin/kafka-reassign-partitions.sh --bootstrap-server
<cluster_name>-kafka-bootstrap:9093 \
--command-config /tmp/config.properties \
--reassignment-json-file /tmp/reassignment.json \
--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 ファイルに渡したターゲットの再割り当てです。

再割り当ての最中にスロットルを変更する必要がある場合は、同じコマンドに別のスロットル率を指定して実行します。以下に例を示します。

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

- 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かどうかを確認します。これは先ほどの手順と同じコマンドですが、**--verify** オプションの代わりに **--execute** 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 コマンドによって、移動した各パーティションが正常に完了したことが報告されると、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。

- 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。
- すべてのパーティション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が、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。ブローカのログディレクトリーに、拡張正規表現 **\[a-z0-9]-delete\$** に一致しないディレクトリーが含まれている場合、ブローカには、まだライブパーティションがあるため、停止しないでください。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$"
```

n は削除された 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ていないか、再割り当て JSON ファイルが適切ではありません。

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

2.2.5. ローリング更新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更新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。

2.2.5.1. 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更新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更新が行われる可能性があります。

2.2.5.2. メンテナンス時間枠の定義

Kafka.spec.maintenanceTimeWindows プロパティ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。

2.2.5.3. 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更新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
- Cluster Operator が稼働中である。

手順

1. **Kafka** リソースの **maintenanceTimeWindows** プロパティ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 **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_configuration_file>
```

関連情報

ローリング更新の実行:

- [「Pod 管理のアノテーションを使用したローリング更新の実行」](#)
- [「Pod アノテーションを使用したローリング更新の実行」](#)

2.2.6. 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、通常の状態では 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

- 利用可能な OpenShift クラスタ
- 稼働中の Kafka クラスタ
- Cluster Operator が稼働中である。

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。以下に例を示します。

```
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。

2.2.7. 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** が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](#)を参照すること。

- [Cluster Operator](#)
- [Kafka クラスタ](#)

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。ここで、`<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。

2.2.8. 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 ノードを削除するには、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** が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**を参照すること。

- [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、`<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-zookeeper-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-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。

2.2.9. 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

以下の ZooKeeper リソースに指定された名前。

- ZooKeeper ノード Pod を管理する StatefulSet または StrimziPodSet ([UseStrimziPodSets](#) [フィーチャーゲート](#) が有効になっている場合)。
- ZooKeeper ノードで使用されるサービスアカウント。
- ZooKeeper ノードに設定された PodDisruptionBudget。

cluster-name-zookeeper-idx

ZooKeeper StatefulSet または StrimziPodSet 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-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

以下の Kafka リソースに指定された名前。

- Kafka ブローカー Pod を管理する StatefulSet または StrimziPodSet ([UseStrimziPodSets フィーチャーゲート](#) が有効になっている場合)。
- Kafka Pod によって使用されるサービスアカウント。
- Kafka ブローカーに設定された PodDisruptionBudget。

cluster-name-kafka-idx

以下の Kafka リソースに指定された名前。

- Kafka StatefulSet または StrimziPodSet によって作成された Pod。
- Kafka ブローカー設定を使用した ConfigMap ([UseStrimziPodSets 機能ゲート](#) が有効になっている場合)。

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-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。

Entity 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

以下の Entity Operator リソースに指定された名前:

- Topic および User Operator とのデプロイメント。
- Entity Operator によって使用されるサービスアカウント。

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-topic-operator-certs

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

cluster-name-entity-user-operator-certs

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

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

以下の Kafka Exporter リソースに指定された名前。

- Kafka Exporter でのデプロイメント。
- コンシューマーラグメトリックの収集に使用されるサービス。
- Kafka Exporter によって使用されるサービスアカウント。

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。

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

以下の Cruise Control リソースに指定された名前。

- Cruise Control でのデプロイメント。
- Cruise Control との通信に使用されるサービス。
- Cruise Control によって使用されるサービスアカウント。

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-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。

2.3. 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設定する方法について説明します。

Kafka Connect は、コネクタ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を統合するためのフレームワークを提供し、コネクタ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。

コネクタプラグインのデプロイの詳細は [コネクタプラグインを使用した Kafka Connect の拡張](#) を参照してください。

2.3.1. 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を使用して、Kafka Connect デプロイメントを設定します。

KafkaConnector の設定

KafkaConnect リソースを使用すると、Kafka Connect のコネクタインスタンスを 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** 設定を追加して、データ接続に必要なコネクタ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を作成、再設定、または削除する方法は、[コネクタの作成および管理](#) を参照してください。

コネクタ設定は、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。

大量のメッセージ処理

設定を調整して、大量のメッセージを処理できます。詳細は、「[大量のメッセージ処理](#)」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](#) を参照すること。

- [Cluster Operator](#)

- Kafka クラスター

手順

1. **KafkaConnect** リソースの **spec** プロパティを編集します。
設定可能なプロパティ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:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e8020bf4187870699819f54ef5859c7846ee4081507f48873479
  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 タスクを実行するワーカーの **レプリカノード数**。
- 4 **OAuth** ベアトークン、SASL ベースの **SCRAM-SHA-256/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に限定されます。
- 8 コネクタプラグインで自動的にコンテナイメージをビルドするための **ビルド設定プロパティ**。
- 9 (必須) 新しいイメージがプッシュされるコンテナレジストリーの設定。
- 10 (必須) 新しいコンテナイメージに追加するコネクタプラグインとそれらのアーティファクトのリスト。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。

コンテナを再起動するタイミング (liveness) およびコンテナ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 メトリック](#)。この例では、Prometheus JMX エクスポートの設定が含まれる 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ックを有効にできます。
- 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 [コンテナイメージの設定](#)。
- 18 特別なオプション: 展開のための [Rack awareness](#) 設定。これは、リージョン間ではなく、同じロケーション内でのデプロイメントを目的とした特殊なオプションです。このオプションは、コネクタがリーダーレプリカではなく、最も近いレプリカから消費する場合に使用できます。場合によっては、最も近いレプリカから消費することで、ネットワークの使用率を改善したり、コストを削減したりできます。 **topologyKey** は、ラック ID を含むノードラベルと一致する必要があります。この設定で使用される例では、標準の [topology.kubernetes.io/zone](#) ラベルを使用するゾーンを指定します。最も近いレプリカから消費するには、Kafka ブローカー設定で **RackAwareReplicaSelector** を有効にします。
- 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化](#) します。

2.3.2. 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

group.id が同じすべての Kafka Connect インスタンスで、これら3つのトピックの値を揃える必要があります。

デフォルト設定を変更しない限り、同じ Kafka クラスターに接続する Kafka Connect インスタンスはそれぞれ同じ値でデプロイされます。事実上、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て同じトピックを使用するようになります。

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の値を変更してください。

2.3.3. 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 でいずれかの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	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: "*"

```

- リソースを作成または更新します。

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

2.3.4. Kafka Connect 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。

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 インターフェイ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。

2.3.5. 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 トピックに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 AMQ Streams on OpenShift 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で Kafka Connect をデプロイします。**KafkaConnector** リソースを設定し、コネクタ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の設定方法を説明します。

2.4. 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 (ドキュメントでは単に **MirrorMaker** と呼ばれる) 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 を導入すると、**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。

2.4.1. Kafka MirrorMaker の設定

KafkaMirrorMaker リソースのプロパティを使用して、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](#) を参照すること。
 - [Cluster Operator](#)
 - [Kafka クラスタ](#)
- ソースおよびターゲットの Kafka クラスタが使用できる必要があります。

手順

1. **KafkaMirrorMaker** リソースの **spec** プロパティを編集します。設定可能なプロパティ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
  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 ベアトークン または SASL ベースの [SCRAM-SHA-256/SCRAM-SHA-512](#) または [PLAIN](#) メカニズムを使用するか、ここで紹介されているような [TLS メカニズム](#) を使用したコンシューマーまたはプロデューサーの認証。
- 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](#)。
- 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](#)。
- 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 [ロガーおよびログレベル](#) 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 コンテナを再起動するタイミング (liveness) およびコンテナ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 メトリック](#)。この例では、Prometheus JMX エクスポートの設定が含まれる 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ックを有効にできます。

- 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>
```

2.4.2. Kafka MirrorMaker 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。

<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。

2.5. 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。**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。

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。

2.5.1. 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** 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 は次のコネクタを使用します。

MirrorSourceConnector

ソースコネクタは、トピックをソースクラスターからターゲットクラスターに複製します。

MirrorCheckpointConnector

チェックポイントコネクタは定期的にオフセットを追跡します。有効にすると、ソースクラスターとターゲットクラスター間のコンシューマーグループオフセットも同期されます。

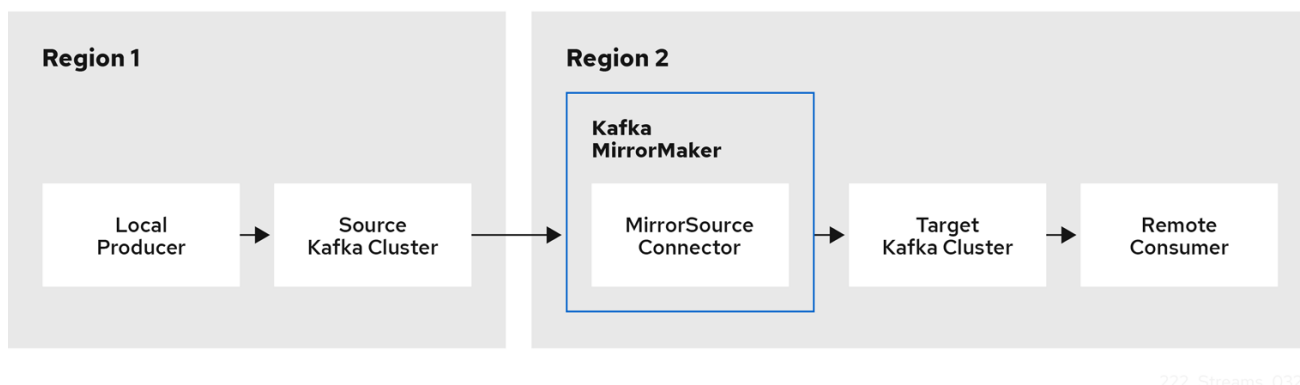
MirrorHeartbeatConnector

ハートビートコネクタは、ソースクラスターとターゲットクラスター間の接続を定期的にチェックします。

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2.12 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。

2.5.2. クラスター設定

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。

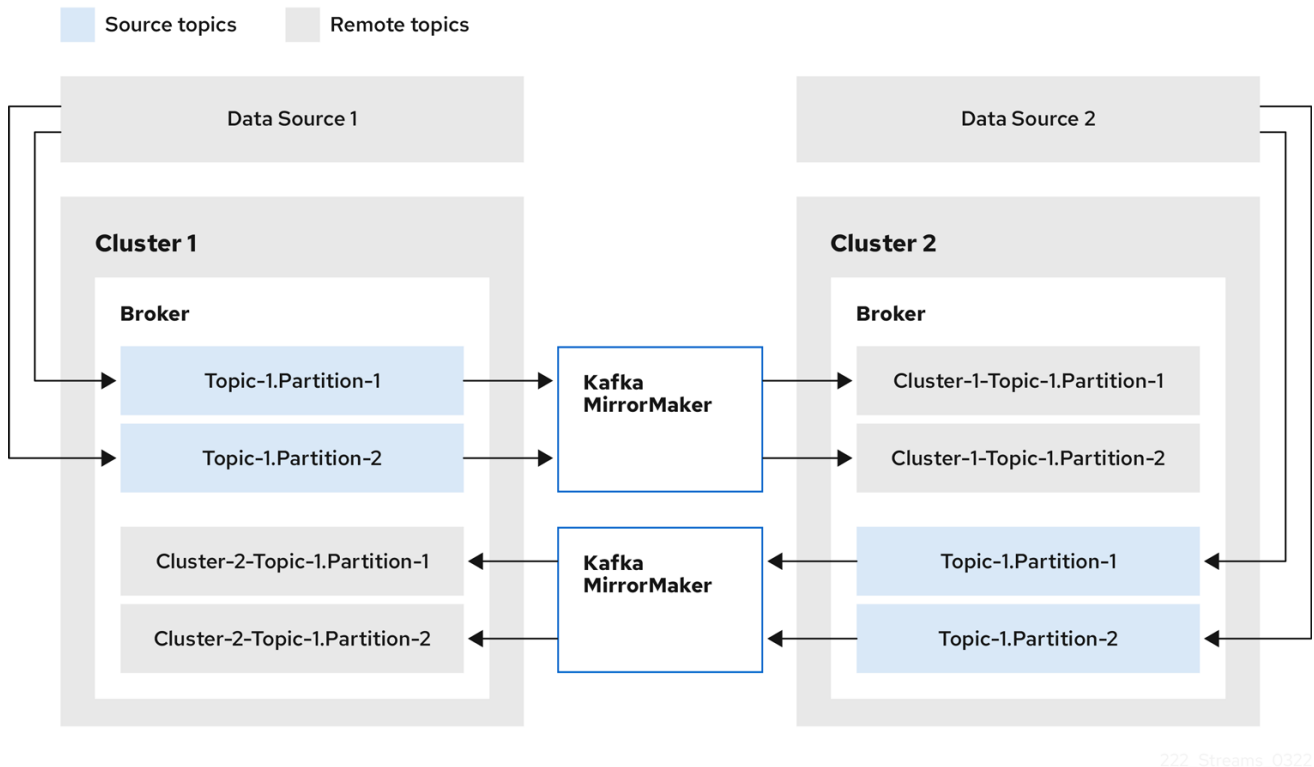
MirrorMaker 2.0 クラスターは、ターゲットの宛先ごとに必要です。

2.5.2.1. 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レプリケートします。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



222_Streams_0322

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レプリケートされません。

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。

2.5.2.2. 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の自動変更は推奨しません。

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。

2.5.2.3. 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。

2.5.2.4. 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。

2.5.2.5. 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。

- **offset-syncs** 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で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。

MirrorCheckpointConnector は、オフセット追跡用の **チェックポイント** を発行します。 **checkpoints** トピックのオフセットは、設定によって事前に決定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。

offset-syncs トピックの場所は、デフォルトで **source** クラスターです。 **offset-syncs.topic.location** コネクター設定を使用して、これを **target** クラスターに変更することができます。トピックが含まれるクラスターへの読み取り/書き込みアクセスが必要です。ターゲットクラスターを **offset-syncs** トピックの場所として使用すると、ソースクラスターへの読み取りアクセスしかない場合でも、MirrorMaker 2.0 を使用できます。

2.5.2.6. 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を **passive** (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** を使用する場合は、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。コンシューマーグループがターゲットクラスターにある場合、Synchronization を実行できず、 **UNKNOWN_MEMBER_ID** エラーが返されます。

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。両方のプロパティのデフォルトは 60 秒です。 **refresh.groups.interval.seconds** プロパティを使用して、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。

2.5.2.7. 接続性チェック

MirrorHeartbeatConnector は **heartbeat** を発行して、クラスター間の接続を確認します。

内部 **heartbeat** トピックは、ソースクラスターからレプリケーションされます。ターゲットクラスターは、**heartbeat** トピックを使用して次のことを確認します。

- クラスター間の接続を管理するコネクタが稼働しているかどうか
- ソースクラスターが利用可能かどうか

2.5.3. コネクタ設定

Kafka クラスター間のデータの同期を調整する内部コネクタの Mirrormaker 2.0 コネクタ設定を使用します。

以下の表は、コネクタプロパティと、これらを使用するために設定するコネクタについて説明しています。

表2.1 MirrorMaker 2.0 コネクタ設定プロパティ

プロパティ	sourceConnector	checkpointConnector	heartbeatConnector
admin.timeout.ms 新規トピックの検出などの管理タスクのタイムアウト。デフォルトは 60000 (1分) です。	✓	✓	✓
replication.policy.class リモートトピックの命名規則を定義するポリシー。デフォルトは org.apache.kafka.connect.mirror.DefaultReplicationPolicy です。	✓	✓	✓
replication.policy.separator ターゲットクラスターのトピックの命名に使用されるセパレーター。デフォルトは . (ドット) です。 replication.policy.class が DefaultReplicationPolicy の場合にのみ使用されます。	✓	✓	✓
consumer.poll.timeout.ms ソースクラスターをポーリングする際のタイムアウト。デフォルトは 1000 (1秒) です。	✓	✓	
offset-syncs.topic.location offset-syncs トピックの場所。これは、 source (デフォルト) または target クラスターになります。	✓	✓	

プロパティ	sourceConnector	checkpointConnector	heartbeatConnector
topic.filter.class レプリケーションするトピックを選択するためのトピックフィルター。デフォルトは org.apache.kafka.connect.mirror.DefaultTopicFilter です。	✓	✓	
config.property.filter.class レプリケーションするトピック設定プロパティを選択するトピックフィルター。デフォルトは org.apache.kafka.connect.mirror.DefaultConfigPropertyFilter です。	✓		
config.properties.exclude レプリケーションすべきでないトピック設定プロパティ。コンマ区切りのプロパティ名と正規表現をサポートします。	✓		
offset.lag.max リモートパーティションが同期されるまでの最大許容 (同期外) オフセットラグ。デフォルトは 100 です。	✓		
offset-syncs.topic.replication.factor 内部 offset-syncs トピックのレプリケーション係数。デフォルトは 3 です。	✓		
refresh.topics.enabled 新しいトピックおよびパーティションの確認を有効にします。デフォルトは true です。	✓		
refresh.topics.interval.seconds トピック更新の頻度。デフォルトは 600 (10 分) です。	✓		

プロパティ	sourceConnector	checkpointConnector	heartbeatConnector
replication.factor 新しいトピックのレプリケーション係数。デフォルトは 2 です。	✓		
sync.topic.acls.enabled ソースクラスターからの ACL の同期を有効にします。デフォルトは true です。User Operator との互換性はありません。	✓		
sync.topic.acls.interval.seconds ACL 同期の頻度。デフォルトは 600 (10 分) です。	✓		
sync.topic.configs.enabled ソースクラスターからのトピック設定の同期を有効にします。デフォルトは true です。	✓		
sync.topic.configs.interval.seconds トピック設定の同期頻度。デフォルトは 600 (10 分) です。	✓		
checkpoints.topic.replication.factor 内部 checkpoints トピックのレプリケーション係数。デフォルトは 3 です。		✓	
emit.checkpoints.enabled コンシューマーオフセットをターゲットクラスターに同期できるようにします。デフォルトは true です。		✓	
emit.checkpoints.interval.seconds コンシューマーオフセット同期の頻度。デフォルトは 60 (1 分) です。		✓	

プロパティ	sourceConnector	checkpointConnector	heartbeatConnector
group.filter.class レプリケーションするコンシューマーグループを選択するためのグループフィルター。デフォルトは org.apache.kafka.connect.mirror.DefaultGroupFilter です。		✓	
refresh.groups.enabled 新規コンシューマーグループの確認を有効にします。デフォルトは true です。		✓	
refresh.groups.interval.seconds コンシューマーグループ更新の頻度。デフォルトは 600 (10分) です。		✓	
sync.group.offsets.enabled ターゲットクラスターの __consumer_offsets トピックへのコンシューマーグループオフセットの同期を有効にします。デフォルトは false です。		✓	
sync.group.offsets.interval.seconds コンシューマーグループオフセット同期の頻度。デフォルトは 60 (1分) です。		✓	
emit.heartbeats.enabled ターゲットクラスターでの接続性チェックを有効にします。デフォルトは true です。			✓
emit.heartbeats.interval.seconds 接続性チェックの頻度。デフォルトは 1 (1秒) です。			✓

プロパティ	sourceConnector	checkpointConnector	heartbeatConnector
heartbeats.topic.replication.factor 内部 heartbeats トピックのレプリケーション係数。デフォルトは 3 です。			✓

2.5.4. コネクタプロデューサーおよびコンシューマーの設定

MirrorMaker 2.0 コネクタは内部プロデューサーおよびコンシューマーを使用します。必要に応じて、これらのプロデューサーおよびコンシューマーを設定して、デフォルト設定を上書きできます。

たとえば、トピックをターゲットの Kafka クラスターに送信するソースプロデューサーの **batch.size** を増やして、大量のデータをより適切に対応できます。



重要

プロデューサーとコンシューマーの設定オプションは、MirrorMaker 2.0 の実装に依存し、変更される可能性があります。

次の表では、各コネクタのプロデューサーとコンシューマー、および設定を追加できる場所について説明します。

表2.2 ソースコネクタのプロデューサーとコンシューマー

タイプ	説明	設定
producer	トピックメッセージをターゲット Kafka クラスターに送信します。大量のデータを処理する場合は、このプロデューサーの設定を調整することを検討してください。	mirrors.sourceConnector.config: producer.override.*
producer	レプリケートされたトピックパーティションのソースオフセットとターゲットオフセットをマップする、 offset-syncs トピックに書き込みます。	mirrors.sourceConnector.config: producer.*

タイプ	説明	設定
コンシューマー	ソース Kafka クラスターからトピックメッセージを取得します。	mirrors.sourceConnector.config: consumer.*

表2.3 チェックポイントコネクターのプロデューサーとコンシューマー

タイプ	説明	設定
producer	コンシューマーオフセットチェックポイントを発行します。	mirrors.checkpointConnector.config: producer.override.*
コンシューマー	offset-syncs トピックを読み込みます。	mirrors.checkpointConnector.config: consumer.*



注記

offset-syncs.topic.location を **target** に設定して、ターゲット Kafka クラスターを **offset-syncs** トピックの場所として使用できます。

表2.4 ハートビートコネクタプロデューサー

タイプ	説明	設定
producer	ハートビートを生成します。	mirrors.heartbeatConnector.config: producer.override.*

次の例は、プロデューサーとコンシューマーを設定する方法を示しています。

コネクターのプロデューサーとコンシューマーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.2.3
  # ...
  mirrors:
    - sourceCluster: "my-cluster-source"
      targetCluster: "my-cluster-target"
      sourceConnector:
```

```

tasksMax: 5
config:
  producer.override.batch.size: 327680
  producer.override.linger.ms: 100
  producer.request.timeout.ms: 30000
  consumer.fetch.max.bytes: 52428800
  # ...
checkpointConnector:
  config:
    producer.override.request.timeout.ms: 30000
    consumer.max.poll.interval.ms: 300000
    # ...
heartbeatConnector:
  config:
    producer.override.request.timeout.ms: 30000
    # ...

```

関連情報

- [「KafkaMirrorMaker2ConnectorSpec スキーマ参照」](#)
- [「KafkaMirrorMaker2MirrorSpec スキーマ参照」](#)

2.5.5. タスクの最大数を指定

コネクタは、Kafka にデータを出し入れするタスクを作成します。各コネクタは、タスクを実行するワーカー Pod のグループ全体に分散される1つ以上のタスクで設定されます。タスクの数を増やすと、多数のパーティションをレプリケーションするとき、または多数のコンシューマーグループのオフセットを同期するときのパフォーマンスの問題に役立ちます。

タスクは並行して実行されます。ワーカーには1つ以上のタスクが割り当てられます。1つのタスクが1つのワーカー Pod によって処理されるため、タスクよりも多くのワーカー Pod は必要ありません。ワーカーよりも多くのタスクがある場合、ワーカーは複数のタスクを処理します。

tasksMax プロパティを使用して、MirrorMaker 設定でコネクタタスクの最大数を指定できます。タスクの最大数を指定しない場合、デフォルト設定のタスク数は1つです。

ハートビートコネクタは常に単一のタスクを使用します。

ソースおよびチェックポイントコネクタに対して開始されるタスクの数は、可能なタスクの最大数と **tasksMax** の値の間の低い値です。ソースコネクタの場合、可能なタスクの最大数は、ソースクラスターからレプリケーションされるパーティションごとに1つです。チェックポイントコネクタの場合、可能なタスクの最大数は、ソースクラスターからレプリケーションされるコンシューマーグループごとに1つです。タスクの最大数を設定するときは、プロセスをサポートするパーティションの数とハードウェアリソースを考慮してください。

インフラストラクチャーが処理のオーバーヘッドをサポートしている場合、タスクの数を増やすと、スループットと待機時間が向上する可能性があります。たとえば、タスクを追加すると、多数のパーティションまたはコンシューマーグループがある場合に、ソースクラスターのポーリングにかかる時間が短縮されます。

チェックポイントコネクタのタスク数を増やすと、多数のパーティションがある場合に役立ちます。

ソースコネクタのタスク数を増やす

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 10
  # ...

```

多数のコンシューマーグループがある場合は、チェックポイントコネクタのタスク数を増やすと便利です。

チェックポイントコネクタのタスク数の増加

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  # ...
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    checkpointConnector:
      tasksMax: 10
  # ...

```

デフォルトでは、MirrorMaker 2.0 は新しいコンシューマーグループを10分ごとにチェックします。**refresh.groups.interval.seconds** 設定を調整して、頻度を変更できます。低く調整するときは注意してください。より頻繁なチェックは、パフォーマンスに悪影響を及ぼす可能性があります。

2.5.5.1. コネクタタスクの動作の確認

Prometheus と Grafana を使用して展開を監視している場合は、MirrorMaker 2.0 のパフォーマンスを確認できます。AMQ Streams で提供される MirrorMaker 2.0 Grafana ダッシュボードの例には、タスクとレイテンシーに関連する次のメトリックが表示されます。

- タスクの数
- レプリケーションのレイテンシー
- オフセット同期のレイテンシー

関連情報

- [Grafana ダッシュボード](#)

2.5.6. 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 **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、このようなりモートトピックへのアクセスをサポートしません。

2.5.7. Kafka MirrorMaker 2.0 の設定

KafkaMirrorMaker2 リソースのプロパティを使用して、Kafka MirrorMaker 2.0 のデプロイメントを設定します。MirrorMaker 2.0 を使用して、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
 - クラスターからクラスターへ
 - トピックからトピックへ



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。

MirrorMaker 2.0 の最小設定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.2.3
  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。

KafkaMirrorMaker2 リソースのソースクラスターからレプリケートするトピックとコンシューマーグループを指定できます。これを行うには、**topicsPattern** および **groupsPattern** プロパティを使用します。名前のリストを指定したり、正規表現を使用したりできます。デフォルトでは、**topicsPattern** および **groupsPattern** プロパティを設定しない場合、すべてのトピックとコンシューマーグループがレプリケートされます。*****を正規表現として使用して、すべてのトピックとコンシューマーグループを複製することもできます。ただし、クラスターに不要な負荷が余分にかかるのを避けるため、必要なトピックとコンシューマーグループのみを指定するようにしてください。

大量のメッセージ処理

設定を調整して、大量のメッセージを処理できます。詳細は、「[大量のメッセージ処理](#)」を参照してください。

前提条件

- AMQ Streams が実行されている
- ソースおよびターゲットの Kafka クラスターが使用できる

手順

1. **KafkaMirrorMaker2** リソースの **spec** プロパティを編集します。設定可能なプロパティ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: 3.2.3 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: "org.apache.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: "org.apache.kafka.connect.mirror.IdentityReplicationPolicy"
  topicsPattern: "topic1|topic2|topic3" 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
  rack:
    topologyKey: topology.kubernetes.io/zone 42
  template: 43
    pod:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: application
                    operator: In
                    values:
                      - postgresql
                      - mongodb
              topologyKey: "kubernetes.io/hostname"
    connectContainer: 44
      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 45
  externalConfiguration: 46
    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 タスクを実行するワーカーのレプリカノード数。
- 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-256/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ポートする場合、この値を大きくするとスループットが向上されます。Kafka Connect は、クラスターのメンバー間でタスクを分散します。ワーカーよりも多くのタスクがある場合は、ワーカーには複数のタスクが割り当てられます。シンクコネクターでは、消費される各トピックパーティションに1つタスクがあるようになることを目指します。ソースコネクターでは、並行して実行できるタスクの数は外部システムによって異なる場合があります。並列処理を実現できない場合、コネクターは最大数より少ないタスクを作成しません。
- 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。この機能は User Operator と互換性がありません。User

Operator を使用している場合は、このプロパティを **false** に設定します。

- 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は、これらのチェックの頻度も調整することを推奨します。
- 35 **コンマ区切りリストまたは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。ソースコネクタは指定のトピックをレプリケーションします。チェックポイントコネクタは、指定されたトピックのオフセットを追跡します。ここでは、3つのトピックを名前でもリクエストします。
- 36 **コンマ区切りリストまたは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のレプリケーション。チェックポイントコネクタは、指定されたコンシューマーグループをレプリケーションします。ここで、3つのコンシューマーグループを名前でも要求します。
- 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 コンテナを再起動するタイミング (liveness) およびコンテナ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 [コンテナイメージの設定](#)。
- 42 特別なオプション: 展開のための [Rack awareness](#) 設定。これは、リージョン間ではなく、同じロケーション内でのデプロイメントを目的とした特殊なオプションです。このオプションは、コネクターがリーダーレプリカではなく、最も近いレプリカから消費する場合に使用できます。場合によっては、最も近いレプリカから消費することで、ネットワークの使用率を改善したり、コストを削減したりできます。**topologyKey** は、ラック ID を含むノードラベルと一致する必要があります。この設定で使用される例では、標準の [topology.kubernetes.io/zone](#) ラベルを使用するゾーンを指定します。最も近いレプリカから消費するには、Kafka ブローカー設定で **RackAwareReplicaSelector** を有効にします。
- 43 [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。
- 44 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。
- 45 [Jaeger](#) の分散トレーシングは有効になっています。
- 46 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
```

2.5.8. Kafka MirrorMaker 2.0 デプロイメントの保護

この手順は、MirrorMaker2.0 のデプロイメントを保護するために必要な設定の概要を説明しています。

ソース Kafka クラスターとターゲット Kafka クラスターには別々の設定が必要です。また、MirrorMaker がソースおよびターゲットの Kafka クラスターに接続するために必要な認証情報を提供するために、個別のユーザー設定が必要です。

Kafka クラスターの場合、OpenShift クラスター内のセキュア接続用の内部リスナーと、OpenShift クラスター外の接続用の外部リスナーを指定します。

認証および許可メカニズムを設定できます。ソースおよびターゲットの Kafka クラスターに実装されているセキュリティーオプションは、MirrorMaker 2.0 に実装されているセキュリティーオプションと互換性がある必要があります。

クラスターとユーザー認証情報を作成したら、セキュアな接続のために MirrorMaker 設定でそれらを指定します。



注記

この手順では、Cluster Operator によって生成された証明書が使用されますが、[独自の証明書をインストール](#) してそれら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。

作業を開始する前の注意事項

この手順を開始する前に、AMQ Streams が提供する [設定ファイルの例](#) を確認してください。TLS または SCRAM-SHA-512 認証を使用して MirrorMaker 2.0 のデプロイメントを保護する場合の例が含まれています。例では、OpenShift クラスター内で接続するための内部リスナーを指定しています。

例には、ソースおよびターゲットの Kafka クラスターでの操作を可能にするために MirrorMaker 2.0 が必要とする ACL も示されています。

前提条件

- AMQ Streams が実行されている
- ソースクラスターとターゲットクラスターの namespace が分離されている

この手順では、ソースとターゲットの Kafka クラスターが別々の namespace にインストールされていることを前提としています。Topic Operator を使用する場合は、これを行う必要があります。Topic Operator は、指定された namespace 内の単一クラスターのみをモニタリングします。

クラスターを namespace に分割することにより、クラスターシークレットをコピーして、namespace の外部からアクセスできるようにする必要があります。MirrorMaker 設定でシークレットを参照する必要があります。

手順

1. 2つの **Kafka** リソースを設定します。1つはソース Kafka クラスターを保護するためのもので、もう1つはターゲット Kafka クラスターを保護するためのものです。認証用のリスナー設定を追加し、認可を有効にすることができます。

この例の場合、内部リスナーは TLS 暗号化と認証を使用して Kafka クラスター用に設定されています。Kafka の **simple** 認証が有効になっています。

TLS 認証を使用したソース Kafka クラスターの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-source-cluster
spec:
  kafka:
    version: 3.2.3
    replicas: 1
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    authorization:
      type: simple
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      default.replication.factor: 1
```

```

min.insync.replicas: 1
inter.broker.protocol.version: "3.2"
storage:
  type: jbod
  volumes:
  - id: 0
    type: persistent-claim
    size: 100Gi
    deleteClaim: false
zookeeper:
  replicas: 1
  storage:
    type: persistent-claim
    size: 100Gi
    deleteClaim: false
entityOperator:
  topicOperator: {}
  userOperator: {}

```

TLS 認証を使用したターゲット Kafka クラスターの設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-target-cluster
spec:
  kafka:
    version: 3.2.3
    replicas: 1
    listeners:
    - name: tls
      port: 9093
      type: internal
      tls: true
      authentication:
        type: tls
    authorization:
      type: simple
    config:
      offsets.topic.replication.factor: 1
      transaction.state.log.replication.factor: 1
      transaction.state.log.min.isr: 1
      default.replication.factor: 1
      min.insync.replicas: 1
      inter.broker.protocol.version: "3.2"
    storage:
      type: jbod
      volumes:
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
  zookeeper:
    replicas: 1
    storage:
      type: persistent-claim

```

```

size: 100Gi
deleteClaim: false
entityOperator:
  topicOperator: {}
  userOperator: {}

```

- 別の namespace で **Kafka** リソースを作成または更新します。

```
oc apply -f <kafka_configuration_file> -n <namespace>
```

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。

証明書は、シークレット **<cluster_name>-cluster-ca-cert** に作成されます。

- 2つの **KafkaUser** リソースを設定します。1つはソース Kafka クラスターのユーザー用で、もう1つはターゲット Kafka クラスターのユーザー用です。
 - 対応するソースおよびターゲットの Kafka クラスターと同じ認証および認可タイプを設定します。たとえば、ソース Kafka クラスターの **Kafka** 設定で **tls** 認証と **simple** 認可タイプを使用した場合は、**KafkaUser** 設定でも同じものを使用します。
 - ソースおよびターゲットの Kafka クラスターでの操作を可能にするために MirrorMaker 2.0 が必要とする ACL を設定します。
ACL は、内部 MirrorMaker コネクター、および基盤となる Kafka Connect フレームワークによって使用されます。

TLS クライアント認証のソースユーザーの設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-source-user
  labels:
    strimzi.io/cluster: my-source-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
  acls:
    # MirrorSourceConnector
    - resource: # Not needed if offset-syncs.topic.location=target
      type: topic
      name: mm2-offset-syncs.my-target-cluster.internal
      operation: Create
    - resource: # Not needed if offset-syncs.topic.location=target
      type: topic
      name: mm2-offset-syncs.my-target-cluster.internal
      operation: DescribeConfigs
    - resource: # Not needed if offset-syncs.topic.location=target
      type: topic
      name: mm2-offset-syncs.my-target-cluster.internal
      operation: Write
    - resource: # Needed for every topic which is mirrored
      type: topic

```

```

    name: "*"
    operation: Read
  - resource: # Needed for every topic which is mirrored
    type: topic
    name: "*"
    operation: DescribeConfigs
  # MirrorCheckpointConnector
  - resource:
    type: cluster
    operation: Describe
  - resource: # Needed for every group for which offsets are synced
    type: group
    name: "*"
    operation: Describe
  - resource: # Not needed if offset-syncs.topic.location=target
    type: topic
    name: mm2-offset-syncs.my-target-cluster.internal
    operation: Read

```

TLS クライアント認証のターゲットユーザーの設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-target-user
  labels:
    strimzi.io/cluster: my-target-cluster
spec:
  authentication:
    type: tls
  authorization:
    type: simple
  acls:
    # Underlying Kafka Connect internal topics to store configuration, offsets, or status
    - resource:
      type: group
      name: mirrmaker2-cluster
      operation: Read
    - resource:
      type: topic
      name: mirrmaker2-cluster-configs
      operation: Read
    - resource:
      type: topic
      name: mirrmaker2-cluster-configs
      operation: Describe
    - resource:
      type: topic
      name: mirrmaker2-cluster-configs
      operation: DescribeConfigs
    - resource:
      type: topic
      name: mirrmaker2-cluster-configs
      operation: Write
    - resource:
      type: topic

```

```
    name: mirrmaker2-cluster-configs
  operation: Create
- resource:
  type: topic
  name: mirrmaker2-cluster-status
  operation: Read
- resource:
  type: topic
  name: mirrmaker2-cluster-status
  operation: Describe
- resource:
  type: topic
  name: mirrmaker2-cluster-status
  operation: DescribeConfigs
- resource:
  type: topic
  name: mirrmaker2-cluster-status
  operation: Write
- resource:
  type: topic
  name: mirrmaker2-cluster-status
  operation: Create
- resource:
  type: topic
  name: mirrmaker2-cluster-offsets
  operation: Read
- resource:
  type: topic
  name: mirrmaker2-cluster-offsets
  operation: Write
- resource:
  type: topic
  name: mirrmaker2-cluster-offsets
  operation: Describe
- resource:
  type: topic
  name: mirrmaker2-cluster-offsets
  operation: DescribeConfigs
- resource:
  type: topic
  name: mirrmaker2-cluster-offsets
  operation: Create
# MirrorSourceConnector
- resource: # Needed for every topic which is mirrored
  type: topic
  name: "*"
  operation: Create
- resource: # Needed for every topic which is mirrored
  type: topic
  name: "*"
  operation: Alter
- resource: # Needed for every topic which is mirrored
  type: topic
  name: "*"
  operation: AlterConfigs
- resource: # Needed for every topic which is mirrored
```

```

    type: topic
    name: "*"
  operation: Write
# MirrorCheckpointConnector
- resource:
  type: cluster
  operation: Describe
- resource:
  type: topic
  name: my-source-cluster.checkpoints.internal
  operation: Create
- resource:
  type: topic
  name: my-source-cluster.checkpoints.internal
  operation: Describe
- resource:
  type: topic
  name: my-source-cluster.checkpoints.internal
  operation: Write
- resource: # Needed for every group for which the offset is synced
  type: group
  name: "*"
  operation: Read
- resource: # Needed for every group for which the offset is synced
  type: group
  name: "*"
  operation: Describe
- resource: # Needed for every topic which is mirrored
  type: topic
  name: "*"
  operation: Read
# MirrorHeartbeatConnector
- resource:
  type: topic
  name: heartbeats
  operation: Create
- resource:
  type: topic
  name: heartbeats
  operation: Describe
- resource:
  type: topic
  name: heartbeats
  operation: Write

```



注記

type を **tls-external** に設定することにより、User Operator の外部で発行された証明書を使用できます。詳細については、[User authentication](#) を参照してください。

4. ソースおよびターゲットの Kafka クラスター用に作成した各 namespace で、**KafkaUser** リソースを作成または更新します。

```
oc apply -f <kafka_user_configuration_file> -n <namespace>
```


User Operator はクライアント (MirrorMaker) 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。

User Operator は、**KafkaUser** リソースと同じ名前の新しいシークレットを作成します。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。

5. ソースおよびターゲットの Kafka クラスターに接続するための認証の詳細を使用して **KafkaMirrorMaker2** リソースを設定します。

TLS 認証を使用した MirrorMaker 2.0 の設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker-2
spec:
  version: 3.2.3
  replicas: 1
  connectCluster: "my-target-cluster"
  clusters:
    - alias: "my-source-cluster"
      bootstrapServers: my-source-cluster-kafka-bootstrap:9093
      tls: ①
        trustedCertificates:
          - secretName: my-source-cluster-cluster-ca-cert
            certificate: ca.crt
        authentication: ②
          type: tls
          certificateAndKey:
            secretName: my-source-user
            certificate: user.crt
            key: user.key
    - alias: "my-target-cluster"
      bootstrapServers: my-target-cluster-kafka-bootstrap:9093
      tls: ③
        trustedCertificates:
          - secretName: my-target-cluster-cluster-ca-cert
            certificate: ca.crt
        authentication: ④
          type: tls
          certificateAndKey:
            secretName: my-target-user
            certificate: user.crt
            key: user.key
  config:
    # -1 means it will use the default replication factor configured in the broker
    config.storage.replication.factor: -1
    offset.storage.replication.factor: -1
    status.storage.replication.factor: -1
  mirrors:
    - sourceCluster: "my-source-cluster"
      targetCluster: "my-target-cluster"
      sourceConnector:

```

```

config:
  replication.factor: 1
  offset-syncs.topic.replication.factor: 1
  sync.topic.acls.enabled: "false"
heartbeatConnector:
  config:
    heartbeats.topic.replication.factor: 1
checkpointConnector:
  config:
    checkpoints.topic.replication.factor: 1
    sync.group.offsets.enabled: "true"
topicsPattern: "topic1|topic2|topic3"
groupsPattern: "group1|group2|group3"

```

- 1 ソース Kafka クラスターの TLS 証明書。それらが別の namespace にある場合は、Kafka クラスターの namespace からクラスターシークレットをコピーします。
 - 2 [TLS mechanism](#) を使用してソース Kafka クラスターにアクセスするためのユーザー認証。
 - 3 ターゲット Kafka クラスターの TLS 証明書。
 - 4 ターゲット Kafka クラスターにアクセスするためのユーザー認証。
6. ターゲット Kafka クラスターと同じ namespace で **KafkaMirrorMaker2** リソースを作成または更新します。

```
oc apply -f <mirrormaker2_configuration_file> -n <namespace_of_target_cluster>
```

関連情報

- [Supported listener authentication options](#)
- [Supported authorization options for a Kafka cluster](#)
- [Kafka ブローカーのセキュア化](#)
- [Kafka へのユーザーアクセスのセキュア化](#)
- [TLS 証明書の管理](#)

2.5.9. 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 が稼働中である。

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 コネクタの名前を見つけます。

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

3. コネクタを再起動するには、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](#)

2.5.10. Kafka MirrorMaker 2.0 コネクタ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 が稼働中である。

手順

1. 再起動する Kafka MirrorMaker 2.0 コネクタを制御する **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](#)

2.6. 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 には、HTTP ベースのクライアントと Kafka クラスターを統合する API が含まれています。

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。

2.6.1. 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](#)を参照すること。

- [Cluster Operator](#)
- [Kafka クラスター](#)

手順

1. **KafkaBridge** リソースの **spec** プロパティを編集します。設定可能なプロパティ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: <cluster_name>-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](#)。Kafka クラスターの名前は `<cluster_name>` を使用します。
- 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 ベアトークン](#)、SASL ベースの [SCRAM-SHA-256/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](#)。要求に別の HTTP ヘッダーを追加して、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 コンテナを再起動するタイミング (liveness) およびコンテナ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: [コンテナイメージの設定](#)。これは、特別な状況でのみ推奨されます。

- 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
```

関連情報

- [AMQ Streams Kafka Bridge の使用](#)

2.6.2. 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 インターフェイ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。

2.7. 大量のメッセージ処理

AMQ Streams デプロイメントで大量のメッセージを処理する必要がある場合は、設定オプションを使用してスループットとレイテンシーを最適化できます。

Kafka プロデューサーおよびコンシューマー設定は、Kafka ブローカーへのリクエストのサイズおよび頻度を制御するのに役立ちます。設定オプションの詳細は、以下を参照してください。

- [プロデューサーのスループットおよびレイテンシーの最適化](#)
- [スループットおよびレイテンシーに対するコンシューマーの最適化](#)

Kafka Connect ランタイムソースコネクター (MirrorMaker 2.0 を含む) とシンクコネクターで使用されるプロデューサーとコンシューマーで同じ設定オプションを使用することもできます。

ソースコネクター

- Kafka Connect ランタイムのプロデューサーは、メッセージを Kafka クラスターに送信します。
- MirrorMaker 2.0 の場合、ソースシステムが Kafka であるため、コンシューマーはソース Kafka クラスターからメッセージを取得します。

シンクコネクター

- Kafka Connect ランタイムのコンシューマーは、Kafka クラスターからメッセージを取得します。

コンシューマーの場合、1回のフェッチリクエストでフェッチされるデータの量を増やして、レイテンシーを短縮することができます。**fetch.max.bytes** および **max.partition.fetch.bytes** プロパティを使用して、フェッチ要求のサイズを増やします。**max.poll.records** プロパティを使用して、コンシューマーバッファから返されるメッセージ数の上限を設定することもできます。

MirrorMaker 2.0 の場合、ソースコネクターレベル (**consumer.***) で **fetch.max.bytes**、**max.partition.fetch.bytes**、および **max.poll.records** の値を設定します。ソース。

プロデューサーの場合、1つの生成リクエストで送信されるメッセージバッチのサイズを増やすことができます。**batch.size** プロパティを使用してバッチサイズを増やします。バッチサイズを大きくすると、送信する準備ができていない未処理のメッセージの数と、メッセージキュー内のバックログのサイズが減少します。同じパーティションに送信されるメッセージはまとめてバッチ処理されます。バッチサイズに達すると、プロデューサーリクエストがターゲットクラスターに送信されます。バッチサイズを大きくすると、プロデューサーリクエストが遅延し、より多くのメッセージがバッチに追加され、同時にブローカーに送信されます。これにより、多数のメッセージを処理するトピックパーティションが複数ある場合に、スループットが向上します。

プロデューサーが適切なプロデューサーバッチサイズに対して処理するレコードの数とサイズを考慮します。

linger.ms を使用してミリ秒単位の待機時間を追加し、プロデューサーの負荷が減少したときにプロデューサーリクエストを遅らせます。遅延は、最大バッチサイズ未満の場合に、より多くのレコードをバッチに追加できることを意味します。

ソースコネクターレベル (**producer.override.***) で **batch.size** および **linger.ms** の値を設定します。これは、ターゲット Kafka クラスターにメッセージを送信する特定のプロデューサーに関連するためです。

Kafka Connect ソースコネクターでは、ターゲット Kafka クラスターへのデータストリーミングパイプラインは以下ようになります。

Kafka Connect ソースコネクターのデータストリーミングパイプライン

外部データソース → (Kafka Connect タスク) ソースメッセージキュー → プロデューサーバッファ → ターゲット Kafka トピック

Kafka Connect シンクコネクターの場合、ターゲット外部データソースへのデータストリーミングパイプラインは次のとおりです。

Kafka Connect シンクコネクターのデータストリーミングパイプライン

ソース Kafka トピック → (Kafka Connect タスク) シンクメッセージキュー → コンシューマーバッファ → 外部データソース

MirrorMaker 2.0 の場合、ターゲット Kafka クラスターへのデータミラーリングパイプラインは次のとおりです。

MirrorMaker 2.0 のデータミラーリングパイプライン

ソース Kafka トピック → (Kafka Connect タスク) ソースメッセージキュー → プロデューサーバッファ → ターゲット Kafka トピック

プロデューサーは、バッファ内のメッセージをターゲット Kafka クラスター内のトピックに送信します。これが発生している間、Kafka Connect タスクは引き続きデータソースをポーリングして、ソースメッセージキューにメッセージを追加します。

ソースコネクタのプロデューサーバッファのサイズは、**producer.override.buffer.memory** プロパティを使用して設定されます。タスクは、バッファがフラッシュされる前に、指定されたタイムアウト期間 (**offset.flush.timeout.ms**) 待機します。これは、送信されたメッセージがブローカーによって確認され、コミットされたデータがオフセットされるのに十分な時間です。ソースタスクは、シャットダウン中を除き、オフセットをコミットする前にプロデューサーがメッセージキューを空にするのを待ちません。

プロデューサーがソースメッセージキュー内のメッセージのスループットについていけない場合、バッファリングは、**max.block.ms** で制限された期間内にバッファに使用可能なスペースができるまでブロックされます。バッファ内に未確認のメッセージがあれば、この期間中に送信されます。これらのメッセージが確認されてフラッシュされるまで、新しいメッセージはバッファに追加されません。

次の設定変更を試して、未処理メッセージの基になるソースメッセージキューを管理可能なサイズに保つことができます。

- **offset.flush.timeout.ms** のデフォルト値 (ミリ秒) を増やす
- 十分な CPU およびメモリーリソースがあることの確認
- 以下を実行して、並行して実行されるタスクの数を増やします。
 - **tasksMax** プロパティを使用して並行して実行するタスクの数を増やす
 - **replicas** プロパティを使用してタスクを実行するワーカーノードの数の増加

使用可能な CPU とメモリーリソース、およびワーカーノードの数に応じて、並列実行できるタスクの数を検討してください。必要な効果が得られるまで、設定値を調整し続けることを推奨します。

2.7.1. 大量メッセージ用の Kafka Connect の設定

Kafka Connect は、ソースの外部データシステムからデータをフェッチし、それを Kafka Connect ランタイムプロデューサーに渡して、ターゲットクラスターにレプリケートします。

次の例は、**KafkaConnect** カスタムリソースを使用した Kafka Connect の設定を示しています。

大量のメッセージを処理するための Kafka Connect 設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  replicas: 3
  config:
    offset.flush.timeout.ms: 10000
    # ...
  resources:
    requests:
      cpu: "1"
      memory: 2Gi
    limits:
```

```
cpu: "2"
memory: 2Gi
# ...
```

プロデューサー設定は、**KafkaConnector** カスタムリソースを使用して管理されるソースコネクタ用に追加されます。

大量のメッセージを処理するためのソースコネクタの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector
  tasksMax: 2
  config:
    producer.override.batch.size: 327680
    producer.override.linger.ms: 100
  # ...
```



注記

FileStreamSourceConnector および **FileStreamSinkConnector** は、コネクタの例として提供されています。それらを **KafkaConnector** リソースとしてデプロイする方法については、[サンプル KafkaConnector リソースのデプロイ](#) を参照してください。

シンクコネクタのコンシューマー設定が追加されます。

大量のメッセージを処理するためのシンクコネクタの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector
  tasksMax: 2
  config:
    consumer.fetch.max.bytes: 52428800
    consumer.max.partition.fetch.bytes: 1048576
    consumer.max.poll.records: 500
  # ...
```

KafkaConnector カスタムリソースの代わりに Kafka Connect API を使用してコネクタを管理している場合は、コネクタ設定を JSON オブジェクトとして追加できます。

大量のメッセージを処理するためのソースコネクタ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"
    "producer.override.batch.size": 327680
    "producer.override.linger.ms": 100
  }
}'

```

2.7.2. 大量のメッセージ用に MirrorMaker 2.0 を設定する

MirrorMaker 2.0 は、ソースクラスターからデータをフェッチし、それを Kafka Connect ランタイムプロデューサーに渡して、ターゲットクラスターにレプリケートします。

次の例は、**KafkaMirrorMaker2** カスタムリソースを使用した MirrorMaker 2.0 の設定を示しています。

大量のメッセージを処理するための MirrorMaker 2.0 の設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.2.3
  replicas: 1
  connectCluster: "my-cluster-target"
  clusters:
  - alias: "my-cluster-source"
    bootstrapServers: my-cluster-source-kafka-bootstrap:9092
  - alias: "my-cluster-target"
    config:
      offset.flush.timeout.ms: 10000
      bootstrapServers: my-cluster-target-kafka-bootstrap:9092
  mirrors:
  - sourceCluster: "my-cluster-source"
    targetCluster: "my-cluster-target"
    sourceConnector:
      tasksMax: 2
      config:
        producer.override.batch.size: 327680
        producer.override.linger.ms: 100
        consumer.fetch.max.bytes: 52428800
        consumer.max.partition.fetch.bytes: 1048576
        consumer.max.poll.records: 500
    # ...
  resources:
    requests:
      cpu: "1"
      memory: Gi

```

```
limits:  
  cpu: "2"  
  memory: 4Gi
```

2.7.3. MirrorMaker 2.0 メッセージフローの確認

Prometheus と Grafana を使用してデプロイメントをモニタリングしている場合は、MirrorMaker 2.0 メッセージフローを確認できます。

AMQ Streams で提供される MirrorMaker 2.0 Grafana ダッシュボードの例は、フラッシュパイプラインに関連する次のメトリクスを示しています。

- Kafka Connect の未処理メッセージキューにあるメッセージの数
- プロデューサーバッファの使用可能なバイト数
- オフセットコミットタイムアウト(ミリ秒)

これらのメトリックを使用して、メッセージの量に基づいて設定を調整する必要があるかどうかを判断できます。

関連情報

- [Grafana ダッシュボード](#)
- [コネクターの作成および管理](#)

2.8. OPENSIFT リソースのカスタマイズ

AMQ Streams デプロイメントでは、**Deployments**、**StatefulSets**、**Pods**、**Services** などの OpenShift リソースを作成します。これらのリソースは AMQ Streams Operator が管理します。特定の OpenShift リソースの管理を担当する operator のみは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、operator はその変更を元に戻します。

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** プロパティを使用して追加します。**template** プロパティ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

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 スキーマ参照](#)」を参照

以下の例では、**template** プロパティを使用して Kafka ブローカーの Pod 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:
    pod:
      metadata:
        labels:
          mylabel: myvalue
    # ...
```

2.8.1. 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更新が実行されます。

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。

2.8.2. 終了時の猶予期間の適用

終了時の猶予期間を適用し、Kafka クラスターが正常にシャットダウンされるように十分な時間を確保します。

terminationGracePeriodSeconds プロパティを使用して時間を指定します。プロパティを **Kafka** カスタムリソースの **template.pod** 設定に追加します。

追加する時間は Kafka クラスターのサイズによって異なります。終了猶予期間の OpenShift のデフォルト値は 30 秒です。クラスターが正常にシャットダウンしていないことが判明した場合には、終了までの猶予期間を増やすことができます。

終了時の猶予期間は、Pod が再起動されるたびに適用されます。この期間は、OpenShift が Pod で実行されているプロセスに **term** (中断) シグナルを送信すると開始します。この期間は、終了する Pod のプロセスを、停止する前に別の Pod に転送するのに必要な時間を反映する必要があります。期間の終了後、**kill** シグナルにより、Pod で実行中のプロセスはすべて停止します。

以下の例では、終了猶予期間 120 秒を **Kafka** カスタムリソースに追加します。他の Kafka コンポーネントのカスタムリソースで設定を指定することもできます。

終了猶予期間の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      pod:
        terminationGracePeriodSeconds: 120
      # ...
    # ...
```

2.9. 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。

2.9.1. 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** フィーチャー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](#)

2.9.1.1. 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。

2.9.1.2. 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設定されます。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ル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。

2.9.1.3. 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。

2.9.2. 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。以下に例を示します。

```
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。以下に例を示します。

```

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。

- リソースを作成または更新します。

```
oc apply -f <kafka_configuration_file>
```

2.9.3. 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

- 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。以下に例を示します。

```

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

```

- 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_configuration_file>
```

2.9.4. 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

- 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。

- クラスターデプロイメントを指定するリソースの **affinity** プロパティを編集します。以下に例を示します。

```
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:
        # ...
```

- 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_configuration_file>
```

2.9.5. 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** プロパティを編集します。
以下に例を示します。

```
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_configuration_file>
```

2.10. 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** プロパティに直接指定できます。あるいは、**configMapKeyRef** プロパティを使用し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。

2.10.1. 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 のログ設定の詳細は、次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](#)

2.10.2. 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"
# ...
```

- 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
```

- リソースを作成または更新します。

```
oc apply -f <kafka_configuration_file>
```

2.10.3. 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。ログ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、このレベルでログ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 ❶
appender.console.filter.filter1.onMatch=ACCEPT ❷
appender.console.filter.filter1.onMismatch=DENY ❸
appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster) ❹
```

- 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。
- onMatch** プロパティは、マーカーが一致するとログを受け入れます。
- onMismatch** プロパティは、マーカーが一致しない場合にログを拒否します。
- 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を ConfigMap に追加します。
この例では、フィルタープロパティは **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。これを使用して、プロデューサーやコンシューマーを含む、すべての Kafka コンポーネントの設定データを読み込むことができます。たとえば、これを使用して、KafkaConnect コネクタ設定のクレデンシャルを提供します。

OpenShift 設定プロバイダー

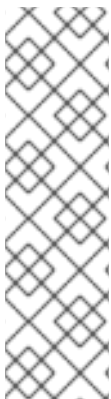
OpenShift Configuration Provider プラグインは、OpenShift シークレットまたは設定マップから設定データを読み込みます。

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シークレットがあるとします。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シークレットの値を参照できます。使用するシークレットをプロバイダーに伝え、アクセス権限を提供する必要があります。プロバイダーは、新しい **Secret** または **ConfigMap** 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ロードします。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を使用して、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、たとえば、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** 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 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、外部 **ConfigMap** はコネクタの設定プロパティを提供します。

前提条件

- 利用可能な OpenShift クラスター
- 稼働中の Kafka クラスター
- Cluster Operator が稼働中である。

手順

1. 設定プロパティが含まれる **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** にはコネクタプロパティが含まれます。

コネクタプロパティのある 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。
ここで示される仕様は、シークレットおよび ConfigMap 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せん。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 Secret から値を指定します。
- 3** **KubernetesConfigMapConfigProvider** は設定マップから値を指定します。

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。

```
oc apply -f <kafka_connect_configuration_file>
```

4. 外部の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、`<cluster_name>` は **KafkaConnect** の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、外部の ConfigMap から `option1` プロパティの値を読み込んで抽出します。

3.2. 環境変数から設定値の読み込み

この手順では、環境変数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の設定プロパティを提供します。データベースのパスワードは環境変数として指定します。

前提条件

- 利用可能な OpenShift クラスタ
- 稼働中の Kafka クラスタ
- Cluster Operator が稼働中である。

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 設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
  annotations:
    strimzi.io/use-connector-resources: "true"
spec:
  # ...
  config:
    # ...
    config.providers: env ❶
    config.providers.env.class: io.strimzi.kafka.EnvVarConfigProvider ❷
  # ...
  externalConfiguration:
    env:
      - name: DB_PASSWORD ❸
        valueFrom:
          secretKeyRef:
            name: db-creds ❹
            key: dbPassword ❺
  # ...
```

- ❶ 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。
- ❷ `EnvVarConfigProvider` は、環境変数から値を指定します。

- 3 DB_PASSWORD 環境変数は、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。

```
oc apply -f <kafka_connect_configuration_file>
```

3. コネクタ設定の環境変数を参照してください。

環境変数を参照するコネクタ設定の例

```
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 OPENSIFT クラスター外の KAFKA へのアクセス

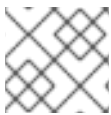
外部リスナーを使用して AMQ Streams の Kafka クラスターを OpenShift 環境外のクライアントに公開します。

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](#) を使用していません。
- **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 schema reference](#) を参照してください。

各接続タイプの長所と短所については、[Strimzi 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。以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: nodeport
        tls: true
      authentication:
        type: tls
```



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

- リソースを作成または更新します。

```
oc apply -f <kafka_configuration_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 ブローカーのアイデンティティを検証するクラスター CA 証明書もシークレット **<cluster_name>-cluster-ca-cert** に作成されます。

- Kafka** リソースのステータスから、Kafka クラスター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[?
(@.name=="<listener_name>")].bootstrapServers}{"\n"}
```

以下に例を示します。

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="external")].bootstrapServers}{"\n"}
```

- 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\.cert}' | 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

- 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。以下に例を示します。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: external
        port: 9094
        type: loadbalancer
        tls: true
      # ...
    # ...
  zookeeper:
    # ...

```

- リソースを作成または更新します。

```
oc apply -f <kafka_configuration_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 ブローカーのアイデンティティを検証するクラスター CA 証明書もシークレット **<cluster_name>-cluster-ca-cert** に作成されます。

- Kafka** リソースのステータスから、Kafka クラスター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[?(@.name=="<listener_name>")].bootstrapServers}{"\n"}
```

以下に例を示します。

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?(@.name=="external")].bootstrapServers}{"\n"}
```

- 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。

以下に例を示します。

```
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。

- リソースを作成または更新します。

```
oc apply -f <kafka_configuration_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 コントローラーを使用し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 ブローカーのアイデンティティを検証するクラスター CA 証明書もシークレット **<cluster_name>-cluster-ca-cert** に作成されます。

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。

- 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. OPENSIFT 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

- 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。以下に例を示します。

```
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 の名前を設定されます。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。

- リソースを作成または更新します。

```
oc apply -f <kafka_configuration_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用し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 ブローカーのアイデンティティを検証するクラスター CA 証明書もシークレット **<cluster_name>-cluster-ca-cert** に作成されます。

- Kafka** リソースのステータスから、Kafka クラスター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[?(@.name=="<listener_name>")].bootstrapServers}'{"\n"}
```

以下に例を示します。

```
oc get kafka my-cluster -o=jsonpath='{.status.listeners[?(@.name=="listener1")].bootstrapServers}'{"\n"}
```

- 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\.cert}' | 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認可

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。

外部クライアントを接続するための設定オプションの詳細は、[OpenShift クラスター外からの 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](#)
4. [カスタム認証](#)

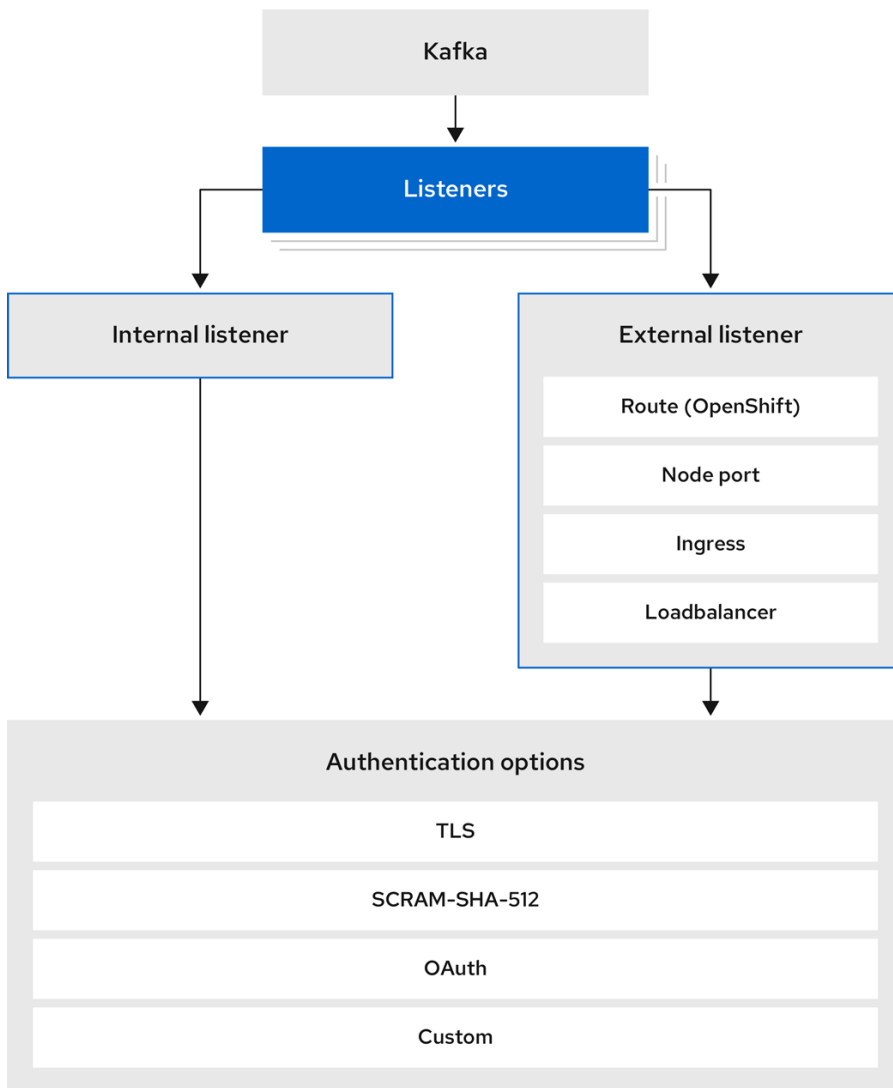
選択する認証オプションは、Kafka ブローカーへのクライアントアクセスを認証する方法によって異なります。



注記

カスタム認証を使用する前に、標準の認証オプションを試してみてください。カスタム認証では、kafka でサポートされているあらゆるタイプの認証が可能です。柔軟性を高めることができますが、複雑さも増します。

図5.1 Kafka リスナーの認証オプション



221_streams_0372

リスナーの **authentication** プロパティは、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。

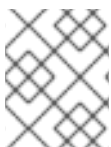
authentication プロパティが指定されていない場合、リスナーはそのリスナー経由で接続するクライアントを認証しません。認証がないと、リスナーではすべての接続が許可されます。

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) およびメトリック (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。

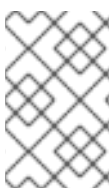
リスナー認証設定の例

```
# ...
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。

- 通信チャンネル上では、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 文字と数字で設定されるランダムな 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 の設定が 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。

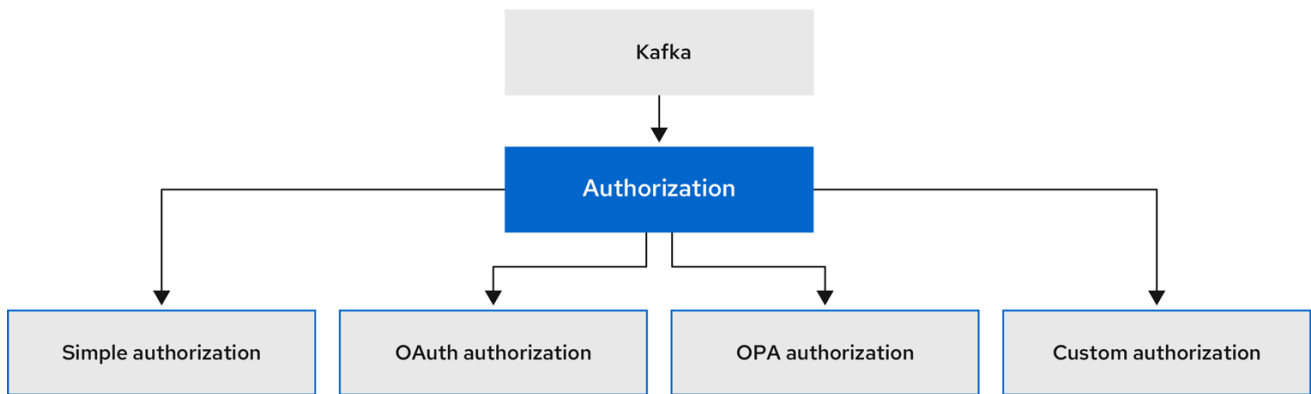
5.1.2. Kafka の承認

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 クラスタ承認オプション



222_Streams_0322

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 クラスタのスーパーユーザーを指定するには、**superUsers** プロパティ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コモンネームになります。

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のセキュリティーを確保する設定

Kafka ブローカーに安全にアクセスするための **KafkaUser** リソースの設定の詳細は、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 TLS クライアント認証を使用することが予想されますが、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 以外で発行された証明書を使用して TLS クライアントの認証を行います。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様に、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 証明書で指定したコモンネームです。

関連情報

- 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 スキーマ参照](#)

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、[10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= ❶
  sasl.jaas.config:
b3JnLmFwYWNoZS5rYWZrYS5jb21tb24uc2VjdXJpdHkuc2NyYW0uU2NyYW1Mb2dpcmk1vZHVzZSByZ
XF1aXJlZCB1c2VybmFtZT0ibXktdXNlciIgcGFzc3dvcmQ9ImdlbmVvYXRlZHBhc3N3b3JkIjsK ❷
```

❶ 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せん。
- 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せん。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まなう 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。

アクセス設定プロパティの詳細は、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** は **KafkaUser.spec.authorization** と一致します。

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** プロパティを設定します。
 - b. **listeners** プロパティを設定し、認証でリスナーを作成します。以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    authorization: ❶
      type: simple
      superUsers: ❷
        - CN=client_1
        - user_2
        - CN=client_3
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls ❸
    # ...
  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コモンネームです。
- 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_configuration_file>
```

Kafka クラスタは、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 ブローカーのアイデンティティを検証するクラスター CA 証明書もシークレット **<cluster_name>-cluster-ca-cert** に作成されます。

5.3.2. Kafka へのユーザーアクセスのセキュア化

KafkaUser を作成または変更して、Kafka クラスタへのセキュアなアクセスを必要とするクライアントを表します。

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。
以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication: ❶
    type: tls
  authorization:
    type: simple ❷
  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
```

❶ 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。

❷ 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 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

- Ingress NetworkPolicies をサポートする OpenShift クラスタ。
- Cluster Operator が稼働中である。

手順

1. **Kafka** リソースを開きます。
2. **networkPolicyPeers** プロパティで、Kafka クラスタ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 の設定](#)
- [NetworkPolicyPeer API リファレンス](#)

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て、[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認可を有効にし、中央の認可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 認可](#) 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。認可サーバーは、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 認可サーバーと通信し、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、認可サーバー、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。

クライアントの接続に OAuth 2.0 認証を使用するように Kafka ブローカーリスナーを設定します。必要な場合は、同じ **oauth** リスナーで OAUTHBEARER および PLAIN メカニズムを使用できます。各メカニズムをサポートするプロパティは、**oauth** リスナー設定で明示的に指定する必要があります。

OAUTHBEARER の概要

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
```

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サポートするために、アップストリームの Kafka Client Java ライブラリーに **OAuth コールバックハンドラー** が提供されます (ただし、他のライブラリーは対象外)。そのため、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認可サーバーに接続し、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ータ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



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 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
```

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単な認証メカニズムです。PLAIN を OAuth 2.0 認証で使用できるようにするために、AMQ Streams では **OAuth 2.0 over PLAIN** サーバー側のコールバックが提供されます。

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認可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

どちらの方法でも、クライアントは Kafka ブローカーにクレデンシャルを渡すために、PLAIN **username** および **password** プロパティを提供する必要があります。クライアントはこれらのプロパティを使用してクライアント ID およびシークレット、または、ユーザー名およびアクセストークンを渡します。

クライアント ID およびシークレットは、アクセストークンの取得に使用されます。

アクセストークンは、**password** プロパティの値として渡されます。**\$accessToken**: 接頭辞の有無に関わらずアクセストークンを渡します。

- リスナー設定でトークンエンドポイント (**tokenEndpointUri**) を設定する場合は、接頭辞が必要です。
- リスナー設定でトークンエンドポイント (**tokenEndpointUri**) を設定しない場合は、接頭辞は必要ありません。Kafka ブローカーは、パスワードを raw アクセストークンとして解釈します。

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。**userNameClaim**、**fallbackUserNameClaim**、**fallbackUsernamePrefix**、および **userInfoEndpointUri** プロパティを使用すると、リスナーにユーザー名抽出オプションを指定できます。ユーザー名の抽出プロセスも、認可サーバーによって異なります。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異なります。

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。

- 認可サーバーでの 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

認可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. 認可サーバーの 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、認可サーバーで 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

認可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、パブリック認可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、使用しないことを推奨します。

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 (期限切れでない) かどうかを確認します。
- 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、認可サーバーの通信は必要はありません。OAuth 2.0 の認可サーバーによって公開されるエンドポイントの **jwtEndpointUri** 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

認可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、通常は一意的な 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>
          jwtEndpointUri: <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 **jwtEndpointUri** 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、認可サーバーに応じて **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接続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** のトークン検証と共に使用できます。

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、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** 認可を使用したり、カスタム authorizer をインストールして、認可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。

- 認可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)。
- 認可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認可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、認可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、認可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、認可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、クライアントは認可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 クライアント認証フロー

OAuth 2.0 認証フローは、基礎となる Kafka クライアントおよび Kafka ブローカー設定によって異なります。フローは、使用する認可サーバーによってもサポートされる必要があります。

Kafka ブローカーリスナー設定は、クライアントがアクセストークンを使用して認証する方法を決定します。クライアントはクライアント ID およびシークレットを渡してアクセストークンをリクエストできます。

リスナーが PLAIN 認証を使用するように設定されている場合、クライアントはクライアント ID およびシークレット、または、ユーザー名およびアクセストークンで認証できます。これらの値は PLAIN メカニズムの **username** および **password** プロパティとして渡されます。

リスナー設定は、以下のトークン検証オプションをサポートします。

- 認可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を使用できます。認可サーバーは、トークンで署名を検証するため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します。
- 認可サーバーが提供するトークンイントロスペクションエンドポイントへの呼び出しを使用することができます。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認可サーバーに渡します。Kafka ブローカーは応答を確認して、トークンが有効かどうかを確認します。



注記

認可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。

Kafka クライアントクレデンシャルは、以下のタイプの認証に対して設定することもできます。

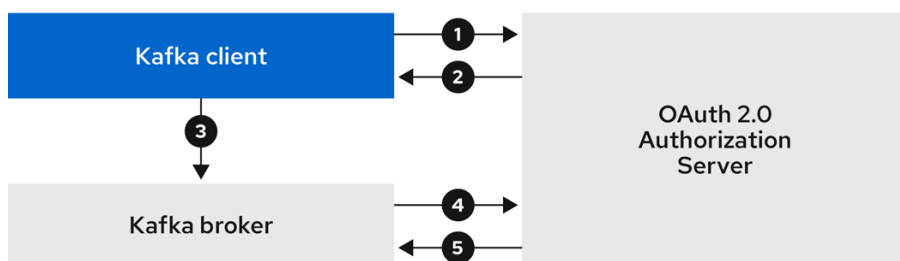
- 以前に生成された有効期間の長いアクセストークンを使用した直接ローカルアクセス
- 新しいアクセストークンを発行するための承認サーバーとの通信 (クライアント ID およびシークレットまたは更新トークンを使用)

5.4.5.1. SASL OAUTHBEARER メカニズムを使用したクライアント認証フローの例

SASL OAUTHBEARER メカニズムを使用して、Kafka 認証に以下の通信フローを使用できます。

- クライアントがクライアント ID とシークレットを使用し、ブローカーが検証を認可サーバーに委任する場合
- クライアントがクライアント ID およびシークレットを使用し、ブローカーが高速のローカルトークン検証を実行する場合
- クライアントが有効期限の長いアクセストークンを使用し、ブローカーが検証を認可サーバーに委任する場合
- クライアントが有効期限の長いアクセストークンを使用し、ブローカーが高速のローカル検証を実行する場合

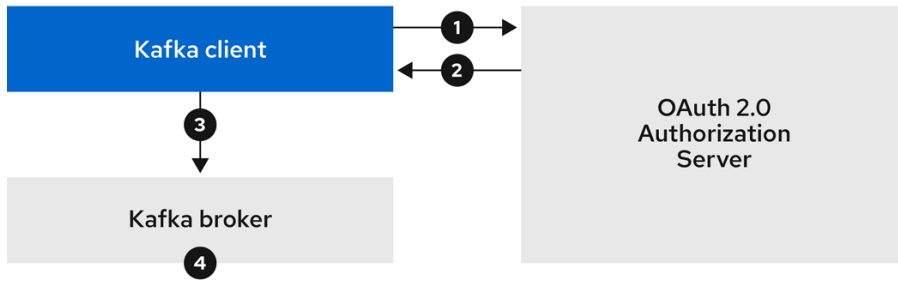
クライアントがクライアント ID とシークレットを使用し、ブローカーが検証を認可サーバーに委任する場合



222_Streams_0322

1. Kafka クライアントは、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からアクセストークンを要求し、必要に応じて更新トークンを要求します。
2. 承認サーバーは新しいアクセストークンを生成します。
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すことで Kafka ブローカーで認証されます。
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、認可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すことで、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、Kafka クライアントセッションが確立されます。

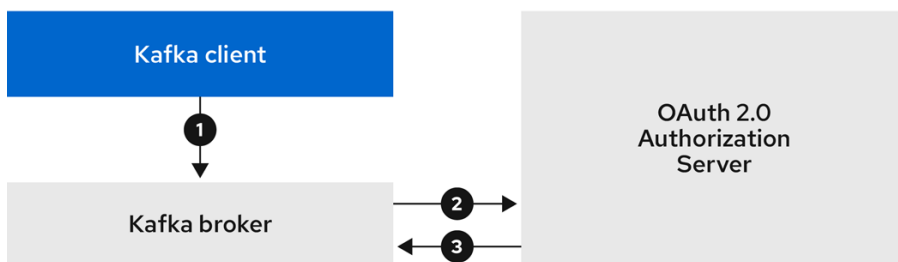
クライアントがクライアント ID およびシークレットを使用し、ブローカーが高速のローカルトークン検証を実行する場合



222_Streams_0322

1. Kafka クライアントは、クライアント ID およびシークレットを使用し、オプションで更新トークンを使用して、トークンエンドポイントから承認サーバーで認証します。
2. 承認サーバーは新しいアクセストークンを生成します。
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すことで Kafka ブローカーで認証されます。
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。

クライアントが有効期限の長いアクセストークンを使用し、ブローカーが検証を認可サーバーに委任する場合



222_Streams_0322

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すために Kafka ブローカーで認証します。
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、認可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、Kafka クライアントセッションが確立されます。

クライアントが有効期限の長いアクセストークンを使用し、ブローカーが高速のローカル検証を実行する場合



222_Streams_0322

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すために Kafka ブローカーで認証します。

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

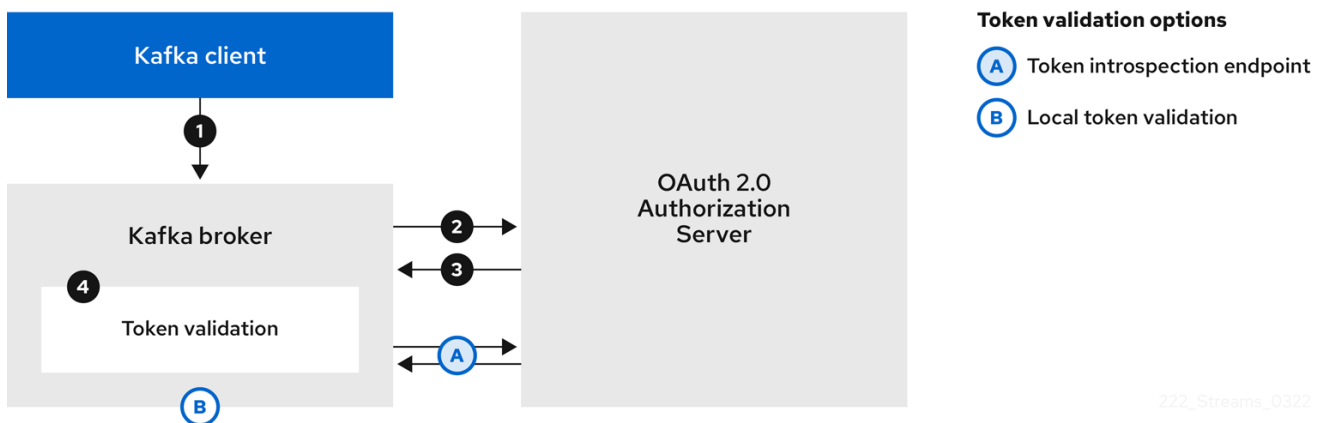
トークンが取り消された場合に認可サーバーとのチェックが行われなため、高速のローカル JWT トークン署名の検証は有効期限の短いトークンにのみ適しています。トークンの有効期限はトークンに書き込まれますが、失効はいつでも発生する可能性があるため、認可サーバーと通信せずに対応することはできません。発行されたトークンはすべて期限切れになるまで有効とみなされます。

5.4.5.2. SASL PLAIN メカニズムを使用したクライアント認証フローの例

OAuth PLAIN メカニズムを使用して、Kafka 認証に以下の通信フローを使用できます。

- クライアントがクライアント ID およびシークレットを使用し、ブローカーがクライアントのアクセストークンを取得する場合
- クライアントが、クライアント ID およびシークレットなしで有効期限の長いアクセストークンを使用する場合

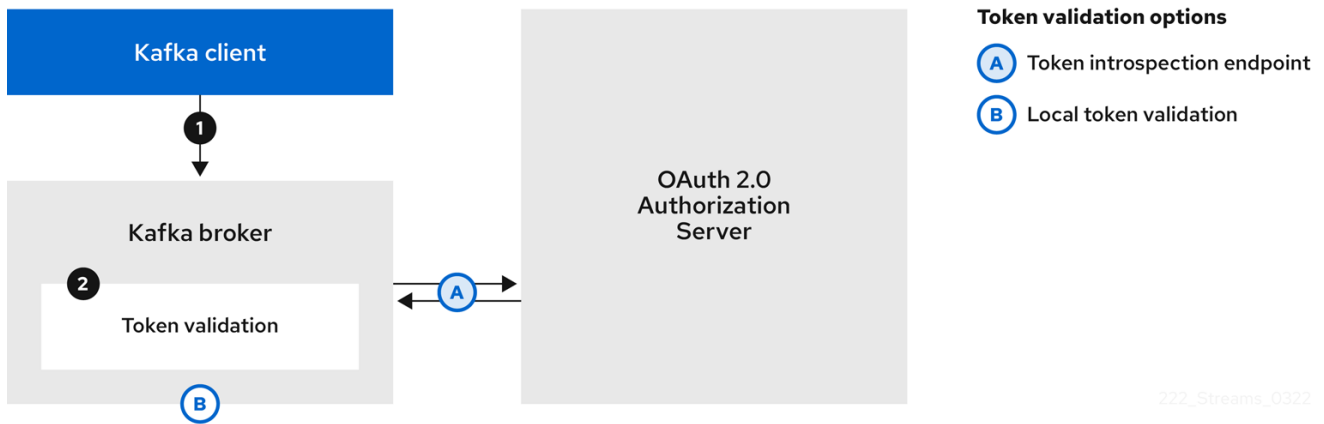
クライアントがクライアント ID およびシークレットを使用し、ブローカーがクライアントのアクセストークンを取得する場合



1. Kafka クライアントは、**clientId** をユーザー名として、**secret** をパスワードとして渡します。
2. Kafka ブローカーは、トークンエンドポイントを使用して **clientId** および **secret** を認可サーバーに渡します。
3. 認可サーバーは、新しいアクセストークンまたはエラー (クライアントクレデンシャルが有効でない場合) を返します。
4. Kafka ブローカーは、以下のいずれかの方法でトークンを検証します。
 - a. トークンイントロスペクションエンドポイントが指定されている場合、Kafka ブローカーは認可サーバーでエンドポイントを呼び出すことで、アクセストークンを検証します。トークンの検証に成功した場合には、セッションが確立されます。

- b. ローカルトークンのイントロスペクションが使用される場合、要求は認可サーバーに対して行われません。Kafka ブローカーは、JWT トークン署名チェックを使用して、アクセストークンをローカルで検証します。

クライアントが、クライアント ID およびシークレットなしで有効期限の長いアクセストークンを使用する場合



1. Kafka クライアントはユーザー名とパスワードを渡します。パスワードは、クライアントを実行する前に手動で取得および設定されたアクセストークンの値を提供します。
2. Kafka ブローカーリスナーが認証のトークンエンドポイントで設定されているかどうかに応じて、**\$accessToken**: 文字列の接頭辞の有無にかかわらず、パスワードは渡されます。
 - a. トークンエンドポイントが設定されている場合、パスワードの前に **\$accessToken** を付け、password パラメーターにクライアントシークレットではなくアクセストークンが含まれていることをブローカーに知らせる必要があります。Kafka ブローカーは、ユーザー名をアカウントのユーザー名として解釈します。
 - b. トークンエンドポイントが Kafka ブローカーリスナーで設定されていない場合 (**no-client-credentials mode** を強制)、パスワードは接頭辞なしでアクセストークンを提供する必要があります。Kafka ブローカーは、ユーザー名をアカウントのユーザー名として解釈します。このモードでは、クライアントはクライアント ID およびシークレットを使用せず、**password** パラメーターは常に raw アクセストークンとして解釈されます。
3. Kafka ブローカーは、以下のいずれかの方法でトークンを検証します。
 - a. トークンイントロスペクションエンドポイントが指定されている場合、Kafka ブローカーは認可サーバーでエンドポイントを呼び出すことで、アクセストークンを検証します。トークンの検証に成功した場合には、セッションが確立されます。
 - b. ローカルトークンイントロスペクションが使用されている場合には、認可サーバーへの要求はありません。Kafka ブローカーは、JWT トークン署名チェックを使用して、アクセストークンをローカルで検証します。

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. 認可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
3. OAuth 2.0 を使用するよう 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 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 認可サーバーの設定

この手順では、AMQ Streams と統合するために認可サーバーを設定するために必要な一般的な手順について説明します。

これらの手順は製品固有のものではありません。

手順は、選択した認可サーバーによって異なります。OAuth 2.0 アクセスの設定方法については、認可サーバーの製品ドキュメントを参照してください。



注記

認可サーバーがすでにデプロイされている場合は、デプロイ手順をスキップして、現在のデプロイを使用できます。

手順

1. 認可サーバーをクラスターにデプロイします。
2. 認可サーバーの CLI または管理コンソールにアクセスして、AMQ Streams 用に OAuth 2.0 を設定します。
AMQ Streams で動作するように認可サーバーを準備します。
3. **kafka-broker** クライアントを設定します。
4. アプリケーションの Kafka クライアントコンポーネントごとにクライアントを設定します。

次のステップ

認可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認可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イスで 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、プロパ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](#)
- [OAuth 2.0 認証メカニズム](#)

前提条件

- AMQ Streams および Kafka が稼働している。
- OAuth 2.0 の認可サーバーがデプロイされている。

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よりも短い場合、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): 認可サーバーへの 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、 **jwtRefreshSeconds** 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

OAuth 2.0 認証の適用方法や、認可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
  connectTimeoutSeconds: 60 13
  readTimeoutSeconds: 60 14
  groupsClaim: "$.groups" 15
  groupsClaimDelimiter: "," 16
```

- 1 認可サーバーが **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 認可サーバーは、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することがあります。**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
 - 6 (introspectionEndpointUri を使用する場合のみ該当): イン트로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、認可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。
 - 9 PLAIN メカニズムの追加設定。これが指定されている場合、クライアントは **\$accessToken**: 接頭辞を使用してアクセストークンを **password** として渡すことで、PLAIN 経由で認証できます。実稼働環境の場合は、常に **https://** urls を使用してください。
 - 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 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、inter-broker 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** を使用した 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、認可サーバーに応じて、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 **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、inter-broker 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** を使用した 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、認可サーバーに応じて、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。
 - 13 認可サーバーへの接続時のタイムアウト (秒単位)。デフォルト値は 60 です。
 - 14 認可サーバーへの接続時の読み取りタイムアウト (秒単位)。デフォルト値は 60 です。
 - 15 JWT トークンまたはイントロスペクションエンドポイントの応答からグループ情報を抽出するために使用される JsonPath クエリー。デフォルトでは設定されません。これは、カスタム承認者がユーザーグループに基づいて認可を決定するために使用できます。
 - 16 1つのコンマ区切りの文字列として返されるときにグループ情報を解析するために使用される区切り文字。デフォルト値は ,(コンマ) です。
3. エディターを保存して終了し、ローリング更新の完了を待ちます。
 4. 更新をログで確認するか、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
```

ローリング更新によって、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 を使用するための Kafka Java クライアントの設定

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を設定します。

前提条件

- AMQ Streams および Kafka が稼働している。
- OAuth 2.0 認可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。
- Kafka ブローカーが OAuth 2.0 に対して設定されている。

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>0.10.0.redhat-00014</version>
</dependency>
```

2. コールバックのシステムプロパティを設定します。
以下に例を示します。

```
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
System.setProperty(ClientConfig.OAUTH_SCOPE, "<scope_value>"); 4
System.setProperty(ClientConfig.OAUTH_AUDIENCE, "<audience_value>"); 5
```

- 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。認可サーバーで **クライアント** を作成するときに使用される名前です。
- 3 認可サーバーで **クライアント** 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。
- 4 (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 **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。
- 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。

サーバーでは、シリアルポートによるクライアントへの接続が必要になることがあります。

3. Kafka クライアント設定の TLS で暗号化された接続で **OAUTHBEARER** または **PLAIN** メカニズムを有効にします。
以下に例を示します。

Kafka クライアントの OAUTHBEARER の有効化

```
props.put("sasl.jaas.config",
"org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required;");
props.put("security.protocol", "SASL_SSL");
props.put("sasl.mechanism", "OAUTHBEARER");
props.put("sasl.login.callback.handler.class",
"io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHandler");
```

Kafka クライアントの PLAIN の有効化

```
props.put("sasl.jaas.config", "org.apache.kafka.common.security.plain.PlainLoginModule
required username=\"$CLIENT_ID_OR_ACCOUNT_NAME\"
password=\"$SECRET_OR_ACCESS_TOKEN\" ;");
props.put("security.protocol", "SASL_SSL"); ❶
props.put("sasl.mechanism", "PLAIN");
```

- ❶ この例では、TLS 接続で **SASL_SSL** を使用します。ローカル開発のみでは、暗号化されていない接続で **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。

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、認可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 コンポーネントと認可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。

- OAuth 2.0 認可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。
- Kafka ブローカーが OAuth 2.0 に対して設定されている。

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: MGQ1OTRmMzYtZTIIZS00MDY2LWI5OGEtMTM5MzM2NjdIZjQw 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 認可サーバーへの TLS 接続用の信用できる証明書。

OAuth 2.0 認証の適用方法や、認可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
accessTokensIsJwt: false 3
scope: any 4
audience: kafka 5
connectTimeoutSeconds: 60 6
readTimeoutSeconds: 60 7
```

- 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 認可サーバーによって、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 **accessTokensIsJwt: false** を適用することができます。
- 4 (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 **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。
- 6 (オプション) 認可サーバーへの接続時のタイムアウト (秒単位)。デフォルト値は 60 です。
- 7 (オプション): 認可サーバーへの接続時の読み取りタイムアウト (秒単位)。デフォルト値は 60 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、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
```

ローリング更新では、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 [認証サービス](#) 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。

関連情報

- [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リストを提供します。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 authorizer

AMQ Streams では、Red Hat Single Sign-On の **authorizer (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 authorizer を設定します。

authorizer は必要に応じて付与された権限のリストを認可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、認可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る。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、認可サーバーと Red Hat Single Sign-On の Authorization Services にアクセスできるようにします。以下に例を示します。

```

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
      connectTimeoutSeconds: 60 10
      readTimeoutSeconds: 60 11
    #...

```

- 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。実稼働環境の場合は、常に **https://** urls 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、**jwtEndpointUri** 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。
- 10 (オプション): Red Hat Single Sign-On トークンエンドポイントへの接続時のタイムアウト (秒単位)。デフォルト値は 60 です。

- 11 (オプション): Red Hat Single Sign-On トークンエンドポイントへの接続時の読み取りタイムアウト (秒単位)。デフォルト値は 60 です。

6. エディターを保存して終了し、ローリング更新の完了を待ちます。
7. 更新をログで確認するか、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
```

ローリング更新によって、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。

トピックの作成やリスト表示など、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、名前が同じ接頭辞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**)、承認スコープのリストから省略できます。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)
- 接頭辞一致 (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。

クラスター接頭辞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
```

トピックのリスト表示

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、トピックがリスト表示されません。

```
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 \
  --bootstrap-server 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、コンシューマーグループにコンシューマー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ー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。以下に例を示します。

```
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。以下に例を示します。

```
Topic:my-topic
Cluster:kafka-cluster
```

```
bin/kafka-console-producer.sh --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --producer.config=/tmp/config.properties --
  producer-property enable.idempotence=true --request-required-acks -1
```

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リスト表示時に、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

コンシューマー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

トピックの設定を変更するには、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 を使用し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 ブローカーの設定を変更するには、**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リストを提供します。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リストが更新されます。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 は [設定ファイルの例](#) を提供しています。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almの例。レalm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、**jwtEndpointUri** 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لم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لمをインポートします。管理コンソールのレلم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لم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لم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](#) の説明にしたがって、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イスを使用して、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の対応するロールも取得するようにします。

Clients

Clients は特定の設定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、および承認スコア

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 クラスターをデプロイメントします。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 認可サーバーが 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 認可サーバーが OpenShift クラスターにデプロイされ、サンプルレー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インタラクティブな Pod コンテナを新たに実行し、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-32-rhel8:2.2.2
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。

- Pod コンテナにアタッチします。

```
oc attach -ti kafka-cli -n $NS
```

- 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
```

- 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
```

- 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
```

- 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=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.client.id="team-a-client" \
  oauth.client.secret="team-a-client-secret" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-
connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHar
dler
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=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.client.id="team-b-client" \
  oauth.client.secret="team-b-client-secret" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$STOREPASS" \
  oauth.ssl.truststore.type="PKCS12" \
  oauth.token.endpoint.uri="https://$SSO_HOST/auth/realms/kafka-authz/protocol/openid-
connect/token" ;
sasl.login.callback.handler.class=io.strimzi.kafka.oauth.client.JaasClientOAuthLoginCallbackHar
dler
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=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.refresh.token="$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.JaasClientOAuthLoginCallbackHar
dler
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 の設定を使用して、**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 --bootstrap-server 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 --bootstrap-server 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. トピックをリスト表示します。

```
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リスト表示します。

```
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** が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 --bootstrap-server 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 --bootstrap-server 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 --bootstrap-server 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 --bootstrap-server 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リスト表示します。

```
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リスト表示できますが、**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リスト表示できます。

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 --bootstrap-server 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 --bootstrap-server 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** を使用して、**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** を使用して、**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** を使用して、**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** を使用して、**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 STRIMZI OPERATOR の使用

Strimzi 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. AMQ STREAMS OPERATOR を使用した名前空間の監視

Operator は、ネームスペース内の AMQ Streams リソースを監視および管理します。Cluster Operator は、OpenShift クラスター内の単一の名前空間、複数の名前空間、またはすべての名前空間を監視できます。Topic Operator と User Operator は、単一の名前空間を監視できます。

- Cluster Operator は **Kafka** リソースを監視します
- Topic Operator は **KafkaTopic** リソースを監視します
- User Operator は **KafkaUser** リソースを監視します

Topic Operator と User Operator は、名前空間内の単一の Kafka クラスターのみを監視できます。また、単一の Kafka クラスターにのみ接続できます。

複数のトピックオペレーターが同じ名前空間を監視すると、名前の衝突やトピックの削除が発生する可能性があります。これは、各 Kafka クラスターが同じ名前 (**__consumer_offsets** など) を持つ Kafka トピックを使用するためです。特定の名前空間を監視するトピック Operator が1つだけであることを確認してください。

単一の名前空間で複数の User Operator を使用する場合、特定のユーザー名を持つユーザーは複数の Kafka クラスターに存在できます。

Cluster Operator を使用して Topic Operator と User Operator をデプロイすると、デフォルトで Cluster Operator によってデプロイされた Kafka クラスターが監視されます。Operator 設定で **watchedNamespace** を使用して名前空間を指定することもできます。

各 Operator のスタンドアロンデプロイの場合、設定で監視する名前空間と Kafka クラスターへの接続を指定します。

6.2. 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。

6.2.1. 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トのコンテナ設定に関連します。**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 のコ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

オプションです。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_ZOOKEEPER_ADMIN_SESSION_TIMEOUT_MS

任意設定、デフォルトは 10000 ミリ秒です。

Cluster Operator の ZooKeeper 管理クライアントのセッションタイムアウト (ミリ秒単位)。タイムアウトの問題が原因で Cluster Operator からの ZooKeeper 要求が定期的に失敗する場合は、この値を大きくする必要があります。**maxSessionTimeout** 設定で ZooKeeper サーバー側に最大許容セッション時間が設定されます。デフォルトでは、このセッションの最大値は、**tickTime** (デフォルトは 2000) のデフォルト値の 20 倍、つまり 40000 ミリ秒です。タイムアウト時間を伸ばす必要がある場合は、**maxSessionTimeout** ZooKeeper サーバー設定値を変更する必要があります。

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

オプション: 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

オプションです。デフォルトの正規表現パターンは `^app.kubernetes.io/(?!part-of).*` です。メインのカスタムリソースからサブリソースへの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}` カスタムリソースで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、カスタムリソースの作成時か、次の調整時に 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、空白またはコンマ区切りの `<version>=<image>` ペアです。例: `3.2.3=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.3.0`, `3.3.1=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0` 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

オプションです。デフォルトは `registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0` です。**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、空白またはコンマ区切りの `<version>=<image>` ペアです。例: `3.2.3=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.3.0`, `3.3.1=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0`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、空白またはコンマ区切りの `<version>=<image>` ペアです。例: `3.2.3=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.3.0`, `3.3.1=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0`これは、`KafkaMirrorMaker.spec.version` プロパティが指定されていても `KafkaMirrorMaker.spec.image` プロパティが指定されていない場合に使用されます。

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

オプションです。デフォルトは `registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0` です。**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

オプションです。デフォルトは `registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0` です。**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

オプションです。デフォルトは `registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0` です。**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更新が実行されます。

STRIMZI_IMAGE_PULL_SECRETS

オプション:`Secret` 名のコ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、コンテナイメージがプルされるコンテナ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更新が実行されます。

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 接尾辞を上書きします。
デフォルトでは、OpenShift クラスタで割り当てられるサービスに、デフォルトの接尾辞 **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。

クラスタで異なる DNS 接尾辞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 接尾辞に変更します。

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が含まれるコンテナイメージをビルドしたり、低速なコンテナ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せん。

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_DNS_CACHE_TTL

オプションです。デフォルトは **30** です。ローカル DNS リゾルバーで成功した名前のルックアップをキャッシュする秒数。負の値を指定すると、キャッシュの期限はありません。ゼロはキャッシュされないことを意味します。これは、長いキャッシュポリシーが適用されることが原因の接続エラーを回避するのに役立ちます。

STRIMZI_POD_SET_RECONCILIATION_ONLY

オプションで、デフォルトは **false** です。**true** に設定すると、Cluster Operator は **StrimziPodSet** リソースのみを調整し、他のカスタムリソース (**Kafka**、**KafkaConnect** など) への変更は無視されます。このモードは、必要に応じて Pod が再作成されることを保証するのに役立ちますが、クラスタに他の変更は発生しません。

STRIMZI_FEATURE_GATES

オプション。**フィーチャーゲート**で制御される機能を有効または無効にします。

6.2.1.1. 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。

クラスター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せん。

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。

6.2.1.2. 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設定されています。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 Streams 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
#...
```

6.2.1.3. 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。

6.2.1.4. 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** などの管理されたり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。

6.2.1.5. 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。

6.2.1.6. 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。

6.2.1.7. ClusterRoles

Cluster Operator は、必要なり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

```

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:
  - "core.strimzi.io"
resources:
  # The cluster operator uses StrimziPodSets to manage the Kafka and ZooKeeper pods
  - strimzipodsets
  - strimzipodsets/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
  - replicaset
  # The cluster operator needs to access and manage replication controllers to manage replicaset
  - 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
```

```
- replicasetsets
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

```

6.2.1.8. 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トウェアネスの 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トウェアネスの 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

```

6.2.2. デフォルトのプロキシ設定を使用した Cluster Operator の設定

HTTP プロキシの背後で Kafka クラスタを実行している場合は、クラスタとの間でデータを出し入れできます。たとえば、プロキシ外からデータをプッシュおよびプルするコネクタで Kafka Connect を実行できます。または、プロキシを使用して認可サーバーに接続できます。

プロキシ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。プロキシ設定はすべての AMQ Streams コンテナに適用されます。

プロキシアドレスの形式は `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** 設定 (`install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml`) を更新します。

Cluster Operator のプロキシ設定の例

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
          # ...
          - name: "HTTP_PROXY"
            value: "http://proxy.com" ❶
          - name: "HTTPS_PROXY"
            value: "https://proxy.com" ❷
          - name: "NO_PROXY"
            value: "internal.com, other.domain.com" ❸
        # ...
```

- ❶ プロキシサーバーのアドレス。
- ❷ プロキシサーバーのセキュアなアドレス。
- ❸ プロキシ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 はコ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 の管理者の指定](#)

6.2.3. Cluster Operator での FIPS モードの設定

FIPS(Federal Information Processing Standards) は、コンピューターセキュリティーおよび相互運用性の標準です。FIPS 対応の OpenShift クラスタで AMQ Streams を実行する場合に、AMQ Streams コンテナイメージで使用される OpenJDK は自動的に FIPS モードに切り替わります。これにより、AMQ Streams がクラスタで実行されなくなります。AMQ Streams をクラスタにデプロイする場合、以下のようなエラーが表示されます。

```
Exception in thread "main" io.fabric8.kubernetes.client.KubernetesClientException: An error has occurred.
```

```
...
```

```
Caused by: java.security.KeyStoreException: sun.security.pkcs11.wrapper.PKCS11Exception: CKR_SESSION_READ_ONLY
```

```
...
```

```
Caused by: sun.security.pkcs11.wrapper.PKCS11Exception: CKR_SESSION_READ_ONLY
```

```
...
```

FIPS 対応のクラスタで AMQ Streams を実行する必要がある場合には、Cluster Operator のデプロイメント設定で **FIPS_MODE** 環境変数を **disabled** に設定すると、OpenJDK FIPS モードを無効にできます。AMQ Streams デプロイメントは FIPS に準拠しませんが、AMQ Streams Operator とそのすべてのオペランドは FIPS 対応の OpenShift クラスタで実行できます。

手順

1. Cluster Operator で FIPS モードを無効にするには、**Deployment** 設定 (**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml**) を更新し、**FIPS_MODE** 環境変数を追加します。

Cluster Operator の FIPS 設定例

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
          # ...
```

```
- name: "FIPS_MODE"
  value: "disabled" ❶
# ...
```

- ❶ FIPS モードを無効にします。

または、**Deployment** を直接編集します。

```
oc edit deployment strimzi-cluster-operator
```

- 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 apply -f install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml
```

関連情報

- [Federal Information Processing Standards \(FIPS\) の概要](#)

6.3. 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 クラスタで反映されます。

KafkaTopic リソースの詳細については、[KafkaTopic スキーマリファレンス](#) を参照してください。

Topic Operator のデプロイ

Cluster Operator を使用して、またはスタンドアロン Operator として Topic Operator をデプロイできます。Cluster Operator によって管理されていない Kafka クラスタで、スタンドアロンの Topic Operator を使用します。

デプロイ手順については、次を参照してください。

- [Cluster Operator を使用した Topic Operator のデプロイ \(推奨\)](#)
- [スタンドアロン Topic Operator のデプロイ](#)



重要

スタンドアロンの Topic Operator をデプロイするには、環境変数を設定して Kafka クラスタに接続する必要があります。これらの環境変数は、Cluster Operator によって設定されるため、Cluster Operator を使用して Topic Operator をデプロイする場合には設定する必要はありません。

6.3.1. 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。

6.3.1.1. トピック処理用の Kafka クラスタの特定

KafkaTopic リソースには、それが属する Kafka クラスターの名前 (**Kafka** リソースの名前から派生) を指定するラベルが含まれています。

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。

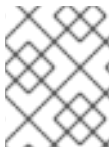
ラベルが Kafka クラスターと一致しない場合、Topic Operator は **KafkaTopic** を識別できず、トピックは作成されません。

6.3.1.2. 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しないでください。

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 での識別子と名前の要件については、[Object Names and IDs](#) を参照してください。

6.3.1.3. 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
# ...
```

6.3.2. 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。

6.3.2.1. 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。

6.3.2.2. 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 2.2 にアップグレードする場合は、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。

6.3.2.3. 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 より前のバージョンの AMQ Streams に戻す場合でも、Cluster Operator を前のバージョンにダウングレードしてから、Kafka ブローカーおよびクライアントアプリケーションを前の 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があります。以下に例を示します。

```
oc run kafka-admin -ti --image=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.2 --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。

6.3.2.4. 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。

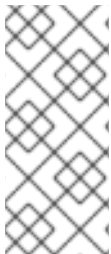
```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
```

```

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** ツールを使用し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** ツールを使用し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。

6.3.2.5. 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、このトピックストアにより、Kafka トピックでトピックが変更された場合、**および** OpenShift **KafkaTopic** カスタムリソースで更新された場合に、変更が矛盾しない限り、Topic Operator による管理を許可します。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。

関連情報

- [AMQ Streams のダウングレード](#)
- [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](#)

6.3.3. Kafka トピックの設定

KafkaTopic リソースのプロパティを使用して、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 はパーティションの数を減らすことをサポートしません。
- 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。
- AMQ Streams では、**KafkaTopic** リソースによる以下の変更はサポートされません。
 - **spec.replicas** を使用して、最初に指定されたレプリカの数を変更する
 - **spec.topicName** を使用したトピック名の変更

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 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. **KafkaTopic** リソースを設定します。

Kafka トピックの設定例

```
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>
```

3. トピックの準備完了ステータスが **True** に変わるまで待ちます。

```
oc get kafkatopics -o wide -w -n <namespace>
```

Kafka トピックのステータス

NAME	CLUSTER	PARTITIONS	REPLICATION	FACTOR	READY
my-topic-1	my-cluster	10	3	True	
my-topic-2	my-cluster	10	3		
my-topic-3	my-cluster	10	3	True	

READY 出力が **True** を示す場合、トピックの作成は成功です。

4. **READY** 列が空白のままの場合は、リソース YAML または Topic Operator ログからステータスの詳細を取得してください。
メッセージは、現在のステータスの理由に関する詳細を提供します。

```
oc get kafkatopics my-topic-2 -o yaml
```

NotReady ステータスのトピックの詳細

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-13T10:14:43.351550Z"
    message: Number of partitions cannot be decreased
    reason: PartitionDecreaseException
    status: "True"
    type: NotReady
```

この例では、トピックの準備ができていない理由は、**KafkaTopic** 設定で元のパーティション数が減ったためです。Kafka はこれをサポートしていません。

トピック設定をリセットした後、ステータスはトピックの準備ができていることを示します。

```
oc get kafkatopics my-topic-2 -o wide -w -n <namespace>
```

トピックのステータス更新

```
NAME          CLUSTER   PARTITIONS  REPLICATION  FACTOR  READY
my-topic-2    my-cluster  10          3             True
```

詳細のフェッチではメッセージが表示されない

```
oc get kafkatopics my-topic-2 -o yaml
```

READY ステータスのトピックの詳細

```
# ...
status:
  conditions:
  - lastTransitionTime: '2022-06-13T10:15:03.761084Z'
    status: 'True'
    type: Ready
```

6.3.4. 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 が稼働中である。

手順

- 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_configuration_file>
```

6.4. 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 クラスタで反映されます。

KafkaUser リソースの詳細については、[KafkaUser スキーマリファレンス](#) を参照してください。

User Operator のデプロイ

Cluster Operator を使用して、またはスタンドアロン Operator として User Operator をデプロイできます。Cluster Operator によって管理されない Kafka クラスタでは、スタンドアロンの User Operator を使用します。

デプロイ手順については、次を参照してください。

- [Cluster Operator を使用した User Operator のデプロイ \(推奨\)](#)
- [スタンドアロン User Operator のデプロイ](#)



重要

スタンドアロン User Operator をデプロイするには、環境変数を設定して Kafka クラスタに接続する必要があります。これらの環境変数は、Cluster Operator によって設定されるため、Cluster Operator を使用して User Operator をデプロイする場合には設定する必要はありません。

6.4.1. Kafka ユーザーの設定

KafkaUser リソースのプロパティを使用して、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。

以下に例を示します。

- `oc apply -f <user_config_file>`
- `oc delete KafkaUser <user_name>`

ユーザーは Kafka クライアントを表します。Kafka ユーザーを設定するとき、クライアントが Kafka にアクセスするのに必要なユーザーの認証および承認メカニズムを有効にします。使用するメカニズムは、同等の **Kafka** 設定と一致する必要があります。**Kafka** および **KafkaUser** リソースを使用して Kafka ブローカーへのアクセスを保護する方法の詳細については、Kafka ブローカーへの [アクセスの保護](#) を参照してください。

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。

手順

1. **KafkaUser** リソースを設定します。
この例では、TLS 認証と、ACL を使用した単純な承認を指定します。

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:
    type: tls
  authorization:
    type: simple
  acls:
    # Example consumer Acls for topic my-topic using consumer group my-group
    - resource:
        type: topic
        name: my-topic
        patternType: literal
        operation: Read
        host: "*"
    - resource:
        type: topic
        name: my-topic
        patternType: literal
        operation: Describe
        host: "*"
    - resource:
        type: group
        name: my-group
        patternType: literal
        operation: Read
        host: "*"
    # Example Producer Acls for topic my-topic
```

```

- 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: "*"

```

- OpenShift で **KafkaUser** リソースを作成します。

```
oc apply -f <user_config_file>
```

- ユーザーの Ready ステータスが **True** に変わるまで待ちます。

```
oc get kafkausers -o wide -w -n <namespace>
```

Kafka ユーザーの状態

NAME	CLUSTER	AUTHENTICATION	AUTHORIZATION	READY
my-user-1	my-cluster	tls	simple	True
my-user-2	my-cluster	tls	simple	
my-user-3	my-cluster	tls	simple	True

READY 出力が **True** を示す場合、ユーザーの作成は成功です。

- READY** 列が空白のままの場合は、リソース YAML またはユーザー Operator ログからステータスの詳細を取得します。
メッセージは、現在のステータスの理由に関する詳細を提供します。

```
oc get kafkausers my-user-2 -o yaml
```

NotReady ステータスのユーザーの詳細

```

# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:07:37.238065Z"
    message: Simple authorization ACL rules are configured but not supported in the
      Kafka cluster configuration.
    reason: InvalidResourceException
    status: "True"
    type: NotReady

```

この例では、ユーザーの準備ができていない理由は、**Kafka** 設定で簡易認証が有効になっていないためです。

簡単な承認のための Kafka 設定

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

Kafka 設定を更新した後、ステータスはユーザーの準備ができていることを示します。

```
oc get kafkausers my-user-2 -o wide -w -n <namespace>
```

ユーザーのステータス更新

```
NAME      CLUSTER  AUTHENTICATION  AUTHORIZATION  READY
my-user-2 my-cluster  tls             simple         True
```

詳細を取得してもメッセージは表示されません。

```
oc get kafkausers my-user-2 -o yaml
```

READY ステータスのユーザーの詳細

```
# ...
status:
  conditions:
  - lastTransitionTime: "2022-06-10T10:33:40.166846Z"
    status: "True"
    type: Ready
```

6.4.2. 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 が稼働中である。

手順

- 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。

6.5. フィーチャーゲートの設定

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。

フィーチャー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名と接頭辞のコンマ区切りリストを指定します。**+**接頭辞はフィーチャーゲートを有効にし、**-**接頭辞を無効にします。

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

```

6.5.1. ControlPlaneListener フィーチャーゲート

ControlPlaneListener フィーチャーゲートのデフォルトの状態は **enabled** です。

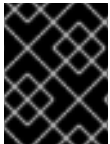
ControlPlaneListener フィーチャーゲートを使用して、Kafka クラスター内のブローカー間通信に使用される通信パスを変更します。AMQ Streams では、コントロールプレーンのトラフィックは、Kafka クラスターの必要な状態を維持するコントローラーコネクションで設定されます。データプレーントラフィックは、主にリーダーブローカーとフォロワーブローカー間のデータレプリケーションで設定されます。

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。

ControlPlaneListener フィーチャーゲートの無効化

ControlPlaneListener フィーチャーゲートを無効にするには、Cluster Operator 設定の **STRIMZI_FEATURE_GATES** 環境変数に **-ControlPlaneListener** を指定します。 **ControlPlaneListener** フィーチャー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。



重要

AMQ Streams 1.7 以前のバージョンのアップグレードまたはダウングレード時に、**ControlPlaneListener** フィーチャーゲートを無効にする必要があります。

6.5.2. ServiceAccountPatching フィーチャーゲート

ServiceAccountPatching フィーチャーゲートのデフォルトの状態は **enabled** です。

デフォルトで、Cluster Operator はサービスアカウントを調整して必要に応じて更新します。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。サービスアカウントのパッチ適用を無効にするには、**ServiceAccountPatching** フィーチャーゲートを無効にします。

ServiceAccountPatching フィーチャーゲートの無効化

ServiceAccountPatching フィーチャーゲートを無効にするには、Cluster Operator 設定の **STRIMZI_FEATURE_GATES** 環境変数に **-ServiceAccountPatching** を指定します。

6.5.3. UseStrimziPodSets フィーチャーゲート

UseStrimziPodSets フィーチャーゲートのデフォルトの状態は **disabled** です。

現在、AMQ Streams は StatefulSets に依存し、ZooKeeper および Kafka クラスターの Pod を作成し、管理します。AMQ Streams は StatefulSet を作成し、OpenShift は StatefulSet 定義に応じて Pod を作成します。Pod が削除されると、OpenShift は Pod を再作成します。StatefulSets の使用には以下の制限があります。

- Pod は常にインデックス番号に基づいて作成または削除される
- StatefulSet のすべての Pod には同様の設定が必要である
- StatefulSet での Pod のストレージ設定の変更が複雑になる

UseStrimziPodSets フィーチャーゲートは、**StrimziPodSet** と呼ばれる Pod の管理用のリソースを導入します。フィーチャーゲートが有効な場合には、StatefulSets の代わりにこのリソースが使用されます。AMQ Streams は、OpenShift ではなく Pod の作成および管理を処理します。StatefulSets の代わりに StrimziPodSets を使用すると、機能の制御が強化されます。

UseStrimziPodSets フィーチャーゲートの有効化

UseStrimziPodSets フィーチャーゲートを有効にするには、Cluster Operator 設定の **STRIMZI_FEATURE_GATES** 環境変数に **+UseStrimziPodSets** を指定します。

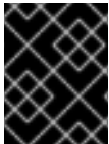
**重要**

AMQ Streams 2.0 以前のバージョンへのダウングレード時に、**UseStrimziPodSets** フィーチャーゲートを無効にする必要があります。

6.5.4. (プレビュー) UseKRaft フィーチャーゲート

UseKRaft フィーチャーゲートのデフォルト状態は **disabled** です。

UseKRaft フィーチャーゲートは、ZooKeeper なしで KRaft (Kafka Raft メタデータ) モードで Kafka クラスターをデプロイします。このフィーチャーゲートは現在、開発とテストのみを目的としています。

**重要**

KRaft モードは、Apache Kafka または AMQ Streams での運用の準備ができていません。

UseKRaft フィーチャーゲートが有効になっている場合、Kafka クラスターは ZooKeeper なしでデプロイされます。Kafka カスタムリソースの **.spec.zookeeper** プロパティは無視されますが、存在する必要があります。**UseKRaft** フィーチャーゲートは、Kafka クラスターノードとそのロールを設定する API を提供します。API はまだ開発中であり、KRaft モードが本番環境に対応する前に変更される予定です。

現在、AMQ Streams の KRaft モードには、次の主要な制限があります。

- ZooKeeper を使用する Kafka クラスターから KRaft クラスターへの移動、またはその逆の移動はサポートされていません。
- Apache Kafka バージョンまたは AMQ Streams Operator のアップグレードとダウングレードはサポートされていません。ユーザーは、クラスターを削除し、Operator をアップグレードして、新しい Kafka クラスターをデプロイする必要がある場合があります。
- Entity Operator (User Operator と Topic Operator を含む) はサポートされていません。**spec.entityOperator** プロパティは、Kafka カスタムリソースから削除する必要があります。
- **simple** 認証はサポートされていません。
- SCRAM-SHA-512 認証はサポートされていません。
- JBOD ストレージはサポートされていません。**type: jbod** ストレージを使用できますが、JBOD アレイに含めることができるディスクは1つだけです。
- Liveness および Readiness プロブは無効になっています。
- すべての Kafka ノードには、**controller** と **broker** の両方の KRaft ロールがあります。個別の **controller** と **broker** ノードを持つ Kafka クラスターはサポートされていません。

UseKRaft フィーチャーゲートの有効化

UseKRaft フィーチャーゲートを有効にするには、Cluster Operator 設定の **STRIMZI_FEATURE_GATES** 環境変数に **+UseKRaft** を指定します。

**重要**

UseKRaft フィーチャーゲートは、**UseSrimziPodSets** フィーチャーゲートに依存します。**UseKRaft** 機能ゲートを有効にする場合は、**UseSrimziPodSets** 機能ゲートも有効になっていることを確認してください。

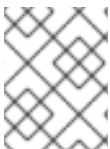
6.5.5. フィーチャーゲートリリース

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): 通常は常に有効

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ありません。Alpha または Beta 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。

- **ControlPlaneListener** フィーチャーゲートは AMQ Streams 2.0 のベータステージに移行しました。
- **ServiceAccountPatching** フィーチャーゲートは、AMQ Streams 2.0 のベータステージに移行しました。
- **UseStrimziPodSets** フィーチャーゲートは現在アルファステージにあります。
- **UseKRaft** フィーチャーゲートは開発用にのみ利用可能であり、現在、ベータフェーズに移行する予定のリリースはありません。

**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	2.0	-
ServiceAccountPatching	1.8	2.0	-
UseStrimziPodSets	2.1	-	-
UseKRaft	2.2	-	-

フィーチャーゲートが有効な場合は、特定の AMQ Streams バージョンからアップグレードまたはダウ

ングレードを行う前に無効にする必要がある場合があります。以下の表は、AMQ Streams バージョンのアップグレードまたはダウングレード時に無効にする必要のあるフィーチャーゲートを示しています。

表6.2 AMQ Streams のアップグレードまたはダウングレード時に無効にするフィーチャーゲート

フィーチャーゲートの無効化	AMQ Streams バージョンからのアップグレード	AMQ Streams バージョンへのダウングレード
ControlPlaneListener	1.7 以前	1.7 以前
UseStrimziPodSets	-	2.0 以前

6.6. PROMETHEUS メトリクスを使用した OPERATOR の監視

AMQ Streams の operator は Prometheus メトリックを公開します。メトリック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 メトリック

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 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 へのメトリックの導入](#)を参照してください。

第7章 CRUISE CONTROL によるクラスターのリバランス

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再分散、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 で設定されます。

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。**Kafka** リソースの設定を通じて Cruise Control をデプロイします。最適化の提案を生成して適用する **KafkaRebalance** リソースを介してリバランスを実行します。

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。

- 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。

最適化ゴール

最適化ゴールは、リバランスから達成する特定のゴールを表します。たとえば、トピックのレプリカをブローカー間でより均等に分散することがゴールになる場合があります。設定から追加するゴールを変更できます。ゴールは、ハードゴールまたはソフトゴールとして定義されます。Cruise Control 展開設定を使用してハード目標を追加できます。また、これらの各カテゴリーに適合するメイン、デフォルト、およびユーザー提供の目標もあります。

- **ハードゴール** は事前設定されており、最適化プロポーザルが正常に実行されるには満たされる必要があります。
- 最適化プロポーザルが正常に実行されるには、**ソフトゴール** を満たす必要はありません。これは、すべてのハードゴールが一致することを意味します。
- **メインゴール** は Cruise Control から継承されます。ハードゴールとして事前設定されているものもあります。メインゴールは、デフォルトで最適化プロポーザルで使用されません。
- **デフォルトのゴール** は、デフォルトでメインゴールと同じです。デフォルトゴールのセットを指定できます。
- **ユーザー提供のゴール** は、特定の最適化プロポーザルを生成するために設定されるデフォルトゴールのサブセットです。

最適化プロポーザル

最適化プロポーザルは、リバランスから達成するゴールで構成されます。最適化プロポーザルを生成して、提案された変更の概要と、リバランス可能な結果を作成します。ゴールは特定の優先順位で評価されます。その後、プロポーザルの承認または拒否を選択できます。プロポーザルを拒否し、調整したゴールセットを使用して再度実行できます。3 つのモードのいずれかで最適化プロポーザルを生成できます。

- **full** はデフォルトのモードで、完全なリバランスを実行します。
- **add-brokers** は、Kafka クラスターをスケールアップするときにブローカーを追加した後に使用するモードです。
- **remove-brokers** は、Kafka クラスターを縮小するときにブローカーを削除する前に使用するモードです。

自己修復、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、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 は [設定ファイルの例](#) を提供しています。Cruise Control の YAML 設定ファイルのサンプルは、[examples/cruise-control/](#) にあります。

7.1. CRUISE CONTROL とは

Cruise Control は、分散された Kafka クラスタ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 クラスタに適用します。クラスタ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 クラスタはより均等に分散されます。

関連情報

- [Cruise Control の Wiki](#)

7.2. 最適化ゴールの概要

最適化ゴールは、Kafka クラスタ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。Cruise Control は Kafka クラスタ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。

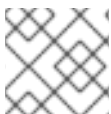
7.2.1. 優先度によるゴールの順序

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. 一連のトピック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. 容量ゴール
 - ディスク容量
 - ネットワークのインバウンド容量
 - 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. リソース配分ゴール
 - 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
8. リーダーへの単位時間あたりバイト流入量の分布
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. 優先リーダーエレクトション
12. ブローカー内のディスク容量
13. ブローカー内のディスク使用量の分散

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

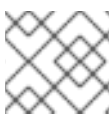
独自のゴールの記述および Kafka アサイナーゴールはまだサポートされていません。

7.2.2. 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、満たなければならない厳しい最適化ゴールのほか、メイン、デフォルト、およびユーザーが指定した最適化ゴールの設定があります。

最適化ゴールは、以下の設定で指定できます。

- Main goals – `Kafka.spec.cruiseControl.config.goals`
- Hard goals – `Kafka.spec.cruiseControl.config.hard.goals`
- Default goals – `Kafka.spec.cruiseControl.config.default.goals`
- ユーザー提供のゴール – `KafkaRebalance.spec.goals`



注記

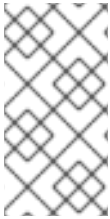
リソース配布の目標は、ブローカーリソース [の容量制限の](#) 対象となります。

7.2.3. ハードおよびソフト最適化ゴール

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、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。

7.2.4. 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
```

これらの目標の一部は、[ハードゴール](#) として事前設定されています。

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。

7.2.5. デフォルトの最適化ゴール

Cruise Control 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、メインの最適化ゴールがデフォルトの最適化ゴール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。

- 主な最適化ゴール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。

7.2.6. 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のゴールが含まれる **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。そうしないと、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。

関連情報

- [Configuring and deploying Cruise Control with Kafka](#)
- Cruise Control Wiki の [Configurations](#)

7.3. 最適化プロポーザルの概要

KafkaRebalance リソースを使用して、最適化の提案で提案された変更を生成および適用します。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、それを生成するために使用された [最適化ゴール](#) のセットに基づいており、ブローカーリソースに設定された [容量制限](#) が適用されます。

すべての最適化プロポーザルは、提案されたリバランスの影響の [見積もり](#) です。提案は、承認または却下できます。最初に最適化プロポーザルを生成しなければ、クラスターのリバランスは承認できません。

次のリバランスモードのいずれかで最適化の提案を実行できます。

- **full**
- **add-brokers**
- **remove-brokers**

7.3.1. リバランスモード

KafkaRebalance カスタムリソースの **spec.mode** プロパティを使用して、リバランスモードを指定します。

full

full モードでは、クラスター内のすべてのブローカー間でレプリカを移動することにより、完全なリバランスが実行されます。これは、**KafkaRebalance** カスタムリソースで **spec.mode** プロパティが定義されていない場合のデフォルトモードです。

add-brokers

add-brokers モードは、1つ以上のブローカーを追加して Kafka クラスターをスケールアップした後に使用されます。通常、Kafka クラスターをスケールアップした後、新しいブローカーは、新しく作成されたトピックのパーティションのみをホストするために使用されます。新しいトピックが作成されないと、新たに追加されたブローカーは使用されず、既存のブローカーは同じ負荷のままになります。クラスターにブローカーを追加した直後に **add-brokers** モードを使用すると、リバランス操作によってレプリカが既存のブローカーから新しく追加されたブローカーに移動します。**KafkaRebalance** カスタムリソースの **spec.brokers** プロパティを使用して、新しいブローカーをリストとして指定します。

remove-brokers

remove-brokers モードは、1つ以上のブローカーを削除して Kafka クラスターをスケールダウンする前に使用されます。Kafka クラスターをスケールダウンすると、レプリカをホストする場合でもブローカーはシャットダウンされます。これにより、レプリケートが不十分なパーティションとなる可能性があり、一部のパーティションが最小 In-Sync レプリカ (ISR) を下回る可能性があります。こ

の潜在的な問題を回避するために、**remove-brokers** モードは、削除されるブローカーからレプリカを移動します。これらのブローカーがレプリカをホストしなくなった場合は、スケールダウン操作を安全に実行できます。**KafkaRebalance** カスタムリソースの **spec.brokers** プロパティで、削除するブローカーをリストとして指定します。

一般に、**full** リバランスモードを使用して、ブローカー間で負荷を分散することにより Kafka クラスターをリバランスします。**add-brokers** および **remove-brokers** モードは、クラスターをスケールアップまたはスケールダウンし、それに応じてレプリカを再調整する場合にのみ使用してください。

リバランスを実行する手順は、実際には3つの異なるモードで同じです。唯一の違いは、**spec.mode** プロパティを介してモードを指定することと、必要に応じて、**spec.brokers** プロパティを介して追加または削除されるブローカーを一覧表示することです。

7.3.2. 最適化提案の結果

最適化の提案が生成されると、概要とブローカーの負荷が返されます。

概要

要約は **KafkaRebalance** リソースに含まれています。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。正常に生成された最適化プロポーザルの要約は、**KafkaRebalance** リソースの **Status.OptimizationResult** プロパティに含まれています。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。

ブローカーの負荷

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。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。

7.3.3. 最適化プロポーザルの承認または拒否

最適化プロポーザルの要約は、提案された変更の範囲を示しています。

KafkaRebalance リソースの名前を使用して、コマンドラインから要約を返すことができます。

最適化プロポーザルの要約を返す方法

```
oc describe kafkarebalance <kafka_rebalance_resource_name> -n <namespace>
```

jq コマンドライン JSON パーサーツールを使用することもできます。

jq を使用して最適化プロポーザルの要約を返す方法

```
oc get kafkarebalance -o json | jq <jq_query>.
```

サマリーを使用して、最適化プロポーザルを承認するか拒否するかを決定します。

最適化プロポーザルの承認

最適化プロポーザルを承認するには、**KafkaRebalance** リソースの **strimzi.io/rebalance** アノテーションを **approve** するように設定します。Cruise Control は、プロポーザルを Kafka クラスターに適用し、クラスターのリバランス操作を開始します。

最適化プロポーザルの拒否

最適化プロポーザルを承認しないことを選択した場合は、[最適化ゴールの変更](#) または [任意のリバランスパフォーマンスチューニングオプションの更新](#) を行い、その後で別のプロポーザルを生成でき

まず、strimzi.io/refresh アノテーションを使用して、**KafkaRebalance** リソースの新しい最適化プロポーザルを生成できます。

最適化プロポーザルを使用して、リバランスに必要な動作を評価します。たとえば、要約ではブローカー間およびブローカー内の動きについて記述します。ブローカー間のリバランスは、別々のブローカー間でデータを移動します。JBOD ストレージ設定を使用していると、ブローカー内のリバランスでは同じブローカー上のディスク間でデータが移動します。このような情報は、プロポーザルを承認しない場合でも有用な場合があります。

リバランスの際には Kafka クラスターに追加の負荷がかかるため、最適化プロポーザルを却下したり、承認を遅らせたりする場合があります。

次の例では、プロポーザルは別々のブローカー間のデータのリバランスを提案しています。リバランスには、ブローカー間での 55 個のパーティションレプリカ (合計 12 MB のデータ) の移動が含まれます。パーティションレプリカのブローカー間の移動は、パフォーマンスに大きな影響を与えますが、データ総量はそれほど多くありません。合計データが膨大な場合は、プロポーザルを却下するか、リバランスを承認するタイミングを考慮して Kafka クラスターのパフォーマンスへの影響を制限できます。

リバランスパフォーマンスチューニングオプションは、データ移動の影響を減らすのに有用です。リバランス期間を延長できる場合は、リバランスをより小さなバッチに分割できます。一回のデータ移動が少なくなると、クラスターの負荷も軽減できます。

最適化プロポーザルサマリーの例

```
Name:      my-rebalance
Namespace: myproject
Labels:    strimzi.io/cluster=my-cluster
Annotations: API Version: kafka.strimzi.io/v1alpha1
Kind:      KafkaRebalance
Metadata:
# ...
Status:
Conditions:
  Last Transition Time: 2022-04-05T14:36:11.900Z
  Status:              ProposalReady
  Type:                State
Observed Generation:  1
Optimization Result:
  Data To Move MB:    0
  Excluded Brokers For Leadership:
  Excluded Brokers For Replica Move:
  Excluded Topics:
  Intra Broker Data To Move MB:      12
  Monitored Partitions Percentage:   100
  Num Intra Broker Replica Movements: 0
  Num Leader Movements:              24
  Num Replica Movements:             55
  On Demand Balancedness Score After: 82.91290759174306
  On Demand Balancedness Score Before: 78.01176356230222
  Recent Windows:                   5
Session Id:                         a4f833bd-2055-4213-bfdd-ad21f95bf184
```

このプロポーザルでは、24 のパーティションリーダーも別のブローカーに移動します。これには、パフォーマンスへの影響が少ない ZooKeeper の設定を変更する必要があります。

バランススコアは、最適化プロポーザルが承認される前後の Kafka クラスターの全体的なバランスの測

定値です。バランススコアは、最適化ゴールに基づいています。すべてのゴールが満たされていると、スコアは 100 になります。達成されないゴールごとにスコアが減少します。バランススコアを比較して、Kafka クラスターのバランスがリバランス後よりも悪いかどうかを確認します。

7.3.4. 最適化プロポーザルサマリーのプロパティー

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7.1 最適化プロポーザルに含まれるプロパティーの概要

JSON プロパティー	説明
numIntraBrokerReplicaMovements	<p>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。</p> <p>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。</p>
excludedBrokersForLeadership	サポートされていません。空のリストが返されます。
numReplicaMovements	<p>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。</p> <p>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。</p>
onDemandBalancednessScore Before, onDemandBalancednessScore After	<p>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。</p> <p>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の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。</p> <p>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。</p>
intraBrokerDataToMoveMB	<p>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)。</p> <p>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。値が大きいくほど、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)。</p>
recentWindows	最適化プロポーザルの基になるメトリックウィンドウの数。

JSON プロパティ	説明
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)。 <p>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。値が大きいくほど、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。</p>
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。 <p>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。</p>
excludedBrokersForReplicaMove	サポートされていません。空のリストが返されます。

7.3.5. ブローカーのロードプロパティ

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ックにリンクする各ブローカー ID のキーを持つ JSON オブジェクトで設定されます。各メトリックは 3 つの値で設定されます。1 つ目は、最適化プロポーザルの適用前のメトリックの値です。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** 状態にある则表示され、リバランスが完了すると残ります。

ConfigMap の名前を使用して、コマンドラインからデータを表示できます。

ConfigMap データを返す方法

```
oc describe configmaps <my_rebalance_configmap_name> -n <namespace>
```

jq コマンドライン JSON パーサーツールを使用して、ConfigMap から JSON 文字列を抽出することもできます。

jq を使用した ConfigMap からの JSON 文字列の抽出

```
oc get configmaps <my_rebalance_configmap_name> -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について説明しています。

JSON プロパティ	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。

7.3.6. 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。

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」](#)

7.4. 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行われる方法を制御し、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。

7.4.1. 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設定されています。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設定されます。

- **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。

7.4.2. 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を行うと、このストラテジーではその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。

7.4.3. ブローカー内のディスクバランシング

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。Kafka デプロイメントで、同じブローカーにディスクが複数割り当てられた JBOD ストレージを使用している場合には、Cruise Control はディスク間でパーティションを分散できます。



注記

1つのディスクで JBOD ストレージを使用している場合は、分散するディスクがないため、ブローカー内でディスク分散すると、パーティションの移動が 0 と提案されます。

ブローカー内のディスク分散を実行するには、**KafkaRebalance.spec** の下で **rebalanceDisk** を **true** に設定します。**rebalanceDisk** を **true** に設定する場合は、Cruise Control はブローカー内のゴールを自動的に設定し、ブローカー間のゴールを無視するため、**KafkaRebalance.spec** の **goals** フィールドを設定しないでください。Cruise Control はブローカー間およびブローカー内の分散を同時に実行しません。

7.4.4. 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、Kafka または [最適化提案 レベル](#) で [Cruise Control を設定および展開する](#) ときに設定できます。

- Cruise Control のサーバー設定は、Kafka カスタムリソースの下の **Kafka.spec.cruiseControl.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。

関連する設定を以下の表にまとめています。

表7.2 リバランスパフォーマンスチューニングの設定

Cruise Control プロパティ	KafkaRebalance プロパティ	デフォルト	説明
num.concurrent.partition.movement.per.broker	concurrentPartitionMovementsPerBroker	5	各パーティション再割り当てバッチにおける inter-broker パーティション移動の最大数。
num.concurrent.intra.broker.partition.movement	concurrentIntraBrokerPartitionMovements	2	各パーティション再割り当てバッチにおけるブローカー内パーティション移動の最大数。
num.concurrent.leader.movement	concurrentLeaderMovements	1000	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。

Cruise Control プロパティ	KafkaRebalance プロパティ	デフォルト	説明
default.replication.throttle	replicationThrottle	Null (制限なし)	パーティションの再割り当てに割り当てる帯域幅 (バイト/秒単位)。
default.replica.movement.strategies	replicaMovementStrategies	Base ReplicaMovementStrategy	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コ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。

Cruise Control プロパティ	KafkaRebalance プロパティ	デフォルト	説明
-	rebalanceDisk	false	ブローカー内のディスク分散を有効にし、同じブローカーのディスク間でディスク領域の使用率を分散します。ディスクが複数割り当てられた JBOD ストレージを使用する Kafka デプロイメントにのみ適用されます。

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ます。その逆も同様です。

関連情報

- [「CruiseControlSpec スキーマ参照」](#)
- [「KafkaRebalanceSpec スキーマ参照」](#)

7.5. CONFIGURING AND DEPLOYING CRUISE CONTROL WITH KAFKA

Kafka リソース設定を通じて、クラスター Operator は Kafka クラスターのデプロイ時に Cruise Control をデプロイできます。**Kafka** リソースの **CruiseControl** プロパティを使用して、デプロイを設定できます。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。

最適化の提案を生成するための最適化ゴールを指定するには、Cruise Control **config** で **goals** 設定を使用します。**brokerCapacity** を使用して、リソース配分に関連するゴールのデフォルトの容量制限を変更できます。ブローカーが異種ネットワークリソースを持つノードで実行されている場合、**overrides** を使用して各ブローカーのネットワーク容量制限を設定できます。

空のオブジェクト ({}) が **CruiseControl** 設定に使用されている場合、すべてのプロパティはデフォルト値を使用します。

前提条件

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

手順

1. **Kafka** リソースの **cruiseControl** プロパティを編集します。設定可能なプロパティ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: > 3
      com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
    # ...
    hard.goals: >
      com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
      com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
    # ...
    cpu.balance.threshold: 1.1
    metadata.max.age.ms: 300000
    send.buffer.bytes: 131072
    webserver.http.cors.enabled: true 4
    webserver.http.cors.origin: "*"
    webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
    # ...
  resources: 5
    requests:
      cpu: 1
      memory: 512Mi
    limits:
      cpu: 2
      memory: 2Gi
  logging: 6
    type: inline
    loggers:
      rootLogger.level: "INFO"
  template: 7
    pod:
      metadata:
        labels:
          label1: value1
      securityContext:
        runAsUser: 1000001
        fsGroup: 0
        terminationGracePeriodSeconds: 120
  readinessProbe: 8
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  metricsConfig: 9
```

```

type: jmxPrometheusExporter
valueFrom:
  configMapKeyRef:
    name: cruise-control-metrics
    key: metrics-config.yml
# ...

```

- 1 ブローカーリソースの容量制限。
- 2 Cruise Control の設定 AMQ Streams によって直接管理されないプロパティに限り、標準 Cruise Control 設定の提供が可能です。
- 3 デフォルトの最適化目標 (**default.goals**)、主な最適化 **goals** (goal)、およびハード目標 (**hard.goals**) の設定を含めることができる **最適化目標** の設定。
- 4 CORS が有効で、Cruise Control API への読み取り専用アクセス用に設定 されています。
- 5 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。
- 6 ConfigMap を介して直接 (**inline**) または間接 (**external**) に追加された **Cruise Control ログとログレベル**。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グがあります。ログレベルは INFO、ERROR、WARN、TRACE、DEBUG、FATAL、または OFF に設定できます。
- 7 **テンプレートのカスタマイズ**。ここでは、Pod が追加のセキュリティ属性でスケジューリングされています。
- 8 コンテナを再起動するタイミング (liveness) およびコンテナ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**。
- 9 **Prometheus メトリック** 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ト) に対して設定されます。

2. リソースを作成または更新します。

```
oc apply -f <kafka_configuration_file>
```

3. デプロイメントのステータスを確認します。

```
oc get deployments -n <my_cluster_operator_namespace>
```

デプロイメント名と準備状態が表示されている出力

```

NAME                READY UP-TO-DATE AVAILABLE
my-cluster-cruise-control 1/1    1          1

```

my-cluster は Kafka クラスターの名前です。

READY は、Ready/expected 状態のレプリカ数を表示します。**AVAILABLE** 出力に **1** が表示されれば、デプロイメントは成功しています。

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。指定された設定オプションを使用して、トピックの名前を変更できます。

表7.3 自動作成されたトピック

自動作成されたトピック設定	デフォルトのトピック名	作成元	機能
<code>metric.reporter.topic</code>	<code>strimzi.cruisecontrol.metrics</code>	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。
<code>partition.metric.sample.store.topic</code>	<code>strimzi.cruisecontrol.partitionmetricsamples</code>	Cruise Control	各パーティションの派生されたメトリックを格納します。これは Metric Sample Aggregator によって作成されます。
<code>broker.metrics.sample.store.topic</code>	<code>strimzi.cruisecontrol.modeltrainingsamples</code>	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ック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。



注記

自動作成されたトピックの名前が、すでに Cruise Control が有効になっている Kafka クラスターで変更された場合、古いトピックは削除されないため、手動で削除する必要があります。

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

- [最適化ゴールの概要](#)
- [CruiseControlSpec スキーマ参照](#)

7.6. 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。最適化プロポーザルの結果を使用して Kafka クラスターをリバランスできます。

最適化の提案は、次のいずれかのモードで実行できます。

- **full** (デフォルト)

- **add-brokers**
- **remove-brokers**

使用するモードは、Kafka クラスターですでに実行されているすべてのブローカー間で再調整するかどうかによって異なります。または、Kafka クラスターをスケールアップした後またはスケールダウンする前に再調整したい場合。詳細については、[ブローカーのスケールリングによるモードの再調整](#) を参照してください。

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

1. **KafkaRebalance** リソースを作成し、適切なモードを指定します。

full モード (デフォルト)

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** プロパティを空のままにします。Cruise Control は、デフォルトで **full** モードで Kafka クラスターを再調整します。

デフォルトで完全なリバランスを行う設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec: {}
```

spec.mode プロパティで **full** モードを指定して、完全なリバランスを実行することもできます。

full モードを指定した設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: full
```

add-brokers モード

スケールアップ後に Kafka クラスターを再調整する場合は、**add-brokers** モードを指定します。

このモードでは、既存のレプリカが新しく追加されたブローカーに移動されます。ブローカーをリストとして指定する必要があります。

add-brokers モードを指定した設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: add-brokers
  brokers: [3, 4] ①

```

- ① スケールアップ操作によって追加された、新しく追加されたブローカーのリスト。このプロパティは必須です。

remove-brokers モード

スケールダウンする前に Kafka クラスターを再調整する場合は、**remove-brokers** モードを指定します。

このモードでは、削除されるブローカーからレプリカが移動されます。削除するブローカーをリストとして指定する必要があります。

remove-brokers モードを指定した設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  mode: remove-brokers
  brokers: [3, 4] ①

```

- ① スケールダウン操作によって削除されるブローカーのリスト。このプロパティは必須です。



注記

次の手順と、再調整を承認または停止する手順は、使用している再調整モードに関係なく同じです。

- デフォルトのゴールを使用する代わりに **ユーザー提供の最適化ゴール** を設定するには、**goals** プロパティを追加し、1つ以上のゴールを入力します。
以下の例では、ラックウェアアネス (Rack Awareness) およびレプリカの容量はユーザー提供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

```

- 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

```

- リソースを作成または更新します。

```
oc apply -f <kafka_rebalance_configuration_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。

- 最適化提案のステータスが **ProposalReady** に変わるまで待ちます。

```
oc get kafkarebalance -o wide -w -n <namespace>
```

PendingProposal

PendingProposal ステータスは、最適化プロポーザルの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ることを意味します。

ProposalReady

ProposalReady ステータスは、最適化プロポーザルのレビューおよび承認の準備ができていることを意味します。

ステータスが **ProposalReady** に変わると、最適化プロポーザルを承認する準備が整います。

- 最適化プロポーザルを確認します。
最適化プロポーザルは **KafkaRebalance** カスタムリソースの **Status.Optimization Result** プロパティに含まれます。

```
oc describe kafkarebalance <kafka_rebalance_resource_name>
```

最適化プロポーザルの例

```

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。

CPU 容量が不足している

Kafka クラスターが CPU 使用率の観点から過負荷になっている場合には、**KafkaRebalance** ステータスで CPU 容量が十分でないというエラーが発生する可能性があります。この使用率の値は、**excludedTopics** 設定の影響を受けないことに注意してください。最適化の提案では、除外されたトピックのレプリカは再割り当てされませんが、負荷は使用率の計算で考慮されます。

CPU 使用率エラーの例

```

com.linkedin.kafka.cruisecontrol.exception.OptimizationFailureException:
[CpuCapacityGoal] Insufficient capacity for cpu (Utilization 615.21,
Allowed Capacity 420.00, Threshold: 0.70). Add at least 3 brokers with
the same cpu capacity (100.00) as broker-0. Add at least 3 brokers with
the same cpu capacity (100.00) as broker-0.

```



注記

このエラーは、CPU コアの数ではなく、CPU 容量をパーセンテージで示しています。このため、Kafka カスタムリソースで設定された CPU の数に直接マップされません。これは、**Kafka.spec.kafka.resources.limits.cpu** で設定された CPU のサイクルを持つ、ブローカーごとに単一の **仮想 CPU** を持つようなものです。CPU 使用率と容量の比率は同じであるため、これはリバランスの動作に影響はありません。

次のステップ

[「最適化プロポーザルの承認」](#)

関連情報

- [「最適化プロポーザルの概要」](#)

7.7. 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。
- **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 **strimzi.io/rebalance=refresh** でアノテーションを付けます。

```
oc annotate kafkarebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance=refresh
```

2. 最適化提案のステータスが **ProposalReady** に変わるまで待ちます。

```
oc get kafkarebalance -o wide -w -n <namespace>
```

PendingProposal

PendingProposal ステータスは、最適化プロポーザルの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ることを意味します。

ProposalReady

ProposalReady ステータスは、最適化プロポーザルのレビューおよび承認の準備ができていることを意味します。

ステータスが **ProposalReady** に変わると、最適化プロポーザルを承認する準備が整います。

3. 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 **strimzi.io/rebalance=approve** でアノテーションを付けます。

```
oc annotate kafkarebalance <kafka_rebalance_resource_name>
strimzi.io/rebalance=approve
```

- 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。
- 最適化提案のステータスが **Ready** に変わるまで待ちます。

```
oc get kafkarebalance -o wide -w -n <namespace>
```

Rebalancing

Rebalancing ステータスは、リバランスが進行中であることを意味します。

Ready

Ready ステータスは、リバランスが完了したことを意味します。

NotReady

NotReady ステータスは、エラーが発生したことを意味します – [KafkaRebalance リソースに関する問題の修正](#) を参照してください。

状態が **Ready** に変更されると、リバランスが完了します。

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。

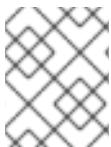
関連情報

- 「[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](#)」

7.8. 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、初期状態よりも低下してい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」](#)

7.9. 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。

クラスター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設定されていません。
- **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8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 インターフェイ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ルライズおよびデシリアル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ルライザーおよびデシリアル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、シリアル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ル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ル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プロジェクトで構築されます。

第9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、トレースが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有用です。

AMQ Streams では、トレース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います。

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)。



注記

OpenTracing のサポートは非推奨となりました。Jaeger クライアントは廃止され、OpenTracing プロジェクトはアーカイブされました。そのため、今後の Kafka バージョンのサポートを保証できません。OpenTelemetry プロジェクトに基づく新しいトレース実装を導入しています。

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イ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 のドキュメント](#) 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。

1. [Kafka Connect のトレーシングを有効にする](#)

- [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ス用にインストルメント化](#)
 - [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は、[Jaeger のドキュメント](#) を参照してください。

9.1. OPENTRACING および JAEGER の概要

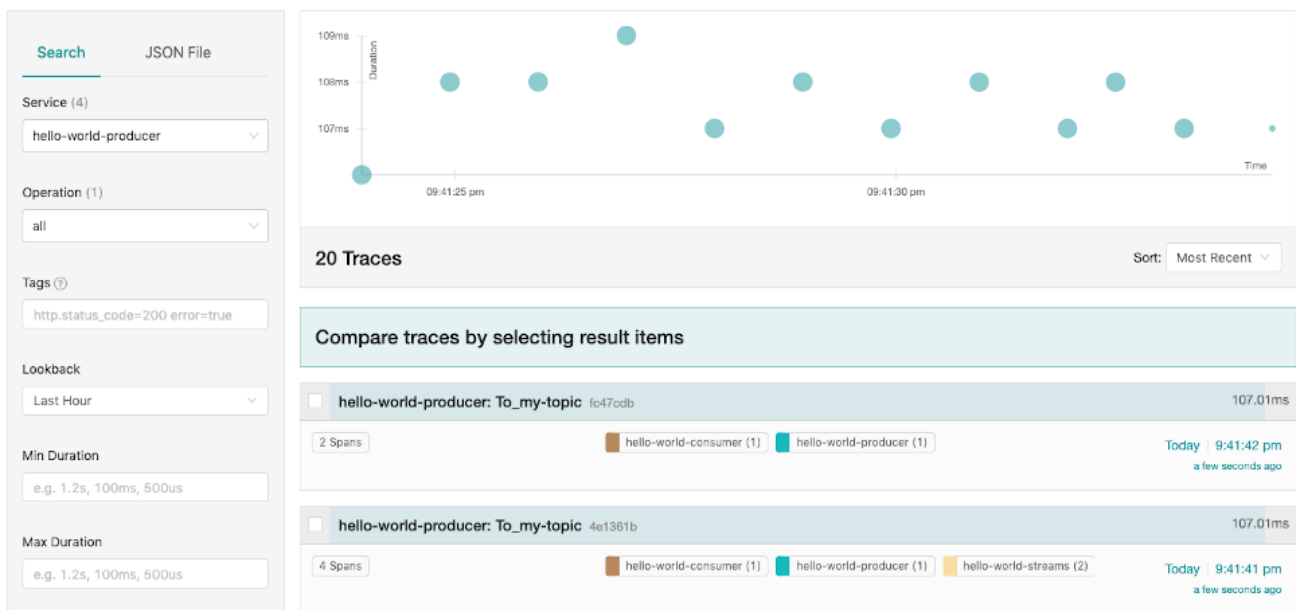
AMQ Streams では OpenTracing および Jaeger プロジェクトが使用されます。

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せん。
- 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** で設定されます。

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イスを使用すると、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](#)

9.2. 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ールメント化します。

9.2.1. 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。

9.2.2. 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	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。
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コ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。

プロパティ	必要性	説明
JAEGER_SAMPLER_TYPE	不要	<p>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。</p> <ul style="list-style-type: none"> ● Constant ● Probabilistic ● Rate Limiting ● Remote (デフォルト) <p>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。</p> <p>Jaeger アーキテクチャーおよびクライアントサンプリング設定パラメーターの概要は、Jaeger のドキュメントを参照してください。</p>
JAEGER_SAMPLER_PARAM	不要	<p>サンプラーのパラメーター (数値)。</p>
JAEGER_SAMPLER_MANAGER_HOST_PORT	不要	<p>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。</p>
JAEGER_TAGS	不要	<p>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コンマ区切りリスト。</p> <p>また、<code>\${envVarName:default}</code> という形式で環境変数を参照することもできます。<code>:default</code> 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。</p>

関連情報

- [「Kafka クライアント用の Jaeger トレーサーの初期化」](#)

9.3. 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。

9.3.1. 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-00006</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);

```

9.3.1.1. 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"

```

9.3.1.2. 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 <code>receive</code> 、プロデューサーの場合は <code>send</code> 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。
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。

BiFunction	説明
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。

9.3.2. 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-00006</version>
</dependency>
```

2. **TracingKafkaClientSupplier** サプライヤーインターフェイ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イスを **KafkaStreams** に提供します。

```
KafkaStreams streams = new KafkaStreams(builder.build(), new StreamsConfig(config),
supplier);
streams.start();
```

9.4. 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。

9.4.1. 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: ❶
    env:
      - name: JAEGER_SERVICE_NAME
        value: my-jaeger-service
      - name: JAEGER_AGENT_HOST
        value: jaeger-agent-name
      - name: JAEGER_AGENT_PORT
        value: "6831"
    tracing: ❷
      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として使用します。
- 2 `spec.tracing.type` プロパティを `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10章 TLS 証明書の管理

AMQ Streams は、Kafka コンポーネントと AMQ Streams コンポーネント間の暗号化通信の TLS をサポートしています。

通信は常に以下のコンポーネント間で暗号化されます。

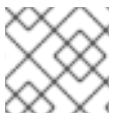
- Kafka と ZooKeeper 間の通信
- Kafka ブローカー間の通信
- ZooKeeper ノード間の通信
- AMQ Streams operator の Kafka ブローカーおよび ZooKeeper ノードとの通信

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。

認証局 (CA) 証明書は、コンポーネントとクライアントの ID 検証にクラスター Operator によって生成されます。クラスター Operator によって生成された CA を使用しない場合は [独自のクラスターおよびクライアント CA 証明書をインストール](#) できます。

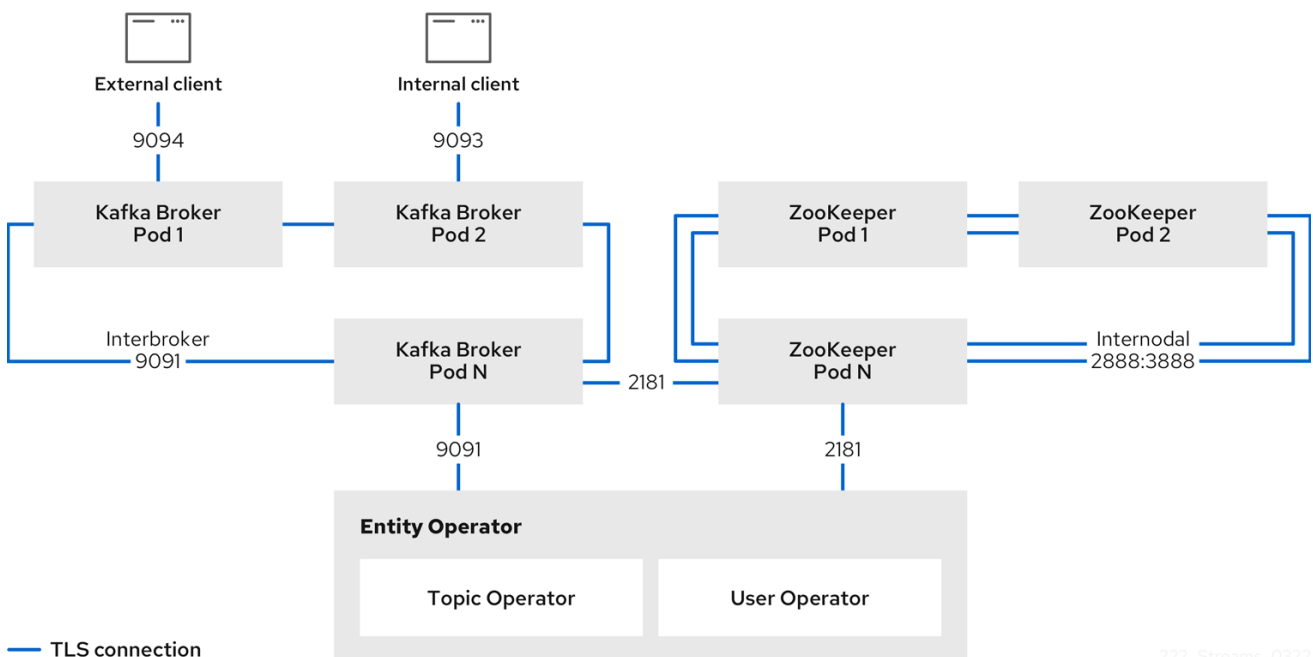
TLS 暗号化が有効になっている TLS リスナーまたは外部リスナーに [Kafka リスナー証明書](#) を指定することもできます。Kafka リスナー証明書を使用して、既存のセキュリティーインフラストラクチャーを組み込みます。



注記

クラスター Operator では、独自に指定した証明書には更新されません。

図10.1 TLS によってセキュリティーが保護された通信のアーキテクチャー例



222_Streams_0322

10.1. 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。

10.1.1. 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。

10.1.2. 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 が稼働中である。
- 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**: クラスター 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**(クラスター CA キー)
- **CLUSTER-NAME-clients-ca**(クラスター 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. シークレットにアノテーションを付けます。

```
oc annotate secret CA-CERTIFICATE-SECRET strimzi.io/ca-cert-generation=CA-CERTIFICATE-GENERATION
```

```
oc annotate secret CA-KEY-SECRET strimzi.io/ca-key-generation=CA-KEY-GENERATION
```


- - **strimzi.io/ca-cert-generation=CA-CERTIFICATE-GENERATION** のアノテーションでは、新しい CA 証明書の生成を定義します。
 - **strimzi.io/ca-key-generation=CA-KEY-GENERATION** のアノテーションは、新しい CA キーの生成を定義します。
 クラスター Operator で自動生成された CA 証明書を置き換える場合は、既存のアノテーションに1つ足した値を使用して、[CA キーの置き換え手順](#)を実行します。クラスター Operator で自動生成された CA 証明書がない場合は、0 を開始点として独自の CA 証明書を増分値で (**strimzi.io/ca-cert-generation=0**) 指定していきます。証明書を更新するときは、値を1つ増やして設定します。
5. 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。
- 以前にインストールした CA 証明書の秘密鍵を置き換えるには、「[独自の CA 証明書で使われる秘密鍵の置き換え](#)」を参照してください。
- 「[独自の Kafka リスナー証明書の指定](#)」

10.2. シークレット

AMQ Streams はシークレットを使用して、Kafka クラスター、クライアント、およびユーザーの秘密鍵と公開鍵の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せん。

クラスターとクライアントのシークレットは常に、公開鍵と、秘密鍵のペアとなっています。

クラスターシークレット

クラスターシークレットには、Kafka ブローカー証明書に署名するための**クラスター CA**が含まれています。接続するクライアントは、証明書を使用して、Kafka クラスターとの TLS 暗号化接続を確立します。証明書はブローカーのアイデンティティを確認します。

クライアントシークレット

クライアントシークレットには、ユーザーが独自のクライアント証明書に署名するための**クライアント CA**が含まれています。これにより、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、証明書を使用してクライアントのアイデンティティを検証します。

ユーザーシークレット

ユーザーシークレットには、秘密鍵と証明書が含まれています。シークレットは、新しいユーザーの作成時にクライアント CA で作成され、署名されます。キーと証明書は、クラスターへのアクセス時にユーザーの認証および承認に使用されます。

10.2.1. PEM および PKCS#12 形式のシークレット

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。クライアントに適した形式を使用してください。PEM 形式の秘密鍵と証明書を使用する場合、ユーザーは Secret からこれらの秘密鍵と証明書を取得し、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。

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。

各シークレットには、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。

すべてのキーのサイズは 2048 ビットで、デフォルトでは最初の生成から 365 日間有効です。有効期間は変更できます。

10.2.2. クラスター Operator で生成されたシークレット

クラスター Operator は、以下の証明書を生成します。これらの証明書は、OpenShift クラスターにシークレットとして保存されます。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。

クラスター CA とクライアント CA には、秘密鍵と公開鍵に別々のシークレットがあります。

<cluster_name>-cluster-ca

クラスター CA の秘密鍵が含まれています。AMQ Streams および Kafka コンポーネントは、秘密鍵を使用してサーバー証明書に署名します。

<cluster_name>-cluster-ca-cert

クラスター CA の公開鍵が含まれています。Kafka クライアントは、公開鍵を使用して、TLS サーバー認証で接続している Kafka ブローカーの ID を確認します。

<cluster_name>-clients-ca

クライアント CA の秘密鍵が含まれています。Kafka クライアントは、秘密鍵を使用して、Kafka ブローカーへの接続時に TLS クライアント認証用の新しいユーザー証明書に署名します。

<cluster_name>-clients-ca-cert

クライアント CA の公開鍵が含まれています。Kafka ブローカーは、TLS クライアント認証が使用されている場合に、公開鍵で Kafka ブローカーにアクセスするクライアントの ID を確認します。

AMQ Streams コンポーネント間の通信のシークレットには、クラスター CA で署名された秘密鍵と公開鍵証明書が含まれています。

<cluster_name>-kafka-brokers

Kafka ブローカーの秘密鍵と公開鍵が含まれています。

<cluster_name>-zookeeper-nodes

ZooKeeper ノードの秘密鍵と公開鍵が含まれています。

<cluster_name>-cluster-operator-certs

クラスター Operator と Kafka または ZooKeeper 間の通信を暗号化するための秘密鍵と公開鍵が含まれています。

<cluster_name>-entity-topic-operator-certs

トピック Operator と Kafka または ZooKeeper 間の通信を暗号化するための秘密鍵と公開鍵が含まれています。

<cluster_name>-entity-user-operator-certs

ユーザー Operator と Kafka または ZooKeeper 間の通信を暗号化するための秘密鍵と公開鍵が含まれています。

<cluster_name>-cruise-control-certs

Cruise Control と Kafka または ZooKeeper の間の通信を暗号化するための秘密鍵と公開鍵が含まれています。

<cluster_name>-kafka-exporter-certs

Kafka Exporter と Kafka または ZooKeeper 間の通信を暗号化するための秘密鍵と公開鍵が含まれています。

**注記**

独自のサーバー証明書および秘密鍵を提供して、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て Kafka ブローカーに接続できます。

10.2.3. クラスター CA シークレット

クラスター CA シークレットは、Kafka クラスターの Cluster Operator によって管理されます。

<cluster_name>-cluster-ca-cert シークレットのみがクライアントに必要です。他のすべてのクラスターシークレットは AMQ Streams コンポーネントによってアクセスされます。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。

**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_name>-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10.1 **<cluster_name>-cluster-ca** シークレットのフィールド

フィールド	説明
ca.key	クラスター CA の現在の秘密鍵。

表10.2 **<cluster_name>-cluster-ca-cert** シークレットのフィールド

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。

表10.3 <cluster_name>-kafka-brokers シークレットのフィールド

フィールド	説明
<cluster_name>-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_name>-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_name>-kafka-<num>.crt	Kafka ブローカー Pod <num> の証明書。<cluster_name>- 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_name>-kafka-<num>.key	Kafka ブローカー Pod <num> の 秘密鍵 。

表10.4 <cluster_name>-zookeeper-nodes シークレットのフィールド

フィールド	説明
<cluster_name>-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_name>-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_name>-zookeeper-<num>.crt	ZooKeeper ノード <num> の証明書。<cluster_name>- 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_name>-zookeeper-<num>.key	ZooKeeper Pod <num> の 秘密鍵 。

表10.5 <cluster_name>-cluster-operator-certs シークレットのフィールド

フィールド	説明
cluster-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。
cluster-operator.password	PKCS #12 アーカイブのファイルを保護するパスワード。

フィールド	説明
cluster-operator.crt	クラスター Operator と Kafka または ZooKeeper との間の TLS 通信の証明書。<cluster_name>-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-operator.key	クラスター Operator と Kafka または ZooKeeper との間の TLS 通信の秘密鍵。

表10.6 <cluster_name>-entity-topic-operator-certs シークレットのフィールド

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	トピック Operator と Kafka または ZooKeeper との間の TLS 通信の証明書。<cluster_name>-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	トピック Operator と Kafka または ZooKeeper との間の TLS 通信の秘密鍵。

表10.7 <cluster_name>-entity-user-operator-certs シークレットのフィールド

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	ユーザー Operator と Kafka または ZooKeeper との間の TLS 通信の証明書。<cluster_name>-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	ユーザー Operator と Kafka または ZooKeeper との間の TLS 通信の秘密鍵。

表10.8 <cluster_name>-cruise-control-certs シークレットのフィールド

フィールド	説明
cruise-control.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
cruise-control.password	PKCS #12 アーカイブのファイルを保護するパスワード。
cruise-control.crt	Cruise Control と Kafka または ZooKeeper との間の TLS 通信の証明書。 <cluster_name>-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。
cruise-control.key	Cruise Control と Kafka または ZooKeeper との間の TLS 通信の秘密鍵。

表10.9 <cluster_name>-kafka-exporter-certs シークレットのフィールド

フィールド	説明
kafka-exporter.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。
kafka-exporter.password	PKCS #12 アーカイブのファイルを保護するパスワード。
kafka-exporter.crt	Kafka Exporter と Kafka または ZooKeeper との間の TLS 通信の証明書。 <cluster_name>-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。
kafka-exporter.key	Kafka Exporter と Kafka または ZooKeeper との間の TLS 通信の秘密鍵。

10.2.4. クライアント CA シークレット

クライアント CA シークレットは、Kafka クラスターの Cluster Operator によって管理されます。

<cluster_name>-clients-ca-cert の証明書は、Kafka ブローカーが信頼する証明書です。

<cluster_name>-clients-ca シークレットは、クライアントアプリケーションの証明書の署名に使用されます。このシークレットは AMQ Streams コンポーネントにアクセスできる必要があります、ユーザー 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 のロールベースアクセス制御を使用して強制できます。

表10.10 <cluster_name>-clients-ca シークレットのフィールド

フィールド	説明
ca.key	クライアント CA の現在の秘密鍵。

表10.11 <cluster_name>-clients-ca-cert シークレットのフィールド

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。
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。

10.2.5. ユーザーのシークレット

ユーザーシークレットは User Operator によって管理されます。

User Operator でユーザーが作成されると、ユーザーの名前を使用してシークレットが生成されます。

表10.12 **user_name** シークレットのフィールド

Secret 名	Secret 内のフィールド	説明
<user_name>	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。

10.2.6. ラベルおよびアノテーションのクラスター CA シークレットへの追加

Kafka カスタムリソースで **clusterCaCert** テンプレートプロパティを設定することで、クラスターオペレータが作成したクラスター 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。

10.2.7. CA シークレットでの `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 シークレットは、**Kafka** カスタムリソースに設定される `ownerReference` プロパティで作成されます。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 を再利用する場合は、**Kafka** 設定で Cluster および Client CA シークレットの `generateSecretOwnerReference` プロパティを `false` に設定して、`ownerReference` を無効にすることができます。`ownerReference` が無効な場合に、対応する **Kafka** カスタムリソースが削除されると、OpenShift では CA シークレットは削除されません。

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
# ...

```

関連情報

- [CertificateAuthority schema reference](#)

10.3. 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。

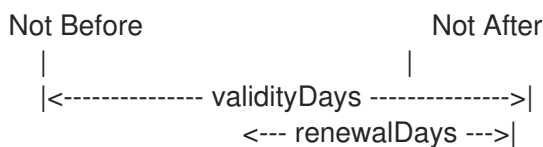
Cluster Operator によって作成される証明書の更新期間を設定できます。

- `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



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

```
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ーである `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。

10.3.1. 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。

10.3.2. 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。

10.3.3. 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、これらの証明書的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 が稼働中である。
- 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10.13 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** の名前です。クラスター 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

- [「シークレット」](#)
- [「ローリング更新のメンテナンス時間枠」](#)
- [「CertificateAuthority スキーマ参照」](#)

10.3.4. 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 が稼働中である。
- 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10.14 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

- [「シークレット」](#)
- [「ローリング更新のメンテナンス時間枠」](#)

10.3.5. 独自の CA 証明書の更新

この手順では、Cluster Operator で証明書を生成する代わりに、使用している CA 証明書を更新する方法を説明します。

対応する CA キーを変更しない場合は、この手順を実行します。それ以外の場合は、[独自の CA 証明書で使用される秘密鍵を置き換える](#) 手順を実行します。

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

- 稼働中の Cluster Operator
- [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。](#)
- クラスターまたはクライアントの PEM 形式による新しい X.509 証明書が必要です。

手順

1. CA 証明書の **Secret** を更新します。
既存のシークレットを編集して新規 CA 証明書を追加し、証明書生成アノテーション値を更新します。

```
oc edit secret <ca_certificate_secret_name>
```

<ca_certificate_secret_name>は**Secret**の名前で、クラスター 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。

以下の例は、**my-cluster** という名前の Kafka クラスターに関連付けられたクラスター CA 証明書のシークレットを示しています。

クラスター CA 証明書のシークレット設定例

```

apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTlBDRVJUSUZJQ0F... ❶
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" ❷
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque

```

❶ 現在の base64 でエンコードされた CA 証明書

❷ 現在の CA 証明書生成アノテーションの値

2. 新規 CA 証明書を base64 にエンコードします。

```
cat <path_to_new_certificate> | base64
```

3. CA 証明書を更新します。
前の手順の base64 でエンコードされた CA 証明書を、**data** の **ca.crt** プロパティの値としてコピーします。
4. CA 証明書生成アノテーションの値を増やします。

strimzi.io/ca-cert-generation アノテーションの値を1つ増分して更新します。たとえば、**strimzi.io/ca-cert-generation=0** を **strimzi.io/ca-cert-generation=1** に変更します。**Secret** にアノテーションがない場合、値は **0** として扱われるため、**1** を指定してアノテーションを追加します。

AMQ Streams が証明書を生成すると、証明書生成アノテーションは Cluster Operator によって自動的に増分されます。独自の CA 証明書を手動で更新するには、値を1増やしたアノテーションを設定します。Cluster Operator が Pod をロールアウトし、証明書を更新できるように、アノテーションには現在のシークレットよりも高い値を指定する必要があります。**strimzi.io/ca-cert-generation** は、各 CA 証明書の更新で値を1増やす必要があります。

5. 新しい CA 証明書と証明書生成のアノテーション値でシークレットを保存します。

新しい CA 証明書で更新されるシークレット設定の例

```

apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F... ❶
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" ❷
  labels:
    strimzi.io/cluster: my-cluster

```

```

strimzi.io/kind: Kafka
name: my-cluster-cluster-ca-cert
#...
type: Opaque

```

- ① 新しい base64 でエンコードされた CA 証明書
- ② 新しい CA 証明書生成アノテーションの値

次の調整時に、Cluster Operator は ZooKeeper、Kafka、およびその他のコンポーネントのローリング更新を実行して、新しい CA 証明書を信頼します。

メンテナンス時間枠が設定されている場合には、Cluster Operator は次のメンテナンス時間枠内で最初の調整時に Pod をローリングします。

10.3.6. 独自の CA 証明書で使用される秘密鍵の置き換え

この手順では、Cluster Operator で証明書と鍵を生成する代わりに、使用している CA 証明書と秘密鍵を更新する方法を説明します。

対応する CA キーも変更する場合は、この手順のステップを実行します。それ以外の場合は、**独自の CA 証明書を更新** する手順を実行します。

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。

以下の手順を実行する前に、新規 CA 証明書の CN(コモンネーム) が現在の CA 証明書とは異なることを確認してください。たとえば、Cluster Operator が証明書を更新する場合には、バージョンの識別に `v<version_number>` 接尾辞を追加します。更新ごとに別の接尾辞を追加して、独自の CA 証明書で同じ作業を行います。別のキーを使用して新しい CA 証明書を生成して、シークレットに保存されている現在の CA 証明書を保持します。

前提条件

- 稼働中の Cluster Operator
- **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。**
- クラスタまたはクライアントの PEM 形式による新しい X.509 証明書と鍵が必要です。

手順

1. **Kafka** カスタムリソースの調整を一時停止します。
 - a. OpenShift でカスタムリソースにアノテーションを付け、**pause-reconciliation** アノテーションを **true** に設定します。

```
oc annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation="true"
```

たとえば、**my-cluster** という名前の **Kafka** カスタムリソースの場合:

```
oc annotate Kafka my-cluster strimzi.io/pause-reconciliation="true"
```

- b. 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 Kafka <name_of_custom_resource>
```

type 条件は、**lastTransitionTime** で **ReconciliationPaused** に変わります。

2. CA 証明書の **Secret** を更新します。

- a. 既存のシークレットを編集して新規 CA 証明書を追加し、証明書生成アノテーション値を更新します。

```
oc edit secret <ca_certificate_secret_name>
```

<ca_certificate_secret_name> は **Secret** の名前です。クラスター 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。

以下の例は、**my-cluster** という名前の Kafka クラスターに関連付けられたクラスター CA 証明書のシークレットを示しています。

クラスター CA 証明書のシークレット設定例

```
apiVersion: v1
kind: Secret
data:
  ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ❶
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" ❷
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

❶ 現在の base64 でエンコードされた CA 証明書

❷ 現在の CA 証明書生成アノテーションの値

- b. 保持する現在の CA 証明書の名前を変更します。
data の配下にある現在の **ca.crt** プロパティ名を **ca-<date>.crt** に変更します。<date> は、証明書の有効期限を **YEAR-MONTH-DAYTHOUR-MINUTE-SECONDZ** の形式で指定します。例: **ca-2022-01-26T17-32-00Z.crt**:現在の CA 証明書を保持するため、プロパティの値を残します。
- c. 新規 CA 証明書を base64 にエンコードします。

```
cat <path_to_new_certificate> | base64
```


- d. CA 証明書を更新します。
data の下に新しい **ca.crt** プロパティを作成し、上の手順から base64 でエンコードされた CA 証明書を **ca.crt** プロパティの値としてコピーします。
- e. CA 証明書生成アノテーションの値を増やします。
strimzi.io/ca-cert-generation アノテーションの値を1つ増分して更新します。たとえば、**strimzi.io/ca-cert-generation=0** を **strimzi.io/ca-cert-generation=1** に変更します。**Secret** にアノテーションがない場合、値は **0** として扱われるため、**1** を指定してアノテーションを追加します。

AMQ Streams が証明書を生成すると、証明書生成アノテーションは Cluster Operator によって自動的に増分されます。独自の CA 証明書を手動で更新するには、値を1増やしたアノテーションを設定します。Cluster Operator が Pod をロールアウトし、証明書を更新できるように、アノテーションには現在のシークレットよりも高い値を指定する必要があります。**strimzi.io/ca-cert-generation** は、各 CA 証明書の更新で値を1増やす必要があります。

- f. 新しい CA 証明書と証明書生成のアノテーション値でシークレットを保存します。

新しい CA 証明書で更新されるシークレット設定の例

```
apiVersion: v1
kind: Secret
data:
  ca.crt: GCa6LS3RTHeKFiFDGBOUDYFAZ0F... ❶
  ca-2022-01-26T17-32-00Z.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... ❷
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" ❸
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

- ❶ 新しい base64 でエンコードされた CA 証明書
- ❷ 古い base64 でエンコードされた CA 証明書
- ❸ 新しい CA 証明書生成アノテーションの値

3. 新しい CA 証明書の署名に使用する CA キーの **Secret** を更新します。
- a. 既存のシークレットを編集して新規 CA キーを追加し、キー生成アノテーション値を更新します。

```
oc edit secret <ca_key_name>
```

<ca_key_name> 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。

以下の例は、**my-cluster** という名前の Kafka クラスターに関連付けられたクラスター CA キーのシークレットを示しています。

クラスター CA キーのシークレット設定例

```
apiVersion: v1
kind: Secret
data:
  ca.key: SA1cKF1GFDzOliPOIUQBHDNFGDFS... ❶
metadata:
  annotations:
    strimzi.io/ca-key-generation: "0" ❷
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque
```

- ❶ 現在の base64 でエンコードされた CA キー
- ❷ 現在の CA キー生成アノテーションの値

b. CA キーを base64 にエンコードします。

```
cat <path_to_new_key> | base64
```

- c. CA キーを更新します。
前の手順の base64 でエンコードされた CA キーを **data** にある **ca.key** プロパティの値としてコピーします。
- d. CA キー生成アノテーションの値を増やします。
strimzi.io/ca-key-generation アノテーションの値を1つ増分して更新します。たとえば、**strimzi.io/ca-key-generation=0** を **strimzi.io/ca-key-generation=1** に変更します。**Secret** にアノテーションがない場合は **0** として扱われるため、**1** の値を指定してアノテーションを追加します。

AMQ Streams が証明書を生成すると、キー生成アノテーションは Cluster Operator によって自動的に増分されます。独自の CA 証明書を新しい CA キーと共に手動で更新するには、増分の値のアノテーションを設定します。Cluster Operator が Pod をロールアウトし、証明書およびキーを更新できるように、アノテーションには現在のシークレットよりも高い値が必要です。**strimzi.io/ca-key-generation** は、CA 証明書の更新ごとにインクリメントする必要があります。

4. 新しい CA キーおよびキー生成アノテーション値でシークレットを保存します。

新規 CA キーで更新されるシークレット設定の例

```
apiVersion: v1
kind: Secret
data:
  ca.key: AB0cKF1GFDzOliPOIUQWERZJQ0F... ❶
metadata:
  annotations:
```

```

strimzi.io/ca-key-generation: "1" 2
labels:
  strimzi.io/cluster: my-cluster
  strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca
  #...
type: Opaque

```

- 1 新規の base64 でエンコードされた CA キー
- 2 新しい CA キー生成アノテーションの値

5. 一時停止から再開します。

Kafka カスタムリソースの調整を再開するには、**pause-reconciliation** アノテーションを **false** に設定します。

```

oc annotate --overwrite Kafka <name_of_custom_resource> strimzi.io/pause-
reconciliation="false"

```

pause-reconciliation アノテーションを削除してもこれを実行できます。

```

oc annotate Kafka <name_of_custom_resource> strimzi.io/pause-reconciliation-

```

次の調整時に、Cluster Operator は ZooKeeper、Kafka、およびその他のコンポーネントのローリング更新を実行して、新しい CA 証明書を信頼します。ローリング更新が完了すると、Cluster Operator は新しい CA キーで署名された新しいサーバー証明書を生成するために新しい証明書を起動します。

メンテナンス時間枠が設定されている場合には、Cluster Operator は次のメンテナンス時間枠内で最初の調整時に Pod をローリングします。

10.4. TLS 接続

10.4.1. 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。

10.4.2. 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。

10.4.3. 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。

10.4.4. 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。

10.4.5. 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` プロパティを使用して設定されます。

10.5. 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/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。

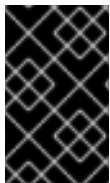
10.6. クラスター CA を信頼する外部クライアントの設定

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 クラスターの **CLUSTER-NAME-cluster-ca-cert** Secret からクラスター 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で設定します。
 - セキュリティ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。

10.7. KAFKA リスナー証明書

TLS 暗号化が有効になっているリスナーに、独自のサーバー証明書および秘密鍵を提供できます。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。

Kafka リスナー証明書を提供すると、組織のプライベート CA やパブリック CA などの既存のセキュリティインフラストラクチャーを利用できます。Kafka クライアントは、リスナー証明書の署名に使用された CA を信頼する必要があります。

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。

10.7.1. 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
- 稼働中の Cluster Operator。
- 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
# ...
```

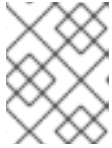
TLS リスナーの設定例

```
# ...
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更新を開始し、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、ローリング更新が開始されます。

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](#)

10.7.2. 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。

10.7.2.1. 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
```

10.7.2.2. 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10.15 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。
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11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。

11.1. 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。

11.1.1. 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11.1 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
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

11.1.1.1. 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リスト表示されます。

```
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。

11.1.1.2. 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。**-o json** を使用すると JSON として返されます。

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[?(@.name=="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。

この場合、コマンドは **tls** という名前のリスナーの **bootstrapServers** 値を検索します。

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

名前の条件を変更することで、他の Kafka リスナーのアドレスも取得できます。

jsonpath を使用して、カスタムリソースから他のプロパティまたはプロパティのグループを抽出できます。

11.1.2. 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** プロパティがあります。

表11.2 カスタムリソースの status プロパティ

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。
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。
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 クラスタの Kafka トピック
KafkaUser	「 KafkaUserStatus スキーマ参照」	Kafka クラスタの 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** プロパティの **Current state** (現在の状態)。
- **status.observedGeneration** プロパティの **Last observed generation** (最後に確認された生成)。

status プロパティによって、リソース固有の情報も提供されます。以下に例を示します。

- **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。定期的には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 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リスト表示されても、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。

11.1.3. 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
- Cluster Operator が稼働中である。

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。

11.2. 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](#)

11.3. AMQ STREAMS DRAIN CLEANER での POD の退避

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に、AMQ Streams 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。OpenShift ではなく AMQ Streams Drain Cleaner がエビクションを処理するため、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。その後、OpenShift は Pod を自動的に削除できなくなります。

AMQ Streams Drain Cleaner をデプロイすることで、Cluster Operator を使用して OpenShift ではなく Kafka Pod を移動できます。Cluster Operator は、トピックの複製の数が最低数未満にならないようにします。Kafka はエビクションプロセス中も稼働状態を維持できます。Cluster Operator は、OpenShift ワーカーノードが連続してドレイン (解放) されるため、トピックが同期するのを待ちます。

受付 Webhook は、AMQ Streams Drain Cleaner に Kubernetes API への Pod エビクション要求を通知します。AMQ Streams Drain Cleaner は次に、ドレイン (解放) する Pod にローリング更新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 のローリング更新を実行するように指示します。



注記

AMQ Streams Drain Cleaner を使用していない場合は、[Pod アノテーションを追加して手動でローリング更新を実行](#) できます。

Webhook の設定

AMQ Streams Drain Cleaner デプロイメントファイルには、**ValidatingWebhookConfiguration** リソースファイルが含まれます。リソースでは、Kubernetes API で Webhook を登録する設定が可能です。

この設定は、Pod のエビクション要求の場合に使用する 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** 操作だけ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ている場合、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、認証を必要とする、セキュアな TLS 接続を使用します。**caBundle** プロパティは、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==
# ...
```

11.3.1. 前提条件

AMQ Streams Drain Cleaner 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、[AMQ Streams ソフトウェアダウンロードページ](#) から入手できます。

11.3.2. AMQ Streams Drain Cleaner のデプロイ

Cluster Operator および Kafka クラスターが実行中の OpenShift クラスターに、AMQ Streams Drain Cleaner をデプロイします。

前提条件

- [AMQ Streams Drain Cleaner デプロイメントファイルをダウンロード](#) しておく。
- 更新する OpenShift ワーカー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がある。
- トピックを複製して高可用性に対応する少なくとも 3 つのレプリケーション係数と、レプリケーション係数よりも 1 つ少ない In-Sync レプリカの最小数を指定するトピック設定。

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 を含めない場合は、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" ❶
          # ...

```

- ❶ このオプションを削除して、ZooKeeper を AMQ Streams Drain Cleaner 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 を **0** に設定します。

Pod の Disruption Budget 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、自発的な中断が発生した場合に OpenShift が Pod を自動的にエビクトし、AMQ Streams Drain Cleaner および AMQ Streams Cluster Operator を残して、別のワーカーノードで OpenShift によってスケジュールされる Pod をロールします。

AMQ Streams Drain Cleaner 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
```

11.3.3. AMQ Streams Drain Cleaner の使用

AMQ Streams Drain Cleaner を Cluster Operator と組み合わせて使用し、ドレイン (解放) されているノードから 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 にローリング更新 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更新を実行します。

前提条件

- [AMQ Streams Drain Cleaner がデプロイ](#) 済みである。

手順

1. Kafka ブローカーまたは ZooKeeper Pod をホストする、特定の 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 ログのエビクションイベントを確認し、Pod が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。

Pod のアノテーションを示す AMQ Streams Drain Cleaner ログ

```
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
```

- Cluster Operator ログで調整イベントを確認し、ローリング更新を確認します。

Cluster Operator log shows rolling updates(クラスター Operator ログによるローリング更新の表示)

```
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
```

11.4. KAFKA および ZOOKEEPER クラスターの手動によるローリング更新の開始

AMQ Streams は、Cluster Operator 経由で Kafka および ZooKeeper クラスターのローリング更新を手動でトリガーするために、リソースでアノテーションの使用をサポートします。ローリング更新により、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 や 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 経由でローリング更新を実行すると、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。

11.4.1. 前提条件

手動でローリング更新を実行するには、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](#)を参照すること。

- [Cluster Operator](#)
- [Kafka クラスター](#)

11.4.2. Pod 管理のアノテーションを使用したローリング更新の実行

この手順では、Kafka クラスターまたは ZooKeeper クラスターのローリング更新をトリガーする方法を説明します。

更新をトリガーするには、クラスターで実行されている Pod の管理に使用するリソースにアノテーションを追加します。**StatefulSet** または **StrimziPodSet** リソースにアノテーションを付けます ([UseStrimziPodSets 機能ゲート](#) を有効にした場合)。

手順

1. 手動で更新する Kafka または ZooKeeper Pod を制御するリソース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** の場合には、対応する名前は **my-cluster-kafka** および **my-cluster-zookeeper** になります。
2. **oc annotate** を使用して、OpenShift で適切なリソースにアノテーションを付けます。

StatefulSet のアノテーション

```
oc annotate statefulset <cluster_name>-kafka strimzi.io/manual-rolling-update=true
oc annotate statefulset <cluster_name>-zookeeper strimzi.io/manual-rolling-update=true
```

StrimziPodSet のアノテーション

```
oc annotate strimzipodset <cluster_name>-kafka strimzi.io/manual-rolling-update=true
oc annotate strimzipodset <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リソース内のすべての Pod でローリング更新がトリガーされます。すべての Pod のローリング更新が完了すると、アノテーションはリソースから削除されます。

11.4.3. Pod アノテーションを使用したローリング更新の実行

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更新を手動でトリガーする方法を説明します。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更新は同じ調整実行内で実行されます。

前提条件

使用され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更新を実行できます。ただし、更新中に Kafka を稼働し続けるには、以下が必要になります。

- 更新するノードで実行されている高可用性 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。
少なくとも 3 つのレプリケーション係数と、レプリケーション係数よりも 1 つ少ない In-Sync レプリカの最小数を指定するトピック設定。

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、レプリカの総数マイナス1で終わります。
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** のローリング更新がトリガーされます。Pod のローリング更新が完了すると、アノテーションは **Pod** から削除されます。

11.5. 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

```

11.5.1. 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。

11.6. 永続ボリュームからのクラスタの復元

Kafka クラスタは、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。

11.6.1. 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 にマウントされません。

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。

11.6.2. 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。

11.6.3. 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。

作業を開始する前に

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。

- [永続ボリューム要求の命名](#)
- [JBOD および 永続ボリューム要求](#)



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。以下に例を示します。

```

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** プロパティを削除します。以下に例を示します。

```

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

```

- Cluster Operator をデプロイします。

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

- 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 クラスタをデプロイします。
以下に例を示します。

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

オプション 2: クラスタ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-32-rhel8:2.2.2 --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 クラスタを **topicOperator** プロパティで再デプロイして TopicOperator を有効にし、**KafkaTopic** リソースを再作成します。
以下に例を示します。

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

- ❶ ここで示すデフォルト設定には、追加のプロパティはありません。「[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
```

11.7. 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 1TB ディスクを使用し、クォータが 1.1TB の場合は、1つのディスクがいっぱいになり、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: 40000000000 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1MBps 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1MBps 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_configuration_file>
```

関連情報

- [ユーザークォータの設定](#)

11.8. よくある質問

11.8.1. Cluster Operator に関する質問

11.8.1.1. 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？](#)

11.8.1.2. 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。

11.8.1.3. 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 管理者の指定](#)を参照してください。

11.8.1.4. 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、ローリング更新、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。

11.8.1.5. 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12章 カスタムリソース API のリファレンス

12.1. 共通の設定プロパティ

共通設定プロパティは複数のリソースに適用されます。

12.1.1. replicas

replicas プロパティ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、可用性とスケラビリティを向上します。



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。

12.1.2. bootstrapServers

bootstrapServers プロパティ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。

12.1.3. ssl

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" ❶
    ssl.enabled.protocols: "TLSv1.2" ❷
    ssl.protocol: "TLSv1.2" ❸
    ssl.endpoint.identification.algorithm: HTTPS ❹
# ...
```

- ❶ 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。
- ❷ SSI プロトコル **TLSv1.2** は有効になります。
- ❸ **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。
- ❹ 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。

12.1.4. 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リストを提供します。

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。

12.1.5. resources

AMQ Streams コンテナのリソースを制御するために、リソース **要求** および **制限** を設定します。メモリーおよび **cpu** リソースの要求および制限を指定できます。要求には、Kafka の安定したパフォーマンスを確保できる十分な値が必要です。

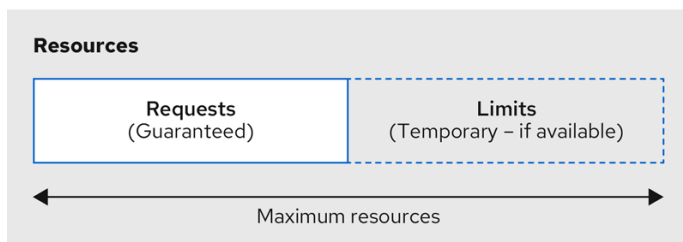
実稼働環境でリソースを設定する方法は、さまざまな要因によって異なります。たとえば、アプリケーションは OpenShift クラスタでリソースを共有する可能性があります。

Kafka では、デプロイメントの以下の要素が、必要なリソースに影響を与える可能性があります。

- メッセージのスループットとサイズ
- メッセージを処理するネットワークスレッドの数
- プロデューサーおよびコンシューマーの数
- トピックおよびパーティションの数

リソース要求に指定の値は予約され、常にコンテナで利用可能になります。リソース制限によって、指定のコンテナが消費可能な最大リソースが指定されます。要求数から制限数の間は予約されず、常に利用できるとは限りません。コンテナは、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。リソースの制限は一時的で、再割り当てが可能です。

リソース要求および制限



212_Streams_0322

要求なしに制限を設定する場合や、その逆の場合、OpenShift は両方に同じ値を使用します。OpenShift は制限を超えない限りコンテナを強制終了しないので、リソースに対して、要求と制限を同じ数に設定すると、QoS (Quality of Service) が保証されます。

サポート対象のリソース 1 つまたは複数に対して、リソース要求および制限を設定できます。

リソース設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    #...
    resources:
      requests:
        memory: 64Gi
        cpu: "8"
      limits:
        memory: 64Gi
```

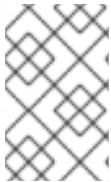
```

cpu: "12"
entityOperator:
  #...
topicOperator:
  #...
resources:
  requests:
    memory: 512Mi
    cpu: "1"
  limits:
    memory: 512Mi
    cpu: "1"

```

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。

リソース要求が OpenShift クラスタ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。



注記

AMQ Streams では、OpenShift 構文を使用して **メモリー** および **cpu** リソースを指定します。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。

メモリーリソース

メモリーリソースを設定する場合は、コンポーネントの合計要件を考慮してください。Kafka は JVM 内で実行され、オペレーティングシステムページキャッシュを使用してディスクに書き込む前にメッセージデータを保存します。Kafka のメモリー要求は、JVM ヒープおよびページキャッシュに適合する必要があります。[jvmOptions プロパティを設定](#) すると、最小および最大ヒープサイズを制御できます。

他のコンポーネントはページキャッシュに依存しません。メモリーリソースは、ヒープサイズを制御する **jvmOptions** を指定せずに、設定できます。

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。仕様では、以下の接尾辞を使用します。

- **M** (メガバイト)
- **G** (ギガバイト)
- **Mi** (メビバイト)
- **Gi** (ギビバイト)

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。

CPU リソース

常に信頼できるパフォーマンスを発揮させるには、CPU 要求を十分に指定する必要があります。CPU の要求および制限は、**コア** または **ミリ cpu/ミリコア** として指定します。CPU コアは、**整数 (5 CPU コア)** または **小数 (2.5 CPU コア)** で指定します。1000 ミリコアは 1 CPU コアと同じです。

CPU の単位の例

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

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。

12.1.6. image

image プロパティを使用して、コンポーネントによって使用されるコンテナイメージを設定します。

コンテナイメージのオーバーライドは、別のコンテナ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レジス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** プロパティを使用すると、コンポーネントに使用するコンテナ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** プロパティの設定

他のカスタムリソースの **image** プロパティでは、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イメージ。
 2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.0** コンテナイメージ。
- 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イメージ。
 2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.0** コンテナイメージ。
- 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イメージ。
 2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.0** コンテナイメージ。

コンテナイメージ設定の例

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


12.1.7. livenessProbe および readinessProbe healthcheck

livenessProbe および **readinessProbe** プロパティを使用して、AMQ Streams でサポートされる healthcheck プロブを設定します。

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。

12.1.8. metricsConfig

metricsConfig プロパティを使用して、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 メトリックを Prometheus メトリックに変換するために、Prometheus JMX エクスポートを使用した Prometheus メトリックがサポートされます。

追加設定なしで Prometheus メトリック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、すべてのメトリックが公開されます。

Kafka のメトリック設定が含まれる 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 のメトリック設定例

```

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ックは、9404 番ポートで公開されます。

metricsConfig (または非推奨の **metrics**) プロパティがリソースに定義されていない場合、Prometheus メトリックは無効になります。

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 へのメトリックの導入](#) を参照してください。

12.1.9. 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` プロパティを設定します。

- `Kafka.spec.kafka`
- `Kafka.spec.zookeeper`
- `Kafka.spec.entityOperator.userOperator`
- `Kafka.spec.entityOperator.topicOperator`
- `Kafka.spec.cruiseControl`
- `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

gcLoggingEnabled

ガベージコレクターのログを有効にします



注記

-Xmx や **-Xms** などの JVM 設定で使用する単位は、対応するイメージの JDK **java** バイナリーで使用する単位と同じです。そのため、**1g** または **1G** は 1,073,741,824 バイトを意味し、**Gi** は接尾辞として有効な単位ではありません。これは、**1G** が 1,000,000,000 バイトを意味し、**1Gi** が 1,073,741,824 バイトを意味する OpenShift 規則に従う、[メモリー要求および制限](#) に使用される単位とは異なります。

-Xms および -Xmx オプション

コンテナのメモリー要求および制限値を設定するだけでなく、**-Xms** および **-Xmx** JVM オプションを使用して、JVM に特定のヒープサイズを設定できます。**-Xms** オプションを使用して初期ヒープサイズを設定し、**-Xmx** オプションを使用して最大ヒープサイズを設定します。

JVM に割り当てられたメモリーをより詳細に制御するには、ヒープサイズを指定します。ヒープサイズは、コンテナの [メモリー制限 \(および要求\)](#) を最大限に活用し、それを超えないようにする必要があります。ヒープサイズとその他のメモリー要件は、指定されたメモリー制限内に収まる必要があります。

す。設定でヒープサイズを指定せずに、メモリーリソースの制限 (および要求) を設定する場合、Cluster Operator はデフォルトのヒープサイズを自動的に適用します。Cluster Operator は、メモリーリソース設定の割合に基づいて、デフォルトの最大および最小ヒープ値を設定します。

次の表に、デフォルトのヒープ値を示します。

表12.1 コンポーネントのデフォルトのヒープ設定

コンポーネント	ヒープに割り当てられた使用可能なメモリーの割合	上限
Kafka	50%	5 GB
ZooKeeper	75%	2 GB
Kafka Connect	75%	なし
MirrorMaker 2.0	75%	なし
MirrorMaker	75%	なし
Cruise Control	75%	なし
Kafka Bridge	50%	31 Gi

メモリー制限 (および要求) が指定されていない場合、JVM の最小ヒープサイズは **128M** に設定されます。JVM の最大ヒープサイズは、必要に応じてメモリーを増やすことができるように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。

適切なメモリー要求を設定すると、次のことを防ぐことができます。

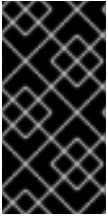
- ノードで実行されている他の Pod からのメモリーに圧力がかかる場合、OpenShift はコンテナを強制終了します。
- OpenShift がメモリー不足のノードにコンテナをスケジューリングする。-Xms が -Xmx に設定されている場合には、コンテナはすぐにクラッシュし、存在しない場合、コンテナは後でクラッシュします。

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。JVM メモリー使用量の合計は、最大ヒープサイズを超える可能性があります。

-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など、多数のディスク I/O を実行するコンテナには、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が必要です。このようなコンテナの場合、要求されるメモリは、JVM が使用するメモリよりも大幅に大きくする必要があります。

-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
```

jvmOptions の詳細は、[JvmOptions スキーマ参照](#) を参照してください。

12.1.10. ガベー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
# ...
```

12.2. スキーマプロパティ

12.2.1. Kafka スキーマ参照

プロパティ	説明
spec	Kafka および ZooKeeper クラスタ、Topic Operator の仕様。
KafkaSpec	
status	Kafka および ZooKeeper クラスタ、Topic Operator のステータス。
KafkaStatus	

12.2.2. KafkaSpec スキーマ参照

[Kafka](#) で使用

プロパティ	説明
kafka	Kafka クラスタの設定。
KafkaClusterSpec	
zookeeper	ZooKeeper クラスタの設定。
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	説明
cruiseControl	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。
CruiseControlSpec	
kafkaExporter	Kafka Exporter の設定。Kafka Exporter は追加のメトリックを提供できます (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)。
KafkaExporterSpec	
maintenanceTimeWindows	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。
string array	

12.2.3. KafkaClusterSpec スキーマ参照

KafkaSpec で使用

KafkaClusterSpec スキーマプロパティの完全リスト

Kafka クラスタを設定します。

12.2.3.1. listeners

listeners プロパティを使用して、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:
    # ...
```

12.2.3.2. config

config プロパティを使用して、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
    # ...
```

12.2.3.3. brokerRackInitImage

ラックウェアネス (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。このコンテナに使用されるコンテナイメージは、 **brokerRackInitImage** プロパティ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イメージ。
2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.0.0** コンテナイメージ。

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

コンテナイメージのオーバーライドは、別のコンテナ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。

12.2.3.4. logging

Kafka には独自の設定可能なロガーがあります。

- `log4j.logger.org.l0ltec.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、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更新がトリガーされます。

ガベージコレクター (GC)

ガベージコレクターのロギングは `jvmOptions` プロパティを使用して有効 (または無効) にすることもできます。

12.2.3.5. KafkaClusterSpec スキーマプロパティ

プロパティ	説明
version	Kafka ブローカーのバージョン。デフォルトは 3.2.3 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	クラスター内の Pod 数。
integer	
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	
config	次の接頭辞のある Kafka ブローカーの config プロパティは設定できません: broker., listener., host.name, port, inter.broker.listener.name, sasl., ssl., security., password., log.dir, zookeeper.connect, zookeeper.set.acl, zookeeper.ssl, zookeeper.clientCnxnSocket, authorizer., super.user, cruise.control.metrics.topic, cruise.control.metrics.reporter.bootstrap.servers,nod e.id, process.roles, controller. (次の例を除く: zookeeper.connection.timeout.ms, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols,cruise.control.metrics.topic.nu m.partitions, cruise.control.metrics.topic.replication.factor, cruise.control.metrics.topic.retention.ms,cruise.contr ol.metrics.topic.auto.create.retries, cruise.control.metrics.topic.auto.create.timeout.ms,cr uise.control.metrics.topic.min.insync.replicas,control ler.quorum.election.backoff.max.ms, controller.quorum.election.timeout.ms, controller.quorum.fetch.timeout.ms).
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	

プロパティ	説明
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 コンテナのイメージ。
string	
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	メトリックの設定。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	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	説明
template	Kafka クラスターリソースのテンプレート。テンプレートを使用すると、ユーザーは StatefulSet 、 Pod 、および Service の生成方法を指定できます。
KafkaClusterTemplate	

12.2.4. Generic KafkaListener スキーマ参照

KafkaClusterSpec で使用

GenericKafkaListener スキーマプロパティの完全リスト

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

```

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
#...
```

12.2.4.1. listeners

Kafka リソースの **listeners** プロパティを使用して **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、小文字と数字で設定されます。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。

12.2.4.2. 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 Pod 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

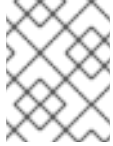
```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external2
        port: 9095
        type: ingress
        tls: true
        authentication:
          type: tls
        configuration:
          bootstrap:
            host: bootstrap.myingress.com
          brokers:
            - 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 Streams 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。

12.2.4.3. 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[?(@.name=="<listener_name>")].bootstrapServers}'{"\n"}
```



注記

ブローカー間通信 (9090 および 9091) およびメトリック (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。

12.2.4.4. tls

TLS プロパティが必要です。

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。

12.2.4.5. 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**)
- Custom(**カスタム**)

12.2.4.6. 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。

```
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。

12.2.4.7. GenericKafkaListener スキーマプロパティ

プロパティ	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。
string	
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。 <ul style="list-style-type: none"> ● internal タイプは、OpenShift クラスター内でのみ 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用して 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 プロパティの値によって異なり、[tls、scram-sha-512、oauth、custom] のいずれかでなければなりません。
KafkaListenerAuthenticationTls 、 KafkaListenerAuthenticationScramSha512 、 KafkaListenerAuthenticationOAuth 、 KafkaListenerAuthenticationCustom	
configuration	追加のリスナー設定。
GenericKafkaListenerConfiguration	

プロパティ	説明
networkPolicyPeers	このリスナーに接続できるピアのリスト。このリストのピアは、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、存在しない場合、このリスナーのすべての接続が許可されます。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、リスナーはこのリストの少なくとも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 アレイ	

12.2.5. KafkaListenerAuthenticationTls スキーマ参照

[GenericKafkaListener](#) で使用

type プロパティは、[KafkaListenerAuthenticationTls](#) タイプと、[KafkaListenerAuthenticationScramSha512](#)、[KafkaListenerAuthenticationOAuth](#)、[KafkaListenerAuthenticationCustom](#) とを区別して使用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	説明
type	tls でなければなりません。
string	

12.2.6. KafkaListenerAuthenticationScramSha512 スキーマ参照

[GenericKafkaListener](#) で使用

type プロパティは、[KafkaListenerAuthenticationScramSha512](#) タイプと、[KafkaListenerAuthenticationTls](#)、[KafkaListenerAuthenticationOAuth](#)、[KafkaListenerAuthenticationCustom](#) とを区別して使用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	説明
type	scram-sha-512 でなければなりません。
string	

12.2.7. KafkaListenerAuthenticationOAuth スキーマ参照

[GenericKafkaListener](#) で使用

type プロパティは、[KafkaListenerAuthenticationOAuth](#) タイプ

と、[KafkaListenerAuthenticationTls](#)、[KafkaListenerAuthenticationScramSha512](#)、[KafkaListenerAuthenticationCustom](#) とを区別して使用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	説明
accessTokensJwt	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。認可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。
boolean	
checkAccessTokenType	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。認可サーバーの 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。
boolean	
checkAudience	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 <code>clientId</code> プロパティで設定する必要があります。Kafka ブローカーは、 <code>aud</code> (オーディエンス)クレームに <code>clientId</code> がいないトークンを拒否します。デフォルト値は false です。
boolean	
checkIssuer	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、 <code>validIssuerUri</code> 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。
boolean	
clientAudience	認可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、 <code>clientId</code> と <code>secret</code> メソッドを用いた PLAIN 上の OAuth 2.0 の設定に使用されます。
string	
clientId	Kafka ブローカーは、OAuth クライアント ID を使用して認可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。
string	
clientScope	認可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、 <code>clientId</code> と <code>secret</code> メソッドを用いた PLAIN 上の OAuth 2.0 の設定に使用されます。
string	
clientSecret	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認可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。
GenericSecretSource	

プロパティ	説明
connectTimeoutSeconds	認可サーバーへの接続時のタイムアウト (秒単位)。設定しない場合は、実際の接続タイムアウトは 60 秒になります。
integer	
customClaimCheck	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。
string	
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	
enableECDSA	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。値は無視されません。
boolean	
enableOauthBearer	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。
boolean	
enablePlain	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。
boolean	
fallbackUserNameClaim	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。
string	
fallbackUserNamePrefix	ユーザー ID を設定するために 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。
string	
groupsClaim	認証中にユーザーのグループ抽出に使用される JSONPath クエリー。抽出したグループは、カスタムオーソライザーで使用できます。デフォルトでは、グループは抽出されません。
string	

プロパティ	説明
groupsClaimDelimiter	グループの解析時に JSON 配列ではなく単一の文字列の値として抽出された場合に使用される区切り文字。デフォルト値は','(コンマ)です。
string	
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	
jwtEndpointUri	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	
jwtExpirySeconds	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、 jwtRefreshSeconds 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	
jwtMinRefreshPauseSeconds	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間隔は待機します。デフォルトは 1 秒です。
integer	
jwtRefreshSeconds	JWKS 証明書が更新される頻度を設定します。更新間隔は、 jwtExpirySeconds 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	
readTimeoutSeconds	認可サーバーへの接続時の読み取りタイムアウト (秒単位)。設定しない場合は、実際の読み取りタイムアウトは 60 秒になります。
integer	
tlsTrustedCertificates	OAuth サーバーへの TLS 接続の信頼済み証明書。

プロパティ	説明
CertSecretSource array	
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 clientsecret に設定するか、 username をアカウントのユーザー名に、 password を \$accessToken: の接頭辞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 (接頭辞なしで) 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	

12.2.8. GenericSecretSource スキーマ参照

KafkaClientAuthenticationOAuth、**KafkaListenerAuthenticationCustom**、**KafkaListenerAuthenticationOAuth** で使用

プロパティ	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	説明
string	
secretName	シークレット値が含まれる OpenShift シークレットの名前。
string	

12.2.9. CertSecretSource スキーマ参照

[ClientTls](#)、[KafkaAuthorizationKeycloak](#)、[KafkaClientAuthenticationOAuth](#)、[KafkaListenerAuthenticationOAuth](#) で使用

プロパティ	説明
certificate	Secret のファイル証明書の名前。
string	
secretName	証明書が含まれる Secret の名前。
string	

12.2.10. KafkaListenerAuthenticationCustom スキーマ参照

[GenericKafkaListener](#) で使用

[KafkaListenerAuthenticationCustom](#) スキーマプロパティの完全リスト

カスタム認証を設定するには、**type** プロパティを **custom** に設定します。

カスタム認証では、kafka でサポートされているあらゆるタイプの認証が使用できます。

カスタム OAuth 認証の設定例

```
spec:
  kafka:
    config:
      principal.builder.class: SimplePrincipal.class
    listeners:
      - name: oauth-bespoke
        port: 9093
        type: internal
        tls: true
        authentication:
          type: custom
          sasl: true
        listenerConfig:
          oauthbearer.sasl.client.callback.handler.class: client.class
          oauthbearer.sasl.server.callback.handler.class: server.class
```

```

oauthbearer.sasl.login.callback.handler.class: login.class
oauthbearer.connections.max.reauth.ms: 999999999
sasl.enabled.mechanisms: oauthbearer
oauthbearer.sasl.jaas.config: |
  org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required ;
secrets:
- name: example

```

sasl および **tls** の値を使用して、リスナーにマップするプロトコルを判別するプロトコルマップが生成されます。

- SASL = True, TLS = True → SASL_SSL
- SASL = False, TLS = True → SSL
- SASL = True, TLS = False → SASL_PLAINTEXT
- SASL = False, TLS = False → PLAINTEXT

12.2.10.1. listenerConfig

listenerConfig を使用して指定されたリスナー設定は、**listener.name.<listener_name>-<port>** の接頭辞が付けられます。たとえば、**sasl.enabled.mechanisms** は **listener.name.<listener_name>-<port>.sasl.enabled.mechanisms** になります。

12.2.10.2. secrets

シークレットは、Kafka ブローカーノードのコンテナの **/opt/kafka/custom-authn-secrets/custom-listener-<listener_name>-<port>/<secret_name>** にマウントされます。

たとえば、設定例ではマウントされたシークレット (**example**) は **/opt/kafka/custom-authn-secrets/custom-listener-oauth-bespoke-9093/example** にあります。

12.2.10.3. プリンシパルビルダー

Kafka クラスター設定でカスタムプリンシパルビルダーを設定できます。ただし、プリンシパルビルダーは以下の要件に依存します。

- 指定されたプリンシパルビルダークラスがイメージに存在している。独自に構築する **前** に、すでに存在しているかどうかを確認します。必要なクラスで AMQ Streams イメージを再構築する必要があります。
- 他のリスナーが **oauth** タイプ認証をしていない。これは、OAuth リスナーが独自のプリンシパルビルダーを Kafka 設定に追加するためです。
- 指定のプリンシパルビルダーは AMQ Streams と互換性がある。

AMQ Streams は Kafka クラスターの管理に使用するため、カスタムプリンシパルビルダーは認証用のピア証明書をサポートする必要があります。



注記

Kafka のデフォルトのプリンシパルビルダークラス は、ピア証明書の名前を基にしたプリンシパルのビルドをサポートします。カスタムプリンシパルビルダーは、SSL ピア証明書の名前を使用して **user** タイプのプリンシパルを指定する必要があります。

以下の例は、AMQ Streams の OAuth 要件を満たすカスタムプリンシパルビルダーを示しています。

カスタム OAuth 設定のプリンシパルビルダーの例

```
public final class CustomKafkaPrincipalBuilder implements KafkaPrincipalBuilder {

    public KafkaPrincipalBuilder() {}

    @Override
    public KafkaPrincipal build(AuthenticationContext context) {
        if (context instanceof SslAuthenticationContext) {
            SSLSession sslSession = ((SslAuthenticationContext) context).session();
            try {
                return new KafkaPrincipal(
                    KafkaPrincipal.USER_TYPE, sslSession.getPeerPrincipal().getName());
            } catch (SSLPeerUnverifiedException e) {
                throw new IllegalArgumentException("Cannot use an unverified peer for authentication", e);
            }
        }

        // Create your own KafkaPrincipal here
        ...
    }
}
```

12.2.10.4. KafkaListenerAuthenticationCustom スキーマ参照

type プロパティは **KafkaListenerAuthenticationCustom** タイプと **KafkaListenerAuthenticationTls**, **KafkaListenerAuthenticationScramSha512**, **KafkaListenerAuthenticationOAuth** とを区別して使用するための識別子です。 **KafkaListenerAuthenticationCustom** タイプには **custom** の値が必要です。

プロパティ	説明
listenerConfig	特定のリスナーに使用される設定。すべての値の前に listener.name.<listener_name> を付けます。
map	
sasl	このリスナーで SASL を有効または無効にします。
boolean	
secrets	/opt/kafka/custom-authn-secrets/custom-listener-<listener_name>-<port>/<secret_name> にマウントされるシークレット。
GenericSecretSource アレイ	
type	custom である必要があります。
string	

12.2.11. GenericKafkaListenerConfiguration スキーマ参照

[GenericKafkaListener](#) で使用

[GenericKafkaListenerConfiguration](#) スキーマプロパティの全リスト

Kafka リスナーの設定。

12.2.11.1. 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
  # ...
```

12.2.11.2. 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。

12.2.11.3. 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
# ...
# ...

```

12.2.11.4. 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
# ...
# ...

```

12.2.11.5. 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
# ...
# ...

```

12.2.11.6. useServiceDnsDomain

`useServiceDnsDomain` プロパティは、**internal** リスナーでのみ使用されます。クラスターサービスの接尾辞 (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接尾辞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接尾辞で生成されます。(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接尾辞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接尾辞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。

12.2.11.7. GenericKafkaListenerConfiguration スキーマプロパティ

プロパティ	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	説明
<p>loadBalancerSourceRanges</p> <p>string array</p>	<p>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) のリスト。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。このフィールドは、loadbalancer のリスナーでのみ使用できます。</p>
<p>bootstrap</p> <p>GenericKafkaListenerConfigurationBootstrap</p>	<p>ブートストラップの設定。</p>
<p>brokers</p> <p>GenericKafkaListenerConfigurationBroker アレイ</p>	<p>ブローカーごとの設定。</p>
<p>ipFamilyPolicy</p> <p>string ([RequireDualStack、SingleStack、PreferDualStack] のいずれか)</p>	<p>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、RequiredDualStack です。SingleStack は単一の IP ファミリー用です。PreferDualStack は、デュアルスタック設定のクラスターでは2つの IP ファミリーを、シングルスタック設定のクラスターでは1つの IP ファミリーを対象としています。RequiredDualStack は、デュアルスタック設定のクラスターに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。</p>
<p>ipFamilies</p> <p>string ([IPv6, IPv4] の1つ以上) array</p>	<p>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。</p>
<p>createBootstrapService</p> <p>boolean</p>	<p>ブートストラップサービスを作成するかどうか。ブートストラップサービスはデフォルトで作成されます (指定されない場合)。このフィールドは、loadBalancer タイプリスナーと併用できません。</p>

プロパティ	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ーのリストです。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	
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):
string ([ExternalDNS、ExternalIP、Hostname、InternalIP、InternalDNS] のいずれか)	
<ul style="list-style-type: none"> ● ExternalDNS ● ExternalIP ● InternalDNS ● InternalIP ● Hostname <p>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。</p>	

プロパティ	説明
useServiceDnsDomain	OpenShift サービス DNS ドメインを使用す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 ドメインの接尾辞が含まれます (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	

12.2.12. CertAndKeySecretSource スキーマ参照

[GenericKafkaListenerConfiguration](#)、[KafkaClientAuthenticationTls](#) で使用

プロパティ	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	
secretName	証明書が含まれる Secret の名前。
string	

12.2.13. GenericKafkaListenerConfigurationBootstrap スキーマ参照

[GenericKafkaListenerConfiguration](#) で使用

[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 で設定されます。

12.2.13.1. 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
# ...

```

12.2.13.2. 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パープロカーサービスで使用されるホスト名を指定します。

Ingress コントローラー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
# ...

```

12.2.13.3. 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
# ...

```

12.2.13.4. 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
# ...
```

12.2.13.5. 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"
# ...

```

12.2.13.6. GenericKafkaListenerConfigurationBootstrap スキーマのプロパティ

プロパティ	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	
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	

12.2.14. GenericKafkaListenerConfigurationBroker スキーマ参照

[GenericKafkaListenerConfiguration](#) で使用

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** プロパティの設定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
  # ...
```

12.2.14.1. GenericKafkaListenerConfigurationBroker スキーマプロパティ

プロパティ	説明
broker	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。
integer	
advertisedHost	ブローカーの advertised.brokers で使用されるホスト名。
string	

プロパティ	説明
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	

12.2.15. 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	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。

プロパティ	説明
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	

12.2.16. 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	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。
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	

プロパティ	説明
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	

12.2.17. PersistentClaimStorageOverride スキーマ参照

PersistentClaimStorage で使用

プロパティ	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。
string	
broker	Kafka ブローカーの ID (ブローカー ID)。
integer	

12.2.18. 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	説明
type	jbod でなければなりません。
string	
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リスト。
EphemeralStorage 、 PersistentClaimStorage array	

12.2.19. 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。

12.2.19.1. 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。

12.2.19.2. 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	説明
type	simple でなければなりません。
string	
superUsers	スーパーユーザーのリスト。無制限のアクセス権を取得する必要があるユーザープリンシパルのリストが含まれなければなりません。
string array	

12.2.20. 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 のプロパティを設定します。AMQ Streams は、Kafka 承認に Open Policy Agent プラグインをオーソライザーとして使用します。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。

12.2.20.1. 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**。

12.2.20.2. 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。

12.2.20.3. 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。

12.2.20.4. 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。

12.2.20.5. 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。

12.2.20.6. 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

```
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
# ...

```

12.2.20.7. 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	説明
type	opa でなければなりません。
string	
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。このオプションは必須です。
string	
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。
boolean	
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	

プロパティ	説明
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を持つユーザープリンシパルのリストです。
string array	
enableMetrics	Open Policy Agent オーソライザープラグインでメトリックを指定するかどうかを定義します。デフォルトは false です。
boolean	

12.2.21. 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	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	認可サーバートークンエンドポイント URI。
string	
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。

プロパティ	説明
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、認可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リスト。無制限のアクセス権を取得する必要があるユーザープリンシパルのリストが含まれなければなりません。
string array	
connectTimeoutSeconds	認可サーバーへの接続時のタイムアウト (秒単位)。設定しない場合は、実際の接続タイムアウトは60秒になります。
integer	
readTimeoutSeconds	認可サーバーへの接続時の読み取りタイムアウト (秒単位)。設定しない場合は、実際の読み取りタイムアウトは60秒になります。
integer	

12.2.22. KafkaAuthorizationCustom スキーマ参照

[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** インターフェイスを実装し、**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。

12.2.22.1. authorizerClass

(必須) カスタム ACL をサポートするための **org.apache.kafka.server.authorizer.Authorizer** インターフェイスを実装した Java クラスです。

12.2.22.2. 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` を使用して、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** ディレクトリーの下にある **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ーライブラリー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、サードパーティーのライブラリーが Maven リポジトリーで利用可能であり、その Maven リポジトリーが 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。

カスタム承認では、**oauth** 認証を使用して **groupsClaim** 設定属性を設定する時に JWT トークンから抽出されたグループメンバーシップ情報を利用できます。グループは、以下のように `authorize()` 呼び出し中に **OAuthKafkaPrincipal** オブジェクトで利用できます。

```
public List<AuthorizationResult> authorize(AuthorizableRequestContext requestContext,
List<Action> actions) {

    KafkaPrincipal principal = requestContext.principal();
    if (principal instanceof OAuthKafkaPrincipal) {
        OAuthKafkaPrincipal p = (OAuthKafkaPrincipal) principal;
```



```

    for (String group: p.getGroups()) {
        System.out.println("Group: " + group);
    }
}
}
}

```

12.2.22.3. 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	説明
type	custom である必要があります。
string	
authorizerClass	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。
string	
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を持つユーザープリンシパルです。
string array	
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。
boolean	

12.2.23. Rack スキーマ参照

以下で使用: [KafkaClusterSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)

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。

12.2.23.1. ラック間でのパーティションレプリカの分散

ラック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、各ブローカーにラック 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 の設定に応じて、**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。

12.2.23.2. 最も近いレプリカからのメッセージの消費

ラック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ウェアネスが設定されており、**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** を使用して、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。

レプリカ対応セクターを有効にした rack 設定例

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

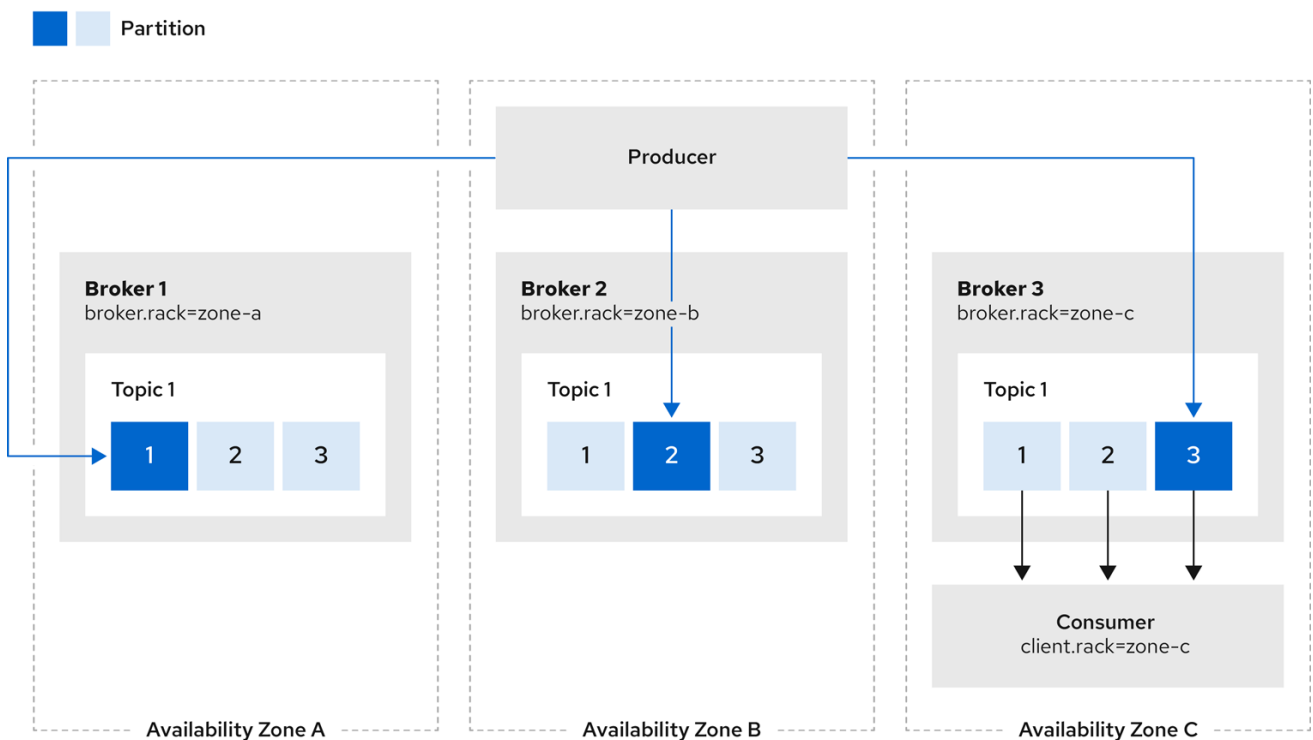
```

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.rack` ID 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12.1 同じアベイラビリティゾーンのリプレカから消費するクライアントの例



222_Streams_0322

コネクタが最も近いレプリカからのメッセージを消費するように、Kafka Connect と MirrorMaker 2.0 を設定することもできます。 `KafkaConnect` および `KafkaMirrorMaker2` カスタムリソースでラック認識を有効にします。この設定ではアフィニティルールは設定されませんが、 `affinity` または `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。

AMQ Streams を使用して Kafka Connect をデプロイメントする場合、 `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:
  # ...
  rack:
    topologyKey: topology.kubernetes.io/zone
  # ...

```

AMQ Streams を使用して MirrorMaker 2 を展開する場合、**KafkaMirrorMaker2** 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できます。

MirrorMaker 2.0 の rack 設定例

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
# ...
spec:
  # ...
  rack:
    topologyKey: topology.kubernetes.io/zone
  # ...

```

12.2.23.3. Rack スキーマのプロパティ

プロパティ	説明
topologyKey	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。ラベルの値は、ブローカーの broker.rack 設定と、Kafka Connect または MirrorMaker 2.0 の client.rack 設定を設定するために使用されます。
string	

12.2.24. Probe スキーマ参照

[CruiseControlSpec](#)、[EntityTopicOperatorSpec](#)、[EntityUserOperatorSpec](#)、[KafkaBridgeSpec](#)、[KafkaClusterSpec](#)、[KafkaConnectSpec](#)、[KafkaExporterSpec](#)、[KafkaMirrorMaker2Spec](#)、[KafkaMirrorMakerSpec](#)、[TlsSidecar](#)、[ZookeeperClusterSpec](#) で使用

プロパティ	説明
failureThreshold	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。
integer	
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。
integer	

プロパティ	説明
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	

12.2.25. JvmOptions スキーマ参照

[CruiseControlSpec](#)、[EntityTopicOperatorSpec](#)、[EntityUserOperatorSpec](#)、[KafkaBridgeSpec](#)、[KafkaClusterSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)、[KafkaMirrorMakerSpec](#)、[ZooKeeperClusterSpec](#) で使用

プロパティ	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。
string	
-Xmx	JVM への -Xmx オプション。
string	
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のマップ。
SystemProperty array	

12.2.26. SystemProperty スキーマ参照

[JvmOptions](#) で使用

プロパティ	説明
name	システムプロパティ名。
string	
value	システムプロパティの値。
string	

12.2.27. KafkaJmxOptions スキーマ参照

[KafkaClusterSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)、[ZookeeperClusterSpec](#) で使用

[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。

ポート 9999 に接続して、Kafka ブローカー、ZooKeeper ノード、Kafka Connect、および MirrorMaker 2.0 から JMX メトリックを取得します。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ックを取得できます。

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ーを有効にするには、**authentication** フィールドの **type** パラメーターを **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

```
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 メトリックを取得できます。

たとえば、ブローカー 0 から JMX メトリックを取得するには、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 Pod の名前、**CLUSTER-NAME-kafka-brokers** はブローカー Pod 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、メトリックを取得できますが、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 設定例

```
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ックの詳細は、[Apache Kafka のドキュメント](#) を参照してください。

12.2.27.1. KafkaJmxOptions スキーマプロパティ

プロパティ	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 プロパティの値によって異なり、[password] の 1 つでなければなりません。
KafkaJmxAuthenticationPassword	

12.2.28. 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	説明
type	password でなければなりません。
string	

12.2.29. JmxPrometheusExporterMetrics スキーマ参照

[CruiseControlSpec](#)、[KafkaClusterSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)、[KafkaMirrorMakerSpec](#)、[ZookeeperClusterSpec](#) で使用

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	説明
type	jmxPrometheusExporter でなければなりません。
string	
valueFrom	Prometheus JMX Exporter 設定が保存される ConfigMap エントリ。この設定の構造の詳細は、 Prometheus JMXExporter を参照してください。
ExternalConfigurationReference	

12.2.30. ExternalConfigurationReference のスキーマ参照

[ExternalLoggingJmxPrometheusExporterMetrics](#) で使用

プロパティ	説明
configMapKeyRef	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。
ConfigMapKeySelector	

12.2.31. InlineLogging スキーマ参照

[CruiseControlSpec](#)、[EntityTopicOperatorSpec](#)、[EntityUserOperatorSpec](#)、[KafkaBridgeSpec](#)、[KafkaClusterSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)、[KafkaMirrorMakerSpec](#)、[ZookeeperClusterSpec](#) で使用

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	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	

12.2.32. ExternalLogging スキーマ参照

[CruiseControlSpec](#)、[EntityTopicOperatorSpec](#)、[EntityUserOperatorSpec](#)、[KafkaBridgeSpec](#)、[KafkaClusterSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)、[KafkaMirrorMakerSpec](#)、[ZooKeeperClusterSpec](#) で使用

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	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	

12.2.33. KafkaClusterTemplate スキーマ参照

[KafkaClusterSpec](#) で使用

プロパティ	説明
statefulset	Kafka StatefulSet のテンプレート。
StatefulSetTemplate	
Pod	Kafka Pod のテンプレート。
PodTemplate	
bootstrapService	Kafka ブートストラップ Service のテンプレート。
InternalServiceTemplate	

プロパティ	説明
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のテンプレート。
ContainerTemplate	
initContainer	Kafka init コンテナのテンプレート。
ContainerTemplate	

プロパティ	説明
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	
podSet	Kafka StrimziPodSet リソースのテンプレート。
ResourceTemplate	

12.2.34. StatefulSetTemplate スキーマ参照

[KafkaClusterTemplate](#)、[ZookeeperClusterTemplate](#) で使用

プロパティ	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)	

12.2.35. MetadataTemplate スキーマ参照

[BuildConfigTemplate](#)、[DeploymentTemplate](#)、[InternalServiceTemplate](#)、[PodDisruptionBudgetTemplate](#)、[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** プロパティで設定されます。

以下に例を示します。

```
# ...
template:
  pod:
    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。

12.2.35.1. MetadataTemplate スキーマのプロパティ

プロパティ	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	

12.2.36. PodTemplate スキーマ参照

[CruiseControlTemplate](#)、[EntityOperatorTemplate](#)、[KafkaBridgeTemplate](#)、[KafkaClusterTemplate](#)、[KafkaConnectTemplate](#)、[KafkaExporterTemplate](#)、[KafkaMirrorMakerTemplate](#)、[ZookeeperClusterTemplate](#) で使用

[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 の設定例

```
# ...
template:
  pod:
    metadata:
      labels:
        label1: value1
      annotations:
        anno1: value1
    imagePullSecrets:
      - name: my-docker-credentials
    securityContext:
      runAsUser: 1000001
```

```

fsGroup: 0
terminationGracePeriodSeconds: 120
# ...

```

12.2.36.1. hostAliases

hostAliases プロパティを使用して、Pod 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"
#...

```

12.2.36.2. PodTemplate スキーマのプロパティ

プロパティ	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リストです。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 アレイ	
securityContext	Pod レベルのセキュリティ属性と共通のコンテナ設定を設定します。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	

プロパティ	説明
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	
tolerations	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。
toleration アレイ	
priorityClassName	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。
string	
schedulerName	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。
string	
hostAliases	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、 external documentation for core/v1 hostaliases を参照してください。
HostAlias アレイ	
tmpDirSizeLimit	一時 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) を定義します。デフォルト値は 5Mi です。
string	
enableServiceLinks	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。
boolean	
topologySpreadConstraints	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。
TopologySpreadConstraint 配列	

12.2.37. InternalServiceTemplate のスキーマ参照

[CruiseControlTemplate](#)、[KafkaBridgeTemplate](#)、[KafkaClusterTemplate](#)、[KafkaConnectTemplate](#)、[ZookeeperClusterTemplate](#) で使用

プロパティ	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	
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 、 RequiredDualStack です。 SingleStack は単一の IP ファミリー用です。 PreferDualStack は、デュアルスタック設定のクラスターでは2つの IP ファミリーを、シングルスタック設定のクラスターでは1つの IP ファミリーを対象としています。 RequiredDualStack は、デュアルスタック設定のクラスターに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 <code>ipFamilyPolicy</code> 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	

12.2.38. ResourceTemplate スキーマ参照

[CruiseControlTemplate](#)、[EntityOperatorTemplate](#)、[KafkaBridgeTemplate](#)、[KafkaClusterTemplate](#)、[KafkaConnectTemplate](#)、[KafkaExporterTemplate](#)、[KafkaMirrorMakerTemplate](#)、[KafkaUserTemplate](#)、[ZookeeperClusterTemplate](#) で使用

プロパティ	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	

12.2.39. PodDisruptionBudgetTemplate スキーマ参照

[CruiseControlTemplate](#)、[KafkaBridgeTemplate](#)、[KafkaClusterTemplate](#)、[KafkaConnectTemplate](#)、[KafkaMirrorMakerTemplate](#)、[ZookeeperClusterTemplate](#) で使用

[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** プロパティのデフォルト値を変更して、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
# ...
```

12.2.39.1. PodDisruptionBudgetTemplate スキーマ参照

プロパティ	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	

12.2.40. ContainerTemplate スキーマ参照

[CruiseControlTemplate](#)、[EntityOperatorTemplate](#)、[KafkaBridgeTemplate](#)、[KafkaClusterTemplate](#)、[KafkaConnectTemplate](#)、[KafkaExporterTemplate](#)、[KafkaMirrorMakerTemplate](#)、[ZookeeperClusterTemplate](#) で使用

[ContainerTemplate](#) スキーマプロパティの全リスト

コンテナ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。

12.2.40.1. ContainerTemplate スキーマのプロパティ

プロパティ	説明
env	コンテナに適用する必要がある環境変数。
ContainerEnvVar array	
securityContext	コンテナ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。
SecurityContext	

12.2.41. ContainerEnvVar スキーマ参照

[ContainerTemplate](#) で使用

プロパティ	説明
name	環境変数のキー。
string	
value	環境変数の値。
string	

12.2.42. 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 クラスタを設定します。

12.2.42.1. config

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"
# ...

```

12.2.42.2. logging

ZooKeeper には設定可能なロガーがあります。

- **zookeeper.root.logger**

ZooKeeper は、Apache **log4j** のロガー実装を使用しています。

logging プロパティ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、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
# ...

```

ガベージコレクター (GC)

ガベージコレクターのロギングは **jvmOptions** プロパティを使用して有効 (または無効) にすることもできます。

12.2.42.3. ZookeeperClusterSpec スキーマプロパティ

プロパティ	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 プロローカーの設定。次の接頭辞のあるプロパティは設定できません: 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。

プロパティ	説明
JvmOptions	
jmxOptions	Zookeeper ノードの 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	メトリックの設定。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。テンプレートを使用すると、ユーザーは StatefulSet 、 Pod 、および Service の生成方法を指定できます。
ZookeeperClusterTemplate	

12.2.43. ZookeeperClusterTemplate スキーマ参照

ZookeeperClusterSpec で使用

プロパティ	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。

プロパティ	説明
InternalServiceTemplate	
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	
podSet	ZooKeeper StrimziPodSet リソースのテンプレート。
ResourceTemplate	

12.2.44. EntityOperatorSpec スキーマ参照

KafkaSpec で使用

プロパティ	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	

プロパティ	説明
template	Entity Operator リソースのテンプレート。テンプレートを使用すると、ユーザーは Deployment と Pod の生成方法を指定できます。
EntityOperatorTemplate	

12.2.45. 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。

12.2.45.1. logging

Topic Operator には設定可能なロガーがあります。

- **rootLogger.level**

Topic Operator では、Apache**log4j2** 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、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
    # ...

```

ガベージコレクター (GC)

ガベージコレクターのロギングは [jvmOptions](#) プロパティを使用して有効 (または無効) にすることもできます。

12.2.45.2. EntityTopicOperatorSpec スキーマプロパティ

プロパティ	説明
watchedNamespace	Topic Operator が監視する必要がある 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	

プロパティ	説明
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	
logging	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	

12.2.46. 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。

12.2.46.1. logging

User Operator には設定可能なロガーがあります。

- **rootLogger.level**

User Operator では、Apache **log4j2** 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、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
# ...

```

ガベージコレクター (GC)

ガベージコレクターのロギングは **jvmOptions** プロパティを使用して有効 (または無効) にすることもできます。

12.2.46.2. EntityUserOperatorSpec スキーマプロパティ

プロパティ	説明
watchedNamespace	User Operator が監視する必要がある 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接頭辞。
string	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	

プロパティ	説明
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	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	

12.2.47. TlsSidecar スキーマ参照

[CruiseControlSpec](#)、[EntityOperatorSpec](#) で使用

[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せん。。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 Entity Operator で使用されます。

TLS サイドカーは、[Kafka.spec.entityOperator](#) の [tlsSidecar](#) プロパティを使用して設定されます。

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 [メモリー](#) 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](#) を設定します。

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
  # ...

```

12.2.47.1. TlsSidecar スキーマのプロパティ

プロパティ	説明
image	コンテナの Docker イメージ。
string	
livenessProbe	Pod の liveness チェック。
Probe	
logLevel	TLS サイドカーのログレベル。デフォルト値は notice です。
string ([emerg、debug、crit、err、alert、warning、notice、info] のいずれか)	

プロパティ	説明
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	

12.2.48. EntityOperatorTemplate スキーマ参照

EntityOperatorSpec で使用

プロパティ	説明
deployment	Entity Operator Deployment のテンプレート。
ResourceTemplate	
Pod	Entity Operator Pod のテンプレート。
PodTemplate	
topicOperatorContainer	Entity Topic Operator コンテナのテンプレート。
ContainerTemplate	
userOperatorContainer	Entity User Operator コンテナのテンプレート。
ContainerTemplate	
tlsSidecarContainer	Entity Operator TLS サイドカーコンテナのテンプレート。
ContainerTemplate	
serviceAccount	Entity Operator サービスアカウントのテンプレート。
ResourceTemplate	

12.2.49. CertificateAuthority スキーマ参照

KafkaSpec で使用

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	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	
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 で設定されます。 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)	

12.2.50. CruiseControlSpec スキーマ参照

[KafkaSpec](#) で使用

[CruiseControlSpec](#) スキーマプロパティの完全なリスト

Cruise Control 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。

- ゴールの設定
- リソース配分目標の容量制限

12.2.50.1. config

config プロパティを使用して、Cruise Control オプションをキーとして設定します。

AMQ Streams によって直接管理されないプロパティに限り、標準 Cruise Control 設定の提供が可能です。

設定できない設定オプションは、次のものに関連しています。

- セキュリティー (暗号化、認証、および承認)
- Kafka クラスターへの接続
- クライアント ID の設定
- ZooKeeper の接続
- Web サーバー設定
- 自己修復

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、[Cruise Control 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。禁止されている接頭辞のリストについては、**config** プロパティの説明を参照してください。

禁止されているオプションが **config** プロパティにある場合、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。サポートされている他のすべてのオプションは、Cruise Control 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に、[許可された ssl プロパティを設定](#) することができます。**webserver** プロパティを設定して、CORS (Cross-Origin Resource Sharing) を有効にすることもできます。

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
```



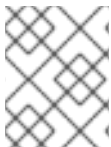
```

send.buffer.bytes: 131072
webserver.http.cors.enabled: true
webserver.http.cors.origin: "*"
webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
# ...

```

12.2.50.2. Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) は、REST API へのアクセスを制御するための HTTP メカニズムです。制限は、アクセス方法またはクライアントアプリケーションの元の URL に対して行うことができます。**config** の **webserver.http.cors.enabled** プロパティを使用して、Cruise Control で CORS を有効にできます。有効にすると、CORS は、AMQ Streams とは異なる元の URL を持つアプリケーションからの Cruise Control REST API への読み取りアクセスを許可します。これにより、指定されたオリジンからのアプリケーションが **GET** リクエストを使用して、Cruise Control API を介して Kafka クラスタに関する情報をフェッチできるようになります。たとえば、アプリケーションは、現在のクラスタ負荷または最新の最適化提案に関する情報を取得できます。**POST** リクエストは許可されていません。



注記

Cruise Control で CORS を使用方法の詳細については、[Cruise Control Wiki の REST API](#) を参照してください。

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 ①
      webserver.http.cors.origin: "*" ②
      webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type" ③
  # ...

```

- ① CORS を有効にします。
- ② **Access-Control-Allow-Origin** HTTP 応答ヘッダーの許可されるオリジンを指定します。ワイルドカードを使用するか、単一のオリジンを URL として指定できます。ワイルドカードを使用すると、任意のオリジンからのリクエストに続いてレスポンスが返されます。
- ③ **Access-Control-Expose-Headers** HTTP 応答ヘッダーの指定されたヘッダー名を公開します。許可されたオリジンのアプリケーションは、指定されたヘッダーで応答を読み取ることができます。

12.2.50.3. Cruise Control REST API のセキュリティー

Cruise Control REST API は、HTTP 基本認証と SSL で保護されており、Kafka ブローカーの廃止など、破壊的な可能性のある Cruise Control 操作からクラスターを保護します。AMQStreams の Cruise Control は、これらの設定が有効になっている場合にのみ使用することを推奨します。

ただし、次の Cruise Control 設定を指定することで、これらの設定を無効にすることができます。

- 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
  # ...
```

12.2.50.4. brokerCapacity

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** - ディスク使用量の分布
- **CpuUsageDistributionGoal** - CPU 使用量の分布
- **NetworkInboundUsageDistributionGoal** - ネットワーク受信使用量の分布
- **NetworkOutboundUsageDistributionGoal** - ネットワーク送信使用量の分布

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、以下のブローカーリソースに設定できます。

- **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)

ネットワークスループットの場合、1 秒あたりの標準の OpenShift バイト単位 (K、M、G) またはそれに相当するピバイト (2 の累乗)(Ki、Mi、Gi) の整数値を使用します。



注記

ディスクと CPU の容量制限は AMQ Streams で自動的に生成されるので、設定する必要はありません。



注記

CPU ゴールの使用時に正確なリバランスプロポーザルを保证するには、**Kafka.spec.kafka.resources** で CPU 制限と同等の CPU 要求を設定できます。これにより、すべての CPU リソースが事前に予約されて常に利用できます。この設定を使用すると、CPU 目標に基づいてリバランス提案を準備するときに Cruise Control が CPU 使用率を適切に評価できます。

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:
      inboundNetwork: 10000KiB/s
      outboundNetwork: 10000KiB/s
    # ...
```

12.2.50.5. 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 **log4j2** ロガー実装が使用されます。

logging プロパティ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、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
    # ...
```

ガベージコレクター (GC)

ガベージコレクターのロギングは [jvmOptions](#) プロパティを使用して有効 (または無効) にすることもできます。

12.2.50.6. CruiseControlSpec スキーマプロパティ

プロパティ	説明
image	Pod の Docker イメージ。
string	
tlsSidecar	tlsSidecar プロパティは廃止されました。TLS サイドカーの設定。
TlsSidecar	
resources	Cruise Control コンテナ用に予約された 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。
ResourceRequirements	
livenessProbe	Cruise Control コンテナの Pod liveness チェック
Probe	
readinessProbe	Cruise Control コンテナの Pod readiness チェック
Probe	

プロパティ	説明
jvmOptions	Cruise Control コンテナの JVM オプション
JvmOptions	
logging	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	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。
CruiseControlTemplate	
brokerCapacity	Cruise Control の brokerCapacity の設定。
BrokerCapacity	
config	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。次の接頭辞を持つプロパティは設定できません: failed.brokers.zk.path,webserver.http,webserver.api.urlprefix,webserver.session.path,webserver.accesslog, two.step, request.reason.required,metric.reporter.sampler.bootstrap.servers, capacity.config.file, self.healing, ssl, kafka.broker.failure.detection.enable, topic.config.provider.class (例外: ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols, webserver.http.cors.enabled, webserver.http.cors.origin, webserver.http.cors.exposeheaders, webserver.security.enable, webserver.ssl.enable).
map	
metricsConfig	メトリックの設定。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	

12.2.51. CruiseControlTemplate スキーマ参照

CruiseControlSpec で使用

プロパティ	説明
deployment	Cruise Control Deployment のテンプレート。

プロパティ	説明
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	
podDisruptionBudget	Cruise Control PodDisruptionBudget のテンプレート。
PodDisruptionBudgetTemplate	
cruiseControlContainer	Cruise Control コンテナのテンプレート。
ContainerTemplate	
tlsSidecarContainer	tlsSidecarContainer プロパティは廃止されました。Cruise Control TLS サイドカーコンテナのテンプレート。
ContainerTemplate	
serviceAccount	Cruise Control サービスアカウントのテンプレート。
ResourceTemplate	

12.2.52. BrokerCapacity スキーマ参照

CruiseControlSpec で使用

プロパティ	説明
disk	disk プロパティは非推奨になりました。Cruise Control のディスク容量設定は非推奨になり、無視されています。今後、ディスクのブローカー容量 (バイト単位) で削除される予定です。標準の Open Shift バイト単位 (K、M、G、または T)、それらのピバイト (2 の累乗) に相当するもの (Ki、Mi、Gi、または Ti) の数値、または E 表記の有無にかかわらずバイト値を使用します。例: 100000M、100000Mi、104857600000、または 1e+11。
string	

プロパティ	説明
cpuUtilization	cpuUtilization プロパティは非推奨になりました。Cruise Control の CPU 容量設定は非推奨になり、無視されています。今後、CPU リソースの使用率 (0-100) に対するブローカー容量 (バイト単位) で削除される予定です。
integer	
inboundNetwork	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 (バイト/秒)。整数値は、標準の OpenShift バイト単位 (K、M、G) またはそれと同等のピバイト (Ki、Mi、Gi)/秒を使用します。たとえば、10000KiB/s です。
string	
outboundNetwork	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 (バイト/秒)。整数値は、標準の OpenShift バイト単位 (K、M、G) またはそれと同等のピバイト (Ki、Mi、Gi)/秒を使用します。たとえば、10000KiB/s です。
string	

12.2.53. KafkaExporterSpec スキーマ参照

KafkaSpec で使用

プロパティ	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	

プロパティ	説明
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	

12.2.54. KafkaExporterTemplate スキーマ参照

KafkaExporterSpec で使用

プロパティ	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	service プロパティは非推奨になりました。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	

12.2.55. KafkaStatus スキーマ参照

Kafka で使用

プロパティ	説明
conditions	ステータス条件のリスト。
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	

12.2.56. Condition スキーマ参照

使用先:

[KafkaBridgeStatus](#)、[KafkaConnectorStatus](#)、[KafkaConnectStatus](#)、[KafkaMirrorMaker2Status](#)、[KafkaMirrorMakerStatus](#)、[KafkaRebalanceStatus](#)、[KafkaStatus](#)、[KafkaTopicStatus](#)、[KafkaUserStatus](#)

プロパティ	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。

プロパティ	説明
string	

12.2.57. ListenerStatus スキーマ参照

[KafkaStatus](#) で使用

プロパティ	説明
type	type プロパティは非推奨となり、 name を使用して設定する必要があります。リスナーの名前。
string	
name	リスナーの名前。
string	
addresses	このリスナーのアドレスリスト。
ListenerAddress array	
bootstrapServers	このリスナーを使用して Kafka クラスタに接続するための host:port ペアのコンマ区切りリスト。
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を検証するために使用できる TLS 証明書 のリスト。 tls リスナーと external リスナーに対してのみ設定します。
string array	

12.2.58. ListenerAddress スキーマ参照

[ListenerStatus](#) で使用

プロパティ	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	

12.2.59. KafkaConnect スキーマ参照

プロパティ	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	

12.2.60. 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。

12.2.60.1. config

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 インターフェイスの設定
- 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。

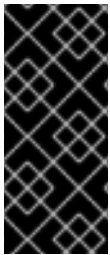
- 文字列
- 数値
- ブール値

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。この状況で、**KafkaConnect.spec.config** オブジェクトの設定を修正すると、Cluster Operator は新しい設定をすべての Kafka Connect ノードにロールアウトできます。

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**

このようなオプションは、**KafkaConnect.spec.config** プロパティにない場合に自動的に設定されません。

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て、クライアント接続に許可される 3 つの **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。

12.2.60.2. logging

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、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更新がトリガーされます。

ガベージコレクター (GC)

ガベージコレクターのロギングは [jvmOptions](#) プロパティを使用して有効 (または無効) にすることもできます。

12.2.60.3. KafkaConnectSpec スキーマプロパティ

プロパティ	説明
version	Kafka Connect のバージョン。デフォルトは 3.2.3 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。
string	

プロパティ	説明
bootstrapServers	接続するブートストラップサーバー。これは <code><hostname>:<port>_pairs</code> のコンマ区切りリストとして指定する必要があります。
string	
tls	TLS 設定。
ClientTls	
authentication	Kafka Connect の認証設定。タイプは、指定のオブジェクト内の authentication.type プロパティの値によって異なり、[tls, scram-sha-256, scram-sha-512, plain, oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls, KafkaClientAuthenticationScramSha256, KafkaClientAuthenticationScramSha512, KafkaClientAuthenticationPlain, KafkaClientAuthenticationOAuth	
config	Kafka Connect の設定。次の接頭辞を持つプロパティは設定できません: 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	
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	

プロパティ	説明
logging	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	
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	
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	
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 および Kafka Mirror Maker 2 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 、 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を設定します。
ExternalConfiguration	
build	Connect コンテナ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	
metricsConfig	メトリックの設定。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	

12.2.61. ClientTls スキーマ参照

使用先:

KafkaBridgeSpec、**KafkaConnectSpec**、**KafkaMirrorMaker2ClusterSpec**、**KafkaMirrorMakerConsumerSpec**、**KafkaMirrorMakerProducerSpec**

ClientTls スキーマプロパティの完全リスト

KafkaConnect、KafkaBridge、KafkaMirror、KafkaMirrorMaker2 をクラスターに接続するための TLS 信頼証明書を設定します。

12.2.61.1. trustedCertificates

trustedCertificates プロパティを使用してシークレットのリストを提供する。

12.2.61.2. ClientTls スキーマプロパティ

プロパティ	説明
trustedCertificates	TLS 接続の信頼済み証明書。
CertSecretSource array	

12.2.62. KafkaClientAuthenticationTls スキーマ参照

使用先:

[KafkaBridgeSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2ClusterSpec](#)、[KafkaMirrorMakerConsumerSpec](#)、[KafkaMirrorMakerProducerSpec](#)

[KafkaClientAuthenticationTls](#) スキーマプロパティの全リスト

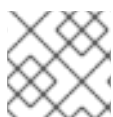
TLS クライアント認証を設定するには、**type** プロパティを **tls** の値に指定します。TLS クライアント認証は TLS 証明書を使用して認証します。

12.2.62.1. certificateAndKey

証明書は **certificateAndKey** プロパティで指定され、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。

User Operator によって作成されたシークレットを使用できます。または、認証に使用される鍵で独自の TLS 証明書ファイルを作成し、ファイルから **Secret** 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
```

12.2.62.2. KafkaClientAuthenticationTls スキーマプロパティ

type プロパティは、**KafkaClientAuthenticationTls** タイプ

と、[KafkaClientAuthenticationScramSha256](#), [KafkaClientAuthenticationScramSha512](#), [KafkaClientAuthenticationPlain](#), [KafkaClientAuthenticationOAuth](#) の使用を区別するための識別子です。[KafkaClientAuthenticationTls](#) タイプには `tls` の値が必要です。

プロパティ	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	

12.2.63. KafkaClientAuthenticationScramSha256 スキーマ参照

使用先:

[KafkaBridgeSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2ClusterSpec](#)、[KafkaMirrorMakerConsumerSpec](#)、[KafkaMirrorMakerProducerSpec](#)

[KafkaClientAuthenticationScramSha256](#) スキーマプロパティの全リスト

SASL ベースの SCRAM-SHA-256 認証を設定するには、**type** プロパティを **scram-sha-256** に設定します。SCRAM-SHA-256 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。

12.2.63.1. username

username プロパティでユーザー名を指定します。

12.2.63.2. passwordSecret

passwordSecret プロパティで、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
```

次に、テキストファイルから **Secret** を作成し、パスワードに独自のフィールド名 (鍵) を設定できます。

```
oc create secret generic MY-CONNECT-SECRET-NAME --from-file=MY-PASSWORD-FIELD-NAME=./MY-PASSWORD.txt
```

Kafka Connect の SCRAM-SHA-256 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: LFTlyFRFIMmU2N2Tm

```

secretName プロパティには **Secret** の名前が含まれ、**password** プロパティには **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 プロパティには、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-256 クライアント認証の設定例

```

authentication:
  type: scram-sha-256
  username: my-connect-username
  passwordSecret:
    secretName: my-connect-secret-name
    password: my-connect-password-field

```

12.2.63.3. KafkaClientAuthenticationScramSha256 スキーマプロパティ

プロパティ	説明
passwordSecret	パスワードを保持する Secret への参照。
PasswordSecretSource	
type	scram-sha-256 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	

12.2.64. PasswordSecretSource スキーマ参照

使用先:

[KafkaClientAuthenticationPlain](#)、[KafkaClientAuthenticationScramSha256](#)、[KafkaClientAuthenticationScramSha512](#)

プロパティ	説明
password	パスワードが保存される Secret のキーの名前。
string	

プロパティ	説明
secretName	パスワードを含むシークレットの名前。
string	

12.2.65. KafkaClientAuthenticationScramSha512 スキーマ参照

使用先:

[KafkaBridgeSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2ClusterSpec](#)、[KafkaMirrorMakerConsumerSpec](#)、[KafkaMirrorMakerProducerSpec](#)

[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。

12.2.65.1. username

username プロパティでユーザー名を指定します。

12.2.65.2. passwordSecret

passwordSecret プロパティで、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
```

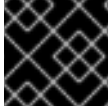
次に、テキストファイルから **Secret** を作成し、パスワードに独自のフィールド名 (鍵) を設定できます。

```
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: LFTlyFRFIMmU2N2Tm
```

secretName プロパティには **Secret** の名前が含まれ、**password** プロパティには **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 プロパティには、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
```

12.2.65.3. KafkaClientAuthenticationScramSha512 スキーマプロパティ

プロパティ	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。
string	
username	認証に使用されるユーザー名。
string	

12.2.66. KafkaClientAuthenticationPlain スキーマ参照

使用先:

[KafkaBridgeSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2ClusterSpec](#)、[KafkaMirrorMakerConsumerSpec](#)、[KafkaMirrorMakerProducerSpec](#)

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。

12.2.66.1. username

username プロパティでユーザー名を指定します。

12.2.66.2. passwordSecret

passwordSecret プロパティで、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
```

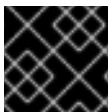
次に、テキストファイルから **Secret** を作成し、パスワードに独自のフィールド名 (鍵) を設定できます。

```
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: LFTlyFRFIMmU2N2Tm
```

secretName プロパティには **Secret** の名前が含まれ、**password** プロパティには **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 プロパティには、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
```

12.2.66.3. KafkaClientAuthenticationPlain スキーマプロパティ

type プロパティは、**KafkaClientAuthenticationPlain** タイプと、**KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationOAuth** の使用を区別するための識別子で

す。 **KafkaClientAuthenticationPlain** タイプには **plain** の値が必要です。

プロパティ	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	

12.2.67. KafkaClientAuthenticationOAuth スキーマ参照

使用先:

[KafkaBridgeSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2ClusterSpec](#)、[KafkaMirrorMakerConsumerSpec](#)、[KafkaMirrorMakerProducerSpec](#)

[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設定するには、**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** プロパティで認可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、**scope** と **audience** を指定できます。

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リストを設定できます。**tlsTrustedCertificates** プロパティには、証明書が保存されるキーの名前を持つシークレットのリストが含まれます。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せん。

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
```

12.2.67.1. KafkaClientAuthenticationOAuth スキーマプロパティ

type プロパティは、**KafkaClientAuthenticationOAuth** タイプと、**KafkaClientAuthenticationTls**、**KafkaClientAuthenticationScramSha256**、**KafkaClientAuthenticationScramSha512**、**KafkaClientAuthenticationPlain** の使用を区別するための識別子です。**KafkaClientAuthenticationOAuth** タイプには **oauth** の値が必要です。

プロパティ	説明
accessToken	認可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。
GenericSecretSource	
accessTokenIsJwt	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。認可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。
boolean	
audience	認可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。一部の認可サーバーでは、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、認可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 audience は指定されません。
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	
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	

プロパティ	説明
connectTimeoutSeconds	認可サーバーへの接続時のタイムアウト (秒単位)。設定しない場合は、実際の接続タイムアウトは 60 秒になります。
integer	
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、認可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	
readTimeoutSeconds	認可サーバーへの接続時の読み取りタイムアウト (秒単位)。設定しない場合は、実際の読み取りタイムアウトは 60 秒になります。
integer	
refreshToken	認可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	認可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。一部の認可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、認可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	認可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	

12.2.68. JaegerTracing スキーマ参照

使用先: [KafkaBridgeSpec](#)、[KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)、[KafkaMirrorMakerSpec](#)

type プロパティは、**JaegerTracing** タイプを使用する際に、今後追加される可能性のある他のサブタイプと区別する識別子です。**JaegerTracing** タイプには **jaeger** の値が必要です。

プロパティ	説明
type	jaeger でなければなりません。
string	

12.2.69. KafkaConnectTemplate スキーマ参照

使用先: [KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)

プロパティ	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 コンテナのテンプレート。
ContainerTemplate	
initContainer	Kafka init コンテナのテンプレート。
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	
clusterRoleBinding	Kafka Connect ClusterRoleBinding のテンプレート。
ResourceTemplate	

プロパティ	説明
buildPod	Kafka Connect Build Pods のテンプレート。build Pod は OpenShift でのみ使用されます。
PodTemplate	
buildContainer	Kafka Connect Build コンテナのテンプレート。build コンテナは OpenShift でのみ使用されます。
ContainerTemplate	
buildConfig	新しいコンテナ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	

12.2.70. DeploymentTemplate スキーマ参照

[KafkaBridgeTemplate](#)、[KafkaConnectTemplate](#)、[KafkaMirrorMakerTemplate](#) で使用

プロパティ	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 および Recreate です。デフォルトは RollingUpdate です。
string ([RollingUpdate、Recreate] のいずれか)	

12.2.71. BuildConfigTemplate スキーマ参照

使用先: [KafkaConnectTemplate](#)

プロパティ	説明
metadata	PodDisruptionBudgetTemplate リソースに適用するメタデータ。

プロパティ	説明
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 Container Registry Secret。
string	

12.2.72. ExternalConfiguration スキーマ参照

使用先: [KafkaConnectSpec](#)、[KafkaMirrorMaker2Spec](#)

ExternalConfiguration スキーマプロパティの完全リスト

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。ボリュームおよび環境変数は、**KafkaConnect.spec** の **externalConfiguration** プロパティで設定されます。

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。

12.2.72.1. env

env プロパティを使用して1つ以上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: QUtJQVhYWfYWFhYWfYWFg=
  awsSecretAccessKey: YlhSD1IYTnpkMjI5WkE=
```



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。

シークレットから環境変数に値をマウントするには、**valueFrom** プロパティおよび **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、コネクタが Amazon AWS と通信するためのものです。コネクタは **AWS_ACCESS_KEY_ID** および **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

```

12.2.72.2. 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 コネクタの設定に使用されるプロパティ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、パス **/opt/kafka/external-configuration/<volume-name>** の Kafka Connect コンテナ内にマウントされます。たとえば、**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** ディレクトリーにあります。

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 Connect REST インターフェイスを介して渡されないようにします。

- **FileConfigProvider** ファイルのプロパティから設定値をロードします。
- **DirectoryConfigProvider** ディレクトリー構造内で個別のファイルから設定値をロードします。

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 を使用したプロパティ値の読み込み

以下の例では、**mysecret** という名前の Secret には、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のある Secret の例

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

- ❶ プロパティファイル形式のコネクタ設定。
- ❷ 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。

Secret および **FileConfigProvider** 設定プロバイダーは Kafka Connect 設定に指定されます。

- Secret は **connector-config** という名前のボリュームにマウントされます。
- **FileConfigProvider** にはエイリアス **ファイル** が付与されます。

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 ❶
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider ❷
  #...
  externalConfiguration:
    volumes:
      - name: connector-config ❸
        secret:
          secretName: mysecret ❹
```

- ❶ 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。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
- 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 またはシークレットを参照する必要があります。
- 4 Secret の名前。

Secret のプロパティ値のプレースホルダーは、コネクタ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。**FileConfigProvider** は、コネクタ設定でマウントされた Secret からデータベースの **username** および **password** プロパティの値を読み取りおよびデプロイメントします。

外部値のプレースホルダーを示すコネクタ設定の例

```
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 を使用した個別ファイルからのプロパティ値のロード

この例の **Secret** には個別のファイルに TLS トラストストアとキーストアユーザーのクレデンシャルが含まれています。

ユーザークレデンシャルのある Secret の例

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  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
```


Secret および **DirectoryConfigProvider** 設定プロバイダーは Kafka Connect 設定に指定されます。

- Secret は **connector-config** という名前のボリュームにマウントされます。
- **DirectoryConfigProvider** にはエイリアスの **ディレクトリー** が付与されます。

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
  #...
  externalConfiguration:
    volumes:
      - name: cluster-ca
        secret:
          secretName: my-cluster-cluster-ca-cert
      - name: my-user
        secret:
          secretName: my-user
```

- 1** **1** **DirectoryConfigProvider** はディレクトリー内のファイルからの値を提供します。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。

クレデンシャルのプレースホルダーはコネクタ設定で参照されます。プレースホルダー構造は **directory:PATH:FILE-NAME** です。**DirectoryConfigProvider** は、コネクタ設定でマウントされた Secret からクレデンシャルを読み取りおよびデプロイメントします。

外部値のプレースホルダーを示すコネクタ設定の例

```
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.history.producer.security.protocol: SSL
    database.history.producer.ssl.truststore.type: PEM
    database.history.producer.ssl.truststore.certificates: "${directory:/opt/kafka/external-configuration/cluster-ca:ca.crt}"
    database.history.producer.ssl.keystore.type: PEM
    database.history.producer.ssl.keystore.certificate.chain: "${directory:/opt/kafka/external-
```

```
configuration/my-user:user.crt)"
  database.history.producer.ssl.keystore.key: "${directory:/opt/kafka/external-configuration/my-
user:user.key}"
  #...
```

12.2.72.3. ExternalConfiguration スキーマプロパティ

プロパティ	説明
env	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。
ExternalConfigurationEnv array	
volumes	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。
ExternalConfigurationVolumeSource array	

12.2.73. ExternalConfigurationEnv スキーマ参照

[ExternalConfiguration](#) で使用

プロパティ	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	

12.2.74. ExternalConfigurationEnvVarSource スキーマ参照

[ExternalConfigurationEnv](#) で使用

プロパティ	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	

12.2.75. ExternalConfigurationVolumeSource スキーマ参照

[ExternalConfiguration](#) で使用

プロパティ	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	
secret	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。
SecretVolumeSource	

12.2.76. Build スキーマ参照

使用先: [KafkaConnectSpec](#)

Build 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を設定します。

12.2.76.1. 出力

追加のコネクタプラグインで新しいコンテナ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レジストリーが AMQ Streams に必要です。AMQ Streams は独自のコンテナ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ーは、**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。**output** 設定は必須で、**docker** と **imagestream** の 2 つのタイプをサポートします。

Docker レジストリーの使用

Docker レジストリーを使用するには、**type** を **docker** として指定し、**image** 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)
- 新しいコンテナイメージのタグ

有効なコンテナイメージ名の例:

- **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。

レジストリーに認証が必要な場合は、**pushSecret** を使用してレジストリーのクレデンシャルで Secret の名前を設定します。Secret には、`kubernetes.io/dockerconfigjson` タイプと `.dockerconfigjson` ファイルを使用して Docker 認証情報を追加します。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。

output 設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      type: docker ❶
      image: my-registry.io/my-org/my-connect-cluster:latest ❷
      pushSecret: my-registry-credentials ❸
  #...
```

- ❶ (必須) AMQ Streams によって使用される出力のタイプ。
- ❷ (必須) リポジトリとタグを含む、使用されるイメージのフルネーム。
- ❸ (任意) コンテナ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** プロパティを使用して ImageStream と使用するタグの名前を指定します。例: `my-connect-image-stream:latest`

output 設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      type: imagestream ❶
      image: my-connect-build:latest ❷
  #...
```

- 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。

12.2.76.2. plugins

コネクタプラグインは、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。各コネクタプラグインは、異なるコネクタ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持つ plugins の設定例

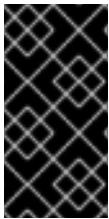
```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins: 1
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb548045e115e
33a9e69b1b2a352bee24df035a0447cb820077af00c03
          - 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:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e8020bf4
187870699819f54ef5859c7846ee4081507f48873479
            #...
```

- 1 (必須) コネクタプラグインおよびそれらのアーティファクトのリスト。

AMQ Streams では、以下のタイプのアーティファクトがサポートされます。

- 直接ダウンロードして使用する JAR ファイル
- ダウンロードおよび解凍された TGZ アーカイブ

- ダウンロードおよび解凍された ZIP アーカイブ
- Maven コーディネートを使用する Maven アーティファクト
- 直接ダウンロードおよび使用されるその他のアーティファクト



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** プロパティーを **jar** に設定し、**url** プロパティーを使用してダウンロードする場所を指定します。

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設定される場合でも、TGZ アーティファクトに Kafka Connect コネクタ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デプロイメントされます。TGZ アーティファクトを使用するには、**type** プロパティを **tgz** に設定し、**url** プロパティを使用してダウンロードする場所を指定します。

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、デプロイメント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 ❶
          url: https://my-domain.tld/my-connector-archive.tgz ❷
          sha512sum: 158...jg10 ❸
  #...
```

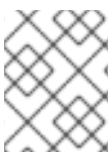
- ❶ (必須) アーティファクトのタイプ。
- ❷ (必須) アーカイブのダウンロード元 URL。
- ❸ (任意) アーティファクトを検証する SHA-512 チェックサム。

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 リポジトリから検索および取得できるようにします。



注記

コネクタビルドプロセスがアーティファクトをコンテナイメージに追加するには、Maven リポジトリへのアクセス権が必要です。

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 ❶
      repository: https://mvnrepository.com ❷
      group: org.apache.camel.kafkaconnector ❸
      artifact: camel-kafka-connector ❹
      version: 0.11.0 ❺
  #...

```

- ❶ (必須) アーティファクトのタイプ。
- ❷ (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。リポジトリを指定しないと、デフォルトで [Maven Central リポジトリ](#) が使用されます。
- ❸ (必須) Maven グループ ID。
- ❹ (必須) Maven アーティファクトタイプ。
- ❺ (必須) Maven バージョン番号。

other アーティファクトの使用

other アーティファクトは、コンテナイメージにダウンロードおよび追加されたファイルの種類を表します。結果とな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する場合は、**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。

other 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 ❶

```



```
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 (任意) 結果となるコンテナイメージに保存されるファイルの名前。

12.2.76.3. Build スキーマのプロパティ

プロパティ	説明
output	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、指定のオブジェクト内の output.type プロパティの値によって異なり、[docker、imagestream] のいずれかでなければなりません。
DockerOutput , ImageStreamOutput	
resources	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。
ResourceRequirements	
plugins	Kafka Connect に追加する必要があるコネクタプラグインのリスト。必須。
Plugin アレイ	

12.2.77. DockerOutput スキーマ参照

Build で使用

type プロパティは、**DockerOutput** タイプの使用を、**ImageStreamOutput** と区別するための識別子です。**DockerOutput** タイプには **docker** の値が必要です。

プロパティ	説明
image	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: quay.io/my-organization/my-custom-connect:latest 必須。
string	
pushSecret	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。
string	

プロパティ	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 でなければなりません。
string	

12.2.78. ImageStreamOutput スキーマ参照

Build で使用

type プロパティは、**InlineLogging** タイプの使用と、**DockerOutput** を区別するための識別子です。**ImageStreamOutput** タイプには **imagestream** の値が必要です。

プロパティ	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: my-custom-connect:latest 必須。
string	
type	ImageStream の例
string	

12.2.79. Plugin スキーマ参照

Build で使用

プロパティ	説明
-------	----

プロパティ	説明
name	コネクタプラグインの一意名。コネクタ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。この名前は、 <code>^[a-z][-_a-z0-9]*[a-z]\$</code> のパターンに従う必要があります。必須。
string	
artifacts	このコネクタプラグインに属するアーティファクトのリスト。必須。
JarArtifact , TgzArtifact , ZipArtifact , MavenArtifact , OtherArtifact アレイ	

12.2.80. JarArtifact スキーマ参照

[Plugin](#) で使用

プロパティ	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スキャンは行いません。セキュリティ上の理由から、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 および other アーティファクトで必要です。 maven アーティファクトタイプには該当しません。
string	
sha512sum	アーティファクトの SHA512 チェックサム。オプション:指定すると、新しいコンテナ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該当しません。
string	
insecure	デフォルトでは、TLS を使用する接続を検証して安全かどうかを確認します。使用するサーバー証明書は有効で信頼でき、サーバー名が含まれる必要があります。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、サーバーが安全ではないと見なされる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	jar でなければなりません。
string	

12.2.81. TgzArtifact スキーマ参照

Plugin で使用

プロパティ	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スキャンは行いません。セキュリティ上の理由から、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 および other アーティファクトが必要です。 maven アーティファクトタイプには該当しません。
string	
sha512sum	アーティファクトの SHA512 チェックサム。オプション:指定すると、新しいコンテナ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該当しません。
string	
insecure	デフォルトでは、TLS を使用する接続を検証して安全かどうかを確認します。使用するサーバー証明書は有効で信頼でき、サーバー名が含まれる必要があります。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、サーバーが安全ではないと見なされる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	

12.2.82. ZipArtifact スキーマ参照

Plugin で使用

プロパティ	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スキャンは行いません。セキュリティ上の理由から、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 および other アーティファクトが必要です。 maven アーティファクトタイプには該当しません。
string	

プロパティ	説明
sha512sum	アーティファクトのSHA512 チェックサム。オプション:指定すると、新しいコンテナ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該当しません。
string	
insecure	デフォルトでは、TLS を使用する接続を検証して安全かどうかを確認します。使用するサーバー証明書は有効で信頼でき、サーバー名が含まれる必要があります。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、サーバーが安全ではないと見なされる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 でなければなりません。
string	

12.2.83. MavenArtifact スキーマ参照

Plugin で使用

type プロパティは、**MavenArtifact** タイプと **JarArtifact**, **TgzArtifact**, **ZipArtifact**, **OtherArtifact** の使用を区別するための識別子です。**MavenArtifact** タイプには **maven** の値が必要です。

プロパティ	説明
repository	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。 maven アーティファクトタイプにのみ適用されます。
string	
group	Maven グループ ID。 maven アーティファクトタイプにのみ適用されます。
string	
artifact	Maven artifact 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	

プロパティ	説明
type	maven でなければなりません。
string	

12.2.84. OtherArtifact スキーマ参照

Plugin で使用

プロパティ	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スキャンは行いません。セキュリティ上の理由から、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 および other アーティファクトが必要です。 maven アーティファクトタイプには該当しません。
string	
sha512sum	アーティファクトの SHA512 チェックサム。オプション:指定すると、新しいコンテナ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該当しません。
string	
fileName	保存されるアーティファクトの名前。
string	
insecure	デフォルトでは、TLS を使用する接続を検証して安全かどうかを確認します。使用するサーバー証明書は有効で信頼でき、サーバー名が含まれる必要があります。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、サーバーが安全ではないと見なされる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	other でなければなりません。
string	

12.2.85. KafkaConnectStatus スキーマ参照

KafkaConnect で使用

プロパティ	説明
conditions	ステータス条件のリスト。
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プラグインのリスト。
ConnectorPlugin array	
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	

12.2.86. ConnectorPlugin スキーマ参照

使用先: [KafkaConnectStatus](#)、[KafkaMirrorMaker2Status](#)

プロパティ	説明
type	コネクタ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。
string	
version	コネクタプラグインのバージョン。
string	
class	コネクタプラグインのクラス。
string	

12.2.87. KafkaTopic スキーマ参照

プロパティ	説明
spec	トピックの仕様。
KafkaTopicSpec	
status	トピックのステータス。
KafkaTopicStatus	

12.2.88. KafkaTopicSpec スキーマ参照

[KafkaTopic](#) で使用

プロパティ	説明
partitions	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。これがない場合、デフォルトは num.partitions のブローカー設定になります。
integer	
replicas	トピックのレプリカ数。これがない場合、デフォルトは default.replication.factor のブローカー設定になります。
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 <code>metadata.name</code> 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	

12.2.89. KafkaTopicStatus スキーマ参照

[KafkaTopic](#) で使用

プロパティ	説明
conditions	ステータス条件のリスト。
Condition array	

プロパティ	説明
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	

12.2.90. KafkaUser スキーマ参照

プロパティ	説明
spec	ユーザーの仕様。
KafkaUserSpec	
status	Kafka User のステータス。
KafkaUserStatus	

12.2.91. KafkaUserSpec スキーマ参照

[KafkaUser](#) で使用

プロパティ	説明
-------	----

プロパティ	説明
<p>authentication</p> <p>KafkaUserTlsClientAuthentication, KafkaUserTlsExternalClientAuthentication, KafkaUserScramSha512ClientAuthentication</p>	<p>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。サポートされる認証メカニズムは、scram-sha-512、tls、および tls-external です。</p> <ul style="list-style-type: none"> ● SCRAM-sha-512 は、SASL SCRAM-SHA-512 認証情報でシークレットを生成します。 ● TLS は、相互 TLS 認証のユーザー証明書でシークレットを生成します。 ● TLS-external はユーザー証明書を生成しません。ただし、User Operator の外部で生成されるユーザー証明書を使用して相互 TLS 認証を使用するようにユーザーを準備します。このユーザーに設定された ACL およびクォータは CN=<username> 形式で設定されます。 <p>認証はオプションです。認証が設定されていない場合には、認証情報は生成されません。ユーザーに設定された ACL およびクォータは、SASL 認証に適した <username> 形式で設定されます。タイプは、指定のオブジェクト内の authentication.type プロパティの値によって異なり、[tls, tls-external, scram-sha-512] のいずれかでなければなりません。</p>
<p>認可</p> <p>KafkaUserAuthorizationSimple</p>	<p>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 プロパティの値によって異なり、[simple] の1つでなければなりません。</p>
<p>quotas</p> <p>KafkaUserQuotas</p>	<p>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。</p>
<p>template</p> <p>KafkaUserTemplate</p>	<p>Kafka User Secrets の生成方法を指定するテンプレート。</p>

12.2.92. KafkaUserTlsClientAuthentication スキーマ参照

KafkaUserSpec で使用

type プロパティは **KafkaUserTlsClientAuthentication** タイプと、**KafkaUserTlsExternalClientAuthentication**, **KafkaUserScramSha512ClientAuthentication** の使用を区別するための識別子です。**KafkaUserTlsClientAuthentication** タイプには **tls** の値が必要です。

プロパティ	説明
type	tls でなければなりません。
string	

12.2.93. KafkaUserTlsExternalClientAuthentication スキーマ参照

KafkaUserSpec で使用

type プロパティは **KafkaUserTlsExternalClientAuthentication** タイプと **KafkaUserTlsClientAuthentication**, **KafkaUserScramSha512ClientAuthentication** の使用を区別するための識別子です。**KafkaUserTlsExternalClientAuthentication** タイプには **tls-external** の値が必要です。

プロパティ	説明
type	tls-external でなければなりません。
string	

12.2.94. KafkaUserScramSha512ClientAuthentication スキーマ参照

KafkaUserSpec で使用

type プロパティは **KafkaUserScramSha512ClientAuthentication** タイプと **KafkaUserTlsClientAuthentication**, **KafkaUserTlsExternalClientAuthentication** の使用を区別するための識別子です。**KafkaUserScramSha512ClientAuthentication** タイプには **scram-sha-512** の値が必要です。

プロパティ	説明
password	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、新規パスワードは User Operator によって生成されます。
Password	
type	scram-sha-512 でなければなりません。
string	

12.2.95. Password スキーマ参照

使用先: **KafkaUserScramSha512ClientAuthentication**

プロパティ	説明
valueFrom	パスワードを読み取る必要のあるシークレット。
PasswordSource	

12.2.96. PasswordSource スキーマ参照

使用先: [Password](#)

プロパティ	説明
secretKeyRef	リソースの namespace で 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。
SecretKeySelector	

12.2.97. KafkaUserAuthorizationSimple スキーマ参照

[KafkaUserSpec](#) で使用

type プロパティは、**KafkaUserAuthorizationSimple** タイプを使用する際に、今後追加される可能性のある他のサブタイプと区別する識別子です。**KafkaUserAuthorizationSimple** タイプには **simple** の値が必要です。

プロパティ	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 ルールのリスト。
AclRule array	

12.2.98. 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** を使用する場合に **KafkaUser** のアクセス制御ルールを設定します。

認証を使用した 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

```

12.2.98.1. resource

resource プロパティを使用して、ルールが適用されるリソースを指定します。

簡易承認は、**type** プロパティに指定される、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 (**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。
- 接頭辞名では、**name** 値を接頭辞として使用し、名前がその値で始まるすべてのリソースにルールを適用します。

patternType が **literal** として設定されている場合、名前を * に設定して、ルールがすべてのリソースに適用されるように指示します。

ユーザーがすべてのトピックからメッセージを読み込める ACL ルールの例

```

acls:
  - resource:

```

```
type: topic
name: "*"
patternType: literal
operation: Read
```

12.2.98.2. type

ルールの **type**。 **allow** (操作の許可) または **deny** (操作の拒否、現在未サポート) です。

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。

12.2.98.3. operation

ルールが許可または拒否する **operation** 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るリソースと操作の組み合わせの詳細は、[Authorization and ACL](#) を参照してください。

12.2.98.4. host

host プロパティを使用して、ルールが許可または拒否され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。

12.2.98.5. AclRule スキーマのプロパティ

プロパティ	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 プロパティの値によって異なり、[topic、group、cluster、transactionalld] のいずれかでなければなりません。
AcIRuleTopicResource 、 AcIRuleGroupResource 、 AcIRuleClusterResource 、 AcIRuleTransactionalldResource	
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)	

12.2.99. AcIRuleTopicResource スキーマ参照

AcIRule で使用

type プロパティは、**AcIRuleTopicResource** タイプを使用する際に **AcIRuleGroupResource**、**AcIRuleClusterResource**、**AcIRuleTransactionalldResource** タイプと区別する識別子です。**AcIRuleTopicResource** タイプには **topic** の値が必要です。

プロパティ	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	
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)	

12.2.100. AclRuleGroupResource スキーマ参照

AclRule で使用

type プロパティは、**AclRuleGroupResource** タイプを使用する際に **AclRuleTopicResource**、**AclRuleClusterResource**、**AclRuleTransactionalIdResource** タイプと区別する識別子です。**AclRuleGroupResource** タイプには **group** の値が必要です。

プロパティ	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)	

12.2.101. AclRuleClusterResource スキーマ参照

AclRule で使用

type プロパティは、**AclRuleClusterResource** タイプを使用する際に **AclRuleTopicResource**、**AclRuleGroupResource**、**AclRuleTransactionalIdResource** タイプと区別する識別子です。**AclRuleClusterResource** タイプには **cluster** の値が必要です。

プロパティ	説明
type	cluster でなければなりません。
string	

12.2.102. AclRuleTransactionalIdResource スキーマ参照

AclRule で使用

type プロパティは、**AclRuleTransactionalIdResource** タイプを使用する際に **AclRuleTopicResource**、**AclRuleGroupResource**、**AclRuleClusterResource** タイプと区別する識別子です。**AclRuleTransactionalIdResource** タイプには **transactionalId** の値が必要です。

プロパティ	説明
type	transactionalId 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。
string ([prefix, literal] のいずれか)	

12.2.103. 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、ユーザーは **quotas** を設定してクライアントによるリソースの使用を制御できます。

12.2.103.1. 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 クラスタが同時にトピック操作に圧倒されないようになります。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ォータを設定して、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 クラスタ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。

12.2.103.2. KafkaUserQuotas スキーマ参照

プロパティ	説明
consumerByteRate	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) のクォータ。ブローカーごとに定義されます。
integer	
controllerMutationRate	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 of クォータ。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。
number	
producerByteRate	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) のクォータ。ブローカーごとに定義されます。
integer	
requestPercentage	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。
integer	

12.2.104. KafkaUserTemplate スキーマ参照

[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](#) を示す例

```

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
# ...

```

12.2.104.1. KafkaUserTemplate スキーマのプロパティ

プロパティ	説明
secret	KafkaUser リソースのテンプレート。テンプレートを使用すると、ユーザーはパスワードまたは TLS 証明書のある Secret の生成方法を指定できます。
ResourceTemplate	

12.2.105. KafkaUserStatus スキーマ参照

[KafkaUser](#) で使用

プロパティ	説明
conditions	ステータス条件のリスト。
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	

12.2.106. KafkaMirrorMaker スキーマ参照

KafkaMirrorMaker タイプ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	説明
spec	Kafka MirrorMaker の仕様。
KafkaMirrorMakerSpec	
status	Kafka MirrorMaker のステータス。
KafkaMirrorMakerStatus	

12.2.107. KafkaMirrorMakerSpec スキーマ参照

KafkaMirrorMaker で使用

KafkaMirrorMakerSpec スキーマプロパティの完全リスト

Kafka MirrorMaker を設定します。

12.2.107.1. include

include プロパティを使用して、Kafka MirrorMaker がソースからターゲット Kafka クラスターにミラーリングするトピックのリストを設定します。

このプロパティでは、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。たとえば、**A|B** を使用してトピック A と B をミラーリングでき、***** を使用してすべてのトピックをミラーリングできます。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。

12.2.107.2. KafkaMirrorMakerConsumerSpec および KafkaMirrorMakerProducerSpec

KafkaMirrorMakerConsumerSpec および **KafkaMirrorMaker ProducerSpec** を使用して、ソース (コンシューマー) およびターゲット (プロデューサー) クラスターを設定します。

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、ソースおよびターゲット Kafka クラスターのブートストラップサーバーは **HOSTNAME:PORT** ペアのコンマ区切りリストとして指定されます。それぞれのコンマ区切りリストには、**HOSTNAME:PORT** ペアとして指定された 1 つ以上の Kafka ブローカーまたは Kafka ブローカーを示す 1 つの **Service** が含まれます。

12.2.107.3. logging

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、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` プロパティを使用して 有効 (または無効) にすることもできます。

12.2.107.4. KafkaMirrorMakerSpec スキーマプロパティ

プロパティ	説明
version	Kafka MirrorMaker のバージョン。デフォルトは 3.2.3 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。
string	
replicas	Deployment の Pod 数。
integer	

プロパティ	説明
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	whitelist プロパティは非推奨となり、 spec.include を使用して設定する必要があります。ミラーリングに含まれるトピックのリスト。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。式 ' A B ' を使用して、A と B という名前の 2 つのトピックをミラーリングすることができます。または、特殊なケースとして、正規表現 * を使用してすべてのトピックをミラーリングできます。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リスト。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。式 ' A B ' を使用して、A と B という名前の 2 つのトピックをミラーリングすることができます。または、特殊なケースとして、正規表現 * を使用してすべてのトピックをミラーリングできます。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	説明
metricsConfig	メトリックの設定。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	
tracing	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 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。
KafkaMirrorMakerTemplate	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	

12.2.108. KafkaMirrorMakerConsumerSpec スキーマ参照

KafkaMirrorMakerSpec で使用

KafkaMirrorMakerConsumerSpec スキーマプロパティの完全リスト

MirrorMaker コンシューマーを設定します。

12.2.108.1. numStreams

consumer.numStreams プロパティを使用して、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。

12.2.108.2. offsetCommitInterval

consumer.offsetCommitInterval プロパティを使用して、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。

12.2.108.3. config

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 コンシューマー設定オプションが鍵として含まれ、値は以下の JSON タイプのいずれかに設定されます。

- 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 ID
- 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.** (特定の例外は除外)
- **sasl.**
- **security.**

禁止されているオプションが **config** プロパティにある場合、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** オブジェクトの設定を修正し、Cluster Operator によって Kafka MirrorMaker の新しい設定がロールアウトされるようにします。

12.2.108.4. groupId

consumer.groupId プロパティを使用して、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。

12.2.108.5. KafkaMirrorMakerConsumerSpec スキーマプロパティ

プロパティ	説明
numStreams	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。
integer	
offsetCommitInterval	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。
integer	
bootstrapServers	Kafka クラスターへの最初の接続を確立するための host:port ペアのリスト。
string	
groupId	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。
string	
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 プロパティの値によって異なり、[tls, scram-sha-256, scram-sha-512, plain, oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls, KafkaClientAuthenticationScramSha256, KafkaClientAuthenticationScramSha512, KafkaClientAuthenticationPlain, KafkaClientAuthenticationOAuth	

プロパティ	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	

12.2.109. KafkaMirrorMakerProducerSpec スキーマ参照

KafkaMirrorMakerSpec で使用

KafkaMirrorMakerProducerSpec スキーマプロパティの完全リスト

MirrorMaker プロデューサーを設定します。

12.2.109.1. abortOnSendFailure

producer.abortOnSendFailure プロパティを使用して、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 コンテナが OpenShift で終了します。
- その後、コンテナが再作成されます。

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。

12.2.109.2. config

producer.config プロパティを使用して、プロデューサーの 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鍵として含まれ、値は以下の JSON タイプのいずれかに設定されます。

- 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。

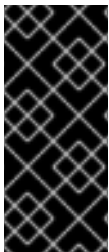
しかし、以下に関連する AMQ Streams によって自動的に設定され、直接管理されるオプションには例外があります。

- 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.** (特定の例外は除外)
- **sasl.**
- **security.**

禁止されているオプションが **config** プロパティにある場合、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** オブジェクトの設定を修正し、Cluster Operator によって Kafka MirrorMaker の新しい設定がロールアウトされるようにします。

12.2.109.3. KafkaMirrorMakerProducerSpec スキーマプロパティ

プロパティ	説明
bootstrapServers	Kafka クラスタへの最初の接続を確立するための host:port ペアのリスト。
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	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 プロパティの値によって異なり、[tls, scram-sha-256, scram-sha-512, plain, oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls , KafkaClientAuthenticationScramSha256 , 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	

12.2.110. KafkaMirrorMakerTemplate スキーマ参照

[KafkaMirrorMakerSpec](#) で使用

プロパティ	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	

12.2.111. KafkaMirrorMakerStatus スキーマ参照

[KafkaMirrorMaker](#) で使用

プロパティ	説明
conditions	ステータス条件のリスト。
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	

12.2.112. KafkaBridge スキーマ参照

プロパティ	説明
spec	Kafka Bridge の仕様。
KafkaBridgeSpec	
status	Kafka Bridge のステータス。
KafkaBridgeStatus	

12.2.113. 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

12.2.113.1. logging

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>**

logger.<operation-id> ロガーの **<operation-id>** を置き換えると、特定の操作のログレベルを設定できます。

- **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。

各ロガーはその名前を **http.openapi.operation.<operation-id>** として割り当てる必要があります。たとえば、**send** 操作ロガーのロギングレベルを設定すると、以下が定義されます。

```
logger.send.name = http.openapi.operation.send
logger.send.level = DEBUG
```

Kafka Bridge では Apache **log4j2** ロガー実装が使用されます。ロガーは **log4j2.properties** ファイルで定義されます。このファイルには **healthy** および **ready** エンドポイントの以下のデフォルト設定が含まれています。

```
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、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** および **logging.valueFrom.configMapKeyRef.key** プロパティはいずれも必須です。**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

```
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更新がトリガーされます。

ガベージコレクター (GC)

ガベージコレクターのロギングは **jvmOptions** プロパティを使用して有効 (または無効) にすることもできます。

12.2.113.2. KafkaBridgeSpec スキーマプロパティ

プロパティ	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 ペアのリスト。
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 プロパティの値によって異なり、[tls, scram-sha-256, scram-sha-512, plain, oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls, KafkaClientAuthenticationScramSha256, KafkaClientAuthenticationScramSha512, KafkaClientAuthenticationPlain, KafkaClientAuthenticationOAuth	
http	HTTP 関連の設定。
KafkaBridgeHttpConfig	
adminClient	Kafka AdminClient 関連の設定。
KafkaBridgeAdminClientSpec	
consumer	Kafka コンシューマーに関連する設定。
KafkaBridgeConsumerSpec	

プロパティ	説明
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。
JvmOptions	
logging	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	
enableMetrics	Kafka Bridge のメトリックを有効にします。デフォルトは false です。
boolean	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	
template	Kafka Bridge リソースのテンプレート。テンプレートを使用すると、ユーザーは Deployment と Pod の生成方法を指定できます。
KafkaBridgeTemplate	
tracing	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 プロパティの値によって異なり、[jaeger] の1つでなければなりません。
JaegerTracing	

12.2.114. KafkaBridgeHttpConfig スキーマ参照

[KafkaBridgeSpec](#) で使用

[KafkaBridgeHttpConfig](#) スキーマプロパティの完全リスト

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。

12.2.114.1. cors

HTTP プロパティは、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"
  # ...
```

12.2.114.2. KafkaBridgeHttpConfig スキーマプロパティ

プロパティ	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	

12.2.115. KafkaBridgeHttpCors スキーマ参照

[KafkaBridgeHttpConfig](#) で使用

プロパティ	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。
string array	
allowedMethods	許可される HTTP メソッドのリスト。
string array	

12.2.116. KafkaBridgeAdminClientSpec スキーマプロパティ

[KafkaBridgeSpec](#) で使用

プロパティ	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	

12.2.117. KafkaBridgeConsumerSpec スキーマ参照

[KafkaBridgeSpec](#) で使用

[KafkaBridgeConsumerSpec](#) スキーマプロパティの完全リスト

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 に渡されます。



重要

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 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
    # ...

```

12.2.117.1. KafkaBridgeConsumerSpec スキーマプロパティ

プロパティ	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	

12.2.118. KafkaBridgeProducerSpec スキーマ参照

[KafkaBridgeSpec](#) で使用

[KafkaBridgeProducerSpec](#) スキーマプロパティの完全リスト

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。

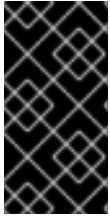
- 文字列
- 数値
- ブール値

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 に渡されます。



重要

`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 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
    # ...
```

12.2.118.1. KafkaBridgeProducerSpec スキーマプロパティ

プロパティ	説明
<code>config</code>	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は設定できません: <code>ssl</code> 、 <code>bootstrap.servers</code> 、 <code>sasl</code> 、 <code>security</code> 。(次の例外を除く: <code>ssl.endpoint.identification.algorithm</code> 、 <code>ssl.cipher.suites</code> 、 <code>ssl.protocol</code> 、 <code>ssl.enabled.protocols</code>).
<code>map</code>	

12.2.119. KafkaBridgeTemplate スキーマ参照

KafkaBridgeSpec で使用

プロパティ	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。
PodDisruptionBudgetTemplate	
bridgeContainer	Kafka Bridge コンテナのテンプレート。
ContainerTemplate	
serviceAccount	Kafka Bridge サービスアカウントのテンプレート。
ResourceTemplate	

12.2.120. KafkaBridgeStatus スキーマ参照

KafkaBridge で使用

プロパティ	説明
conditions	ステータス条件のリスト。
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	

プロパティ	説明
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	

12.2.121. KafkaConnector スキーマ参照

プロパティ	説明
spec	Kafka Connector の仕様。
KafkaConnectorSpec	
status	Kafka Connector のステータス。
KafkaConnectorStatus	

12.2.122. KafkaConnectorSpec スキーマ参照

[KafkaConnector](#) で使用

プロパティ	説明
class	Kafka Connector のクラス。
string	
tasksMax	Kafka Connector のタスクの最大数。
integer	
config	Kafka Connector の設定。次のプロパティは設定できません: connector.class、tasks.max
map	
pause	コネクタを一時停止すべきかどうか。デフォルトは false です。
boolean	

12.2.123. KafkaConnectorStatus スキーマ参照

[KafkaConnector](#) で使用

プロパティ	説明
conditions	ステータス条件のリスト。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
connectorStatus	Kafka Connect REST API によって報告されるコネクタのステータス。
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。
string array	

12.2.124. KafkaMirrorMaker2 スキーマ参照

プロパティ	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	

12.2.125. KafkaMirrorMaker2Spec スキーマ参照

[KafkaMirrorMaker2](#) で使用

プロパティ	説明
-------	----

プロパティ	説明
version	Kafka Connect のバージョン。デフォルトは 3.2.3 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。
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 にあるリスト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	

プロパティ	説明
logging	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	
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	
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	
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 および Kafka Mirror Maker 2 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 、 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を設定します。
ExternalConfiguration	
metricsConfig	メトリックの設定。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	

12.2.126. KafkaMirrorMaker2ClusterSpec スキーマ参照

[KafkaMirrorMaker2Spec](#) で使用

[KafkaMirrorMaker2ClusterSpec](#) スキーマプロパティの完全リスト

ミラーリング用の Kafka クラスタを設定します。

12.2.126.1. config

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。

12.2.126.2. KafkaMirrorMaker2ClusterSpec スキーマプロパティ

プロパティ	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 プロパティの値によって異なり、[tls, scram-sha-256, scram-sha-512, plain, oauth] のいずれかでなければなりません。
KafkaClientAuthenticationTls, KafkaClientAuthenticationScramSha256, 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は設定できません: ssl.、sas.、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	

12.2.127. KafkaMirrorMaker2MirrorSpec スキーマ参照

KafkaMirrorMaker2Spec で使用

プロパティ	説明
sourceCluster	Kafka MirrorMaker 2.0 コネクタ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 にあるリストのクラスターと一致する必要があります。
string	

プロパティ	説明
targetCluster	Kafka MirrorMaker 2.0 コネクタ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 にあるリストのクラスターと一致する必要があります。
string	
sourceConnector	Kafka MirrorMaker 2.0 ソースコネクタの仕様。
KafkaMirrorMaker2ConnectorSpec	
heartbeatConnector	Kafka MirrorMaker 2.0 ハートビートコネクタの仕様。
KafkaMirrorMaker2ConnectorSpec	
checkpointConnector	Kafka MirrorMaker 2.0 チェックポイントコネクタの仕様。
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 プロパティは非推奨となり、 .spec.mirrors.topicsExcludePattern を使用して設定する必要があります。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り、 .spec.mirrors.groupsExcludePattern を使用して設定する必要があります。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	

12.2.128. KafkaMirrorMaker2ConnectorSpec スキーマ参照

KafkaMirrorMaker2MirrorSpec で使用

プロパティ	説明
tasksMax	Kafka Connector のタスクの最大数。
integer	
config	Kafka Connector の設定。次のプロパティは設定できません: connector.class、tasks.max
map	
pause	コネクタを一時停止すべきかどうか。デフォルトは false です。
boolean	

12.2.129. KafkaMirrorMaker2Status スキーマ参照

KafkaMirrorMaker2 で使用

プロパティ	説明
conditions	ステータス条件のリスト。
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プラグインのリスト。
ConnectorPlugin array	
connectors	Kafka Connect REST API によって報告される MirrorMaker 2.0 コネクタステータスのリスト。
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	説明
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	

12.2.130. KafkaRebalance スキーマ参照

プロパティ	説明
spec	Kafka のリバランス (再分散) の仕様。 KafkaRebalanceSpec
status	
KafkaRebalanceStatus	Kafka のリバランス (再分散) のステータス。

12.2.131. KafkaRebalanceSpec スキーマ参照

[KafkaRebalance](#) で使用

プロパティ	説明
モード	リバランスを実行するモード。サポートされているモードは full 、 add-brokers 、 remove-brokers です。指定しない場合、 full モードがデフォルトで使用されます。 <ul style="list-style-type: none"> ● full モードでは、クラスター内のすべてのブローカーでリバランスが実行されます。 ● add-brokers モードは、クラスターをスケールアップした後に使用して、一部のレプリカを新しく追加されたブローカーに移動できます。 ● remove-brokers モードは、クラスターをスケールダウンして削除するブローカーからレプリカを移動する前に使用できます。
文字列 (remove-brokers、full、add-brokers のいずれか)	
brokers	スケールアップの場合に新しく追加されたブローカーのリスト、またはリバランスに使用するためにスケールダウンの場合に削除されるブローカーのリスト。このリストは、リバランスモードの add-brokers および removed-brokers でのみ使用できます。これは、 full モードで無視されます。
整数配列	

プロパティ	説明
goals	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。
string array	
skipHardGoalCheck	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。
boolean	
rebalanceDisk	ブローカー内のディスク分散を有効にし、同じブローカーのディスク間でディスク領域の使用率を分散します。ディスクが複数割り当てられた JBOD ストレージを使用する Kafka デプロイメントにのみ適用されます。有効にすると、ブローカー間の分散は無効になります。デフォルトは false です。
boolean	
excludedTopics	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 <code>java.util.regex.Pattern</code> 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。
string	
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 <code>BaseReplicaMovementStrategy</code> が使用されます。
string array	

12.2.132. KafkaRebalanceStatus スキーマ参照

KafkaRebalance で使用

プロパティ	説明
conditions	ステータス条件のリスト。
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。
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. **My Subscriptions** 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、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** カテゴリで、**AMQ Streams for Apache Kafka** 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。

DNF を使用したパッケージのインストール

パッケージとすべてのパッケージ依存関係をインストールするには、以下を使用します。

```
dnf install <package_name>
```

ローカルディレクトリーからダウンロード済みのパッケージをインストールするには、以下を使用します。

```
dnf install <path_to_download_package>
```

改訂日時: 2023-10-30