Red Hat AMQ Streams 2.3

OpenShift での AMQ Streams の設定

OpenShift ContainerPlatform での AMQ Streams 2.3 のデプロイメントの設定および管理

Last Updated: 2023-04-10

Red Hat AMQ Streams 2.3 OpenShift での AMQ Streams の設定

OpenShift ContainerPlatform での AMQ Streams 2.3 のデプロイメントの設定および管理

法律上の通知

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

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

第1章 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ーを説明しています。

1.2. CONFIGMAP を使用した設定の追加

ConfigMap リソースを使用して、特定の設定を AMQ Streams デプロイメントに追加します。ConfigMap はキーと値のペアを使用して機密ではないデータを保存します。ConfigMap に追加された設定データは 1 か所に保持され、コンポーネント間で再利用できます。

ConfigMap は、以下に関連する設定データのみを保存できます。

- ロギングの設定
- メトリクスの設定
- Kafka Connect コネクターの外部設定

設定の他の領域に ConfigMap を使用することはできません。

コンポーネントを設定する場合、**configMapKeyRef** プロパティーを使用して ConfigMap への参照を追加できます。

たとえば、**configMapKeyRef** を使用してロギングの設定を提供する ConfigMap を参照できます。ConfigMap を使用して Log4j 設定ファイルを渡すことができます。参照を **logging** 設定に追加します。

ロギングの ConfigMap の例

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

メトリクス設定に ConfigMap を使用するには、同じ方法でコンポーネントの **metricsConfig** 設定への参照を追加します。

ExternalConfiguration プロパティーは、Pod にマウントされた ConfigMap (またはシークレット) からのデータを環境変数またはボリュームとして使用できるようにします。Kafka Connect によって使用されるコネクターの外部設定データを使用できます。データは外部データソースに関連する可能性があり、コネクターがそのデータソースと通信するために必要な値を指定します。

たとえば、**configMapKeyRef** プロパティーを使用して、ConfigMap から設定データを環境変数として渡すことができます。

環境変数の値を提供する ConfigMap 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
```

外部で管理される ConfigMap を使用している場合は、設定プロバイダーを使用して ConfigMap にデータを読み込みます。設定プロバイダーの使用の詳細は、3章*外部ソースからの設定値の読み込み* を参照してください。

1.2.1. カスタム ConfigMap の命名

AMQ Streams は、OpenShift にデプロイされると、独自の ConfigMap およびその他のリソースを作成します。ConfigMap には、コンポーネントの実行に必要なデータが含まれます。AMQ Streams によって作成された ConfigMap は編集しないでください。

作成するカスタム ConfigMap にはこれらのデフォルト ConfigMap と同じ名前がないことを確認します。名前が同じ場合は上書きされます。たとえば、ConfigMap が Kafka クラスターの ConfigMap と同じ名前である場合、Kafka クラスターの更新がある場合に上書きされます。

関連情報

- 「Kafka クラスターリソースのリスト」 (ConfigMap を含む)
- 「ロギングの設定」
- 「**metricsConfig**」
- 「**ExternalConfiguration** スキーマ参照」
- *3章外部ソースからの設定値の読み込み*

1.3. 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 ブローカーへのクライアント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ーを備えたジェネリックな **GenericKafkaListener** スキーマを提供しています。

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

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

1.3.1. 内部リスナーの設定

内部リスナーは、クライアントを OpenShift クラスター内の Kafka ブローカーに接続します。**internal** タイプのリスナー設定は、ヘッドレスサービスと、ブローカー Pod に指定された DNS 名を使用します。

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場合があります。その場合、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** タイプのリスナーを (**useServiceDnsDomain** プロパティーを使用して) 設定できます。

ブローカーごとの **ClusterIP** サービスに基づいて Kafka クラスターを公開する **cluster-ip** タイプのリスナーを設定することもできます。これは、ヘッドレスサービスを介してルーティングできない場合や、カスタムアクセスメカニズムを組み込みたい場合に便利なオプションです。たとえば、特定の Ingress コントローラーまたは OpenShift Gateway API 用に独自のタイプの外部リスナーを構築するときに、このリスナーを使用できます。

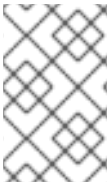
1.3.2. 外部リスナーの設定

異なる認証メカニズムを必要とするネットワークから Kafka クラスターへのアクセスを処理するように、外部リスナーを設定します。

ロードバランサーやルート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。

1.3.3. リスナー証明書の提供

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リングする場合、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。

関連情報

- 「**Generic KafkaListener** スキーマ参照」
- 5章*OpenShift クラスター外の Kafka へのアクセス*
- 「Kafka ブローカーへのアクセスのセキュア化」

1.4. 本書の表記慣例

ユーザーが置き換えた値

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
```

1.5. 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 OPENSHIFT 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クスの紹介 を参照してください。

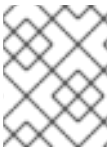
2.1. 標準の KAFKA 設定プロパティーの使用

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

プロパティーは、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
- トピック
- 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、承認、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ーは AMQ Streams によって完全に管理されており、変更できません。

Kafka 設定プロパティーの詳細と、プロパティー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 設定プロパティー
- Kafka 設定のチューニング

2.2. KAFKA クラスターの設定

Kafka リソースを使用して Kafka デプロイメントを設定します。Kafka クラスターは ZooKeeper クラスターと共にデプロイされるため、**Kafka** リソース内の ZooKeeper の設定オプションも使用できます。

Entity Operator は Topic Operator と User Operator で設定されます。**Kafka** リソースの **entityOperator** プロパティーを設定して、Topic Operator と User Operator をデプロイメントに含めることもできます。

「**Kafka** スキーマ参照」 **Kafka** リソースの完全なスキーマについて説明します。

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、更新期間の開始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ーを使用して、Kafka デプロイメントを設定します。

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

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

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

Kafka バージョン

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

Kafka 3.0.0 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** プロパティーを編集します。
設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    replicas: 3 (1)
    version: 3.3.1 (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: "3.3"
    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. コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するための ヘルスチェック。
6. Kafka を実行している仮想マシン (VM) のパフォーマンスを最適化するための JVM 設定オプション。
7. 高度な任意設定: 特別な場合のみ推奨される コンテナーイメージの設定。
8. リスナーは、ブートストラップアドレスでクライアントが Kafka クラスターに接続する方法を設定します。リスナーは、OpenShift クラスター内部または外部からの接続の **内部** または **外部** リスナーとして設定 されます。
9. リスナーを識別するための名前。Kafka クラスター内で一意である必要があります。
10. Kafka 内でリスナーによって使用されるポート番号。ポート番号は指定の Kafka クラスター内で一意である必要があります。許可されるポート番号は 9092 以上ですが、すでに Prometheus および JMX によって使用されているポート 9404 および 9999 以外になります。リスナーのタイプによっては、ポート番号は Kafka クライアントに接続するポート番号と同じではない場合があります。
11. **internal** または **cluster-ip** (ブローカーごとの **ClusterIP** サービスを使用して Kafka を公開するため) として指定されたリスナータイプ、または **route**、**loadbalancer**、**nodeport** または **ingress** として指定された外部リスナーのタイプ。
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. mTLS、SCRAM-SHA-512、またはトークンベースの OAuth 2.0 として指定された 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 CA (認証局) によって管理される Kafka リスナー証明書 のオプションの設定。**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ーに限り、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 プロパティー。

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

21 永続ボリュームのストレージサイズは拡張可能 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 メトリクス は有効になっています。この例では、メトリクスは Prometheus JMX Exporter (デフォルトのメトリクスエクスポーター) に対して設定されます。

25 Prometheus JMX Exporter 経由でメトリクスを Grafana ダッシュボードにエクスポートする Prometheus ルール。Prometheus JMX Exporter の設定が含まれる ConfigMap を参照することで有効になります。**metricsConfig.valueFrom.configMapKeyRef.key** 配下に空のファイルが含まれる ConfigMap の参照を使用して、追加設定なしでメトリクスを有効にできます。

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

27 ZooKeeper ノードの数。通常、ZooKeeper クラスターまたはアンサンブルは、一般的に 3、5、7 個の奇数個のノードで実行されます。効果的なクォーラムを維持するには、過半数のノードが利用可能である必要があります。ZooKeeper クラスターでクォーラムを失うと、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。Kafka Exporter が適切に機能できるようにするには、コンシューマーグループを使用する必要があります。

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。

2. 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ーを管理します。

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 の設定プロパティー

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

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

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

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

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

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

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

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

両方の Operator を有効にする基本設定の例

■

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

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

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

2.2.2.2. Topic Operator 設定プロパティー

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

watchedNamespace

Topic Operator が **KafkaTopic** リソースを監視する OpenShift 名前空間。デフォルトは、Kafka クラスターがデプロイされた namespace です。

reconciliationIntervalSeconds

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

zookeeperSessionTimeoutSeconds

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

topicMetadataMaxAttempts

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

image

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

resources

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

logging

logging プロパティーは、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 設定プロパティー

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

watchedNamespace

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

resources

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

logging

logging プロパティーは、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 プロパティーは、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 ストレージの設定

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します。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 (ZooKeeper ではなく **Kafka のみ**)

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** プロパティーを使用して、ストレージタイプを設定します。

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

ストレージタイプは **type** フィールドで設定されます。

ストレージ設定プロパティーの詳細は、スキーマ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 クラスターをデプロイした後に、ストレージタイプを変更することはできません。

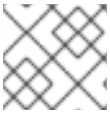
2.2.3.1. データストレージに関する留意事項

AMQ Streams がうまく機能するには、効率的なデータストレージインフラストラクチャーが不可欠です。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。
- ローカル永続ボリューム を使用した永続ストレージ

- **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

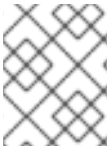
Kafka は、メッセージの保存にファイルシステムを使用します。AMQ Streams は、Kafka で一般的に使用される XFS および ext4 ファイルシステムと互換性があります。ファイルシステムを選択して設定するときは、デプロイメントの基盤となるアーキテクチャーと要件を考慮してください。

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. ディスク使用率

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

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



注記

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

2.2.3.2. 一時ストレージ

一時データストレージは一時的なものです。ノード上のすべての Pod は、ローカルの一時ストレージスペースを共有します。データは、それを使用する Pod が実行されている限り保持されます。Pod が削除されると、データは失われます。ただし、Pod は高可用性環境でデータを回復できます。

その一時的な性質のため、一時ストレージは開発とテストにのみ推奨されます。

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。Pod がノードに割り当てられると、**emptyDir** ボリュームが作成されます。**sizeLimit** プロパティーを使用して、**emptyDir** のストレージの合計量を設定できます。

重要

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。

一時ストレージを使用するには、**Kafka** または **ZooKeeper** リソースのストレージタイプ設定を **ephemeral** に設定します。

一時ストレージ設定の例

```
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. Kafka ログディレクトリーのマウントパス

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-log
```

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

永続的なデータストレージは、システムが中断した場合でもデータを保持します。永続的なデータストレージを使用する Pod の場合、データは Pod の障害や再起動後も保持されます。

動的プロビジョニングフレームワークにより、永続的なストレージを使用してクラスターを作成できます。Pod 設定では、永続ボリューム要求 (PVC) を使用して、永続ボリューム (PV) でストレージ要求を行います。PV は、ストレージボリュームを表すストレージリソースです。PV は、それを使用する Pod から独立しています。PVC は、Pod の作成時に必要なストレージの量を要求します。PV の基盤となるストレージインフラストラクチャーを理解する必要はありません。PV がストレージ基準に一致する場合、PVC は PV にバインドされます。

永続的な性質のため、本番環境には永続ストレージをお勧めします。

PVC は、StorageClass を指定することにより、さまざまなタイプの永続ストレージを要求できます。ストレージクラスはストレージプロファイルを定義し、PV を動的にプロビジョニングします。ストレージクラスが指定されていない場合、デフォルトのストレージクラスが使用されます。永続ストレージオプションには、SAN ストレージタイプまたは ローカル永続ボリューム が含まれる場合があります。

永続ストレージを使用するには、**Kafka** または **ZooKeeper** リソースのストレージタイプ設定を **persistent-claim** に設定します。

本番環境では、次の設定が推奨されます。

- Kafka の場合、**type: jbod** を 1 つ以上の **type: persistent-claim** ボリュームで設定します
- ZooKeeper の場合は、**type: persistent-claim** を設定します。

永続ストレージには、次の設定オプションもあります。

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 (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。ストレージ **class** の設定には、ボリュームのプロファイルを詳細に記述するパラメーターが含まれます。

selector (任意)

特定の PV を指定する設定。選択したボリュームのラベルを表す key:value ペアを提供します。

deleteClaim (任意)

クラスターのアンインストール時に PVC を削除す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。

Kafka と ZooKeeper の永続ストレージ設定の例

```
# ...
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:
    storage:
```

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

ストレージクラスを指定しない場合、デフォルトが使用されます。次の例では、ストレージクラスを指定します。

特定のストレージクラスを使用した永続ストレージ設定の例

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

selector を使用して、SSD などの特定の機能を提供するラベル付き永続ボリュームを指定します。

セレクターを使用した永続ストレージ設定の例

```
# ...
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ーゾーンやデータセンターに制限されている場合などに便利です。この場合、**overrides** フィールドを使用できます。

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

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  labels:
    app: my-cluster
  name: my-cluster
  namespace: myproject
spec:
  # ...
  kafka:
    replicas: 3
    storage:
      type: jbod
      volumes:
```

```
      - id: 0
        type: persistent-claim
        size: 100Gi
        deleteClaim: false
        class: my-storage-class
        overrides:
        - broker: 0
          class: my-storage-class-zone-1a
        - broker: 1
          class: my-storage-class-zone-1b
        - broker: 2
          class: my-storage-class-zone-1c
      # ...
  # ...
  zookeeper:
    replicas: 3
    storage:
      deleteClaim: true
      size: 100Gi
      type: persistent-claim
      class: my-storage-class
      overrides:
        - broker: 0
          class: my-storage-class-zone-1a
        - broker: 1
          class: my-storage-class-zone-1b
        - broker: 2
          class: my-storage-class-zone-1c
  # ...
```

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

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

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

2.2.3.3.2. 永続ストレージ用の PVC リソース

永続ストレージを使用すると、次の名前で PVC が作成されます。

data-cluster-name-kafka-idx

Kafka ブローカー Pod **idx** のデータを格納するために使用されるボリュームの PVC。

data-cluster-name-zookeeper-idx

ZooKeeper ノード Pod **idx** のデータを格納するために使用されるボリュームの PVC。

2.2.3.3.3. Kafka ログディレクトリーのマウントパス

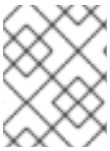
永続ボリュームは、以下のパスにマウントされるログディレクトリーとして Kafka ブローカーによって使用されます。

```
/var/lib/kafka/data/kafka-log**IDX**
```

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 が稼働中です。
- ボリューム拡張をサポートするストレージクラスを使用して作成された永続ボリュームを使用する Kafka クラスター。

手順

1. クラスターの **Kafka** リソースを編集します。
   size プロパティーを変更して、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 クラスターの場合は、**spec.kafka.storage** の下にある **size** プロパティーを更新します。
   - ZooKeeper クラスターの場合は、**spec.zookeeper.storage** の下にある **size** プロパティーを更新します。

   ボリュームサイズを 2000Gi に増やす Kafka 設定

```
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。

3. クラスター上の関連する Pod のストレージ容量が増加したことを確認します。

```
oc get pv
```

ストレージが増加した Kafka ブローカー Pod

```
NAME             CAPACITY   CLAIM
pvc-0ca459ce-...   2000Gi     my-project/data-my-cluster-kafka-2
pvc-6e1810be-...   2000Gi     my-project/data-my-cluster-kafka-0
pvc-82dc78c9-...   2000Gi     my-project/data-my-cluster-kafka-1
```

出力には、ブローカー Pod に関連付けられた各 PVC の名前が表示されます。

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

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。

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。

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、プロビジョニング後に永続ストレージのボリュームのサイズを縮小することはできません。また、タイプが **ephemeral** の場合は、**sizeLimit** の値を変更することはできません。

JBOD ストレージを使用するには、**Kafka** リソースのストレージタイプ設定を **jbod** に設定します。**volumes** プロパティーを使用すると、JBOD ストレージアレイまたは設定を設定するディスクを記述できます。

JBOD ストレージ設定の例

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

JBOD ボリュームが作成されると、ID を変更することはできません。JBOD 設定からボリュームを追加または削除できます。

2.2.3.5.1. JBOD ストレージの PVC リソース

永続ストレージを使用して JBOD ボリュームを宣言すると、次の名前の PVC が作成されます。

data-id-cluster-name-kafka-idx

Kafka ブローカー Pod **idx** のデータを格納するために使用されるボリュームの PVC。**id** は、Kafka ブローカー Pod のデータを格納するために使用されるボリュームの ID になります。

2.2.3.5.2. Kafka ログディレクトリーのマウントパス

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-logidx
```

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** プロパティーを編集します。新しいボリュームを **volumes** アレイに追加します。たとえば、id が **2** の新しいボリュームを追加します。

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

2. リソースを作成または更新します。

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

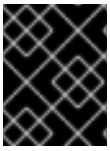
3. 新しいトピックを作成するか、既存のパーティションを新しいディスクに再度割り当てます。

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** プロパティーを編集します。**volumes** アレイから 1 つまたは複数のボリュームを削除します。たとえば、ID が **1** と **2** のボリュームを削除します。

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

3. リソースを作成または更新します。

```
oc apply -f <kafka_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 クラスターで JBOD ストレージを使用する場合は、特定のボリュームとログディレクトリー (各ボリュームに単一のログディレクトリーがある) との間でパーティション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。

このツールは、Kafka クラスターに接続された対話型 Pod コンテナーから実行します。

この手順では、mTLS 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 暗号化と mTLS 認証を使用する Kafka クラスターが必要です。

接続を確立するには、次のものが必要です。

- Kafka クラスターの作成時に Cluster Operator によって生成されたクラスター CA 証明書とパスワード
- ユーザーが Kafka クラスターへのクライアントアクセス用に作成されたときに User Operator によって生成されたユーザー CA 証明書とパスワード

この手順では、CA 証明書と対応するパスワードが、PKCS #12 (**.p12** および **.password**) 形式で含まれているクラスターとユーザーシークレットから抽出されます。パスワードは、証明書を含む **.p12** ストアへのアクセスを許可します。**.p12** ストアを使用してトラストストアとキーストアを指定し、Kafka クラスターへの接続を認証します。

前提条件

- Cluster Operator が実行中である。
- 内部 TLS 暗号化と mTLS 認証で設定された **Kafka** リソースに基づいて実行中の Kafka クラスターがあります。

  TLS 暗号化と mTLS 認証を使用した 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** として指定された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: 1
    type: tls
  authorization:
    type: simple 2
    acls:
      # access to the topic
      - resource:
          type: topic
          name: my-topic
        operations:
          - Create
          - Describe
          - Read
          - AlterConfigs
        host: "*"
      # access to the cluster
      - resource:
          type: cluster
        operations:
          - Alter
          - AlterConfigs
        host: "*"
      # ...
  # ...
```

1 相互 **tls** として定義されたユーザー認証メカニズム。

2 ACL ルールの承認および付随するリスト。

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 コンテナーを新たに実行し、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-33-rhel8:2.3.0 <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 コンテナーにコピーします。

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

4. Kafka ブローカーへのアクセス権限を持つ Kafka ユーザーのシークレットから、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 コンテナーにコピーします。

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

CA 証明書を使用すると、インタラクティブな Pod コンテナーが TLS を使用して Kafka ブローカーに接続できます。

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

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

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

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

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

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

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

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

9. **topics.json** ファイルをインタラクティブな Pod コンテナーにコピーします。

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

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

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

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

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

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

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

関連情報

- 「Kafka の設定」
- 「Kafka トピックの設定」
- 「Kafka ユーザーの設定」

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 暗号化と mTLS 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 暗号化と mTLS 認証で設定された **Kafka** リソースに基づいて実行中の Kafka クラスターがあります。
- **reassignment.json**という名前の再割り当て JSON ファイルを生成している。
- 実行中の Kafka ブローカーに接続されている対話型 Pod コンテナーを実行している。
- **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 コンテナーを実行 して **reassignment.json** という名前の再割り当てJSONファイルを生成します。
4. **reassignment.json** ファイルをインタラクティブな Pod コンテナーにコピーします。

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

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

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

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

6. インタラクティブな Pod コンテナーから **kafka-reassign-partitions.sh** スクリプトを使用して、パーティションの再割り当てを実行します。

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

<cluster_name> 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
```

7. ブローカー Pod のいずれかから **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了した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リングプロセスについて説明します。TLS 暗号化と mTLS 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 暗号化と mTLS 認証で設定された **Kafka** リソースに基づいて実行中の Kafka クラスターがあります。
- **reassignment.json**という名前の再割り当て JSON ファイルを生成している。
- 実行中の Kafka ブローカーに接続されている対話型 Pod コンテナーを実行している。
- **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 コンテナーを実行 して **reassignment.json** という名前の再割り当てJSONファイルを生成します。
2. **reassignment.json** ファイルをインタラクティブな Pod コンテナーにコピーします。

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

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

3. インタラクティブな Pod コンテナーでシェルプロセスを開始します。

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

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

4. インタラクティブな Pod コンテナーから **kafka-reassign-partitions.sh** スクリプトを使用して、パーティションの再割り当てを実行します。

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

-

<cluster_name> 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
```

5. ブローカー Pod のいずれかから **kafka-reassign-partitions.sh** コマンドラインツールを使用して、再割り当てが完了した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。

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

7. すべてのパーティションの再割り当てが終了すると、削除されるブローカーはクラスター内のいずれのパーティションにも対応しないはずです。これは、ブローカーのデータログディレクトリーにライブパーティションのログが含まれていないことを確認すると検証できます。ブローカ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。

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

2.2.5. ローリング更新のメンテナーンス時間枠

メンテナー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. メンテナーンス時間枠の概要

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。メンテナー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。メンテナー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. メンテナーンス時間枠の定義

Kafka.spec.maintenanceTimeWindows プロパティーに文字列の配列を入力して、メンテナー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、単一のメンテナー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ーンス時間枠内で完了できるように、**Kafka** リソースの **Kafka.spec.clusterCa.renewalDays** および **Kafka.spec.clientsCa.renewalDays** プロパティーとともにメンテナー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ー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ー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ーンス操作の開始が、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、メンテナー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、「環境変数を使用した Cluster Operator の設定」 を参照してください。

2.2.5.3. メンテナーンス時間枠の設定

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ーンス時間枠を設定できます。

前提条件

- OpenShift クラスター
- Cluster Operator が稼働中です。

手順

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  maintenanceTimeWindows:
    - "* * 8-10 * * ?"
    - "* * 14-15 * * ?"
```

2. リソースを作成または更新します。

```
oc apply -f <kafka_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、通常の状況では ZooKeeper に接続する必要はありません。ZooKeeper サービスは暗号化および認証でセキュア化され、AMQ Streams の一部でない外部アプリケーションでの使用は想定されていません。

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

前提条件

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

手順

1. OpenShift コンソールを使用してターミナルを開くか、CLI から **exec** コマンドを実行します。以下に例を示します。

```
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。
   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ーの検証に使用できます。

cluster-name-clients-ca

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

cluster-name-clients-ca-cert

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

cluster-name-cluster-operator-certs

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

ZooKeeper ノード

cluster-name-zookeeper

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 (UseSrimziPodSets 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。

Entitiy Operator

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

cluster-name-entity-operator

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クス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

KafkaConnect リソースを使用して Kafka Connect デプロイメントを設定します。Kafka Connect は、

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

「**KafkaConnect** スキーマ参照」 **KafkaConnect** リソースの完全なスキーマについて説明します。

コネクタープラグインのデプロイの詳細は コネクター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ーを使用して、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 のコネクターインスタンスを OpenShift ネイティブに作成および管理できます。

Kafka Connect 設定では、**strimzi.io/use-connector-resources** アノテーションを追加して、Kafka Connect クラスターの KafkaConnectors を有効にします。また、**build** 設定を追加して、データ接続に必要なコネクタープラグインを備えたコンテテナーイメージを 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 コネクターの外部設定は、**externalConfiguration** プロパティーで指定します。

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

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

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** プロパティーを編集します。
設定可能なプロパティーは以下の例のとおりです。

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

```
          url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
          sha512sum: 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. mTLS、トークンベースの OAuth、SASL ベース SCRAM-SHA-256/SCRAM-SHA-512、または PLAIN として指定された Kafka Connect クラスターの認証。デフォルトでは、Kafka Connect はプレーンテキスト接続を使用して Kafka ブローカーに接続します。
5. Kafka Connect クラスターに接続するための ブートストラップサーバー。
6. クラスターの TLS 証明書が X.509 形式で保存されるキー名のある TLS による暗号化。複数の証明書が同じシークレットに保存されている場合は、複数回リストできます。
7. ワーカーの Kafka Connect 設定 (コネクターではない)。標準の Apache Kafka 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。
8. コネクタープラグインで自動的にコンテナーイメージをビルドするための ビルド設定プロパティー。
9. (必須) 新しいイメージがプッシュされるコンテナーレジストリーの設定。
10. (必須) 新しいコンテナーイメージに追加するコネクタープラグインとそれらのアーティファクトの一覧。各プラグインは、1 つ以上の **artifact** で設定する必要があります。
11. ここで示す環境変数や、ボリュームを使用した Kafka コネクターの外部設定。**設定プロバイダープラグイン** を使用して、外部ソースから設定値を読み込む こともできます。
12. サポートされているリソース (現在は **cpu** と **memory**) の予約と、消費可能な最大リソースを指定するための制限を要求します。
13. 指定された Kafka loggers and log levels が ConfigMap を介して直接的に (**inline**) または間接的に (**external**) に追加されます。カスタム ConfigMap は、**log4j.properties** または **log4j2.properties** キー下に配置する必要があります。Kafka Connect **log4j.rootLogger** ロガーでは、ログレベルを INFO、ERROR、WARN、TRACE、DEBUG、FATAL または OFF に設定できます。
14. コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するための ヘルスチェック。
15.

Prometheus メトリクス。この例では、Prometheus JMX エクスポーターの設定が含まれる ConfigMap を参照して有効になりま

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

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

18 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。

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 分散トレース用に環境変数が設定されます。

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。

関連情報

- Introducing distributed tracing

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。

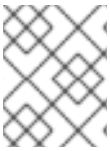
```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: connect-cluster 1
    offset.storage.topic: connect-cluster-offsets 2
    config.storage.topic: connect-cluster-configs 3
    status.storage.topic: connect-cluster-status 4
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

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

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

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

2.3.3. 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ーは AMQ Streams によって自動設定されますが、**KafkaConnect** リソースの **spec** で明示的に指定することもできます。

KafkaConnect リソースの設定プロパティーの例

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

1 Kafka 内の Kafka Connect クラスター ID。

2 コネクターオフセットを保存する Kafka トピック。

3 コネクターおよびタスクステータスの設定を保存する Kafka トピック。

4 コネクターおよびタスクステータスの更新を保存する Kafka トピック。

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

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

注記

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

前提条件

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

手順

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

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  authorization:
    type: simple
    acls:
      # access to offset.storage.topic
      - resource:
          type: topic
          name: connect-cluster-offsets
```

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

2. リソースを作成または更新します。

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

2.3.4. Kafka Connect クラスターリソースの一覧

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

connect-cluster-name-connect

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

connect-cluster-name-connect-api

Kafka Connect クラスターを管理するために REST インターフェ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 の Red Hat ビルドとの統合

Debezium の Red Hat ビルドは、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 の Red Hat ビルドをデプロイおよび統合できます。AMQ Streams のデプロイ後に、Kafka Connect で Debezium をコネクター設定としてデプロイします。Debezium は変更イベントレコードを 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ーで Kafka Connect をデプロイします。**KafkaConnector** リソースを設定し、コネクターインスタンスを定義します。

AMQ Streams で Debezium の Red Hat ビルドをデプロイする方法の詳細は、製品ドキュメント を参照してください。ドキュメントには、**Debezium スタートガイド** が含まれています。このガイドでは、データベース更新の変更イベントレコードの表示に必要なサービスおよびコネクターの設定方法を説明しています。

2.4. KAFKA MIRRORMAKER 2.0 クラスターの設定

KafkaMirrorMaker2 リソースを使用して Kafka MirrorMaker 2.0 デプロイメントを設定します。MirrorMaker 2.0 は、データセンター内またはデータセンター間で、2 つ以上の Kafka クラスター間でデータを複製します。

「**KafkaMirrorMaker2 スキーマ参照**」 **KafkaMirrorMaker2** リソースの完全なスキーマについて説明します。

MirrorMaker 2.0 リソース設定は、以前のバージョンの MirrorMaker 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。

2.4.1. MirrorMaker 2.0 データレプリケーション

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

2.4.1.1. MirrorMaker 2.0 の設定

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

MirrorMaker 2.0 は以下を使用します。

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

MirrorMaker 2.0 は Kafka Connect フレームワークをベースとし、**コネクター** によってクラスター間のデータ転送が管理されます。

MirrorMaker 2.0 を設定して、ソースクラスターとターゲットクラスターの接続の詳細を含む Kafka Connect のデプロイメントを定義し、MirrorMaker 2.0 コネクターのセットを実行して接続を確立します。

MirrorMaker 2.0 は、以下のコネクターで設定されます。

MirrorSourceConnector

ソースコネクターは、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。また、ACL をレプリケートし、**MirrorCheckpointConnector** を実行する必要があります。

MirrorCheckpointConnector

チェックポイントコネクター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

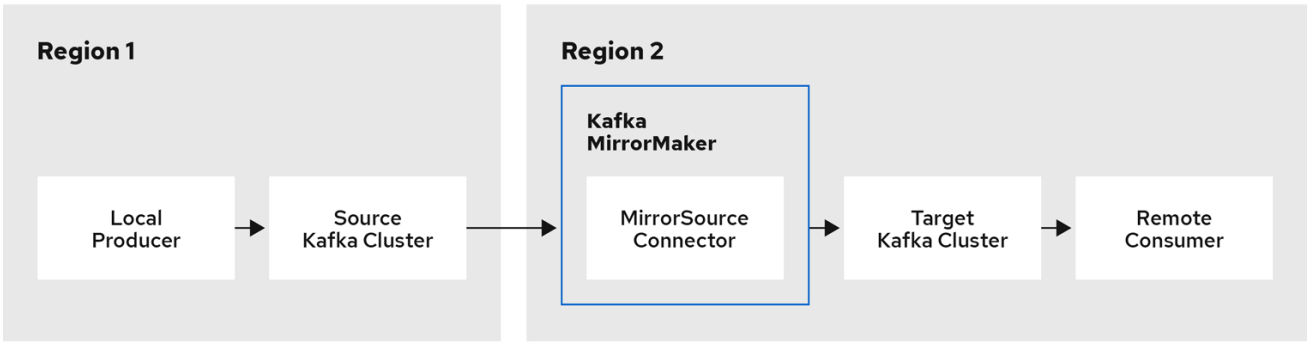
ハートビートコネクターは、ソースクラスターとターゲットクラスター間の接続を定期的にチェックします。

注記

User Operator を使用して ACL を管理する場合、コネクターを介した ACL レプリケーションはできません。

ソースクラスターからターゲットクラスターへのデータの **ミラーリング** プロセスは非同期です。各 MirrorMaker 2.0 インスタンスは、1 つのソースクラスターから 1 つのターゲットクラスターにデータをミラーリングします。複数の MirrorMaker 2.0 インスタンスを使用して、任意の数のクラスター間でデータをミラーリングできます。

図2.1 2 つのクラスターにおけるレプリケーション



デフォルトでは、ソースクラスターの新規トピックのチェックは 10 分ごとに行われます。頻度は、**refresh.topics.interval.seconds** をソースコネクター設定に追加することで変更できます。

2.4.1.1.1. 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。

アクティブ/アクティブのクラスター設定

アクティブ/アクティブ設定には、双方向でデータを複製するアクティブなクラスターが 2 つあります。アプリケーションはいずれかのクラスターを使用できます。各クラスターは同じデータを提供できます。これにより、地理的に異なる場所で同じデータを利用できるようにします。コンシューマーグループは両方のクラスターでアクティブであるため、レプリケートされたトピックのコンシューマーオフセットはソースクラスターに同期されません。

アクティブ/パッシブクラスター設定

アクティブ/パッシブ設定には、パッシブクラスターにデータをレプリケートするアクティブクラスターがあります。パッシブクラスターはスタンバイのままになります。システムに障害が発生した場合に、データ復旧にパッシブクラスターを使用できます。

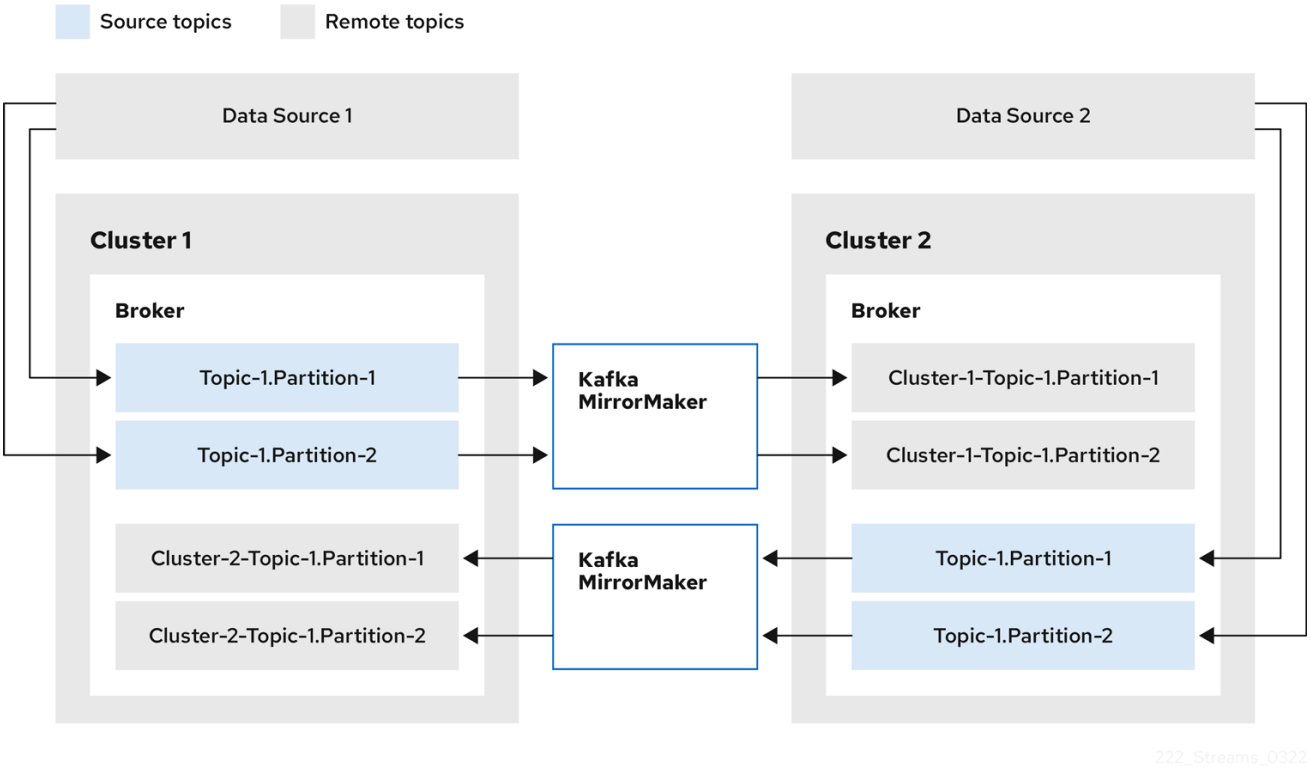
プロデューサーとコンシューマーがアクティブなクラスターのみに接続することを前提とします。MirrorMaker 2.0 クラスターは、ターゲットの宛先ごとに必要です。

2.4.1.1.2. 双方向レプリケーション (active/active)

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

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

図2.2 トピック名の変更



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

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

2.4.1.1.3. 一方向レプリケーション (active/passive)

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

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

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

2.4.1.2. トピック設定の同期

MirrorMaker 2.0 は、ソースクラスターとターゲットクラスター間のトピック設定の Synchronization をサポートしています。MirrorMaker 2.0 設定でソーストピックを指定します。MirrorMaker 2.0 は、ソーストピックを監視します。MirrorMaker 2.0 は、ソーストピックへの変更を検出してリモートトピックに伝播します。変更には、欠けているトピックおよびパーティションの自動作成が含まれる場合があります。

注記

ほとんどの場合、ローカルトピックに書き込み、リモートトピックから読み取ります。リモートトピックでは書き込み操作ができないわけではありませんが、使用しないようにしてください。

2.4.1.3. 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

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。

checkpoints トピック

checkpoints 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。

MirrorMaker 2.0 によって内部で使用されるため、これらのトピックと直接対話することはありません。

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。**チェックポイント** トピックのオフセットは、設定によって事前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。

2.4.1.4. コンシューマーグループオフセットの同期

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

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

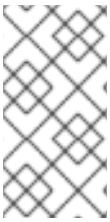
トピックオフセットの同期を使用するには、**sync.group.offsets.enabled** を checkpoint コネクター設定に追加し、プロパティーを **true** に設定して、同期を有効にします。同期はデフォルトで無効になっています。

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

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



注記

Java で作成されたアプリケーションがある場合は、**RemoteClusterUtils.java** ユーティリティーを使用して、アプリケーションを通じてオフセットを同期できます。ユーティリティーは、**checkpoints** トピックからコンシューマーグループのリモートオフセットを取得します。

2.4.1.5. 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トされます。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ー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

2.4.2. コネクター設定

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ーの Mirrormaker 2.0 コネクター設定を使用します。

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

表2.1 MirrorMaker 2.0 コネクター設定プロパティー

プロパティー	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。	✓	✓	✓

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
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。	✓	✓	
topic.filter.class 複製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 複製するトピック設定プロパティー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 複製すべきでないトピック設定プロパティー。コンマ区切りのプロパティー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。	✓		

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
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。	✓		
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 との互換性がありません。	✓		
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。		✓	

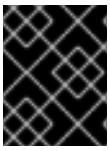
プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
emit.checkpoints.enabled コンシューマーオフセットをターゲットクラスターに同期できるようにします。デフォルトは **true** です。		✓	
emit.checkpoints.interval.seconds コンシューマーオフセット同期の頻度。デフォルトは **60** (1 分) です。		✓	
group.filter.class 複製するコンシューマーグループを選択するためのグループフィルター。デフォルトは **org.apache.kafka.connect.mirror.DefaultGroupFilter** 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。			✓

プロパティー	sourceConnector	checkpointConnector	heartbeatConnector
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。			✓
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。			✓

2.4.3. コネクタープロデューサーおよびコンシューマーの設定

MirrorMaker 2.0 コネクター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、各コネクター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 ソースコネクターのプロデューサーとコンシューマー

タイプ	説明	設定
プロデューサー	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.***

タイプ	説明	設定
プロデューサー	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.***
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
プロデューサー	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ープロデューサー

タイプ	説明	設定
プロデューサー	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: 3.3.1
  # ...
  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」

2.4.4. タスクの最大数を指定

コネクターは、Kafka にデータを出し入れするタスクを作成します。各コネクターは、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トするとき、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 プロパティーを使用して、MirrorMaker 設定でコネクター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ー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ー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。ソースコネクターの場合、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ト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ーの場合、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ト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ー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。

ソースコネクター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ー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ーのタスク数の増加

```
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。

2.4.4.1. コネクター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

2.4.5. 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

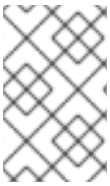
OAuth 2.0 での承認は、このようなリモートトピックへのアクセスをサポートしません。

2.4.6. Kafka MirrorMaker 2.0 の設定

KafkaMirrorMaker2 リソースのプロパティーを使用して、Kafka MirrorMaker 2.0 のデプロイメントを設定します。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
- 認証を含む各クラスターの接続情報
- レプリケーションのフローおよび方向
  - クラスター対クラスター
  - トピック対トピック



注記

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

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

MirrorMaker 2.0 の最小設定

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.3.1
  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: {}
```

mTLS 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 mTLS 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** プロパティーを使用します。名前の一覧を指定したり、正規表現を使用したりできます。既定では、**topicsPattern** および **groupsPattern** プロパティーを設定しない場合、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** プロパティーを編集します。
   設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker2
spec:
  version: 3.3.1 (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 ソースクラスターの認証。mTLS、トークンベースの OAuth、SASL ベース SCRAM-SHA-256/SCRAM-SHA-512、または PLAIN として指定します。

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もあります。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す。

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、このプロパティーを **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、このプロパティーも **checkpointConnector.config** に設定する必要があります。

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

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

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

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

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

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

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

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

35 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。ソースコネクターは指定のトピックを複製します。チェックポイントコネクターは、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ーは、指定されたコンシューマーグループを複製します。ここで、3 つのコンシューマーグループを名前で要求します。

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

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

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

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

42 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 分散トレース用に環境変数が設定されます。

45 Jaeger では分散トレースが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
```

関連情報

- Introducing distributed tracing

2.4.7. 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 CA (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。mTLS 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。

この例では、ミラーメーカー 2.0 がソースおよびターゲットの Kafka クラスターでの操作を許可するために必要なすべての ACL を含む、完全な承認の設定を提供します。

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。Topoic Operator は、指定された namespace 内の単一クラスターのみをモニターリングします。

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 mTLS 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 暗号化と mTLS 認証を使用したソース Kafka クラスター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-source-cluster
spec:
  kafka:
    version: 3.3.1
    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: "3.3"
  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 暗号化と mTLS 認証を使用したターゲット Kafka クラスター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-target-cluster
spec:
  kafka:
    version: 3.3.1
    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: "3.3"
```

```
    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: {}
```

2. 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。

3. 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。

   a. 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。

   b. 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。

mTLS 認証のソースユーザー設定の例

```
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
```

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

mTLS 認証のターゲットユーザー設定の例

```
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: mirrormaker2-cluster
        operations:
          - Read
      - resource:
          type: topic
          name: mirrormaker2-cluster-configs
        operations:
          - Create
          - Describe
          - DescribeConfigs
          - Read
          - Write
```

```
- resource:
    type: topic
    name: mirrormaker2-cluster-status
  operations:
    - Create
    - Describe
    - DescribeConfigs
    - Read
    - Write
- resource:
    type: topic
    name: mirrormaker2-cluster-offsets
  operations:
    - Create
    - Describe
    - DescribeConfigs
    - Read
    - Write
# MirrorSourceConnector
- resource: # Needed for every topic which is mirrored
    type: topic
    name: "*"
  operations:
    - Create
    - Alter
    - AlterConfigs
    - Write
# MirrorCheckpointConnector
- resource:
    type: cluster
  operations:
    - Describe
- resource:
    type: topic
    name: my-source-cluster.checkpoints.internal
  operations:
    - Create
    - Describe
    - Read
    - Write
- resource: # Needed for every group for which the offset is synced
    type: group
    name: "*"
  operations:
    - Read
    - Describe
# MirrorHeartbeatConnector
- resource:
    type: topic
    name: heartbeats
  operations:
    - Create
    - Describe
    - 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、mTLS 認証用の秘密鍵と公開鍵が含まれています。公開鍵は、クライアント 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 暗号化と mTLS 認証を使用した MirrorMaker 2.0 設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker2
metadata:
  name: my-mirror-maker-2
spec:
  version: 3.3.1
  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

2.4.8. 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 が稼働中です。

手順

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

```
oc get KafkaMirrorMaker2
```

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

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

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

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

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

関連情報

- Kafka MirrorMaker 2.0 クラスターの設定

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

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

前提条件

- Cluster Operator が稼働中です。

手順

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

```
oc get KafkaMirrorMaker2
```

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

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

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

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

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

関連情報

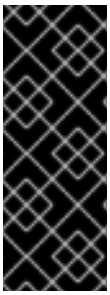
- Kafka MirrorMaker 2.0 クラスターの設定

2.5. KAFKA MIRRORMAKER クラスターの設定

KafkaMirrorMaker リソースを使用して Kafka MirrorMaker デプロイメントを設定します。KafkaMirrorMaker は、Kafka クラスター間でデータを複製します。

「**KafkaMirrorMaker** スキーマ参照」 **KafkaMirrorMaker** リソースの完全なスキーマについて説明します。

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。



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。

2.5.1. Kafka MirrorMaker の設定

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

TLS または SASL 認証を使用して、プロデューサーおよびコンシューマーのアクセス制御を設定できます。この手順では、コンシューマーおよびプロデューサー側で mTLS 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 クラスター

- ソースおよびターゲットの Kafka クラスターが使用できる必要があります。

手順

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
metadata:
  name: my-mirror-maker
spec:
  replicas: 3 (1)
  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. mTLS、トークンベースの OAuth、SASL ベース SCRAM-SHA-256/SCRAM-SHA-512、または PLAIN として指定され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 プロパティー。
10. **HTTPS** に設定することで、ホスト名の検証が有効 になります。空の文字列を指定すると検証が無効になります。
11. **abortOnSendFailure** プロパティー 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、コンテナーは再起動します。
12. TLS バージョンの特定の **暗号スイート** と実行される外部リスナーの SSL プロパティー。
13. **HTTPS** に設定することで、ホスト名の検証が有効 になります。空の文字列を指定すると検証が無効になります。
14. ソースからターゲット Kafka クラスターにミラーリングされた 含まれるトピック。
15. サポートされているリソース (現在は **cpu** と **memory**) の予約と、消費可能な最大リソースを指定するための制限を要求します。
16. 指定された 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. コンテナーを再起動するタイミング (liveness) およびコンテナーがトラフィックを許可できるタイミング (readiness) を把握するための ヘルスチェック。
18. Prometheus メトリクス。この例では、Prometheus JMX エクスポーターの設定が含まれる ConfigMap を参照して有効になります。**metricsConfig.valueFrom.configMapKeyRef.key** 配下に空のファイルが含まれる ConfigMap の参照を使用して、追加設定なしでメトリクスを有効にできます。
19. Kafka MirrorMaker を実行している仮想マシン (VM) のパフォーマンスを最適化するための JVM 設定オプション。
20. 高度な任意設定: 特別な場合のみ推奨される コンテナーイメージの設定。
21. テンプレートのカスタマイズ。ここでは、Pod は非アフィニティーでスケジュールされるため、Pod は同じホスト名のノードではスケジュールされません。

22 分散トレース用に環境変数が設定されます。

23 Jaeger では分散トレースが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>
```

関連情報

- Introducing distributed tracing

2.5.2. Kafka MirrorMaker クラスターリソースの一覧

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

<mirror-maker-name>-mirror-maker

Kafka MirrorMaker Pod の作成を担当するデプロイメント。

<mirror-maker-name>-config

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

<mirror-maker-name>-mirror-maker

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

2.6. KAFKA BRIDGE クラスターの設定

KafkaBridge リソースを使用して Kafka Bridge デプロイメントを設定します。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。

「**KafkaBridge** スキーマ参照」 **KafkaBridge** リソースの完全なスキーマについて説明します。

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ーを使用して、Kafka Bridge デプロイメントを設定します。

クライアントのコンシューマーリクエストが異なる Kafka Bridge インスタンスによって処理された場合に発生する問題を防ぐには、アドレスベースのルーティングを利用して、要求が適切な Kafka Bridge イ

ンスタンスにルーティングされるようにする必要があります。また、独立した各 Kafka Bridge インスタンスにレプリカが必要です。Kafka Bridge インスタンスには、別のインスタンスと共有されない独自の状態があります。

前提条件

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

以下を実行する方法については、**OpenShift での AMQ Streams のデプロイおよびアップグレード**を参照すること。

- Cluster Operator
- Kafka クラスター

手順

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  replicas: 3 (1)
  bootstrapServers: <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

mTLS、トークンベースの OAuth、SASL ベース SCRAM-SHA-256/SCRAM-SHA-512、または PLAIN として指定された Kafka Bridge クラスターの認証。デフォルトでは、Kafka

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

13 オプション: コンテナーイメージの設定。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 分散トレース用に環境変数が設定されます。

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
- Introducing distributed tracing

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。

プロデューサーとコンシューマーの設定は、Kafka ブローカーへの要求のサイズと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。

- プロデューサー向けの Apache Kafka 設定ドキュメント
- コンシューマー向けの Apache Kafka 設定ドキュメント

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** プロパティーを使用して、フェッチ要求のサイズを増やします。**max.poll.records** プロパティーを使用して、コンシューマーバッファー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は、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** プロパティー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る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、プロデュース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、プロデュース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ス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、バッチ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のようになります。

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 → プロデューサーバーッファー → 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ー → 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 → プロデューサーバーッファー → ターゲット Kafka トピック

プロデューサーは、バッファー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。

ソースコネクターのプロデューサーバーッファーのサイズは、**producer.override.buffer.memory** プロパティーを使用して設定されます。タスクは、バッファー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、バッファーリングは、**max.block.ms** で制限された期間内にバッファーに使用可能なスペースができるまでブロックされます。バッファー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ーに追加されません。

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** プロパティーを使用して並行して実行するタスクの数を増やす

- **replicas** プロパティー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必要がある場合があります。

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。

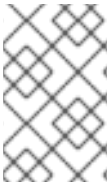
大量のメッセージを処理するための Kafka Connect 設定の例

```
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ー用に追加されます。

大量のメッセージを処理するためのソースコネクターの設定例

```
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、コネクター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。

シンクコネクター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ーの設定例

```
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ーを管理している場合は、コネクター設定を 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ー設定を追加するための curl 要求の例

```
curl -X POST \
  http://my-connect-cluster-connect-api:8083/connectors \
  -H 'Content-Type: application/json' \
  -d '{ "name": "my-source-connector",
    "config":
    {
      "connector.class":"org.apache.kafka.connect.file.FileStreamSourceConnector",
      "file": "/opt/kafka/LICENSE",
      "topic":"my-topic",
      "tasksMax": "4",
      "type": "source"
      "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: 3.3.1
  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ー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
- プロデューサーバーッファー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)

これらのメトリクスを使用して、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. OPENSHIFT 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 のみがそのリソースを変更できます。operator によって管理される OpenShift リソースを手動で変更しようとすると、operator はその変更を元に戻します。

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

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

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

Kafka.spec.kafka

「**KafkaClusterTemplate** スキーマ参照」 を参照

Kafka.spec.zookeeper

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

Kafka.spec.entityOperator

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

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** プロパティーを使用して 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 のコンテナーのイメージプルポリシーをカスタマイズできます。イメージプルポリシーは、Cluster Operator デプロイメントの環境変数 **STRIMZI_IMAGE_PULL_POLICY** を使用して設定されます。**STRIMZI_IMAGE_PULL_POLICY** 環境変数に設定できる値は 3 つあります。

Always

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

IfNotPresent

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

Never

コンテナーイメージはレジストリーからプルされることはありません。

現在、イメージプルポリシーはすべての Kafka、Kafka Connect、および Kafka MirrorMaker クラスターに対してのみ 1 度にカスタマイズできます。ポリシーを変更すると、すべての Kafka、Kafka Connect、および Kafka MirrorMaker クラスターのローリング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 プロパティーを使用して時間を指定します。プロパティーを **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

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** プロパティーを編集します。ワーカーノードが Kafka ブローカーまたは ZooKeeper ノードで共有されないようにするには、**strimzi.io/name** ラベルを使用します。**topologyKey** を **kubernetes.io/hostname** に設定して、選択した Pod が同じホスト名のノードでスケジュールされないように指定します。これにより、同じワーカーノードを単一の Kafka ブローカーと単一の ZooKeeper ノードで共有できます。以下に例を示します。

```
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。

3. 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

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

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

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

```
oc apply -f <kafka_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

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

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

または、既存のラベルによっては再利用が可能です。

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

```
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:
  # ...
```

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

```
oc apply -f <kafka_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** プロパティーを編集します。
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** プロパティーに直接指定できます。あるいは、**configMapKeyRef** プロパティーを使ってカスタムリソースで参照される ConfigMap でロギングプロパティーを定義することもできます。

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

ロギング仕様でロギング **type** を指定します。

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

inline ロギングの設定例

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

external 設定の例

```
spec:
```

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

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

2.10.1. 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ーを定義するには、ConfigMap を作成してから、リソースの **spec** にあるロギング定義の一部としてそれを参照します。

ConfigMap には適切なロギング設定が含まれる必要があります。

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

設定はこれらのプロパティーの配下に配置する必要があります。

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

手順

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

   Kafka のルートロガー定義が含まれる ConfigMap の例:

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

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

   ```
   oc create configmap logging-configmap --from-file=log4j.properties
   ```

   プロパティーファイルではロギング設定が定義されます。

   ```
   # Define the logger
   kafka.root.logger.level="INFO"
   # ...
   ```

2. リソースの **spec** に**external** ロギングを定義し、**logging.valueFrom.configMapKeyRef.name** に ConfigMap の名前を、**logging.valueFrom.configMapKeyRef.key** にこの ConfigMap のキーを設定します。

   ```
   spec:
     # ...
     logging:
       type: external
       valueFrom:
         configMapKeyRef:
           name: logging-configmap
           key: log4j.properties
   ```

3. リソースを作成または更新します。

   ```
   oc apply -f <kafka_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。ロガーのログレベルを DEBUG(**rootLogger.level="DEBUG"**) に設定すると仮定します。ロギングフィルターは、このレベルでロガーに対して返されるログ数を減らし、特定のリソースに集中できるようにします。フィルターが設定されると、フィルターに一致するログメッセージのみがログに記録されます。

フィルターは**マーカー**を使用して、ログに含まれる内容を指定します。マーカーの種類、namespace、および名前を指定します。たとえば、Kafka クラスターで障害が発生した場合、種類を **Kafka** に指定してログを分離し、障害が発生しているクラスターの namespace および名前を使用します。

以下の例は、**my-kafka-cluster** という名前の Kafka クラスターのマーカーフィルターを示しています。

基本的なロギングフィルターの設定

```
rootLogger.level="INFO"
appender.console.filter.filter1.type=MarkerFilter (1)
appender.console.filter.filter1.onMatch=ACCEPT (2)
appender.console.filter.filter1.onMismatch=DENY (3)
appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster) (4)
```

1. **MarkerFilter** 型は、フィルターを行うために指定されたマーカーを比較します。
2. **onMatch** プロパティーは、マーカーが一致するとログを受け入れます。
3. **onMismatch** プロパティーは、マーカーが一致しない場合にログを拒否します。
4. フィルター処理に使用されるマーカーの形式は **KIND(NAMESPACE/NAME-OF-RESOURCE)** です。

フィルターは 1 つまたは複数作成できます。ここでは、ログは 2 つの Kafka クラスターに対してフィルターされます。

複数のロギングフィルターの設定

```
appender.console.filter.filter1.type=MarkerFilter
appender.console.filter.filter1.onMatch=ACCEPT
appender.console.filter.filter1.onMismatch=DENY
appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster-1)
appender.console.filter.filter2.type=MarkerFilter
appender.console.filter.filter2.onMatch=ACCEPT
appender.console.filter.filter2.onMismatch=DENY
appender.console.filter.filter2.marker=Kafka(my-namespace/my-kafka-cluster-2)
```

フィルターの Cluster Operator への追加

フィルターを Cluster Operator に追加するには、そのロギング ConfigMap YAML ファイルを更新します (**install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml**)。

手順

1. **050-ConfigMap-strimzi-cluster-operator.yaml** ファイルを更新して、フィルタープロパティーを ConfigMap に追加します。
   この例では、フィルタープロパティーは **my-kafka-cluster** Kafka クラスターのログのみを返します。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: strimzi-cluster-operator
data:
  log4j2.properties:
    #...
    appender.console.filter.filter1.type=MarkerFilter
```

```
    appender.console.filter.filter1.onMatch=ACCEPT
    appender.console.filter.filter1.onMismatch=DENY
    appender.console.filter.filter1.marker=Kafka(my-namespace/my-kafka-cluster)
```

または、**ConfigMap** を直接編集することもできます。

```
oc edit configmap strimzi-cluster-operator
```

2. **ConfigMap** を直接編集せずに YAML ファイルを更新する場合は、ConfigMap をデプロイして変更を適用します。

```
oc create -f install/cluster-operator/050-ConfigMap-strimzi-cluster-operator.yaml
```

Topic Operator または User Operator へのフィルターの追加

フィルターを Topic Operator または User Operator に追加するには、ロギング ConfigMap を作成または編集します。

この手順では、ロギング ConfigMap は、Topic Operator のフィルターで作成されます。User Operator に同じアプローチが使用されます。

手順

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

   この例では、フィルタープロパティーは **my-topic** トピックに対してのみログを返します。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: logging-configmap
data:
  log4j2.properties:
    rootLogger.level="INFO"
    appender.console.filter.filter1.type=MarkerFilter
    appender.console.filter.filter1.onMatch=ACCEPT
    appender.console.filter.filter1.onMismatch=DENY
    appender.console.filter.filter1.marker=KafkaTopic(my-namespace/my-topic)
```

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

```
oc create configmap logging-configmap --from-file=log4j2.properties
```

   プロパティーファイルではロギング設定が定義されます。

```
# Define the logger
rootLogger.level="INFO"
# Set the filters
appender.console.filter.filter1.type=MarkerFilter
appender.console.filter.filter1.onMatch=ACCEPT
```

```
appender.console.filter.filter1.onMismatch=DENY
appender.console.filter.filter1.marker=KafkaTopic(my-namespace/my-topic)
# ...
```

2. リソースの **spec** に**external** ロギングを定義し、**logging.valueFrom.configMapKeyRef.name** に ConfigMap の名前を、**logging.valueFrom.configMapKeyRef.key** にこの ConfigMap のキーを設定します。
Topic Operator については、**Kafka** リソースの **topicOperator** 設定でロギングを指定します。

```
spec:
  # ...
  entityOperator:
    topicOperator:
      logging:
        type: external
        valueFrom:
          configMapKeyRef:
            name: logging-configmap
            key: log4j2.properties
```

3. Cluster Operator をデプロイして変更を適用します。

```
create -f install/cluster-operator -n my-cluster-operator-namespace
```

関連情報

- Kafka の設定
- Cluster Operator のロギング
- Topic Operator のロギング
- User Operator のロギング

第3章 外部ソースからの設定値の読み込み

設定プロバイダープラグインを使用して、外部ソースから設定データを読み込みます。プロバイダーは AMQ Streams とは独立して動作します。これを使用して、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 コネクター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 ConfigMap 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ーをホストする場合に中断の発生を防ぎます。

環境変数設定プロバイダー

環境変数の設定プロバイダープラグインを使用して、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 ConfigMap 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、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。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 プロパティー** を使用して設定を追加します。このアプローチでアクセス権限を設定する必要はありません。ただし、コネクターに新しい **Secret** または **ConfigMap** を使用する場合は、Kafka Connect の再起動が必要になります。これにより、すべての Kafka Connect インスタンスのコネクターが中断されます。

3.1. CONFIGMAP 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** はコネクターの設定プロパティーを提供します。

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 が稼働中です。

手順

1. 設定プロパティーが含まれる **ConfigMap** または**シークレット**を作成します。

この例では、**my-connector-configuration** という名前の **ConfigMap** にはコネクタープロパティーが含まれます。

コネクタープロパティーのある ConfigMap の例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-connector-configuration
data:
  option1: value1
  option2: value2
```

2. Kafka Connect 設定で OpenShift Configuration Provider を指定します。
ここで示される仕様は、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す。プロバイダーパラメーターは **config.providers** からのエイリアスを使用し、**config.providers.${alias}.class** の形式を取ります。

(2) **KubernetesSecretConfigProvider** は 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ー値のプレースホルダーは、コネクター設定で参照されます。プレー

スホルダー構造は、**configmaps:<path_and_file_name>:<property>** です。**KubernetesConfigMapConfigProvider** は、外部の ConfigMap から **option1** プロパティー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ーの設定プロパティーを提供します。データベースのパスワードは環境変数として指定します。

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 が稼働中です。

手順

1. Kafka Connect 設定で環境変数設定プロバイダーを指定します。
externalConfiguration プロパティー を使用して環境変数を定義します。

環境変数設定プロバイダーを有効にする Kafka Connect 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 (1)
    config.providers.env.class: io.strimzi.kafka.EnvVarConfigProvider (2)
  # ...
  externalConfiguration:
    env:
      - name: DB_PASSWORD (3)
        valueFrom:
          secretKeyRef:
            name: db-creds (4)
            key: dbPassword (5)
  # ...
```

(1) 設定プロバイダーのエイリアスは、他の設定パラメーターを定義するために使用されます。プロバイダーパラメーターは **config.providers** からのエイリアスを使用し、**config.providers.${alias}.class** の形式を取ります。

(2) **EnvVarConfigProvider** は、環境変数から値を指定します。

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. コネクター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ー設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
  # ...
  config:
    option: ${env:DB_PASSWORD}
    # ...
# ...
```

第4章 AMQ STREAMS POD およびコンテナーへのセキュリティーコンテキストの適用

セキュリティーコンテキストは、Pod とコンテナーの制約を定義します。セキュリティーコンテキストを指定すると、Pod およびコンテナーに必要なパーミッションのみが設定されます。たとえば、パーミッションはランタイム操作やリソースへのアクセスを制御できます。

4.1. OPENSHIFT プラットフォームによるセキュリティーコンテキストの処理

セキュリティーコンテキストの処理は、使用している OpenShift プラットフォームのツールによって異なります。

たとえば、OpenShift はビルトイン SCC (Security Context Constraints)を使用してパーミッションを制御します。SCC は、Pod がアクセスできるセキュリティー機能を制御する設定およびストラテジーです。

デフォルトでは、OpenShift はセキュリティーコンテキスト設定を自動的に注入します。ほとんどの場合、Cluster Operator によって作成される Pod およびコンテナーのセキュリティーコンテキストを設定する必要はありません。ただし、引き続き独自の SCC を作成して管理することはできます。

詳細は、Openshift ドキュメント を参照してください。

第5章 OPENSHIFT クラスター外の KAFKA へのアクセス

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

外部リスナー設定で Kafka を公開するため **type** を指定します。

- **nodeport** は **NodePort** タイプの **Service** を使用します
- **loadbalancer** は **Loadbalancer** タイプの **Service** を使用します
- **ingress** は Kubernetes **Ingress** と Ingress NGINX Controller for Kubernetes を使用します。
- **route** は、OpenShift **Routes** と HAProxy ルーターを使用します。

リスナーの設定の詳細については、**GenericKafkaListener** 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。

5.1. 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:
    # ...
```

2. 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ー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。

3. **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"}'
```

4. TLS による暗号化が有効な場合は、ブローカーの認証局の公開証明書を取得します。

```
oc get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64
-d > ca.crt
```

Kafka クライアントで取得した証明書を使用して TLS 接続を設定します。認証を有効にした場合は、クライアントでも設定する必要があります。

5.2. ロードバランサーを使用した KAFKA へのアクセス

この手順では、ロードバランサーを使用して外部クライアントから AMQ Streams Kafka クラスターにアクセスする方法について説明します。

ブローカーに接続するには、**ブートストラップロードバランサー**のアドレスと、TLS による暗号化に使用される証明書が必要です。

前提条件

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

手順

1. 外部リスナーを **loadbalancer** タイプに設定して **Kafka** リソースを設定します。
以下に例を示します。

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

2. 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ー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。

3. **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"}'
```

4. TLS による暗号化が有効な場合は、ブローカーの認証局の公開証明書を取得します。

```
oc get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca\.crt}' | base64
-d > ca.crt
```

Kafka クライアントで取得した証明書を使用して TLS 接続を設定します。認証を有効にした場合は、クライアントでも設定する必要があります。

5.3. INGRESS NGINX CONTROLLER FOR OPENSHIFT を使用して KAFKA にアクセスする

Ingress NGINX Controller for Kubernetes を使用して、OpenShift クラスター外のクライアントから AMQ Streams Kafka クラスターにアクセスします。

Ingress NGINX Controller for OpenShift を使用できるようにするには、**Kafka** カスタムリソースに **ingress** タイプリスナーの設定を追加します。適用すると、設定により、外部ブートストラップとクラ

スター内の各ブローカーに専用の ingress とサービスが作成されます。クライアントはブートストラップ ingress に接続し、これはブートストラップサービスを経由してクライアントをルーティングし、ブローカーに接続します。その後、ブローカーごとの接続が DNS 名を使用して確立されます。DNS 名は、ブローカー固有の ingress とサービスを介してクライアントからブローカーにトラフィックをルーティングします。

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と TLS 証明書を指定します。認証はオプションです。

ingress を使用したアクセスの場合、Kafka クライアントで使用されるポートは通常 443 です。

TLS パススルー

Ingress NGINX Controller for OpenShift デプロイメントで TLS パススルーを有効にしていることを確認してください。Kafka は TCP 経由でバイナリープロトコルを使用しますが、Ingress NGINX Controller for OpenShift は HTTP プロトコルで動作するように設計されています。TCP トラフィックを ingress 経由でルーティングできるようにするために、AMQ Streams は Server Name Indication (SNI) で TLS パススルーを使用します。

SNI は、Kafka ブローカーへの接続を識別して渡すのに役立ちます。パススルーモードでは、TLS 暗号化が常に使用されます。接続はブローカーに渡されるため、リスナーは、ingress 証明書ではなく、内部クラスター CA によって署名された TLS 証明書を使用します。独自のリスナー証明書を使用するようにリスナーを設定するには、**brokerCertChainAndKey** プロパティーを使用します。

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

- Ingress NGINX Controller for OpenShift が TLS パススルーを有効にして実行されている
- 稼働中の Cluster Operator

この手順では、Kafka クラスター名は **my-cluster** です。

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と Kafka クラスター内の各 Kafka ブローカーの ingress ホスト名を指定します。ブートストラップとブローカーを識別する **bootstrap** および **broker-<index>** 接頭辞に任意のホスト名を追加します。

   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: external
        port: 9094
```

```
      type: ingress
      tls: true (1)
      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
        class: nginx (2)
    # ...
  zookeeper:
    # ...
```

(1) **ingress** タイプのリスナーの場合、TLS 暗号化を有効にする必要があります (**true**)。

(2) (オプション) 使用する ingress コントローラーを指定するクラス。デフォルトを設定しておらず、作成された ingress でクラス名が欠落している場合は、クラスを追加する必要がある場合があります。

2. リソースを作成または更新します。

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

kafka ブローカーの ID を確認するためのクラスター CA 証明書は、シークレットの **my-cluster-cluster-ca-cert** に作成されます。

ClusterIP タイプサービスは、各 Kafka ブローカーと、外部のブートストラップサービスに対して作成されます。

ingress は各サービスに対しても作成され、Ingress NGINX Controller for OpenShift を使用してそれらを公開するための DNS アドレスが指定されます。

ブートストラップとブローカー用に作成された ingress

```
NAME                    CLASS  HOSTS                    ADDRESS            PORTS
my-cluster-kafka-0      nginx  broker-0.myingress.com   external.ingress.com  80,443
my-cluster-kafka-1      nginx  broker-1.myingress.com   external.ingress.com  80,443
my-cluster-kafka-2      nginx  broker-2.myingress.com   external.ingress.com  80,443
my-cluster-kafka-bootstrap  nginx  bootstrap.myingress.com  external.ingress.com  80,443
```

クライアント接続に使用される DNS アドレスは、各 ingress の **status** に伝播されます。

ブートストラップ ingress のステータス

```
status:
  loadBalancer:
    ingress:
```

```
      - hostname: external.ingress.com
  # ...
```

3. ターゲットブローカーを使用して、OpenSSL **s_client** を使用するポート 443 でクライアントサーバーの TLS 接続を確認します。

```
openssl s_client -connect broker-0.myingress.com:443 -servername broker-0.myingress.com
-showcerts
```

サーバー名は、接続をブローカーに渡すための SNI です。

接続が成功すると、ブローカーの証明書が返されます。

ブローカーの証明書

```
Certificate chain
 0 s:O = io.strimzi, CN = my-cluster-kafka
   i:O = io.strimzi, CN = cluster-ca v0
```

4. クラスター CA 証明書を抽出します。

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

5. ブローカーに接続するようにクライアントを設定します。

   a. 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、Kafka クライアントでブートストラップホスト (リスナー **configuration** から) とポート 443 を指定します。たとえば、**bootstrap.myingress.com:443** です。

   b. 抽出した証明書を Kafka クライアントのトラストストアに追加して、TLS 接続を設定します。
   認証を有効にした場合は、クライアントでも設定する必要があります。

注記

独自のリスナー証明書を使用している場合は、CA 証明書をクライアントのトラストストア設定に追加する必要があるかどうかを確認してください。パブリック (外部) CA の場合、通常は追加する必要はありません。

5.4. OPENSHIFT ルートを使用した KAFKA へのアクセス

OpenShift ルートを使用して、OpenShift クラスター外のクライアントから AMQ Streams Kafka クラスターにアクセスします。

ルートを使用できるようにするには、**Kafka** カスタムリソースに **route** タイプリスナーの設定を追加します。適用すると、設定により、外部ブートストラップとクラスター内の各ブローカーに専用のルートとサービスが作成されます。クライアントはブートストラップルートに接続し、ブートストラップサービスを経由してクライアントをルーティングし、ブローカーに接続します。その後、ブローカーごとの接続が DNS 名を使用して確立されます。DNS 名は、ブローカー固有のルートとサービスを介してクライアントからブローカーにトラフィックをルーティングします。

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、認証に使用される証明書を指定します。

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。

警告

OpenShift ルートアドレスは、Kafka クラスターの名前、リスナーの名前、および作成されるプロジェクトの名前で構成されます。たとえば、**my-cluster-kafka-listener1-bootstrap-myproject** (<cluster_name>-kafka-<listener_name>-bootstrap-<namespace>) となります。アドレスの全体の長さが上限の 63 文字を超えないように注意してください。

TLS パススルー

AMQ Streams によって作成されたルートに対して TLS パススルーが有効になります。Kafka は TCP 経由でバイナリープロトコルを使用しますが、ルートは HTTP プロトコルで動作するように設計されています。ルートを介して TCP トラフィックをルーティングできるようにするために、AMQ Streams は Server Name Indication (SNI) で TLS パススルーを使用します。

SNI は、Kafka ブローカーへの接続を識別して渡すのに役立ちます。パススルーモードでは、TLS 暗号化が常に使用されます。接続はブローカーに渡されるため、リスナーは、ingress 証明書ではなく、内部クラスター CA によって署名された TLS 証明書を使用します。独自のリスナー証明書を使用するようにリスナーを設定するには、**brokerCertChainAndKey** プロパティーを使用します。

前提条件

- 稼働中の Cluster Operator

この手順では、Kafka クラスター名は **my-cluster** です。

手順

1. 外部リスナーを **route** タイプに設定した **Kafka** リソースを設定します。
   以下に例を示します。

```
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 (1)
        # ...
```

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

1 **route** タイプリスナーの場合、TLS 暗号化を有効にする必要があります (**true**)。

2. リソースを作成または更新します。

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

kafka ブローカーの ID を確認するためのクラスター CA 証明書は、シークレットの **my-cluster-cluster-ca-cert** に作成されます。

ClusterIP タイプサービスは、各 Kafka ブローカーと、外部のブートストラップサービスに対して作成されます。

デフォルトの OpenShift HAProxy ルーターを使用してサービスを公開するための DNS アドレス (ホスト/ポート) を使用して、サービスごとに **route** も作成されます。

ルートは、TLS パススルーで事前設定されています。

ブートストラップとブローカー用に作成されたルート

```
NAME                                HOST/PORT                                                                SERVICES
PORT  TERMINATION
my-cluster-kafka-listener1-0        my-cluster-kafka-listener1-0-my-project.router.com
my-cluster-kafka-listener1-0        9094  passthrough
my-cluster-kafka-listener1-1        my-cluster-kafka-listener1-1-my-project.router.com
my-cluster-kafka-listener1-1        9094  passthrough
my-cluster-kafka-listener1-2        my-cluster-kafka-listener1-2-my-project.router.com
my-cluster-kafka-listener1-2        9094  passthrough
my-cluster-kafka-listener1-bootstrap  my-cluster-kafka-listener1-bootstrap-my-
project.router.com  my-cluster-kafka-listener1-bootstrap  9094  passthrough
```

クライアント接続に使用される DNS アドレスは、各ルートの **status** に伝播されます。

ブートストラップルートのステータスの例

```
status:
  ingress:
    - host: >-
        my-cluster-kafka-listener1-bootstrap-my-project.router.com
  # ...
```

3. ターゲットブローカーを使用して、OpenSSL **s_client** を使用するポート 443 でクライアントサーバーの TLS 接続を確認します。

```
openssl s_client -connect my-cluster-kafka-listener1-0-my-project.router.com:443 -
servername my-cluster-kafka-listener1-0-my-project.router.com -showcerts
```

サーバー名は、接続をブローカーに渡すための SNI です。

接続が成功すると、ブローカーの証明書が返されます。

ブローカーの証明書

```
Certificate chain
 0 s:O = io.strimzi, CN = my-cluster-kafka
   i:O = io.strimzi, CN = cluster-ca v0
```

4. **Kafka** リソースのステータスから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 my-cluster -o=jsonpath='{.status.listeners[?
(@.name=="listener1")].bootstrapServers}{"\n"}'

my-cluster-kafka-listener1-bootstrap-my-project.router.com:443
```

アドレスは、クラスター名、リスナー名、プロジェクト名、およびルーターのドメイン (この例では **router.com**) で構成されます。

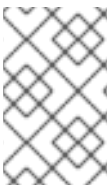
5. クラスター CA 証明書を抽出します。

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

6. ブローカーに接続するようにクライアントを設定します。

   a. 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、Kafka クライアントでブートストラップサービスのアドレスとポート 443 を指定します。

   b. 抽出した証明書を Kafka クライアントのトラストストアに追加して、TLS 接続を設定します。
   認証を有効にした場合は、クライアントでも設定する必要があります。



注記

独自のリスナー証明書を使用している場合は、CA 証明書をクライアントのトラストストア設定に追加する必要があるかどうかを確認してください。パブリック (外部) CA の場合、通常は追加する必要はありません。

第6章 KAFKA へのセキュアなアクセスの管理

各クライアントの Kafka ブローカーへのアクセスを管理することで、Kafka クラスターを保護できます。

Kafka ブローカーとクライアント間のセキュアな接続には、以下が含まれます。

- データ交換の暗号化
- アイデンティティー証明に使用する認証
- ユーザーが実行するアクションを許可または拒否する認可

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

6.1. 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。

6.1.1. リスナー認証

リスナーを作成して、Kafka ブローカーのクライアント認証を設定します。**Kafka** リソースの **Kafka.spec.kafka.listeners.authentication** プロパティーを使用して、リスナー認証タイプを指定します。

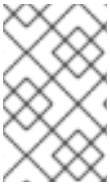
OpenShift クラスター内のクライアントの場合は、**plain** (暗号化なし) または **tls internal** リスナーを作成できます。**internal** リスナータイプは、ヘッドレスサービスと、ブローカー Pod に指定された DNS 名を使用します。ヘッドレスサービスの代わりに、内部リスナーの **cluster-ip** タイプを作成して、ブローカーごとの **ClusterIP** サービスを使用して Kafka を公開することもできます。OpenShift クラスター外のクライアントの場合、**外部** リスナーを作成し、**nodeport**、**loadbalancer**、**ingress**、または **route** (OpenShift の場合) 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. mTLS 認証 (TLS が有効な暗号化を使用するリスナーのみ)
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

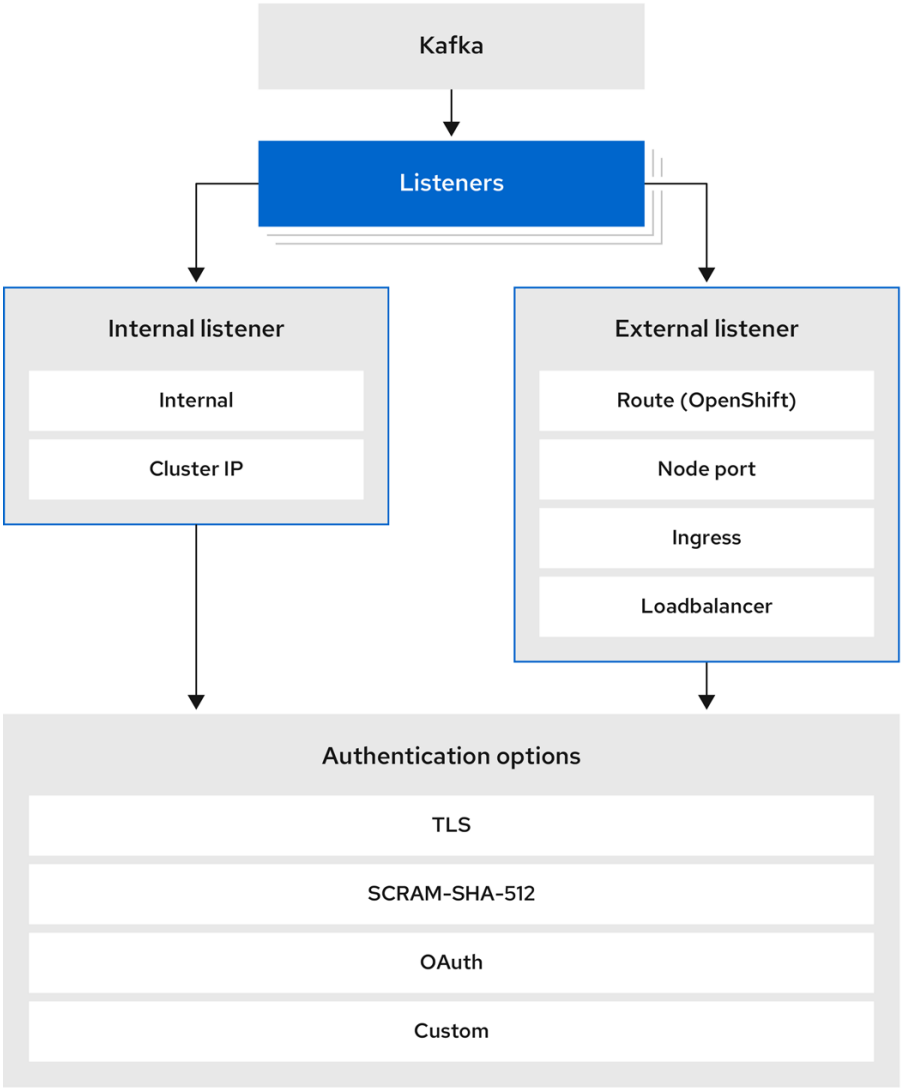
選択する認証オプションは、Kafka ブローカーへのクライアントアクセスを認証する方法によって異なります。



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6.1 Kafka リスナーの認証オプション



リスナーの **authentication** プロパティーは、そのリスナーに固有の認証メカニズムを指定するために使用されます。

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

認証は、User Operator を使用して **KafkaUsers** を管理する場合に設定する必要があります。

以下の例で指定されるものは次のとおりです。

- SCRAM-SHA-512 認証に設定された **plain** リスナー
- **mTLS** 認証を使用する TLS リスナー
- mTLS 認証を使用する **external** リスナー

各リスナーは、Kafka クラスター内で一意の名前およびポートで設定されます。

注記

ブローカー間通信 (9091 または 9090) およびメトリクス (9404) 用に確保されたポートを使用するようにリスナーを設定することはできません。

リスナー認証の設定例

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

6.1.1.1. mTLS 認証

mTLS 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。mTLS 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)。

Kafka リソースの mTLS リスナー設定には、次のものが必要です。

- TLS 暗号化とサーバー認証を指定する場合は **tls: true**
- クライアント認証を指定する場合は **authentication.type: tls**

Cluster Operator によって Kafka クラスターが作成されると、**<cluster_name>-cluster-ca-cert** という

名前の新しいシークレットが作成されます。シークレットには CA 証明書が含まれています。CA 証明書は PEM および PKCS #12 形式 です。Kafka クラスターを検証するには、CA 証明書をクライアント設定のトラストストアに追加します。クライアントを確認するには、クライアント設定のキーストアにユーザー証明書とキーを追加します。mTLS 用のクライアントの設定の詳細については、「ユーザー認証」 を参照してください。

注記

TLS 認証は一般的には一方向で、一方が他方のアイデンティティーを認証します。たとえば、Web ブラウザーと Web サーバーの間で HTTPS が使用される場合、ブラウザーは Web サーバーのアイデンティティーの証明を取得します。

6.1.1.2. 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。

SCRAM-SHA-512 認証が TLS 接続で使用される場合、TLS プロトコルは暗号化を提供しますが、認証には使用されません。

SCRAM の以下のプロパティーは、暗号化されていない接続でも SCRAM-SHA-512 を安全に使用できるようにします。

- 通信チャネル上では、パスワードはクリアテキストで送信されません。代わりに、クライアントとサーバーはお互いにチャレンジを生成し、認証するユーザーのパスワードを認識していることを証明します。
- サーバーとクライアントは、認証を交換するたびに新しいチャレンジを生成します。よって、この交換はリレー攻撃に対する回復性を備えています。

KafkaUser.spec.authentication.type が **scram-sha-512** で設定されている場合、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。

6.1.1.3. ネットワークポリシー

デフォルトでは、AMQ Streams では、Kafka ブローカーで有効になっているリスナーごとに **NetworkPolicy** リソースが自動的に作成されます。この **NetworkPolicy** により、アプリケーションはすべての namespace のリスナーに接続できます。リスナー設定の一部としてネットワークポリシーを使用します。

ネットワークレベルでのリスナーへのアクセスを指定のアプリケーションまたは namespace のみに制限するには、**networkPolicyPeers** プロパティーを使用します。リスナーごとに、異なる **networkPolicyPeers** 設定を指定できます。ネットワークポリシーピアの詳細は、NetworkPolicyPeer API reference を参照してください。

カスタムネットワークポリシーを使用する場合は、Cluster Operator 設定で **STRIMZI_NETWORK_POLICY_GENERATION** 環境変数を **false** に設定できます。詳細は、Cluster Operator configuration を参照してください。

注記

AMQ Streams でネットワークポリシーを使用するためには、OpenShift の設定が ingress **NetworkPolicies** をサポートしている必要があります。

6.1.1.4. 追加のリスナー設定オプション

GenericKafkaListenerConfiguration スキーマのプロパティーを使用して、設定をリスナーに追加できます。

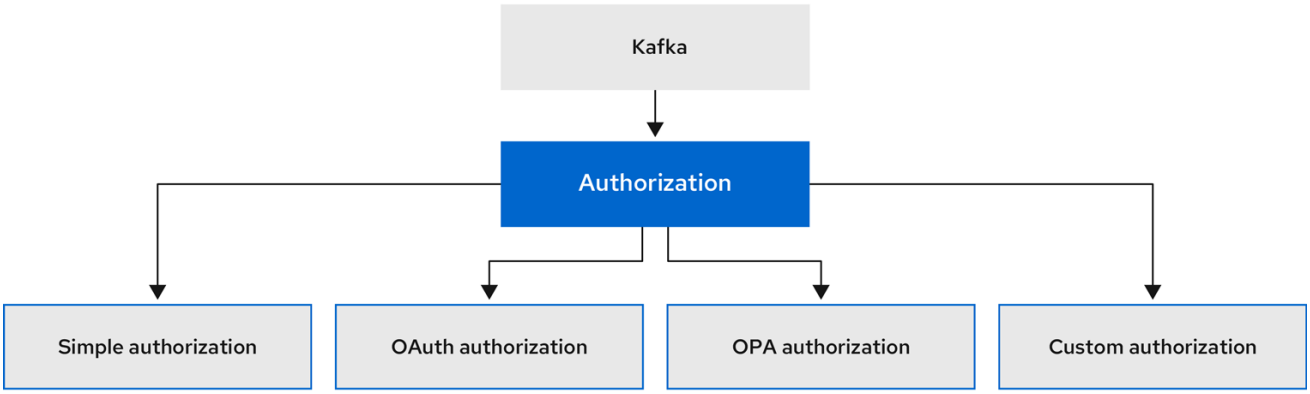
6.1.2. Kafka の承認

Kafka リソースの **Kafka.spec.kafka.authorization** プロパティーを使用して、Kafka ブローカーの承認を設定します。**authorization** プロパティーがないと、承認が有効になりず、クライアントには制限がありません。承認を有効にすると、承認は有効なすべてのリスナーに適用されます。承認方法は **type** フィールドで定義されます。

サポートされる承認オプションは次のとおりです。

- 簡易承認
- OAuth 2.0 での承認 (OAuth 2.0 トークンベースの認証を使用している場合)
- Open Policy Agent (OPA) での承認
- カスタム承認

図6.2 Kafka クラスター承認オプション



6.1.2.1. スーパーユーザー

スーパーユーザーは、アクセスの制限に関係なく Kafka クラスターのすべてのリソースにアクセスでき、すべての承認メカニズムでサポートされます。

Kafka クラスターのスーパーユーザーを指定するには、**superUsers** プロパティーにユーザープリンシパルのリストを追加します。ユーザーが mTLS 認証を使用する場合、ユーザー名は **CN=** で始まる TLS 証明書サブジェクトの共通名です。User Operator を使用せず、mTLS に独自の証明書を使用している場合、ユーザー名は完全な証明書サブジェクトです。完全な証明書サブジェクトには次のフィールドを含めることができます。**CN=user,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=my_country_code**存在しないフィールドは省略します。

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
        - CN=client_4,OU=my_ou,O=my_org,L=my_location,ST=my_state,C=US
        - CN=client_5,OU=my_ou,O=my_org,C=GB
        - CN=client_6,O=my_org
    # ...
```

6.2. KAFKA クライアントのセキュリティーオプション

KafkaUser リソースを使用して、Kafka クライアントの認証メカニズム、承認メカニズム、およびアクセス権限を設定します。セキュリティーの設定では、クライアントはユーザーとして表されます。

Kafka ブローカーへのユーザーアクセスを認証および承認できます。認証によってアクセスが許可され、承認によって許容されるアクションへのアクセスが制限されます。

Kafka ブローカーへのアクセスが制限されない **スーパーユーザー** を作成することもできます。

認証および承認メカニズムは、Kafka ブローカーへのアクセスに使用されるリスナーの仕様 と一致する必要があります。

Kafka ブローカーへのアクセス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
- リスナーを使用して Kafka クラスターへのクライアントアクセスを設定する

6.2.1. 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。

6.2.2. ユーザー認証

KafkaUser カスタムリソースを使用して、Kafka クラスターへのアクセスを必要とするユーザー (クライアント) の認証認証情報を設定します。**KafkaUser.spec** の **authentication** プロパティーを使用して認証情報を設定します。**type** を指定することで、生成される認証情報を制御します。

サポートされる認証タイプ

- mTLS 認証用の **tls**
- 外部証明書を使用した mTLS 認証用の **tls-external**
- **scram-sha-512**(SCRAM-SHA-512 認証用)

tls または **scram-sha-512** が指定された場合、User Operator がユーザーを作成する際に、認証用のクレデンシャルを作成します。**tls-external** が指定されている場合、ユーザーは引き続き mTLS を使用しますが、認証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 の外部で発行された証明書を使用して mTLS で認証できます。User Operator は TLS 証明書またはシークレットを生成しません。**tls** メカニズムを使用する場合と同様に、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。

6.2.2.1. mTLS 認証

mTLS 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。

mTLS 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
  # ...
```

認証タイプは、Kafka クラスターへのアクセスに使用される **Kafka** リスナーの同等の設定と一致する必要があります。

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、mTLS の秘密鍵と公開鍵が含まれています。公開鍵はユーザー証明書に含まれており、作成時にクライアント CA (認証局) によって署名されます。すべての鍵は X.509 形式です。

注記

Cluster Operator によって生成されたクライアント CA を使用している場合、Cluster Operator によってクライアント CA が更新されると、User Operator によって生成されたユーザー証明書も更新されます。

ユーザーシークレットは、キーと証明書を PEM および PKCS #12 形式で提供します 。

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:
  ca.crt: <public_key> # Public key of the clients CA
  user.crt: <user_certificate> # Public key of the user
  user.key: <user_private_key> # Private key of the user
  user.p12: <store> # PKCS #12 store for user certificates and keys
  user.password: <password_for_store> # Protects the PKCS #12 store
```

クライアントを設定するときは、次を指定します。

- Kafka クラスターの ID を検証するためのパブリッククラスター CA 証明書の **トラストストア** プロパティー
- クライアントを検証するためのユーザー認証クレデンシャルの **キーストア** プロパティー

設定は、ファイル形式 (PEM または PKCS #12) によって異なります。この例では、PKCS #12 ストアと、ストア内の認証情報にアクセスするために必要なパスワードを使用しています。

PKCS #12 形式の mTLS を使用したクライアント設定の例

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

1. Kafka クラスターに接続するためのブートストラップサーバーアドレス。
2. 暗号化に TLS を使用する場合のセキュリティープロトコルオプション。
3. トラストストアの場所には、Kafka クラスターの公開鍵証明書 (**ca.p12**) が含まれます。クラスター CA 証明書とパスワードは、Kafka クラスターの作成時に **<cluster_name>-cluster-ca-cert** シークレットで Cluster Operator によって生成されます。
4. トラストストアにアクセスするためのパスワード (**ca.password**)。

5 キーストアの場所には、Kafka ユーザーの公開鍵証明書 (**user.p12**) が含まれます。

6 キーストアにアクセスするためのパスワード (**user.password**)。

6.2.2.2. User Operator の外部で発行された証明書を使用した mTLS 認証

User Operator の外部で発行された証明書を使用して mTLS 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 以外で発行された証明書を使用する mTLS 認証を使用するユーザーの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: tls-external
  # ...
```

6.2.2.3. SCRAM-SHA-512 認証

SCRAM-SHA-512 認証メカニズムを使用するには、**KafkaUser** リソースの **type** フィールドを **scram-sha-512** に設定します。

SCRAM-SHA-512 認証が有効になっているユーザーの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  authentication:
    type: scram-sha-512
  # ...
```

ユーザーが User Operator によって作成されると、**KafkaUser** リソースと同じ名前で新しいシークレットが作成されます。シークレットの **password** キーには、生成されたパスワードが含まれ、base64 でエンコードされます。パスワードを使用するにはデコードする必要があります。

ユーザー認証情報を含むシークレットの例

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
  labels:
    strimzi.io/kind: KafkaUser
```

```
    strimzi.io/cluster: my-cluster
type: Opaque
data:
  password: Z2VuZXJhdGVkcGFzc3dvcmQ= (1)
  sasl.jaas.config:
b3JnLmFwYWNoZS5rYWZrYS5jb21tb24uc2VjdXJpdHkuc2NyYW0uU2NyYW1Mb2dpbk1vZHVsZSByZXF1aXJlZCB1c2VybmFtZT0ibXktdXNlciIgcGFzc3dvcmQ9ImdlbmVyYXRlZHBhc3N3b3JkIjsK (2)
```

(1) base64 でエンコードされた生成されたパスワード。

(2) base64 でエンコードされた SASL SCRAM-SHA-512 認証の JAAS 設定文字列。

生成されたパスワードをデコードします。

```
echo "Z2VuZXJhdGVkcGFzc3dvcmQ=" | base64 --decode
```

6.2.2.3.1. 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。

6.2.3. ユーザーの承認

KafkaUser カスタムリソースを使用して、Kafka クラスターへのアクセスを必要とするユーザー (クライアント) の承認規則を設定します。**KafkaUser.spec** の **authorization** プロパティーを使用してルールを設定します。**type** を指定することで、使用するルールを制御します。

簡易承認を使用するには、**KafkaUser.spec.authorization** で **type** プロパティーを **simple** に設定します。簡易承認は、Kafka Admin API を使用して Kafka クラスター内で ACL ルールを管理します。User Operator の ACL 管理が有効であるかどうかは、Kafka クラスターの承認設定によって異なります。

- 簡易承認では、ACL 管理が常に有効になります。
- OPA 承認の場合、ACL 管理は常に無効になります。承認ルールは OPA サーバーで設定されます。
- Red Hat Single Sign-On の承認では、Red Hat Single Sign-On で ACL ルールを直接管理できます。設定のフォールバックオプションとして、承認を簡単なオーソライザーに委譲することもできます。簡単なオーソライザーへの委譲が有効になっている場合、User Operator は ACL ルールの管理も有効にします。
- カスタム承認プラグインを使用したカスタム承認では、**Kafka** カスタムリソースの **.spec.kafka.authorization** 設定の **supportsAdminApi** プロパティーを使用して、サポートを有効または無効にする必要があります。

承認はクラスター全体です。認証タイプは、**Kafka** カスタムリソースの同等の設定と一致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。

6.2.3.1. ACL ルール

AclAuthorizer は ACL ルールを使用して Kafka ブローカーへのアクセスを管理します。

ACL ルールによって、**acls** プロパティーで指定したユーザーにアクセス権限が付与されます。

AclRule オブジェクトの詳細は、**AclRule** schema reference を参照してください。

6.2.3.2. 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。

6.2.3.3. ユーザークォータ

KafkaUser リソースの **spec** を設定してクォータを強制し、ユーザーが Kafka ブローカーへの設定されたアクセスレベルを超えないようにします。サイズベースのネットワーク使用量と時間ベースの CPU 使用率のしきい値を設定できます。また、パーティション mutation (変更) クォータを追加して、ユーザー要求に対して受け入れられるパーティション変更のリクエストのレートを制御することもできます。

ユーザークォータをともなう KafkaUser の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  name: my-user
  labels:
    strimzi.io/cluster: my-cluster
spec:
  # ...
  quotas:
    producerByteRate: 1048576 (1)
    consumerByteRate: 2097152 (2)
    requestPercentage: 55 (3)
    controllerMutationRate: 10 (4)
```

1. ユーザーが Kafka ブローカーにプッシュできるデータ量の、秒あたりのバイトクォータ。
2. ユーザーが Kafka ブローカーからフェッチできるデータ量の、秒あたりのバイトクォータ。
3. クライアントグループあたりの時間割合で示される、CPU 使用制限。
4. 1 秒あたり許容される同時パーティション作成および削除操作 (mutations) の数

これらのプロパティーの詳細は、**KafkaUserQuotas** schema reference を参照してください。

6.3. KAFKA ブローカーへのアクセスのセキュア化

Kafka ブローカーへのセキュアなアクセスを確立するには、以下を設定し、適用します。

- 以下を行う **Kafka** リソース。
  - 指定された認証タイプでリスナーを作成します。
  - Kafka クラスター全体の承認を設定します。
- Kafka ブローカーにリスナー経由でセキュアにアクセスするための **KafkaUser** リソース。

Kafka リソースを設定して以下を設定します。

- リスナー認証
- Kafka リスナーへのアクセスを制限するネットワークポリシー
- Kafka の承認
- ブローカーへのアクセスが制限されないスーパーユーザー

認証は、リスナーごとに独立して設定されます。承認は、常に Kafka クラスター全体に対して設定されます。

Cluster Operator はリスナーを作成し、クラスターおよびクライアント認証局 (CA) 証明書を設定して Kafka クラスター内で認証を有効にします。

独自の証明書をインストールして、Cluster Operator によって生成された証明書を置き換えることができます。外部 CA (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ーの詳細は、スキーマリファレンス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

6.3.1. Kafka ブローカーのセキュア化

この手順では、AMQ Streams の実行時に Kafka ブローカーをセキュアにするためのステップを説明します。

Kafka ブローカーに実装されたセキュリティーは、アクセスを必要とするクライアントに実装されたセキュリティーとの互換性を維持する必要があります。

- **Kafka.spec.kafka.listeners[*].authentication** matches **KafkaUser.spec.authentication**
- **Kafka.spec.kafka.authorization** は **KafkaUser.spec.authorization** と一致します。

この手順では、mTLS 認証を使用した簡易承認とリスナーの設定を説明します。リスナーの設定の詳細については、**GenericKafkaListener** schema reference を参照してください。

代わりに、リスナー認証 には SCRAM-SHA または OAuth 2.0、 Kafka 承認 には OAuth 2.0 または OPA を使用することができます。

手順

1. **Kafka** リソースを設定します。
   a. 承認には **authorization** プロパティーを設定します。
   b. **listeners** プロパティーを設定し、認証でリスナーを作成します。
   以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
```

```
    # ...
    authorization: 1
      type: simple
      superUsers: 2
        - CN=client_1
        - user_2
        - CN=client_3
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls 3
    # ...
  zookeeper:
    # ...
```

1 Authorization は、**AclAuthorizer** Kafka プラグインを使用して、Kafka ブローカーでの **simple** な承認を可能にします。

2 Kafka へのアクセスを制限されないユーザープリンシパルのリスト。**CN** は、mTLS 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、mTLS、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 クラスターは、mTLS 認証を使用する Kafka ブローカーリスナーと共に設定されます。

Kafka ブローカー Pod ごとにサービスが作成されます。

サービスが作成され、Kafka クラスターに接続するための **ブートストラップアドレス** として機能します。

kafka ブローカーのアイデンティティー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。

6.3.2. 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 クラスターへの安全なアクセスを必要とするクライアントを表します。

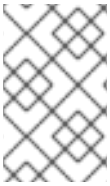
KafkaUser 認証および承認メカニズムを設定する場合、必ず同等の **Kafka** 設定と一致するようにしてください。

- **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。

この手順は、mTLS 認証を使用してユーザーを作成する方法を示しています。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、mTLS が Kafka 設定で有効になっていない場合は、mTLS でユーザーを認証できません。

前提条件

- mTLS 認証と TLS 暗号化を使用する Kafka ブローカーリスナーで設定された 実行中の Kafka クラスター。
- 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: 1
    type: tls
  authorization:
    type: simple 2
    acls:
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operations:
          - Describe
          - Read
      - resource:
          type: group
          name: my-group
          patternType: literal
        operations:
          - Read
```

1 相互 **tls** または **scram-sha-512** として定義されたユーザー認証メカニズム。

2 ACL ルールのリストが必要な簡易承認。

2. **KafkaUser** リソースを作成または更新します。

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

KafkaUser リソースと同じ名前の Secret と共に、ユーザーが作成されます。Secret には、mTLS 認証用の秘密鍵と公開鍵が含まれています。

Kafka ブローカーへの安全な接続のためのプロパティーを使用して Kafka クライアントを設定する方法の詳細は、リスナーを使用した Kafka クラスターへのクライアントアクセスのセットアップ を参照してください。

6.3.3. ネットワークポリシーを使用した Kafka リスナーへのアクセス制限

networkPolicyPeers プロパティーを使用すると、リスナーへのアクセスを指定のアプリケーションのみに制限できます。

前提条件

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

手順

1. **Kafka** リソースを開きます。
2. **networkPolicyPeers** プロパティーで、Kafka クラスターへのアクセスが許可されるアプリケーション Pod または namespace を定義します。
   以下は、ラベル **app** が **kafka-client** に設定されているアプリケーションからの接続のみを許可するよう **tls** リスナーを設定する例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
        networkPolicyPeers:
          - podSelector:
              matchLabels:
                app: kafka-client
    # ...
  zookeeper:
    # ...
```

3. リソースを作成または更新します。

次のように **oc apply** を使用します。

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

関連情報

- **networkPolicyPeers** の設定
- NetworkPolicyPeer API リファレンス

6.4. 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承認を有効にし、中央の承認サーバーを使用してリソースに制限されたアクセス権限を付与するトークンを発行します。

OAuth 2.0 認証を設定した後に OAuth 2.0 承認 を設定できます。

Kafka ブローカーおよびクライアントの両方が OAuth 2.0 を使用するように設定する必要があります。OAuth 2.0 認証は、**simple** または OPA ベースの Kafka authorization と併用することもできます。

OAuth 2.0 のトークンベースの認証を使用すると、アプリケーションクライアントはアカウントのクレデンシャルを公開せずにアプリケーションサーバー (**リソースサーバー** と呼ばれる) のリソースにアクセスできます。

アプリケーションクライアントは、アクセストークンを認証の手段として渡します。アプリケーションサーバーはこれを使用して、付与するアクセス権限のレベルを決定することもできます。承認サーバーは、アクセスの付与とアクセスに関する問い合わせを処理します。

AMQ Streams のコンテキストでは以下が行われます。

- Kafka ブローカーは OAuth 2.0 リソースサーバーとして動作します。
- Kafka クライアントは OAuth 2.0 アプリケーションクライアントとして動作します。

Kafka クライアントは Kafka ブローカーに対して認証を行います。ブローカーおよびクライアントは、必要に応じて OAuth 2.0 承認サーバーと通信し、アクセストークンを取得または検証します。

AMQ Streams のデプロイメントでは、OAuth 2.0 インテグレーションは以下を提供します。

- Kafka ブローカーのサーバー側 OAuth 2.0 サポート
- Kafka MirrorMaker、Kafka Connect、および Kafka Bridge のクライアント側 OAuth 2.0 サポート。

6.4.1. OAuth 2.0 認証メカニズム

AMQ Streams は、OAuth 2.0 認証で OAUTHBEARER および PLAIN メカニズムをサポートします。どちらのメカニズムも、Kafka クライアントが Kafka ブローカーで認証されたセッションを確立できるようにします。クライアント、承認サーバー、および Kafka ブローカー間の認証フローは、メカニズムごとに異なります。

可能な限り、OAUTHBEARER を使用するようにクライアントを設定することが推奨されます。

OAUTHBEARER では、クライアントクレデンシャルは Kafka ブローカーと **共有されることがない** ため、PLAIN よりも高レベルのセキュリティーが提供されます。OAUTHBEARER をサポートしない Kafka クライアントの場合のみ、PLAIN の使用を検討してください。

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ーは、**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

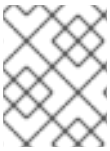
OAUTHBEARER は、Kafka ブローカーの **oauth** リスナー設定で自動的に有効になります。**enableOauthBearer** プロパティーを **true** に設定できますが、これは必須ではありません。

```
# ...
authentication:
  type: oauth
  # ...
  enableOauthBearer: true
```

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承認サーバーに接続し、アクセストークンを取得する必要があります。

OAUTHBEARER を使用する場合、クライアントはクレデンシャルを交換するために Kafka ブローカーでセッションを開始します。ここで、クレデンシャルはコールバックハンドラーによって提供されるベアラートークンの形式を取ります。コールバックを使用して、以下の 3 つの方法のいずれかでトークンの提供を設定できます。

- クライアント ID および Secret (**OAuth 2.0 クライアントクレデンシャルメカニズム** を使用)
- 設定時に手動で取得された有効期限の長いアクセストークン
- 設定時に手動で取得された有効期限の長い更新トークン



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承認サーバーの背後で一元的に処理されます。

OAuth 2.0 over PLAIN コールバックを併用する場合、以下のいずれかの方法を使用して Kafka クライアントは Kafka ブローカーで認証されます。

- クライアント ID およびシークレット (OAuth 2.0 クライアントクレデンシャルメカニズムを使用)
- 設定時に手動で取得された有効期限の長いアクセストークン

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** プロパティー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ー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** プロパティー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** プロパティーを使用すると、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、承認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。

注記

OAuth over PLAIN は、**password grant** メカニズムをサポートしていません。上記のように、SASL PLAIN メカニズムを介して、**client credentials** (clientId + シークレット) またはアクセストークンをプロキシーすることしかできません。

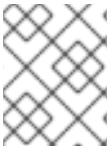
関連情報

- 「Kafka ブローカーの OAuth 2.0 サポートの設定」

6.4.2. OAuth 2.0 Kafka ブローカーの設定

OAuth 2.0 の Kafka ブローカー設定には、以下が関係します。

- 承認サーバーでの OAuth 2.0 クライアントの作成
- Kafka カスタムリソースでの OAuth 2.0 認証の設定



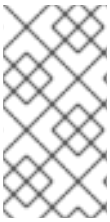
注記

承認サーバーに関連する Kafka ブローカーおよび Kafka クライアントはどちらも OAuth 2.0 クライアントと見なされます。

6.4.2.1. 承認サーバーの OAuth 2.0 クライアント設定

セッションの開始中に受信されたトークンを検証するように Kafka ブローカーを設定するには、承認サーバーで OAuth 2.0 の **クライアント** 定義を作成し、以下のクライアントクレデンシャルが有効な状態で **機密情報** として設定することが推奨されます。

- **kafka** のクライアント ID (例)
- 認証メカニズムとしてのクライアント ID およびシークレット



注記

承認サーバーのパブリックでないイントロスペクションエンドポイントを使用する場合のみ、クライアント ID およびシークレットを使用する必要があります。高速のローカル JWT トークンの検証と同様に、パブリック承認サーバーのエンドポイントを使用する場合は、通常クレデンシャルは必要ありません。

6.4.2.2. 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
      #...
```

リスナーで OAuth 2.0 認証を設定できます。OAuth 2.0 認証と TLS 暗号化 (**tls: true**) を併用することをお勧めします。暗号化を行わないと、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に対して接続が脆弱になります。

external リスナーを **type: oauth** で設定し、セキュアなトランスポート層がクライアントと通信するようにします。

OAuth 2.0 の外部リスナーとの使用

```
# ...
listeners:
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: oauth
    #...
```

tls プロパティーはデフォルトで **false** に設定されているため、有効にする必要があります。

認証のタイプを OAuth 2.0 として定義した場合、検証のタイプに基づいて、 高速のローカル JWT 検証 または イントロスペクションエンドポイントを使用したトークンの検証 のいずれかとして、設定を追加します。

説明や例を用いてリスナー向けに OAuth 2.0 を設定する手順は、Kafka ブローカーの OAuth 2.0 サポートの設定 を参照してください。

6.4.2.3. 高速なローカル JWT トークン検証の設定

高速なローカル JWT トークンの検証では、JWT トークンの署名がローカルでチェックされます。

ローカルチェックでは、トークンに対して以下が確認されます。

- アクセストークンに **Bearer** の (**typ**) 要求値が含まれ、トークンがタイプに準拠することを確認します。
- 有効であるか (期限切れでない) を確認します。
- トークンに **validIssuerURI** と一致する発行元があることを確認します。

リスナーの設定時に **validIssuerURI** 属性を指定することで、認証サーバーから発行されていないトークンは拒否されます。

高速のローカル JWT トークン検証の実行中に、承認サーバーの通信は必要はありません。OAuth 2.0 の承認サーバーによって公開されるエンドポイントの **jwksEndpointUri** 属性を指定して、高速のローカル JWT トークン検証をアクティベートします。エンドポイントには、署名済み JWT トークンの検証に使用される公開鍵が含まれます。これらは、Kafka クライアントによってクレデンシャルとして送信されます。

注記

承認サーバーとの通信はすべて TLS による暗号化を使用して実行する必要があります。

証明書トラストストアを AMQ Streams プロジェクト namespace の OpenShift シークレットとして設定し、**tlsTrustedCertificates** 属性を使用してトラストストアファイルが含まれる OpenShift シークレットを示すことができます。

JWT トークンからユーザー名を適切に取得するため、**userNameClaim** の設定を検討してください。Kafka ACL 承認を使用する場合は、認証中にユーザー名でユーザーを特定する必要があります。JWT トークンの **sub** 要求は、通常は一意な ID でユーザー名ではありません。

高速なローカル JWT トークン検証の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    #...
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          jwksEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-
connect/certs>
          userNameClaim: preferred_username
          maxSecondsWithoutReauthentication: 3600
          tlsTrustedCertificates:
          - secretName: oauth-server-cert
            certificate: ca.crt
    #...
```

6.4.2.4. OAuth 2.0 イントロスペクションエンドポイントの設定

OAuth 2.0 のイントロスペクションエンドポイントを使用したトークンの検証では、受信したアクセストークンは不透明として対処されます。Kafka ブローカーは、アクセストークンをイントロスペクションエンドポイントに送信します。このエンドポイントは、検証に必要なトークン情報を応答として返します。ここで重要なのは、特定のアクセストークンが有効である場合は最新情報を返すことで、トークンの有効期限に関する情報も返します。

OAuth 2.0 のイントロスペクションベースの検証を設定するには、高速のローカル JWT トークン検証に指定された **jwksEndpointUri** 属性ではなく、**introspectionEndpointUri** 属性を指定します。通常、イントロスペクションエンドポイントは保護されているため、承認サーバーに応じて **clientId** および **clientSecret** を指定する必要があります。

イントロスペクションエンドポイントの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    listeners:
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: oauth
          clientId: kafka-broker
          clientSecret:
            secretName: my-cluster-oauth
            key: clientSecret
          validIssuerUri: <https://<auth-server-address>/auth/realms/tls>
          introspectionEndpointUri: <https://<auth-server-address>/auth/realms/tls/protocol/openid-
connect/token/introspect>
```

```
    userNameClaim: preferred_username
    maxSecondsWithoutReauthentication: 3600
    tlsTrustedCertificates:
    - secretName: oauth-server-cert
      certificate: ca.crt
```

6.4.3. Kafka ブローカーの再認証の設定

Kafka クライアントと Kafka ブローカー間の OAuth 2.0 セッションに Kafka **session re-authentication** を使用するように、**oauth** リスナーを設定できます。このメカニズムは、定義された期間後に、クライアントとブローカー間の認証されたセッションを期限切れにします。セッションの有効期限が切れると、クライアントは既存のコネクションを破棄せずに再使用して、新しいセッションを即座に開始します。

セッションの再認証はデフォルトで無効になっています。これを有効にするには、**oauth** リスナー設定で **maxSecondsWithoutReauthentication** の時間値を設定します。OAUTHBEARER および PLAIN 認証では、同じプロパティーを使用してセッションの再認証が設定されます。設定例については、「Kafka ブローカーの OAuth 2.0 サポートの設定」 を参照してください。

セッションの再認証は、クライアントによって使用される Kafka クライアントライブラリーによってサポートされる必要があります。

セッションの再認証は、**高速ローカル JWT** または **イントロスペクションエンドポイント** のトークン検証と使用できます。

クライアントの再認証

ブローカーの認証されたセッションが期限切れになると、クライアントは接続を切断せずに新しい有効なアクセストークンをブローカーに送信し、既存のセッションを再認証する必要があります。

トークンの検証に成功すると、既存の接続を使用して新しいクライアントセッションが開始されます。クライアントが再認証に失敗した場合、さらにメッセージを送受信しようとすると、ブローカーは接続を閉じます。ブローカーで再認証メカニズムが有効になっていると、Kafka クライアントライブラリー 2.2 以降を使用する Java クライアントが自動的に再認証されます。

更新トークンが使用される場合、セッションの再認証は更新トークンにも適用されます。セッションが期限切れになると、クライアントは更新トークンを使用してアクセストークンを更新します。その後、クライアントは新しいアクセストークンを使用して既存のセッションに再認証されます。

OAUTHBEARER および PLAIN のセッションの有効期限

セッションの再認証が設定されている場合、OAUTHBEARER と PLAIN 認証ではセッションの有効期限は異なります。

クライアント ID とシークレットによる方法を使用する OAUTHBEARER および PLAIN の場合:

- ブローカーの認証されたセッションは、設定された **maxSecondsWithoutReauthentication** で期限切れになります。
- アクセストークンが設定期間前に期限切れになると、セッションは設定期間前に期限切れになります。

有効期間の長いアクセストークンによる方法を使用する PLAIN の場合:

- ブローカーの認証されたセッションは、設定された **maxSecondsWithoutReauthentication** で期限切れになります。

- アクセストークンが設定期間前に期限切れになると、再認証に失敗します。セッションの再認証は試行されますが、PLAIN にはトークンを更新するメカニズムがありません。

maxSecondsWithoutReauthentication が **設定されていない** 場合、OAUTHBEARER および PLAIN クライアントは、再認証しなくてもブローカーへの接続を無限に保持できます。認証されたセッションは、アクセストークンの期限が切れても終了しません。ただし、**keycloak** 承認を使用したり、カスタムオーソライザーをインストールして、承認を設定する場合に考慮できます。

関連情報

- 「OAuth 2.0 Kafka ブローカーの設定」
- 「Kafka ブローカーの OAuth 2.0 サポートの設定」
- KafkaListenerAuthenticationOAuth スキーマ参照
- KIP-368

6.4.4. OAuth 2.0 Kafka クライアントの設定

Kafka クライアントは以下のいずれかで設定されます。

- 承認サーバーから有効なアクセストークンを取得するために必要なクレデンシャル (クライアント ID およびシークレット)。
- 承認サーバーから提供されたツールを使用して取得された、有効期限の長い有効なアクセストークンまたは更新トークン。

アクセストークンは、Kafka ブローカーに送信される唯一の情報です。アクセストークンを取得するために承認サーバーでの認証に使用されるクレデンシャルは、ブローカーに送信されません。

クライアントによるアクセストークンの取得後、承認サーバーと通信する必要はありません。

クライアント ID とシークレットを使用した認証が最も簡単です。有効期間の長いアクセストークンまたは更新トークンを使用すると、承認サーバーツールに追加の依存関係があるため、より複雑になります。

注記

有効期間が長いアクセストークンを使用している場合は、承認サーバーでクライアントを設定し、トークンの最大有効期間を長くする必要があります。

Kafka クライアントが直接アクセストークンで設定されていない場合、クライアントは承認サーバーと通信して Kafka セッションの開始中にアクセストークンのクレデンシャルを交換します。Kafka クライアントは以下のいずれかを交換します。

- クライアント ID およびシークレット
- クライアント ID、更新トークン、および (任意の) シークレット
- ユーザー名とパスワード、およびクライアント ID と (オプションで) シークレット

6.4.5. 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、使用する承認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** プロパティーとして渡されます。

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。

- 承認サーバーと通信しない、JWT の署名確認およびローカルトークンのイントロスペクションをベースとした高速なローカルトークン検証を使用できます。承認サーバーは、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。
- 承認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承認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

承認サーバーは不透明なアクセストークンの使用のみを許可する可能性があり、この場合はローカルトークンの検証は不可能です。

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。

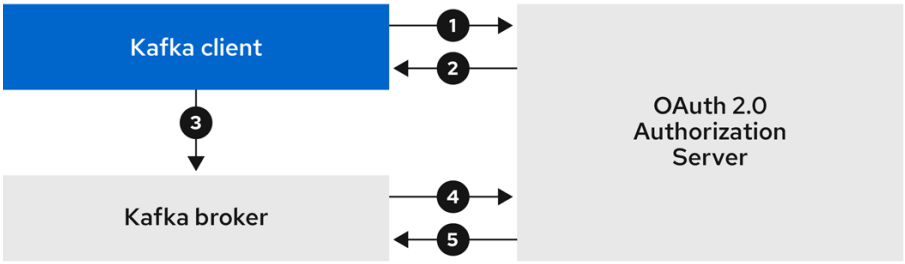
- 以前に生成された有効期間の長いアクセストークンを使用した直接ローカルアクセス
- 新しいアクセストークンを発行するには、認可サーバーに連絡します (クライアント ID とシークレット、または更新トークン、またはユーザー名とパスワードを使用)。

6.4.5.1. 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。

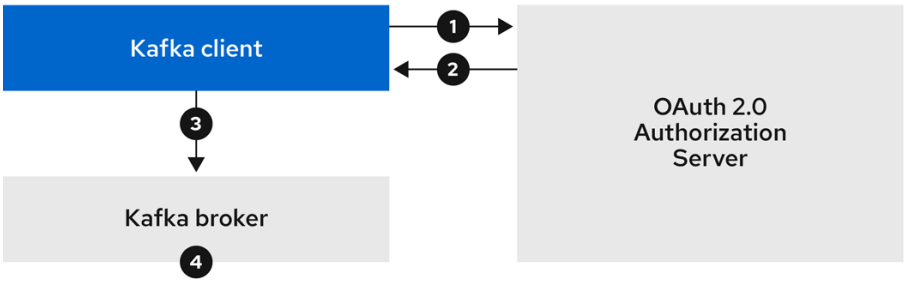
- クライアントがクライアント ID とシークレットを使用し、ブローカーが検証を承認サーバーに委任する場合
- クライアントがクライアント ID およびシークレットを使用し、ブローカーが高速のローカルトークン検証を実行する場合
- クライアントが有効期限の長いアクセストークンを使用し、ブローカーが検証を承認サーバーに委任する場合
- クライアントが有効期限の長いアクセストークンを使用し、ブローカーが高速のローカル検証を実行する場合

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承認サーバーに委任する場合



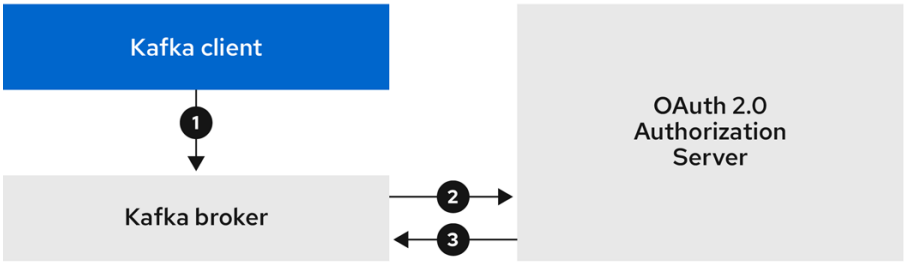
1. Kafka クライアントは、クライアント ID およびシークレットを使用して承認サーバーからアクセストークンを要求し、必要に応じて更新トークンを要求します。または、クライアントはユーザー名とパスワードを使用して認証することもできます。
2. 承認サーバーは新しいアクセストークンを生成します。
3. Kafka クライアントは、SASL OAUTHBEARER メカニズムを使用してアクセストークンを渡すことで Kafka ブローカーで認証されます。
4. Kafka ブローカーは、独自のクライアント ID およびシークレットを使用し、承認サーバーでトークンイントロスペクションエンドポイントを呼び出すことで、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



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。または、クライアントはユーザー名とパスワードを使用して認証することもできます。
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承認サーバーに委任する場合



1. Kafka クライアントは、SASL OAUTHBEARER メカニズムを使用して、有効期限の長いアクセストークンを渡すために 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、承認サーバーでトークンイントロスペクションエンドポイントを呼び出し、アクセストークンを検証します。
3. トークンが有効な場合、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

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

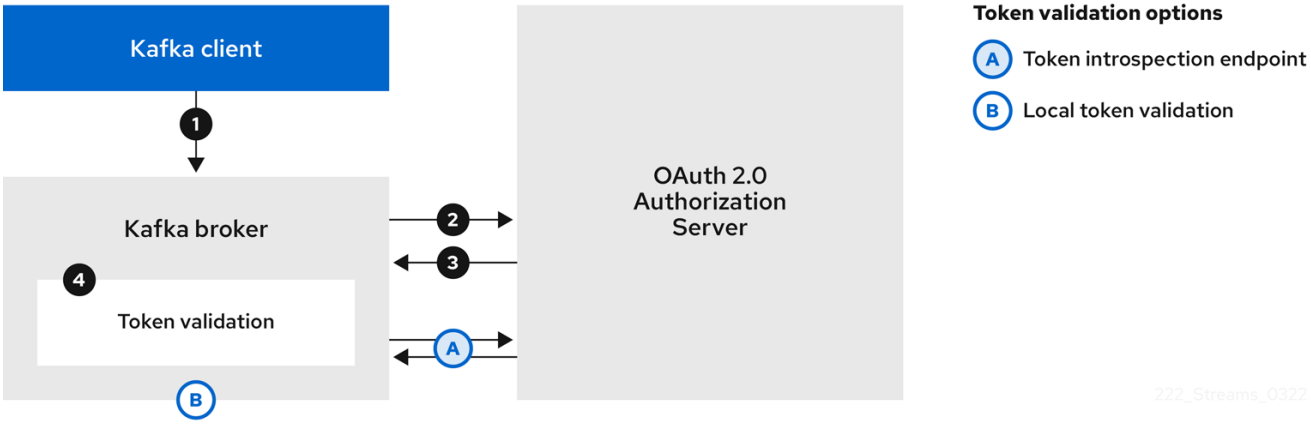
6.4.5.2. SASL PLAIN メカニズムを使用したクライアント認証フローの例

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

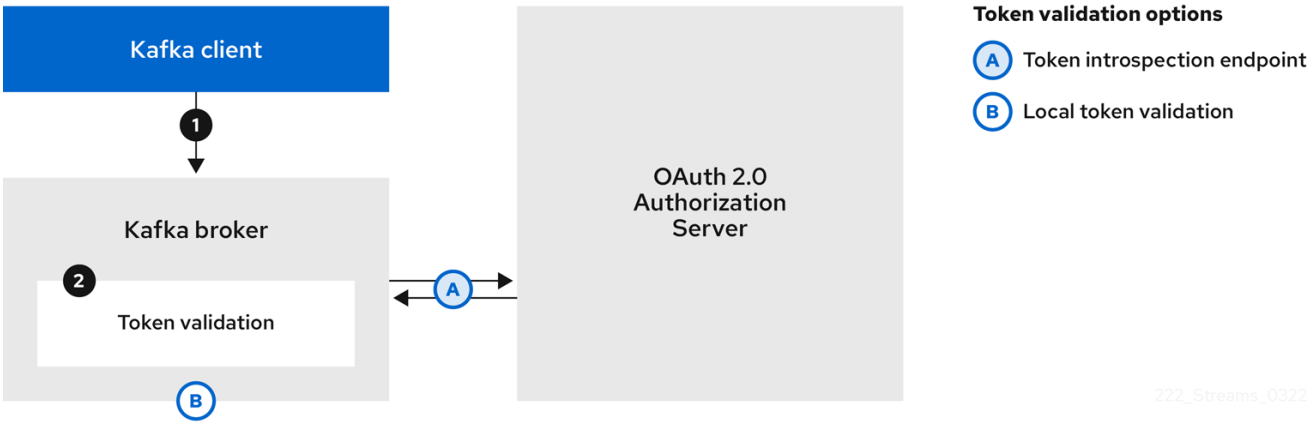
- クライアントが、クライアント ID およびシークレットなしで有効期限の長いアクセストークンを使用する場合

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** を承認サーバーに渡します。
3. 承認サーバーは、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 ブローカーは承認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、要求は承認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 ブローカーは承認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、承認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。

6.4.6. OAuth 2.0 認証の設定

OAuth 2.0 は、Kafka クライアントと AMQ Streams コンポーネントとの対話に使用されます。

AMQ Streams に OAuth 2.0 を使用するには、以下を行う必要があります。

1. 承認サーバーをデプロイし、AMQ Streams と統合するためにそのデプロイメントを設定します。
2. OAuth 2.0 を使用するよう設定された Kafka ブローカーリスナーで Kafka クラスターをデプロイまたは更新
3. OAuth 2.0 を使用するように Java ベースの Kafka クライアントを更新します 。
4. OAuth 2.0 を使用するように Kafka コンポーネントクライアントを更新します 。

6.4.6.1. OAuth 2.0 承認サーバーの設定

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、選択した承認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

承認サーバーが既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

承認サーバーのデプロイおよび設定後に、Kafka ブローカーが OAuth 2.0 を使用するように設定 します。

6.4.6.2. Kafka ブローカーの OAuth 2.0 サポートの設定

この手順では、ブローカーリスナーが承認サーバーを使用して OAuth 2.0 認証を使用するように、Kafka ブローカーを設定する方法について説明します。

tls: true を使用したリスナーを介して、暗号化されたインターフェイスで OAuth 2.0 を使用することをお勧めします。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、適切な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 の承認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未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

(6) (任意設定): 承認サーバーへの TLS 接続用の信用できる証明書。

(7) (任意設定): TLS ホスト名の検証を無効にします。デフォルトは **false** です。

(8) JWKS 証明書が期限切れになる前に有効であるとみなされる期間。デフォルトは **360** 秒です。デフォルトよりも長い時間を指定する場合は、無効になった証明書へのアクセスが許可されるリスクを考慮してください。

(9) JWKS 証明書を更新する間隔。この間隔は、有効期間よりも 60 秒以上短くする必要があります。デフォルトは **300** 秒です。

(10) JWKS 公開鍵の更新が連続して試行される間隔の最小一時停止時間 (秒単位)。不明な署名キーが検出されると、JWKS キーの更新は、最後に更新を試みてから少なくとも指定された期間は一時停止し、通常の定期スケジュール以外でスケジュールされます。キーの更新は指数バックオフ (指数バックオフ) のルールに従い、**jwksRefreshSeconds** に到達する

まで、一時停止を増やして失敗した更新を再試行します。デフォルト値は 1 です。

例 2: イントロスペクションエンドポイントを使用したトークンの検証の設定

```
- name: external
  port: 9094
  type: loadbalancer
  tls: true
  authentication:
    type: oauth
    validIssuerUri: <https://<auth-server-address>/auth/realms/external>
    introspectionEndpointUri: <https://<auth-server-address>/auth/realms/external/protocol/openid-connect/token/introspect> (1)
    clientId: kafka-broker (2)
    clientSecret: (3)
      secretName: my-cluster-oauth
      key: clientSecret
    userNameClaim: preferred_username (4)
    maxSecondsWithoutReauthentication: 3600 (5)
```

(1) トークンイントロスペクションエンドポイントの URI。

(2) クライアントを識別するためのクライアント ID。

(3) 認証にはクライアントシークレットとクライアント ID が使用されます。

(4) トークンの実際のユーザー名が含まれるトークン要求 (またはキー)。ユーザー名は、ユーザーの識別に使用される **principal** です。**userNameClaim** の値は、使用される承認サーバーによって異なります。

(5) (任意設定): セッションの有効期限がアクセストークンと同じ期間になるよう強制する Kafka の再認証メカニズムを有効にします。指定された値がアクセストークンの有効期限が切れるまでの残り時間未満の場合、クライアントは実際にトークンの有効期限が切れる前に再認証する必要があります。デフォルトでは、アクセストークンの期限が切れてもセッションは期限切れにならず、クライアントは再認証を試行しません。

OAuth 2.0 認証の適用方法や、承認サーバーのタイプによっては、追加 (任意) の設定を使用できます。

```
  # ...
  authentication:
    type: oauth
    # ...
    checkIssuer: false (1)
    checkAudience: true (2)
    fallbackUserNameClaim: client_id (3)
    fallbackUserNamePrefix: client-account- (4)
    validTokenType: bearer (5)
    userInfoEndpointUri: https://OAUTH-SERVER-ADDRESS/auth/realms/external/protocol/openid-connect/userinfo (6)
    enableOauthBearer: false (7)
    enablePlain: true (8)
```

```
    tokenEndpointUri: https://OAUTH-SERVER-ADDRESS/auth/realms/external/protocol/openid-connect/token (9)
    customClaimCheck: "@.custom == 'custom-value'" (10)
    clientAudience: AUDIENCE (11)
    clientScope: SCOPE (12)
    connectTimeoutSeconds: 60 (13)
    readTimeoutSeconds: 60 (14)
    groupsClaim: "$.groups" (15)
    groupsClaimDelimiter: "," (16)
```

(1) 承認サーバーが **iss** クレームを提供しない場合は、発行者チェックを行うことができません。このような場合、**checkIssuer** を **false** に設定し、**validIssuerUri** を指定しないようにします。デフォルトは **true** です。

(2) オーソリゼーションサーバーが **aud**(オーディエンス) クレームを提供していて、オーディエンスチェックを実施したい場合は、**checkAudience** を **true** に設定します。オーディエンスチェックによって、トークンの目的の受信者が特定されます。これにより、Kafka ブローカーは **aud** 要求に **clientId** を持たないトークンを拒否します。デフォルトは **false** です。

(3) 承認サーバーは、通常ユーザーとクライアントの両方を識別する単一の属性を提供しない場合があります。クライアントが独自の名前で認証される場合、サーバーによって **クライアント ID** が提供されることがあります。更新トークンまたはアクセストークンを取得するために、ユーザー名およびパスワードを使用してユーザーが認証される場合、サーバーによってクライアント ID の他に **ユーザー名** が提供されることがあります。プライマリーユーザー ID 属性が使用できない場合は、このフォールバックオプションで、使用するユーザー名クレーム (属性) を指定します。

(4) **fallbackUserNameClaim** が適用される場合、ユーザー名クレームの値とフォールバックユーザー名クレームの値が競合しないようにする必要もあることがあります。**producer** というクライアントが存在し、**producer** という通常ユーザーも存在する場合について考えてみましょう。この 2 つを区別するには、このプロパティーを使用してクライアントのユーザー ID に接頭辞を追加します。

(5) (**introspectionEndpointUri** を使用する場合のみ該当): 使用している認証サーバーによっては、イントロスペクションエンドポイントによって**トークンタイプ**属性が返されるかどうかは分からず、異なる値が含まれることがあります。イントロスペクションエンドポイントからの応答に含まれなければならない有効なトークンタイプ値を指定できます。

(6) (**introspectionEndpointUri** を使用する場合のみ該当): イントロスペクションエンドポイントの応答に識別可能な情報が含まれないように、承認サーバーが設定または実装されることがあります。ユーザー ID を取得するには、**userinfo** エンドポイントの URI をフォールバックとして設定します。**userNameClaim**、**fallbackUserNameClaim**、および **fallbackUserNamePrefix** の設定が **userinfo** エンドポイントの応答に適用されます。

(7) これを **false** に設定してリスナーで OAUTHBEARER メカニズムを無効にします。PLAIN または OAUTHBEARER のいずれかを有効にする必要があります。デフォルトは **true** です。

(8) リスナーで PLAIN 認証を有効にするには、**true** に設定します。これは、すべてのプラットフォームのすべてのクライアントでサポートされています。

(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** を使った PLAIN クライアント認証の上にある OAuth 2.0 のクライアント名にも使われています。これは、承認サーバーに応じて、トークンの取得機能とトークンの内容のみに影響します。リスナーによるトークン検証ルールには影響しません。

12 **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** を使った PLAIN クライアント認証の上にある OAuth 2.0 のクライアント名にも使われています。これは、承認サーバーに応じて、トークンの取得機能とトークンの内容のみに影響します。リスナーによるトークン検証ルールには影響しません。

13 承認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 承認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承認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の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、または Pod 状態の遷移を監視して確認します。

```
oc logs -f ${POD_NAME} -c ${CONTAINER_NAME}
oc get pod -w
```

ローリング更新によって、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。

6.4.6.3. OAuth 2.0 を使用するための Kafka Java クライアントの設定

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 API とコンシューマー API を設定します。コールバックプラグインをクライアントの **pom.xml** ファイルに追加してから、OAuth 2.0 用にクライアントを設定します。

クライアント設定で次を指定します。

- SASL (Simple Authentication and Security Layer) セキュリティープロトコル:
  - TLS 暗号化接続を介した認証用の **SASL_SSL**

- 暗号化されていない接続を介した認証用の **SASL_PLAINTEXT**
  プロダクションには **SASL_SSL** を使用し、ローカル開発には **SASL_PLAINTEXT** のみを使用してください。**SASL_SSL** を使用する場合は、追加の **ssl.truststore** 設定が必要です。OAuth 2.0 承認サーバーへの安全な接続 (**https://**) には、トラストストア設定が必要です。OAuth 2.0 承認サーバーを確認するには、承認サーバーの CA 証明書をクライアント設定のトラストストアに追加します。トラストストアは、PEM または PKCS #12 形式で設定できます。

- Kafka SASL メカニズム:
  - ベアラートークンを使用した認証情報交換用の **OAUTHBEARER**
  - クライアント認証情報 (clientId + secret) またはアクセストークンを渡す **PLAIN**

- SASL メカニズムを実装する JAAS (Java Authentication and Authorization Service) モジュール:
  - **org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule** は OAUTHBEARER メカニズムを実装します。
  - **org.apache.kafka.common.security.plain.PlainLoginModule** は PLAIN メカニズムを実装します。

- 以下の認証方法をサポートする SASL 認証プロパティー:
  - OAuth 2.0 クライアント認証情報
  - OAuth 2.0 パスワード付与 (非推奨)
  - アクセストークン
  - トークンの更新

SASL 認証プロパティーを JAAS 設定 (**sasl.jaas.config**) として追加します。認証プロパティーを設定する方法は、OAuth 2.0 承認サーバーへのアクセスに使用している認証方法によって異なります。この手順では、プロパティーはプロパティーファイルで指定されてから、クライアント設定にロードされます。

注記

認証プロパティーを環境変数または Java システムプロパティーとして指定することもできます。Java システムプロパティーの場合は、**setProperty** を使用して設定し、**-D** オプションを使用してコマンドラインで渡すことができます。

前提条件

- AMQ Streams および Kafka が稼働している。
- OAuth 2.0 承認サーバーがデプロイされ、Kafka ブローカーへの OAuth のアクセスが設定されている。
- 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>0.11.0.redhat-00003</version>
</dependency>
```

2. プロパティーファイルで以下の設定を指定して、クライアントプロパティーを設定します。

   - セキュリティープロトコル
   - SASL メカニズム
   - 使用されているメソッドに応じた JAAS モジュールと認証プロパティー
     たとえば、以下を **client.properties** ファイルに追加できます。

     クライアント認証情報メカニズムのプロパティー

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

(1) TLS 暗号化接続用の **SASL_SSL** セキュリティープロトコル。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。

(2) **OAUTHBEARER** または **PLAIN** として指定された SASL メカニズム。

(3) Kafka クラスターへの安全なアクセスのためのトラストストア設定。

(4) 承認サーバーのトークンエンドポイントの URI です。

(5) クライアント ID。承認サーバーで **client** を作成するときに使用される名前です。

(6) 承認サーバーで **client** を作成するときに作成されるクライアントシークレット。

(7) この場所には、承認サーバーの公開鍵証明書 (**truststore.p12**) が含まれています。

(8) トラストストアにアクセスするためのパスワード。

(9) トラストストアのタイプ。

(10) (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。

11 (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

パスワード付与メカニズムのプロパティー

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

1 クライアント ID。承認サーバーで **client** を作成するときに使用される名前です。

2 (オプション) 承認サーバーで **client** を作成するときに作成されるクライアントシークレット。

3 パスワード付与認証のユーザー名。OAuth パスワード付与設定 (ユーザー名とパスワード) は、OAuth 2.0 パスワード付与メソッドを使用します。パスワード付与を使用するには、権限が制限された認可サーバーにクライアント用のユーザーアカウントを作成します。アカウントは、サービスアカウントのように機能する必要があります。認証にユーザーアカウントが必要な環境で使用しますが、最初に更新トークンの使用を検討してください。

4 パスワード付与認証のパスワード。

注記

SASL PLAIN は、OAuth 2.0 パスワード付与メソッドを使用したユーザー名とパスワードの受け渡し (パスワード付与) をサポートしていません。

アクセストークンのプロパティー

```
security.protocol=SASL_SSL
sasl.mechanism=OAUTHBEARER
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule required \
```

```
oauth.token.endpoint.uri="<token_endpoint_url>" \
oauth.access.token="<access_token>" ; (1)
oauth.ssl.truststore.location="/tmp/oauth-truststore.p12" \
oauth.ssl.truststore.password="$STOREPASS" \
oauth.ssl.truststore.type="PKCS12" \
```

(1) Kafka クライアントの有効期間が長いアクセストークン。

トークンのプロパティーを更新する

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

(1) クライアント ID。承認サーバーで **client** を作成するときに使用される名前です。

(2) (オプション) 承認サーバーで **client** を作成するときに作成されるクライアントシークレット。

(3) Kafka クライアントの有効期間が長い更新トークン。

3. OAUTH 2.0 認証のクライアントプロパティーを Java クライアントコードに入力します。

クライアントプロパティーの入力を示す例

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

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。

6.4.6.4. Kafka コンポーネントの OAuth 2.0 の設定

この手順では、承認サーバーを使用して OAuth 2.0 認証を使用するように Kafka コンポーネントを設定する方法を説明します。

以下の認証を設定できます。

- Kafka Connect

- Kafka MirrorMaker
- Kafka Bridge

この手順では、Kafka コンポーネントと承認サーバーは同じサーバーで稼働しています。

作業を開始する前の注意事項

Kafka コンポーネントの OAuth 2.0 認証の設定に関する詳細は、以下を参照してください。

- KafkaClientAuthenticationOAuth スキーマ参照

前提条件

- AMQ Streams および Kafka が稼働している。
- OAuth 2.0 承認サーバーがデプロイされ、Kafka ブローカーへの OAuth のアクセスが設定されている。
- 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: MGQ1OTRmMzYtZTllZS00MDY2LWI5OGEtMTM5MzM2NjdlZjQw (1)
```

(1) **clientSecret** キーは base64 形式である必要があります。

2. Kafka コンポーネントのリソースを作成または編集し、OAuth 2.0 認証が認証プロパティーに設定されるようにします。
   OAuth 2.0 認証では、以下を使用できます。
   - クライアント ID およびシークレット
   - クライアント ID および更新トークン
   - アクセストークン
   - ユーザー名およびパスワード
   - TLS

   KafkaClientAuthenticationOAuth スキーマ参照は、それぞれの例を提供します。

   以下は、クライアント ID、シークレット、および TLS を使用して OAuth 2.0 が Kafka Bridge クライアントに割り当てられる例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  authentication:
    type: oauth (1)
    tokenEndpointUri: https://<auth-server-address>/auth/realms/master/protocol/openid-
connect/token (2)
    clientId: kafka-bridge
    clientSecret:
      secretName: my-bridge-oauth
      key: clientSecret
    tlsTrustedCertificates: (3)
    - secretName: oauth-server-cert
      certificate: tls.crt
```

1. **oauth** に設定された認証タイプ。
2. 認証用のトークンエンドポイントの URI。
3. 承認サーバーへの TLS 接続用の信用できる証明書。

OAuth 2.0 認証の適用方法や、承認サーバーのタイプによって、使用できる追加の設定オプションがあります。

```
# ...
spec:
  # ...
  authentication:
    # ...
    disableTlsHostnameVerification: true (1)
    checkAccessTokenType: false (2)
    accessTokenIsJwt: false (3)
    scope: any (4)
    audience: kafka (5)
    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. 承認サーバーによって、JWT トークン内部で **typ** (タイプ) 要求が返されない場合は、**checkAccessTokenType: false** を適用するとトークンタイプがチェックされず次に進むことができます。デフォルトは **true** です。
3. 不透明なトークンを使用している場合、アクセストークンが JWT トークンとして処理されないように **accessTokenIsJwt: false** を適用することができます。
4. (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。
6. (オプション) 承認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): 承認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、または Pod 状態の遷移を監視して確認します。

```
oc logs -f ${POD_NAME} -c ${CONTAINER_NAME}
oc get pod -w
```

ローリング更新では、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。

6.5. OAUTH 2.0 トークンベース承認の使用

トークンベースの認証に OAuth 2.0 と Red Hat Single Sign-On を使用している場合、Red Hat Single Sign-On を使用して承認ルールを設定し、Kafka ブローカーへのクライアントのアクセスを制限することもできます。認証はユーザーのアイデンティティーを確立します。承認は、そのユーザーのアクセスレベルを決定します。

AMQ Streams は、Red Hat Single Sign-On の 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、セキュリティーポリシーとパーミッションの一元的な管理が可能になります。

Red Hat Single Sign-On で定義されたセキュリティーポリシーおよびパーミッションは、Kafka ブローカーのリソースへのアクセスを付与するために使用されます。ユーザーとクライアントは、Kafka ブローカーで特定のアクションを実行するためのアクセスを許可するポリシーに対して照合されます。

Kafka では、デフォルトですべてのユーザーがブローカーに完全アクセスできます。また、アクセス制御リスト (ACL) を基にして承認を設定するために **AclAuthorizer** プラグインが提供されます。

ZooKeeper には、**ユーザー名** を基にしてリソースへのアクセスを付与または拒否する ACL ルールが保存されます。ただし、Red Hat Single Sign-On を使用した OAuth 2.0 トークンベースの承認では、より柔軟にアクセス制御を Kafka ブローカーに実装できます。さらに、Kafka ブローカーで OAuth 2.0 の承認および ACL が使用されるように設定することができます。

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

6.5.1. OAuth 2.0 の承認メカニズム

AMQ Streams の OAuth 2.0 での承認では、Red Hat Single Sign-On サーバーの Authorization Services REST エンドポイントを使用して、Red Hat Single Sign-On を使用するトークンベースの認証が拡張されます。これは、定義されたセキュリティーポリシーを特定のユーザーに適用し、そのユーザーの異なるリソースに付与されたパーミッションの一覧を提供します。ポリシーはロールとグループを使用して、パーミッションをユーザーと照合します。OAuth 2.0 の承認では、Red Hat Single Sign-On の Authorization Services から受信した、ユーザーに付与された権限のリストを基にして、権限がローカルで強制されます。

6.5.1.1. Kafka ブローカーのカスタムオーソライザー

AMQ Streams では、Red Hat Single Sign-On の **オーソライザー** (**KeycloakRBACAuthorizer**) が提供されます。Red Hat Single Sign-On によって提供される Authorization Services で Red Hat Single Sign-On REST エンドポイントを使用できるようにするには、Kafka ブローカーでカスタムオーソライザーを設定します。

オーソライザーは必要に応じて付与された権限のリストを承認サーバーから取得し、ローカルで Kafka ブローカーに承認を強制するため、クライアントの要求ごとに迅速な承認決定が行われます。

6.5.2. OAuth 2.0 承認サポートの設定

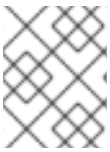
この手順では、Red Hat Single Sign-On の Authorization Services を使用して、OAuth 2.0 承認を使用するように Kafka ブローカーを設定する方法を説明します。

作業を開始する前に

特定のユーザーに必要なアクセス、または制限するアクセスについて検討してください。Red Hat Single Sign-On では、Red Hat Single Sign-On の **グループ**、**ロール**、**クライアント**、および **ユーザー** の組み合わせを使用して、アクセスを設定できます。

通常、グループは組織の部門または地理的な場所を基にしてユーザーを照合するために使用されます。また、ロールは職務を基にしてユーザーを照合するために使用されます。

Red Hat Single Sign-On を使用すると、ユーザーおよびグループを LDAP で保存できますが、クライアントおよびロールは LDAP で保存できません。ユーザーデータへのアクセスとストレージを考慮して、承認ポリシーの設定方法を選択する必要がある場合があります。



注記

スーパーユーザー は、Kafka ブローカーに実装された承認にかかわらず、常に制限なく Kafka ブローカーにアクセスできます。

前提条件

- AMQ Streams は、トークンベースの認証 に Red Hat Single Sign-On と OAuth 2.0 を使用するように設定されている必要がある。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、承認サーバーと Red Hat Single Sign-On の Authorization Services にアクセスできるようにします。
以下に例を示します。

```
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、**jwksEndpointUri** をローカル JWT 検証の URI として指定します。 **tokenEndpointUri** URI のホスト名は同じである必要があります。

(3) 承認サービスが有効になっている Red Hat Single Sign-On の OAuth 2.0 クライアント定義のクライアント ID。通常、**kafka** が ID として使用されます。

(4) (オプション) Red Hat Single Sign-On Authorization Services ポリシーでアクセスが拒否された場合、Kafka **AclAuthorizer** に権限を委譲します。デフォルトは **false** です。

5. (任意設定): TLS ホスト名の検証を無効にします。デフォルトは **false** です。
6. (任意設定): 指定の スーパーユーザー。
7. (任意設定): 承認サーバーへの TLS 接続用の信用できる証明書。
8. (任意設定): 連続する付与 (Grants) 更新実行の間隔。これは、アクティブなセッションが Red Hat Single Sign-On でユーザーのパーミッション変更を検出する最大時間です。デフォルト値は 60 です。
9. (任意設定): アクティブなセッションの付与 (Grants) の更新 (並行して) に使用するスレッドの数。デフォルト値は 5 です。
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、または Pod 状態の遷移を監視して確認します。

```
oc logs -f ${POD_NAME} -c kafka
oc get pod -w
```

ローリング更新によって、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。

6.5.3. Red Hat Single Sign-On の Authorization Services でのポリシーおよびパーミッションの管理

本セクションでは、Red Hat Single Sign-On Authorization Services および Kafka によって使用される承認モデルについて説明し、各モデルの重要な概念を定義します。

Kafka にアクセスするためのパーミッションを付与するには、Red Hat Single Sign-On で **OAuth クライアント仕様**を作成して、Red Hat Single Sign-On Authorization Services オブジェクトを Kafka リソースにマップできます。Kafka パーミッションは、Red Hat Single Sign-On Authorization Services ルールを使用して、ユーザーアカウントまたはサービスアカウントに付与されます。

トピックの作成や一覧表示など、一般的な Kafka 操作に必要なさまざまなユーザーパーミッションの 例を紹介します。

6.5.3.1. 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

6.5.3.2. Red Hat Single Sign-On Authorization Services の Kafka 承認モデルへのマッピング

Kafka 承認モデルは、Kafka へのアクセスを制御する Red Hat Single Sign-On ロールおよびリソースを定義するベースとして使用されます。

ユーザーアカウントまたはサービスアカウントに Kafka パーミッションを付与するには、まず Kafka ブローカーの Red Hat Single Sign-On に **OAuth クライアント仕様**を作成します。次に、クライアントに Red Hat Single Sign-On の Authorization Services ルールを指定します。通常、ブローカーを表す OAuth クライアントのクライアント ID は **kafka** です。AMQ Streams で提供されている 設定ファイルの例では、OAuth のクライアント ID として **kafka** を使用しています。

注記

複数の Kafka クラスターがある場合は、それらすべてに単一の OAuth クライアント (**kafka)** を使用できます。これにより、承認ルールを定義および管理するための単一の統合されたスペースが提供されます。ただし、異なる OAuth クライアント ID(例 **my-cluster-kafka** または **cluster-dev-kafka**) を使用し、各クライアント設定内の各クラスターの承認ルールを定義することもできます。

Kafka **クライアント** 定義では、Red Hat Single Sign-On 管理コンソールで **Authorization Enabled** オプションが有効になっている必要があります。

すべてのパーミッションは、**kafka** クライアントのスコープ内に存在します。異なる OAuth クライアント ID で異なる Kafka クラスターを設定した場合、同じ Red Hat Single Sign-On レルムの一部であっても、それぞれに個別のパーミッションセットが必要です。

Kafka クライアントが OAUTHBEARER 認証を使用する場合、Red Hat Single Sign-On オーソライザー (**KeycloakRBACAuthorizer**) は現在のセッションのアクセストークンを使用して、Red Hat Single Sign-On サーバーからグラントのリストを取得します。許可を取得するために、オーソライザーは Red Hat Single Sign-On の Authorization Services ポリシーおよびパーミッションを評価します。

Kafka パーミッションの承認スコープ

通常、Red Hat Single Sign-On 初期設定では、承認スコープをアップロードして、各 Kafka リソースタイプで実行できるすべての可能なアクションのリストを作成します。この手順は、パーミッションを定義する前に 1 度のみ実行されます。承認スコープをアップロードする代わりに、手動で追加できます。

承認スコープには、リソースタイプに関係なく、可能なすべての Kafka パーミッションが含まれる必要があります。

- **作成**
- **Write**
- **読み取り**
- **Delete**
- **Describe**
- **Alter**
- **DescribeConfig**
- **AlterConfig**
- **ClusterAction**
- **IdempotentWrite**

注記

パーミッションが必要ない場合 (例: **IdempotentWrite**)、承認スコープの一覧から省略できます。ただし、そのパーミッションは Kafka リソースをターゲットにすることはできません。

パーミッションチェックのリソースパターン

リソースパターンは、パーミッションチェックの実行時にターゲットリソースに対するパターンの照合に使用されます。一般的なパターン形式は **RESOURCE-TYPE:PATTERN-NAME** です。

リソースタイプは Kafka 承認モデルをミラーリングします。このパターンでは、次の 2 つの一致オプションが可能です。

- 完全一致 (パターンが * で終了しない場合)
- 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。

6.5.3.3. Kafka 操作に必要なパーミッションの例

以下の例は、Kafka で一般的な操作を実行するために必要なユーザーパーミッションを示しています。

トピックを作成します

トピックを作成するには、特定のトピック、または **Cluster:kafka-cluster** に対して **Create** パーミッションが必要です。

```
bin/kafka-topics.sh --create --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

トピックの一覧表示

指定のトピックでユーザーに **Describe** パーミッションがある場合には、トピックが一覧表示されます。

```
bin/kafka-topics.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

トピックの詳細の表示

トピックの詳細を表示するには、トピックに対して **Describe** および **DescribeConfigs** の権限が必要です。

```
bin/kafka-topics.sh --describe --topic my-topic \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

トピックへのメッセージの生成

トピックへのメッセージを作成するには、トピックに対する **Describe** と **Write** の権限が必要です。

トピックが作成されておらず、トピックの自動生成が有効になっている場合は、トピックを作成するパーミッションが必要になります。

```
bin/kafka-console-producer.sh  --topic my-topic \
  --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

Cluster:kafka-cluster リソースには、トピックをプロデュースするためのアクセス許可だけでなく、**IdempotentWrite** アクセス許可が追加で必要です。

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一覧表示時に、ユーザーが **Describe** 権限を持っているグループのみが返されます。また、ユーザーが **Cluster:kafka-cluster** に対して **Describe** パーミッションを持っている場合は、すべてのコンシューマーグループが返されます。

```
bin/kafka-consumer-groups.sh --list \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

コンシューマーグループの詳細の表示

コンシューマ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 を使ってブローカーの設定を取得するためには、**Cluster:kafka-cluster** に **DescribeConfigs** パーミッションが必要です。

```
bin/kafka-configs.sh --entity-type brokers --entity-name 0 --describe --all \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config=/tmp/config.properties
```

Kafka ブローカー設定の変更

Kafka ブローカーの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
```

パーティションの再割り当てを実行するには、**Cluster:kafka-cluster** に対して **Describe** と **Alter** のパーミッションが必要です。また、関係するトピックには、**Describe の** パーミッションが必要です。

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --execute \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties
```

パーティションの再割り当てを確認するには、**Cluster:kafka-cluster** および関連する各トピックに対して **Describe** および **AlterConfigs** のパーミッションが必要です。

```
bin/kafka-reassign-partitions.sh --reassignment-json-file /tmp/partition-reassignment.json --verify \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 --command-config /tmp/config.properties
```

6.5.4. Red Hat Single Sign-On の Authorization Services の試行

この例では、Red Hat Single Sign-On Authorization Services を **keycloak** 認証で使用する方法を説明します。Red Hat Single Sign-On の Authorization Services を使用して、Kafka クライアントにアクセス制限を強制します。Red Hat Single Sign-On の Authorization Services では、承認スコープ、ポリシー、およびパーミッションを使用してアクセス制御をリソースに定義および適用します。

Red Hat Single Sign-On の Authorization Services REST エンドポイントは、認証されたユーザーのリソースに付与されたパーミッションの一覧を提供します。許可 (パーミッション) のリストは、Kafka クライアントによって認証されたセッションが確立された後に最初のアクションとして Red Hat Single Sign-On サーバーから取得されます。付与の変更が検出されるように、バックグラウンドで一覧が更新されます。付与は、各ユーザーセッションが迅速な承認決定を提供するために、Kafka ブローカーにてローカルでキャッシュおよび適用されます。

AMQ Streams には 設定ファイルの例 が用意されています。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 レルムの例。レルムを Red Hat Single Sign-On インスタンスにインポートし、Kafka にアクセスするための詳細なパーミッションを設定できます。

Red Hat Single Sign-On で例を試す場合は、これらのファイルを使用して、本セクションの順序で説明したタスクを実行します。

1. Red Hat Single Sign-On 管理コンソールへのアクセス
2. Red Hat Single Sign-On 承認をでの Kafka クラスターのデプロイメント
3. CLI Kafka クライアントセッションの TLS 接続の準備
4. CLI Kafka クライアントセッションを使用した Kafka への承認されたアクセスの確認

認証

トークンベースの **oauth** 認証を設定する場合、**jwksEndpointUri** をローカル JWT 検証の URI として指定します。**keycloak** 承認を設定するとき、a **tokenEndpointUri** を Red Hat Single Sign-On トークンエンドポイントの URI として指定します。両方の URI のホスト名は同じである必要があります。

グループまたはロールポリシーを使用した対象パーミッション

Red Hat Single Sign-On では、サービスアカウントが有効になっている機密性の高いクライアントを、クライアント ID とシークレットを使用して、独自の名前のサーバーに対して認証できます。これは、通常、特定ユーザーのエージェント (Web サイトなど) としてではなく、独自の名前で動作するマイクロサービスに便利です。サービスアカウントには、通常のユーザーと同様にロールを割り当てることができます。ただし、グループを割り当てることはできません。そのため、サービスアカウントを使用してマイクロサービスへのパーミッションをターゲットにする場合は、グループポリシーを使用できないため、代わりにロールポリシーを使用する必要があります。逆に、ユーザー名およびパスワードを使用した認証が必要な通常のユーザーアカウントにのみ特定のパーミッションを制限する場合は、ロールポリシーではなく、グループポリシーを使用すると、副次的に実現することができます。これは、**ClusterManager** で始まるパーミッションの例で使用されるものです。通常、クラスター管理の実行は CLI ツールを使用して対話的に行われます。結果的に生成されるアクセストークンを使用して Kafka ブローカーに対して認証を行う前に、ユーザーのログインを要求することは妥当です。この場合、アクセストークンはクライアントアプリケーションではなく、特定のユーザーを表します。

6.5.4.1. Red Hat Single Sign-On 管理コンソールへのアクセス

Red Hat Single Sign-On を設定してから、管理コンソールに接続し、事前設定されたレルムを追加します。**kafka-authz-realm.json** ファイルのサンプルを使用して、レルムをインポートします。管理コンソールのレルムに定義された承認ルールを確認できます。このルールは、Red Hat Single Sign-On レルムの例を使用するよう設定された Kafka クラスターのリソースへのアクセスを許可します。

前提条件

- 実行中の OpenShift クラスター。
- 事前設定されたレルムが含まれる AMQ Streams の **examples/security/keycloak-authorization/kafka-authz-realm.json** ファイル。

手順

1. Red Hat Single Sign-On ドキュメントの Server Installation and Configuration 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 アイデンティティープロバイダーにリンクできます。
Kafka リソースにパーミッションを付与するなど、カスタム LDAP サーバー管理ユーザーインターフェイスを使用して、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 アイデンティティープロバイダーに保存できます。

ロール

Roles は、ユーザーやクライアントが特定の権限を持っていることを示すものです。ロールはグループに似た概念です。通常ロールは、組織ロールでユーザーを **タグ付け** するために

使用され、必要なパーミッションを持ちます。ロールは LDAP アイデンティティープロバイダーに保存できません。LDAP が必須である場合は、代わりにグループを使用し、Red Hat Single Sign-On ロールをグループに追加して、ユーザーにグループを割り当てるときに対応するロールも取得するようにします。

Clients

Clients は特定の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、および承認スコープも含まれます。認証設定は **kafka** クライアントの **Authorization** タブで定義され、**Settings** タブで**Authorization Enabled** をオンにすると表示されます。
- **kafka-cli** クライアントは、アクセストークンまたは更新トークンを取得するためにユーザー名とパスワードを使用して認証するときに Kafka コマンドラインツールによって使用されるパブリッククライアントです。
- **team-a-client** および **team-b-client** クライアントは、特定の Kafka トピックに部分的にアクセスできるサービスを表す機密クライアントです。

11. Red Hat Single Sign-On 管理コンソールで、**Authorization** > **Permissions** の順に移動し、レルムに定義されたリソースおよびポリシーを使用する付与されたパーミッションを確認します。たとえば、**kafka** クライアントには以下のパーミッションがあります。

```
Dev Team A can write to topics that start with x_ on any cluster
Dev Team B can read from topics that start with x_ on any cluster
Dev Team B can update consumer group offsets that start with x_ on any cluster
ClusterManager of my-cluster Group has full access to cluster config on my-cluster
ClusterManager of my-cluster Group has full access to consumer groups on my-cluster
ClusterManager of my-cluster Group has full access to topics on my-cluster
```

Dev Team A

Dev チーム A レルムロールは、任意のクラスターで **x_** で始まるトピックに書き込みできます。これは、**Topic:x_*** というリソース、**Describe** と **Write** のスコープ、そして **Dev Team A** のポリシーを組み合わせたものです。**Dev Team A** ポリシーは、**Dev Team A** というレルムロールを持つすべてのユーザーにマッチします。

Dev Team B

Dev チーム B レルムロールは、任意のクラスターで **x_** で始まるトピックから読み取ることができます。これは、**Topic:x_*、 Group:x_*** のリソース、**Describe** と **Read** のスコープ、および **Dev Team B** のポリシーを組み合わせたものです。**Dev Team B** ポリシーは、**Dev Team B** というレルムロールを持つすべてのユーザーにマッチします。一致するユーザーおよびクライアントはトピックから読み取りでき、名前が **x_** で始まるトピックおよびコンシューマーグループの消費されたオフセットを更新できます。

6.5.4.2. 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 クラスターを展開します。この例では、**keycloak** 承認と **oauth** 認証を使用して単一ノードの Kafka クラスターをデプロイします。

前提条件

- Red Hat Single Sign-On 承認サーバーが OpenShift クラスターにデプロイされ、サンプルレルムでロードされている。
- Cluster Operator が OpenShift クラスターにデプロイされている。
- AMQ Streams の **examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml** カスタムリソース。

手順

1. デプロイした Red Hat Single Sign-On インスタンスのホスト名を使用して、Kafka ブローカーのトラストストア証明書を準備し、Red Hat Single Sign-On サーバーと通信します。

```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.crt
```

   OpenShift Ingress はセキュアな (HTTPS) 接続の確立に使用されるため、証明書が必要です。

2. シークレットとして OpenShift に証明書をデプロイします。

```
oc create secret generic oauth-server-cert --from-file=/tmp/sso.crt -n $NS
```

3. ホスト名を環境変数として設定します。

```
SSO_HOST=SSO-HOSTNAME
```

4. サンプル Kafka クラスターを作成およびデプロイします。

```
cat examples/security/keycloak-authorization/kafka-ephemeral-oauth-single-keycloak-authz.yaml | sed -E 's#\${SSO_HOST}'"#$SSO_HOST#" | oc create -n $NS -f -
```

6.5.4.3. CLI Kafka クライアントセッションの TLS 接続の準備

対話型 CLI セッション用の新規 Pod を作成します。TLS 接続用の Red Hat Single Sign-On 証明書を使用してトラストストアを設定します。トラストストアは、Red Hat Single Sign-On および Kafka ブローカーに接続します。

前提条件

- Red Hat Single Sign-On 承認サーバーが OpenShift クラスターにデプロイされ、サンプルレルムでロードされている。
  Red Hat Single Sign-On 管理コンソールで、クライアントに割り当てられたロールが **Clients** > **Service Account Roles** に表示されることを確認します。
- Red Hat Single Sign-On に接続するように設定された Kafka クラスターが OpenShift クラスターにデプロイされている。

手順

1. AMQ Streams の Kafka イメージを使用してインタラクティブな Pod コンテナーを新たに実行

1. AMQ Streams の Kafka イメージを使用してインタラクティブな Pod コンテナーを新たに実行し、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-33-rhel8:2.3.0
kafka-cli -n $NS -- /bin/sh
```

注記

イメージのダウンロードの待機中に **oc** がタイムアウトする場合、その後の試行によって an **AlreadyExists** エラーが発生することがあります。

2. Pod コンテナーにアタッチします。

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

3. Red Hat Single Sign-On インスタンスのホスト名を使用して、TLS を使用してクライアントコネクションの証明書を準備します。

```
SSO_HOST=SSO-HOSTNAME
SSO_HOST_PORT=$SSO_HOST:443
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $SSO_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/sso.crt
```

4. Kafka ブローカーへの TLS 接続のトラストストアを作成します。

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias sso -storepass $STOREPASS
-import -file /tmp/sso.crt -noprompt
```

5. Kafka ブートストラップアドレスを Kafka ブローカーのホスト名および **tls** リスナーポート (9093) のホスト名として使用し、Kafka ブローカーの証明書を準備します。

```
KAFKA_HOST_PORT=my-cluster-kafka-bootstrap:9093
STOREPASS=storepass

echo "Q" | openssl s_client -showcerts -connect $KAFKA_HOST_PORT 2>/dev/null | awk '
/BEGIN CERTIFICATE/,/END CERTIFICATE/ { print $0 } ' > /tmp/my-cluster-kafka.crt
```

6. Kafka ブローカーの証明書をトラストストアに追加します。

```
keytool -keystore /tmp/truststore.p12 -storetype pkcs12 -alias my-cluster-kafka -storepass
$STOREPASS -import -file /tmp/my-cluster-kafka.crt -noprompt
```

承認されたアクセスを確認するために、セッションを開いたままにします。

6.5.4.4. CLI Kafka クライアントセッションを使用した Kafka への承認されたアクセスの確認

対話型 CLI セッションを使用して、Red Hat Single Sign-On レルムを通じて適用される承認ルールを確認します。Kafka のサンプルプロデューサーおよびコンシューマークライアントを使用してチェックを適用し、異なるレベルのアクセスを持つユーザーおよびサービスアカウントでトピックを作成します。

team-a-client クライアントおよび **team-b-client** クライアントを使用して、承認ルールを確認します。**alice** admin ユーザーを使用して、Kafka で追加の管理タスクを実行します。

この例で使用される AMQ Streams Kafka イメージには、Kafka プロデューサーおよびコンシューマーバイナリーが含まれます。

前提条件

- ZooKeeper および Kafka は OpenShift クラスターで実行され、メッセージを送受信できる。
- 対話型 CLI Kafka クライアントセッション が開始される。
  Apache Kafka のダウンロード。

クライアントおよび管理ユーザーの設定

1. **team-a-client** クライアントの認証プロパティーで Kafka 設定ファイルを準備します。

```
SSO_HOST=SSO-HOSTNAME

cat > /tmp/team-a-client.properties << EOF
security.protocol=SASL_SSL
ssl.truststore.location=/tmp/truststore.p12
ssl.truststore.password=$STOREPASS
ssl.truststore.type=PKCS12
sasl.mechanism=OAUTHBEARER
sasl.jaas.config=org.apache.kafka.common.security.oauthbearer.OAuthBearerLoginModule
required \
  oauth.client.id="team-a-client" \
  oauth.client.secret="team-a-client-secret" \
  oauth.ssl.truststore.location="/tmp/truststore.p12" \
  oauth.ssl.truststore.password="$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 の設定を使って、**a_** や **x_** で始まるトピックへのメッセージを作成できるかどうかを確認します。

1. トピック **my-topic** に書き込みます。

```
bin/kafka-console-producer.sh --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 コンテナーログで **Authorization GRANTED** のデバッグログを確認します。

```
oc logs my-cluster-kafka-0 -f -n $NS
```

承認されたアクセスでのメッセージの消費

team-a-client 設定を使用して、トピック **a_messages** からメッセージを消費します。

1. トピック **a_messages** からメッセージをフェッチします。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties
```

team-a-client の **Dev Team A** ロールは、名前が **a_** で始まるコンシューマーグループのみにアクセスできるため、リクエストはエラーを返します。

2. **team-a-client** プロパティーを更新し、使用が許可されているカスタムコンシューマーグループを指定します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
a_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
a_consumer_group_1
```

コンシューマーは **a_messages** トピックからすべてのメッセージを受信します。

承認されたアクセスでの Kafka の管理

team-a-client はクラスターレベルのアクセスのないアカウントですが、一部の管理操作と使用することができます。

1. トピックを一覧表示します。

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-a-client.properties --list
```

a_messages トピックが返されます。

2. コンシューマーグループを一覧表示します。

```
bin/kafka-consumer-groups.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --
command-config /tmp/team-a-client.properties --list
```

a_consumer_group_1 コンシューマーグループが返されます。

クラスター設定の詳細を取得します。

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-a-client.properties \
  --entity-type brokers --describe --entity-default
```

操作には **team-a-client** にないクラスターレベルのパーミッションが必要なため、リクエストはエラーを返します。

異なるパーミッションを持つクライアントの使用

team-b-client 設定を使用して、**b_** で始まるトピックにメッセージを生成します。

1. トピック **a_messages** に書き込む。

```
bin/kafka-console-producer.sh --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一覧表示します。

```
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/alice.properties --list
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-a-client.properties --list
bin/kafka-topics.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/team-b-client.properties --list
```

管理者ユーザーの **alice** はすべてのトピックを一覧表示できますが、**team-a-client** と **team-b-client** は自分がアクセスできるトピックのみを一覧表示できます。

Dev Team A ロールと **Dev Team B** ロールは、どちらも **x_** で始まるトピックに対する **Describe** 権限を持っていますが、他のチームのトピックに対する **Describe** 権限を持っていないため、他のチームのトピックを見ることができません。

3. **team-a-client** を使用して、**x_messages** トピックにメッセージを生成します。

```
bin/kafka-console-producer.sh --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** を使って、**x_messages** トピックにメッセージを生成します。

```
bin/kafka-console-producer.sh --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** を使って、**x_messages** トピックからメッセージを消費します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-b-client.properties --group
x_consumer_group_b
```

コンシューマーは、**x_messages** トピックからすべてのメッセージを受け取ります。

6. **team-a-client** を使って、**x_messages** トピックからメッセージを消費します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
x_consumer_group_a
```

このリクエストは、**Not authorized to access topics: [x_messages]** エラーを返します。

7. **team-a-client** を使って、**a_** で始まるコンシューマーグループからのメッセージを消費します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/team-a-client.properties --group
a_consumer_group_a
```

このリクエストは、**Not authorized to access topics: [x_messages]** エラーを返します。

Dev Team A には、**x_** で始まるトピックの **Read** 権限がありません。

8. **alice** を使って、**x_messages** トピックへのメッセージを生成します。

```
bin/kafka-console-consumer.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --topic
x_messages \
  --from-beginning --consumer.config /tmp/alice.properties
```

メッセージは Kafka に正常に生成されます。

alice は、すべてのトピックに対して読み取りまたは書き込みを行うことができます。

9. **alice** を使用してクラスター設定を読み取ります。

```
bin/kafka-configs.sh --bootstrap-server my-cluster-kafka-bootstrap:9093 --command-config
/tmp/alice.properties \
  --entity-type brokers --describe --entity-default
```

この例のクラスター設定は空です。

関連情報

- Server Installation and Configuration
- Red Hat Single Sign-On Authorization Services の Kafka 承認モデルへのマッピング

第7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。

7.1. 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。

7.2. 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。

7.2.1. ロールベースアクセス制御 (RBAC) リソース

Cluster Operator は、OpenShift リソースへのアクセスを必要とする AMQ Streams コンポーネントの RBAC リソースを作成し、管理します。

Cluster Operator が機能するには、**Kafka** および **KafkaConnect** などの Kafka リソースや **ConfigMap**、**Pod**、**Deployment**、**StatefulSet**、および **Service** などの管理リソースと対話するために OpenShift クラスター内でパーミッションが必要です。

パーミッションは、OpenShift のロールベースアクセス制御 (RBAC) リソースを使用して指定されます。

- **ServiceAccount**

- **Role** および **ClusterRole**
- **RoleBinding** および **ClusterRoleBinding**

7.2.1.1. AMQ Streams コンポーネントへの権限の委譲

Cluster Operator は **strimzi-cluster-operator** という名前のサービスアカウントで実行されます。このアカウントには、クラスターロールが割り当てられ、AMQ Streams コンポーネントの RBAC リソースを作成するパーミッションが付与されます。ロールバインディングは、クラスターロールをサービスアカウントに関連付けます。

OpenShift は、ある **ServiceAccount** の下で動作するコンポーネントが、付与元の **ServiceAccount** に含まれていない、別の **ServiceAccount** 権限を付与するのを防ぎます。Cluster Operator は管理するリソースが必要とする **RoleBinding** および **ClusterRoleBinding** RBAC リソースを作成するため、これに同じ権限を付与するロールが必要です。

以下の表は、Cluster Operator によって作成される RBAC リソースについて説明しています。

表7.1 ServiceAccount リソース

名前	Used by (使用フィールド)
<cluster_name>-kafka	Kafka ブローカー Pod
<cluster_name>-zookeeper	ZooKeeper Pod
<cluster_name>-cluster-connect	Kafka Connect Pod
<cluster_name>-mirror-maker	MirrorMaker Pod
<cluster_name>-mirrormaker2	MirrorMaker 2.0 Pod
<cluster_name>-bridge	Kafka Bridge Pod
<cluster_name>-entity-operator	Entitiy Operator

表7.2 ClusterRole リソース

名前	Used by (使用フィールド)
strimzi-cluster-operator-namespaced	Cluster Operator
strimzi-cluster-operator-global	Cluster Operator
strimzi-cluster-operator-leader-election	Cluster Operator
strimzi-kafka-broker	Cluster Operator、ラック機能 (使用時)
strimzi-entity-operator	Cluster Operator、Topic Operator、User Operator

名前	Used by (使用フィールド)
strimzi-kafka-client	Cluster Operator、ラック対応の Kafka クライアント

表7.3 **ClusterRoleBinding** リソース

名前	Used by (使用フィールド)
strimzi-cluster-operator	Cluster Operator
strimzi-cluster-operator-kafka-broker-delegation	Cluster Operator、ラック対応の Kafka ブローカー
strimzi-cluster-operator-kafka-client-delegation	Cluster Operator、ラック対応の Kafka クライアント

表7.4 **RoleBinding** リソース

名前	Used by (使用フィールド)
strimzi-cluster-operator	Cluster Operator
strimzi-cluster-operator-kafka-broker-delegation	Cluster Operator、ラック対応の Kafka ブローカー

7.2.1.2. ServiceAccountを使用した Cluster Operator の実行

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** が **serviceAccountName** として指定されています。

7.2.1.3. ClusterRole リソース

Cluster Operator は **ClusterRole** リソースを使用して、リソースに必要なアクセスを提供します。OpenShift クラスターの設定によっては、クラスター管理者がクラスターロールを作成する必要になる場合があります。

注記

クラスター管理者の権限は **ClusterRole** リソース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は実行されません。

ClusterRole リソースは **最小権限の原則** に従い、Cluster Operator が Kafka コンポーネントのクラスターを操作するために必要な権限のみを含みます。最初に割り当てられた一連の権限により、Cluster Operator で **StatefulSet**、**Deployment**、**Pod**、および **ConfigMap** などの OpenShift リソースを管理できます。

Cluster Operator が権限を委任するには、すべてのクラスターロールが必要です。

Cluster Operator は **strimzi-cluster-operator-namespaced** および **strimzi-cluster-operator-global** クラスターロールを使用して、namespace スコープのリソースレベルおよびクラスタースコープのリソースレベルでパーミッションを付与します。

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:
  # Resources in this role are used by the operator based on an operand being deployed in some namespace. When needed, you
  # can deploy the operator as a cluster-wide operator. But grant the rights listed in this role only on the namespaces
  # where the operands will be deployed. That way, you can limit the access the operator has to other namespaces where it
  # does not manage any clusters.
  - 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:
      # The cluster operator needs the extensions api as the operator supports Kubernetes version
1.11+
      # apps/v1 was introduced in Kubernetes 1.14
      - "extensions"
    resources:
```

```
    # The cluster operator needs to access and manage deployments to run deployment based
Strimzi components
    - deployments
    - deployments/scale
    # The cluster operator needs to access replica sets to manage Strimzi components and to
determine error states
    - replicasets
    # The cluster operator needs to access and manage replication controllers to manage replicasets
    - replicationcontrollers
    # The cluster operator needs to access and manage network policies to lock down
communication between Strimzi components
    - networkpolicies
    # The cluster operator needs to access and manage ingresses which allow external access to the
services in a cluster
    - ingresses
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - "apps"
  resources:
    # The cluster operator needs to access and manage deployments to run deployment based
Strimzi components
    - deployments
    - deployments/scale
    - deployments/status
    # The cluster operator needs to access and manage stateful sets to run stateful sets based
Strimzi components
    - statefulsets
    # The cluster operator needs to access replica-sets to manage Strimzi components and to
determine error states
    - replicasets
  verbs:
    - get
    - list
    - watch
    - create
    - delete
    - patch
    - update
- apiGroups:
    - "" # legacy core events api, used by topic operator
    - "events.k8s.io" # new events api, used by cluster operator
  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
```

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-cluster-operator-leader-election クラスターロールは、リーダーの選出に必要な権限を表します。

リーダー選出権限を持つ ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-leader-election
```

```
  labels:
    app: strimzi
rules:
  - apiGroups:
      - coordination.k8s.io
    resources:
      # The cluster operator needs to access and manage leases for leader election
      # The "create" verb cannot be used with "resourceNames"
      - leases
    verbs:
      - create
  - apiGroups:
      - coordination.k8s.io
    resources:
      # The cluster operator needs to access and manage leases for leader election
      - leases
    resourceNames:
      # The default RBAC files give the operator only access to the Lease resource names strimzi-cluster-operator
      # If you want to use another resource name or resource namespace, you have to configure the RBAC resources accordingly
      - strimzi-cluster-operator
    verbs:
      - get
      - list
      - watch
      - delete
      - patch
      - update
```

strimzi-kafka-broker クラスターロールは、ラック対応機能を使用する Kafka Pod の init コンテナーが必要とするアクセス権限を表します。

strimzi- <cluster_name> -kafka-init という名前のロールバインディングは、**<cluster_name> -kafka** サービスアカウントに、**strimzi-kafka-broker** ロールを使用してクラスター内のノードへのアクセスを許可します。ラック機能が使用されておらず、クラスターが **nodeport** 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。

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-entity-operator クラスターロールは、Topic Operator および User Operator が必要とするアクセスを表します。

Topic Operator はステータス情報を含む OpenShift イベントを生成するため、**<cluster_name> -entity-operator** サービスアカウントは **strimzi-entity-operator** ロールにバインドされ、**strimzi-entity-operator** ロールバインディングを介してこのアクセスが許可されます。

Cluster Operator の ClusterRole により、イベントへのアクセスを Topic および User Operator に委任できます。

```
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 クラスターロールは、ラック対応機能を使用する Kafka クライアントが必要とするアクセス権限を表します。

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
```

7.2.1.4. ClusterRoleBinding リソース

Cluster Operator は **ClusterRoleBinding** および **RoleBinding** リソースを使用して **ClusterRole** を **ServiceAccount** に関連付けます。クラスターのロールバインディングは、クラススコープのリソースが含まれるクラスターロールで必要になります。

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
```

権限の委任に使用されるクラスターロールには、クラスターロールバインディングも必要です。

Cluster Operator と Kafka ブローカーのラック対応機能向けの ClusterRoleBinding

```
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
```

Cluster Operator と Kafka クライアントのラック対応機能向けの ClusterRoleBinding

```
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 リソースのみを含むクラスターロールは、ロールバインディングのみを使用してバインドされます。

Cluster Operator の RoleBinding の例

```
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
```

Cluster Operator および Kafka ブローカーのらっく t 相合機能向けの RoleBinding の例

```
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
```

7.2.2. Cluster Operator ロギングの ConfigMap

Cluster Operator のロギングは、**strimzi-cluster-operator** という名前の **ConfigMap** を使用して設定されます。

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
```

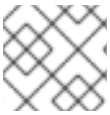
または、**ConfigMap** を直接編集することもできます。

```
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す。

Cluster Operator のデプロイ後に **ConfigMap** が誤って削除される場合、最後に読み込まれたロギング設定が使用されます。新規のロギング設定を読み込むために新規 **ConfigMap** を作成します。



注記

ConfigMap から **monitorInterval** オプションを削除しないでください。

7.2.3. 環境変数を使用した Cluster Operator の設定

環境変数を使用して Cluster Operator を設定できます。サポート対象の環境変数の一覧はをこちらを確認してください。

注記

環境変数は、Cluster Operator イメージのデプロイメンのコンテナー設定に関連します。**image** 設定の詳細については、「**image**」 を参照してください。

STRIMZI_NAMESPACE

Operator が操作する namespace のコンマ区切りリスト。設定されていない場合や、空の文字列や * に設定されている場合には、Cluster Operator はすべての namespace で動作します。
Cluster Operator デプロイメントでは downward API を使用して、これを Cluster Operator がデプロイされる namespace に自動設定することがあります。

Cluster Operator namespace の設定例

```
env:
  - name: STRIMZI_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_FULL_RECONCILIATION_INTERVAL_MS

任意設定、デフォルトは 120000 ミリ秒です。定期的な調整の間隔 (秒単位)。

STRIMZI_OPERATION_TIMEOUT_MS

任意設定、デフォルトは 300000 ミリ秒です。内部操作のタイムアウト (ミリ秒単位)。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、この値を増やします。

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、この値を増やします。**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、最大セッションタイムアウト値はデフォルトの **tickTime** (デフォルトは 2000) 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。Cluster Operator 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_NAME

任意。デフォルトは Pod のホスト名です。Operator 名は、OpenShift イベント を発行するときに AMQ Streams インスタンスを識別します。

STRIMZI_OPERATOR_NAMESPACE

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。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 ラベルを使用して、ネットワークポリシー で namespace セレクターを設定します。ネットワークポリシーを使用すると、AMQ Streams Cluster Operator はこれらのラベルを持つ namespace からのオペランドにのみアクセスできます。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 Cluster Operator にアクセスできるように設定されます。

```
env:
  - name: STRIMZI_OPERATOR_NAMESPACE_LABELS
    value: label1=value1,label2=value2
```

STRIMZI_LABELS_EXCLUSION_PATTERN

任意設定、デフォルトの正規表現パターンは **^app.kubernetes.io/(?!part-of).*** です。メイン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。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、または次の調整時に Pod にラベルを付けます。
ラベルは以下のコンポーネントに適用できます。

- **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:Cluster Operator によって処理されるカスタムリソースをフィルターするラベルセレクター。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。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** プロパティーが指定されていて、**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** リソースで **kafka-init-image** としてイメージが指定されていない場合に、init コンテナーのデフォルトとして使用するイメージ名。init コンテナーは、ラックサポートなど、初期設定用のブローカーが動作する前に開始されます。

STRIMZI_KAFKA_CONNECT_IMAGES

必須。Kafka バージョンから、そのバージョンの Kafka Connect の該当の Docker イメージに対するマッピング。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** プロパティーが指定され、**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 MirrorMakerの該当の Docker イメージに対するマッピング。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** プロパティーが指定されていても **KafkaMirrorMaker.spec.image** プロパティーが指定されていない場合に使用されます。

STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE

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** リソースでイメージが **Kafka.spec.entityOperator.topicOperator.image** として指定されていない場合に、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 のサイドカーコンテナ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。サイドカーは TLS サポートを提供します。

STRIMZI_IMAGE_PULL_POLICY

オプション:Cluster Operator によって管理されるすべての Pod のコンテナーに適用される **ImagePullPolicy**。有効な値は **Always**、**IfNotPresent**、および **Never** です。指定のない場合は、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、コンテナーイメージがプルされるコンテナーレジストリーへのクレデンシャルが含まれます。シークレットは、Cluster Operator によって作成されるすべての Pod の **imagePullSecrets** プロパティーで指定されます。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、OpenShfit クラスターで割り当てられるサービスに、デフォルトの接尾辞 **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。

クラスターで異なる 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。追加のコネクターで新しい Kafka Connect イメージをビルドする場合のタイムアウト (ミリ秒単位)。AMQ Streams を使用して多くのコネクターが含まれるコンテナーイメージをビルドする場合や、低速なコンテナーレジストリーを使用する場合は、この値を増やすことを検討してください。

STRIMZI_NETWORK_POLICY_GENERATION

任意設定、デフォルトは **true** です。リソースのネットワークポリシー。ネットワークポリシーにより、Kafka コンポーネント間の接続が許可されます。
ネットワークポリシーの生成を無効にするには、この環境変数を **false** に設定します。たとえば、カスタムのネットワークポリシーを使用する場合は、これを行うことができます。カスタムネットワークポリシーを使用すると、コンポーネント間の接続をより詳細に制御できます。

STRIMZI_DNS_CACHE_TTL

任意設定で、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場合に便利です。

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 が再作成されるようにするのに役立ちますが、クラスターに他の変更は加えられません。

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。

STRIMZI_POD_SECURITY_PROVIDER_CLASS

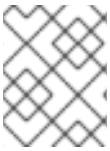
オプション:Pod とコンテナーのセキュリティーコンテキスト設定を提供するために使用できるプラグ可能な **PodSecurityProvider** クラスを設定します。

7.2.3.1. リーダー選択用の環境変数

追加の Cluster Operator レプリカを実行 する場合は、リーダー選出用の環境変数を使用します。大きな障害が原因で中断が発生しないように、追加のレプリカを実行する場合があります。

STRIMZI_LEADER_ELECTION_ENABLED

デフォルトでは無効 (**false**) になります (任意)。リーダーの選出を有効または無効にし、追加の Cluster Operator レプリカをスタンバイで実行できます。



注記

リーダーの選択はデフォルトで無効になっています。インストール時にこの環境変数を適用する場合にのみ有効になります。

STRIMZI_LEADER_ELECTION_LEASE_NAME

リーダー選出が有効な場合に必要です。リーダーの選出に使用される OpenShift **Lease** リソースの名前。

STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE

リーダー選出が有効な場合に必要です。リーダー選出に使用される OpenShift **Lease** リソースが作成される namespace。Downward API を使用して、Cluster Operator がデプロイされている namespace に設定できます。

```
env:
  - name: STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
```

STRIMZI_LEADER_ELECTION_IDENTITY

リーダー選出が有効な場合に必要です。リーダーの選択中に使用される特定の Cluster Operator インスタンスのアイデンティティーを設定します。アイデンティティーは、Operator インスタンスごとに一意である必要があります。Downward API を使用して、Cluster Operator がデプロイされている Pod の名前に設定できます。

```
env:
  - name: STRIMZI_LEADER_ELECTION_IDENTITY
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
```

STRIMZI_LEADER_ELECTION_LEASE_DURATION_MS

オプションで、デフォルトは 15000 ミリ秒です。取得したリースの有効期間を設定します。

STRIMZI_LEADER_ELECTION_RENEW_DEADLINE_MS

任意設定、デフォルトは 10000 ミリ秒です。リーダーがリーダーシップを維持しようと試行する期間を指定します。

STRIMZI_LEADER_ELECTION_RETRY_PERIOD_MS

オプションで、デフォルトは 2000 ミリ秒です。リーダーによるリースロックへの更新頻度を指定します。

7.2.3.2. ネットワークポリシーによる Cluster Operator アクセスの制限

STRIMZI_OPERATOR_NAMESPACE_LABELS 環境変数を使用して Cluster Operator のネットワークポリシーを確立するには、namespace ラベルを使用します。

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、Downward API を使用して、Cluster Operator がどの namespace で実行されているかを検索す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 Sreams によって許可されます。

Cluster Operator が管理するリソースとは別の namespace で実行されている場合、ネットワークポリシーが設定されている場合を除き、OpenShift クラスターのすべての namespace は Cluster Operator へのアクセスが許可されます。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
  #...
```

7.2.3.3. 定期的な調整の時間間隔の設定

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します。

Cluster Operator は、OpenShift クラスターから受信した対象のクラスターリソースに関するすべての

通知に反応します。Operator が実行されていない場合や、何らかの理由で通知を受信しない場合に、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、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して、すべてのリソースで一貫した状態を保つことができます。

関連情報

- Downward API

7.2.4. デフォルトのプロキシー設定を使用した Cluster Operator の設定

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

プロキシー環境変数を指定するように Cluster Operator デプロイメントを設定します。クラスターオペレータは標準的なプロキシー設定 (**HTTP_PROXY**、**HTTPS_PROXY**、**NO_PROXY**) を環境変数として受け入れます。プロキシー設定はすべての AMQ Streams コンテナーに適用されます。

プロキシーアドレスの形式は **http://IP-ADDRESS:PORT-NUMBER** です。名前とパスワードでプロキシーを設定する場合、形式は **http://USERNAME:PASSWORD@IP-ADDRESS:PORT-NUMBER** です。

前提条件

- **CustomResourceDefinition** および RBAC (**ClusterRole** および **RoleBinding**) リソースを作成および管理する権限を持つアカウントが必要です。

手順

1. クラスターオペレータにプロキシー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 のプロキシー設定の例

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      serviceAccountName: strimzi-cluster-operator
      containers:
        # ...
        env:
        # ...
        - name: "HTTP_PROXY"
          value: "http://proxy.com" 1
        - name: "HTTPS_PROXY"
          value: "https://proxy.com" 2
        - name: "NO_PROXY"
          value: "internal.com, other.domain.com" 3
  # ...
```

1. プロキシーサーバーのアドレス。
2. プロキシーサーバーの安全なアドレス。
3. プロキシーサーバーの例外として直接アクセスされるサーバーのアドレス。URL は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

7.2.5. リーダーの選択による複数の Cluster Operator レプリカの実行

デフォルトの Cluster Operator 設定は、リーダー の選択を有効にします。リーダー選択を使用して、Cluster Operator の複数の並列レプリカを実行します。1 つのレプリカがアクティブなリーダーとして選択され、デプロイされたリソースを操作します。他のレプリカはスタンバイモードで実行されます。リーダーが停止またはクラッシュすると、スタンバイレプリカの 1 つが新しいリーダーとして選出され、デプロイされたリソースの操作を開始します。

デフォルトでは、AMQ Streams は、常にリーダーレプリカである単一の Cluster Operator レプリカで実行されます。単一の Cluster Operator レプリカが停止または失敗すると、OpenShift は新しいレプリカを起動します。

複数のレプリカを使用した Cluster Operator の実行は必須ではありません。ただし、大規模な中断が発生した場合に備えて、スタンバイ状態のレプリカを設定しておくと便利です。たとえば、複数のワーカーノードまたはアベイラビリティーゾーン全体に障害が発生したとします。このような障害が発生すると、Cluster Operator Pod と多くの Kafka Pod が同時にダウンする可能性があります。後続の Pod スケジューリングがリソース不足によって輻輳を引き起こす場合、単一の Cluster Operator を実行しているときに操作が遅延する可能性があります。

7.2.5.1. Cluster Operator レプリカの設定

追加の Cluster Operator レプリカをスタンバイモードで実行するには、レプリカ数を増やし、リーダーの選択を有効にする必要があります。リーダーの選択を設定するには、リーダー選択用の環境変数を使用します。

必要な変更を行うには、**install/cluster-operator/** にある以下の Cluster Operator インストールファイルを設定します。

- 060-Deployment-strimzi-cluster-operator.yaml
- 022-ClusterRole-strimzi-cluster-operator-role.yaml
- 022-RoleBinding-strimzi-cluster-operator.yaml

リーダーの選出には、監視している namespace ではなく、Cluster Operator が実行されている namespace を対象とする独自の **ClusterRole** および **RoleBinding** RBAC リソースがあります。

デフォルトのデプロイメント設定は、**strimzi-cluster-operator** という **Lease** リソースを Cluster Operator と同じ namespace に作成します。Cluster Operator はリースを使用してリーダーの選択を管理します。RBAC リソースは、**Lease** リソースを使用するためのパーミッションを提供します。別の **Lease** 名または namespace を使用する場合は、 **ClusterRole** および **RoleBinding** ファイルを適宜更新します。

前提条件

- **CustomResourceDefinition** および RBAC (**ClusterRole** および **RoleBinding**) リソースを作成および管理する権限を持つアカウントが必要です。

手順

Cluster Operator のデプロイに使用される **Deployment** リソースを編集します。これは、**060-Deployment-strimzi-cluster-operator.yaml** ファイルで定義します。

1. **replicas** プロパティーの値は、デフォルトの (1) から、必要なレプリカ数に変更します。

   Cluster Operator レプリカの数の増加

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: strimzi-cluster-operator
     labels:
       app: strimzi
   spec:
     replicas: 3
   ```

2. リーダー選択の **env** プロパティーが設定されていることを確認します。
   設定されていない場合には、設定を行います。

   リーダーの選出を有効にするには、**STRIMZI_LEADER_ELECTION_ENABLED** を **true** (デフォルト) に設定する必要があります。

   この例では、リースの名前は **my-strimzi-cluster-operator** に変更されています。

   Cluster Operator のリーダー選択用の環境変数の設定

   ```
   # ...
   spec
     containers:
       - name: strimzi-cluster-operator
         # ...
         env:
           - name: STRIMZI_LEADER_ELECTION_ENABLED
             value: "true"
           - name: STRIMZI_LEADER_ELECTION_LEASE_NAME
             value: "my-strimzi-cluster-operator"
           - name: STRIMZI_LEADER_ELECTION_LEASE_NAMESPACE
               valueFrom:
                 fieldRef:
                   fieldPath: metadata.namespace
   ```

```
        - name: STRIMZI_LEADER_ELECTION_IDENTITY
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
```

利用可能な環境変数の説明は、「リーダー選択用の環境変数」 を参照してください。

リーダーの選択に使用する **Lease** リソースに別の名前または namespace を指定している場合は、RBAC リソースを更新します。

3. (オプション) **022-ClusterRole-strimzi-cluster-operator-role.yaml** ファイルで **ClusterRole** リソースを編集します。
resourceNames は、**Lease** リソースの名前に更新します。

リースへの ClusterRole 参照の更新

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
rules:
  - apiGroups:
      - coordination.k8s.io
    resourceNames:
      - my-strimzi-cluster-operator
# ...
```

4. (オプション) **022-RoleBinding-strimzi-cluster-operator.yaml** ファイルで **RoleBinding** リソースを編集します。
subjects.name および **subjects.namespace** は **Lease** リソースの名前と、そのリソースが作成された namespace に更新します。

RoleBinding 参照のリースへの更新

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: strimzi-cluster-operator-leader-election
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: my-strimzi-cluster-operator
    namespace: myproject
# ...
```

5. Cluster Operator をデプロイします。

```
oc create -f install/cluster-operator -n myproject
```

6. デプロイメントのステータスを確認します。

```
oc get deployments -n myproject
```

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
strimzi-cluster-operator  3/3    3           3
```

READY は、Ready/expected 状態のレプリカ数を表示します。**AVAILABLE** 出力に正しい数のレプリカが表示されると、デプロイは成功です。

7.2.6. 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 クラスターで AMQ Streams を実行する場合に、AMQ Streams コンテナー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 がクラスターで実行されなくなります。AMQ Streams をクラスター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 対応のクラスターで 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 クラスターで実行できます。

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" 1
# ...
```

1 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
```

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

関連情報

- Federal Information Processing Standards (FIPS) の概要

7.3. 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 クラスターで反映されます。

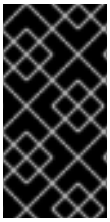
KafkaTopic リソースの詳細については、**KafkaTopic** スキーマリファレンス を参照してください。

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 クラスターで、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 クラスター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設定する必要はありません。

7.3.1. 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。

7.3.1.1. トピック処理用の Kafka クラスターの特定

KafkaTopic リソースには、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。

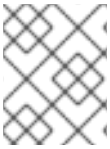
ラベルが Kafka クラスターと一致しない場合、Topic Operator は **KafkaTopic** を識別できず、トピックは作成されません。

7.3.1.2. 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。

7.3.1.3. 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 プロパティーを使用すると、OpenShift の Kafka トピックでは無効な名前を使用して、Kafka で有効なトピックを作成できます。

spec.topicName プロパティーは Kafka の命名検証ルールを継承します。

- 249 文字を超える名前は使用できません。
- Kafka トピックの有効な文字は ASCII 英数字、**.**、**_**、および **-** です。
- 名前を **.** または **..** にすることはできませんが、**.** は **exampleTopic.** や **.exampleTopic** のように名前で使用できます。

spec.topicName は変更しないでください。

以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: topicName-1 1
  # ...
```

1 OpenShift では大文字は無効です。

上記は下記のように変更できません。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: topic-name-1
spec:
  topicName: name-2
  # ...
```

注記

Kafka Streams など一部の Kafka クライアントアプリケーションは、プログラムを使用して Kafka でトピックを作成できます。これらのトピックに、OpenShift リソース名として無効な名前がある場合、Topic Operator はそれらのトピックに Kafka 名に基づく有効な **metadata.name** を提供します。無効な文字が置き換えられ、ハッシュが名前に追加されます。以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: mytopic---c55e57fe2546a33f9e603caf57165db4072e827e
spec:
  topicName: myTopic
  # ...
```

7.3.2. 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 のキーバリューメカニズムを基にしています。

トピックメタデータはインメモリーでキャッシュされ、Topic Operator 内にてローカルでアクセスされます。ローカルのインメモリーキャッシュに適用される操作からの更新は、ディスク上のバックアップトピックストアに永続化されます。トピックストアは、Kafka トピックまたは OpenShift **KafkaTopic** カスタムリソースからの更新と継続的に同期されます。操作は、このような方法で設定されたトピックストアで迅速に処理されますが、インメモリーキャッシュがクラッシュした場合は、永続ストレージから自動的にデータが再入力されます。

7.3.2.1. 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。

7.3.2.2. 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 2.3 にアップグレードする場合、Topic Operator によってトピックストアが制御されるようにシームレスに移行されます。メタデータは ZooKeeper から検出および移行され、古いストアは削除されます。

7.3.2.3. 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もあります。以下に例を示します。

```
oc run kafka-admin -ti --image=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0 --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。

7.3.2.4. 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** ツールを使って変更することができます。

(3) メッセージが正常に書き込まれる必要があるレプリカパーティションの最小数。この条件を満たさない場合は例外が発生します。

注記

インシンクレプリカは、プロデューサーアプリケーションの **acks** 設定と組み合わせて使用します。**acks** 設定は、メッセージが正常に受信されたことを確認するまでに、メッセージを複製しなければならないフォロワーパーティションの数を決定します。トピックオペレータは **acks=all** で動作します。これにより、メッセージは同期しているすべてのレプリカに確認されなければなりません。

ブローカーを追加または削除して Kafka クラスターをスケーリングする場合、レプリケーション係数設定は変更されず、レプリカは自動的に再割り当てされません。しかし、**kafka-reassign-partitions.sh** ツールを使ってレプリケーション係数を変更し、手動でレプリカをブローカーに再割り当てすることができます。

また、AMQ Streams の Cruise Control の統合ではトピックのレプリケーション係数を変更することはできませんが、Kafka をリバランスするために生成された最適化プロポーザルには、パーティションレプリカを転送し、パーティションリーダーを変更するコマンドが含まれます。

7.3.2.5. 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

7.3.3. Kafka トピックの設定

KafkaTopic リソースのプロパティーを使用して、Kafka トピックを設定します。

oc apply を使用すると、トピックを作成または編集できます。**oc delete** を使用すると、既存のトピックを削除できます。

以下に例を示します。

- **oc apply -f <topic_config_file>**
- **oc delete KafkaTopic <topic_name>**

この手順では、10 個のパーティションと 2 つのレプリカがあるトピックを作成する方法を説明します。

作業を開始する前の注意事項

以下を考慮してから変更を行うことが重要になります。

- Kafka は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

- mTLS 認証と TLS 暗号化を使用する Kafka ブローカーリスナーで設定された 実行中の Kafka クラスター。
- 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
```

7.3.4. 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 が稼働中です。

手順

1. 必要に応じてエディターで Kafka クラスター設定を更新します。

```
oc edit kafka MY-CLUSTER
```

2. **Kafka** リソースの **spec.entityOperator.topicOperator.resources** プロパティーで、Topic Operator のリソース要求および制限を設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # Kafka and ZooKeeper sections...
  entityOperator:
    topicOperator:
      resources:
        requests:
          cpu: "1"
          memory: 500Mi
        limits:
          cpu: "1"
          memory: 500Mi
```

3. 新しい設定を適用してリソースを作成または更新します。

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

7.4. 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 クラスターで反映されます。

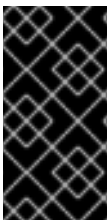
KafkaUser リソースの詳細については、**KafkaUser** スキーマリファレンス を参照してください。

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 クラスターでは、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 クラスター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設定する必要はありません。

7.4.1. Kafka ユーザーの設定

KafkaUser リソースのプロパティーを使用して、Kafka ユーザーを設定します。

oc apply を使用すると、ユーザーを作成または編集できます。**oc delete** を使用すると、既存のユーザーを削除できます。

以下に例を示します。

- **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

- mTLS 認証と TLS 暗号化を使用する Kafka ブローカーリスナーで設定された 実行中の Kafka クラスター。
- 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、mTLS 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
        operations:
          - Describe
          - Read
        host: "*"
      - resource:
          type: group
          name: my-group
          patternType: literal
        operations:
          - Read
        host: "*"
      # Example Producer Acls for topic my-topic
      - resource:
          type: topic
          name: my-topic
```

```
      patternType: literal
    operations:
      - Create
      - Describe
      - Write
    host: "*"
```

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

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

3. 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。

4. **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
```

7.4.2. 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 が稼働中です。

手順

1. 必要に応じてエディターで Kafka クラスター設定を更新します。

```
oc edit kafka MY-CLUSTER
```

2. **Kafka** リソースの **spec.entityOperator.userOperator.resources** プロパティーで、User Operator のリソース要求および制限を設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # Kafka and ZooKeeper sections...
  entityOperator:
    userOperator:
      resources:
        requests:
```

```
      cpu: "1"
      memory: 500Mi
    limits:
      cpu: "1"
      memory: 500Mi
```

ファイルを保存して、エディターを終了します。Cluster Operator によって変更が自動的に適用されます。

7.5. 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
```

7.5.1. ControlPlaneListener フィーチャーゲート

ControlPlaneListener フィーチャーゲートは GA に移動されたので、完全に有効になり、無効にすることはできません。**ControlPlaneListener** が有効にされている場合、Kafka コントローラーとブローカー間の接続はポート 9090 の内部 **コントロールプレーンリスナー** を使用します。ブローカー間のデータのレプリケーション、および AMQ Streams Operator、Cruise Control、または Kafka Exporter からの内部接続では、ポート 9091 で **レプリケーションリスナー** を使用します。

重要

ControlPlaneListener フィーチャーゲートを永続的に有効にすると、AMQ Streams 1.7 以前と AMQ Streams 2.3 以降の間で直接的にアップグレードまたはダウングレードができなくなります。以下の AMQ Streams バージョンのいずれかをアップグレードまたはダウングレードする必要があります。

7.5.2. ServiceAccountPatching フィーチャーゲート

ServiceAccountPatching 機能ゲートは GA に移行しました。つまり、永続的に有効になり、無効にすることはできません。**ServiceAccountPatching** を有効にすると、Cluster Operator は常にサービスアカウントを調整し、必要に応じて更新します。たとえば、カスタムリソースの **template** プロパティーを使用してサービスアカウントのラベルまたはアノテーションを変更すると、Operator はそれらを既存のサービスアカウントリソースで自動的に更新します。

7.5.3. UseStrimziPodSets フィーチャーゲート

UseSrimziPodSets 機能ゲートのデフォルト状態は **enabled** です。

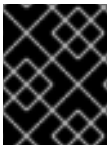
UseStrimziPodSets フィーチャーゲートは、**StrimziPodSet** と呼ばれる Pod の管理用のリソースを導入します。フィーチャーゲートが有効な場合には、StatefulSets の代わりにこのリソースが使用されます。AMQ Streams は、OpenShift ではなく Pod の作成および管理を処理します。StatefulSets の代わりに StrimziPodSets を使用すると、機能の制御が強化されます。

この機能ゲートが無効になっている場合、AMQ Streams は StatefulSets に依存して、ZooKeeper および Kafka クラスターの Pod を作成および管理します。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

UseSrimziPodSets 機能ゲートの無効化

UseSrimziPodSets 機能ゲートを無効にするには、Cluster Operator 設定の **STRIMZI_FEATURE_GATES** 環境変数で **-UseSrimziPodSets** 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。

7.5.4. (プレビュー) UseKRaft 機能ゲート

UseKRaft 機能ゲートのデフォルト状態は **disabled** です。

UseKRaft 機能ゲートは、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。この機能ゲートは現在、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 機能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 プロパティーは無視されますが、存在する必要があります。UseKRaft** 機能ゲートは、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。

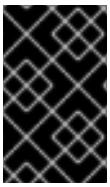
現在、AMQ Streams の KRaft モードには、次の主要な制限があります。

- 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。
- Topic Operator はサポートされていません。**spec.entityOperator.topicOperator** プロパティーを **Kafka** カスタムリソースから **削除する必要があります**。

- 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 プロー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。

UseStrimziPodSets 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。

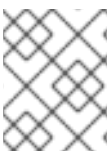
7.5.5. フィーチャーゲートリリース

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 Bata ステージの機能は、有用であることが証明されない場合は削除されます。

- **ControlPlaneListener** 機能ゲートは、AMQ Streams 2.3 で GA に移行しました。現在は永続的に有効になっており、無効にすることはできません。
- **ServiceAccountPatching** 機能ゲートは、AMQ Streams 2.3 で GA に移行しました。現在は永続的に有効になっており、無効にすることはできません。
- **UseSrimziPodSets** 機能ゲートは、AMQ Streams 2.3 でベータ段階に移行しました。
- **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7.5 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	2.3
ServiceAccountPatching	1.8	2.0	2.3
UseStrimziPodSets	2.1	2.3	-
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7.6 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

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

AMQ Streams の operator は Prometheus メトリクスを公開します。メトリクスは自動で有効になり、以下の情報が含まれます。

- 調整の数
- operator が処理しているカスタムリソースの数
- 調整の期間
- operator からの JVM メトリクス

この他に、Grafana ダッシュボードのサンプルが提供されます。

Prometheus に関する詳細は、**OpenShift での AMQ Streams のデプロイおよびアップグレード**のKafka へのメトリクスの導入を参照してください。

第8章 CRUISE CONTROL によるクラスターのリバランス

Cruise Control は、次の Kafka 操作をサポートするオープンソースシステムです。

- クラスターワークロードのモニタリング
- 定義済みの制約に基づくクラスターの再調整

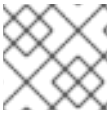
この操作は、ブローカー Pod をより効率的に使用する、よりバランスの取れた Kafka クラスターを実行するのに役立ちます。

通常、クラスターの負荷は時間とともに不均等になります。大量のメッセージトラフィックを処理するパーティションは、使用可能なブローカー全体で均等に分散されない可能性があります。クラスターを再分散するには、管理者はブローカーの負荷を監視し、トラフィックの多いパーティションを容量に余裕のあるブローカーに手作業で再割り当てします。

Cruise Control はクラスターのリバランス処理を自動化します。CPU、ディスク、およびネットワーク負荷を基にして、クラスターにおけるリソース使用の**ワークロードモデル**を構築し、パーティションの割り当てをより均等にする、最適化プロポーザル (承認または拒否可能) を生成します。これらのプロポーザルの算出には、設定可能な最適化ゴールが複数使用されます。

特定のモードで最適化の提案を生成できます。デフォルトの **full** モードでは、すべてのブローカー間でパーティションが再調整されます。**add-brokers** および **remove-brokers** モードを使用して、クラスターをスケールアップまたはスケールダウンするときの変更に対応することもできます。

最適化プロポーザルを承認すると、Cruise Control はそのプロポーザルを Kafka クラスターに適用します。**KafkaRebalance** リソースを使用して、最適化の提案を設定および生成します。最適化の提案が自動または手動で承認されるように、アノテーションを使用してリソースを設定できます。



注記

AMQ Streams には、Cruise Control の設定ファイルのサンプル が含まれています。

8.1. CRUISE CONTROL のコンポーネントと機能

Cruise Control は、Load Monitor、Analyzer、Anomaly Detector、Executor の 4 つの主要コンポーネントと、クライアントとの対話用の REST API で設定されています。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

最適化の目標は、リバランスから達成する特定の目標を表します。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目標になる場合があります。設定を通じて、含める目標を変更できます。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す。
- **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。調整されたゴールのセットを使用して、プロポーザルを拒否し、再度実行するようプロポーザルを拒否することができます。
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。

関連情報

- Cruise Control のドキュメント

8.2. 最適化ゴールの概要

最適化ゴールは、Kafka クラスター全体のワークロード再分散およびリソース使用の制約です。Cruise Control は Kafka クラスターをリバランスするために、最適化ゴールを使用して、承認または拒否可能な 最適化プロポーザル を生成します。

8.2.1. 優先度のゴールの順序

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. トピックのセットに対するブローカーごとのリーダーレプリカの最小数
3. レプリカの容量
4. キャパシティの目標
   - 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. リソース分散ゴール
   - 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

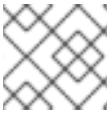
独自のゴールの記述および Kafka アサイナーゴールはまだサポートされていません。

8.2.2. 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、満たす必要のある厳しい最適化目標のほか、メイン、デフォルト、およびユーザー指定の最適化目標の設定があります。

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**
- **ユーザー提供の目標 – KafkaRebalance.spec.goals**



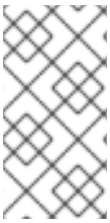
注記

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。

8.2.3. ハードおよびソフト最適化目標

ハードゴールは最適化プロポーザルで **必ず** 満たさなければならないゴールです。ハードゴールとして設定されていないゴールは **ソフトゴール** と呼ばれます。ソフトゴールは **ベストエフォート** 型のゴールと解釈できます。最適化プロポーザルで満たす必要は**ありません**が、最適化の計算に含まれます。すべてのハードゴールを満たし、1 つ以上のソフトゴールに違反する最適化プロポーザルは有効です。

Cruise Control は、すべてのハードゴールを満たし、優先度順にできるだけ多くのソフトゴールを満たす最適化プロポーザルを算出します。すべてのハードゴールを**満たさない**最適化プロポーザルは Cruise Control によって拒否され、ユーザーには送信されません。



注記

たとえば、クラスター全体でトピックのレプリカを均等に分散するソフトゴールがあるとします (トピックレプリカ分散のゴール)。このソフトゴールを無視すると、設定されたハードゴールがすべて有効になる場合、Cruise Control はこのソフトゴールを無視します。

Cruise Control では、以下の メイン最適化ゴール がハードゴールとして事前設定されています。

```
RackAwareGoal; MinTopicLeadersPerBrokerGoal; ReplicaCapacityGoal; DiskCapacityGoal;
NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal
```

Kafka.spec.cruiseControl.config の **hard.goals** プロパティーを編集し、Cruise Control のデプロイメント設定でハードゴールを設定します。

- Cruise Control から事前設定されたハードゴールを継承する場合は、**Kafka.spec.cruiseControl.config** に **hard.goals** プロパティーを指定しないでください。
- 事前設定されたハードゴールを変更するには、完全修飾ドメイン名を使用して、希望のゴールを **hard.goals** プロパティーに指定します。

ハード最適化ゴールの Kafka 設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      hard.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkInboundCapacityGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.NetworkOutboundCapacityGoal
      # ...
```

ハードゴールの数を増やすと、Cruise Control が有効な最適化プロポーザルを生成する可能性が低くなります。

skipHardGoalCheck: true が **KafkaRebalance** カスタムリソースに指定された場合、Cruise Control はユーザー提供の最適化ゴールのリスト (**KafkaRebalance.spec.goals** 内) に設定済みのハードゴール (**hard.goals**) が**すべて**含まれていることをチェック**しません**。そのため、すべてではなく一部のユーザー提供の最適化ゴールが **hard.goals** リストにある場合、**skipHardGoalCheck: true** が指定されていてもハードゴールとして処理されます。

8.2.4. 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。

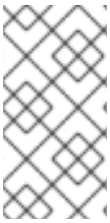
> RackAwareGoal; ReplicaCapacityGoal; DiskCapacityGoal; NetworkInboundCapacityGoal; NetworkOutboundCapacityGoal; CpuCapacityGoal; ReplicaDistributionGoal; PotentialNwOutGoal; DiskUsageDistributionGoal; NetworkInboundUsageDistributionGoal; NetworkOutboundUsageDistributionGoal; CpuUsageDistributionGoal; 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** プロパティーを **Kafka.spec.cruiseControl.config** に指定しないでください。
- 継承した主な最適化目標を変更する必要がある場合は、**goals** 設定オプションで、優先順位の高い順に目標のリストを指定します。



注記

継承された主な最適化ゴールを変更する場合、**Kafka.spec.cruiseControl.config** の **hard.goals** プロパティーに設定されたハードゴールがあれば、設定済みの主な最適化ゴールのサブセットになるようにする必要があります。そうでないと、最適化プロポーザルの生成時にエラーが発生します。

8.2.5. デフォルトの最適化ゴール

Cruise Conrol は**デフォルトの最適化ゴール** を使用して **キャッシュされた最適化プロポーザル** を生成します。キャッシュされた最適化プロポーザルの詳細は、「最適化プロポーザルの概要」 を参照してください。

ユーザー提供の最適化ゴール を **KafkaRebalance** カスタムリソースに設定すると、デフォルトの最適化ゴールを上書きできます。

Cruise Control のデプロイメント設定で **default.goals** を指定しない限り、メインの最適化目標がデフォルトの最適化目標として使用されます。この場合、メイン最適化ゴールを使用して、キャッシュされた最適化プロポーザルが生成されます。

- 主な最適化目標をデフォルトの目標として使用するには、**Kafka.spec.cruiseControl.config** に **default.goals** プロパティーを指定しないでください。
- デフォルトの最適化ゴールを編集するには、**Kafka.spec.cruiseControl.config** の **default.goals** プロパティーを編集します。メイン最適化ゴールのサブセットを使用する必要があります。

デフォルト最適化ゴールの Kafka 設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    topicOperator: {}
    userOperator: {}
  cruiseControl:
    brokerCapacity:
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
    config:
      default.goals: >
        com.linkedin.kafka.cruisecontrol.analyzer.goals.RackAwareGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.ReplicaCapacityGoal,
        com.linkedin.kafka.cruisecontrol.analyzer.goals.DiskCapacityGoal
      # ...
```

デフォルトの最適化ゴールの指定がない場合、メイン最適化ゴールを使用して、キャッシュされたプロポーザルが生成されます。

8.2.6. 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** プロパティーを **KafkaRebalance** カスタムリソースに追加します。「最適化プロポーザルの生成」 を参照してください。

関連情報

- Configuring and deploying Cruise Control with Kafka
- Cruise Control Wiki の Configurations

8.3. 最適化プロポーザルの概要

KafkaRebalance リソースを設定して、最適化の提案を生成し、提案された変更を適用します。**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に、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**

8.3.1. リバランスモード

KafkaRebalance カスタムリソースの **spec.mode** プロパティーを使用して、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** プロパティー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場合には、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** プロパティーを使用して、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、パーティションが複製されない可能性があり、一部のパーティションが最小 ISR(同期レプリカ) 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** プロパティーで、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** プロパティーを介してモードを指定することと、必要に応じて、**spec.brokers** プロパティー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。

8.3.2. 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** プロパティーに含まれています。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。

8.3.3. 最適化プロポーザルの手動承認または拒否

最適化プロポーザルのサマリーは、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、最適化目標の変更 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す。**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場合、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役立つ場合があります。

リバランス時には 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役立ちます。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場合、スコアは 100 です。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。

8.3.4. 最適化プロポーザルの自動承認

時間を節約するために、最適化プロポーザルの承認プロセスを自動化できます。自動化により、最適化の提案を生成すると、クラスターのリバランスに直接進みます。

最適化プロポーザルの自動承認メカニズムを有効にするには、**strimzi.io/rebalance-auto-approval** アノテーションを **true** に設定して **KafkaRebalance** リソースを作成します。アノテーションが設定されていないか、または **false** に設定されている場合、最適化プロポーザルには手動承認が必要です。

自動承認メカニズムが有効になっているリバランス要求の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: true
spec:
  mode: # any mode
  # ...
```

最適化の提案を自動的に承認する場合でも、ステータスを確認できます。リバランスが完了すると、**KafkaRebalance** リソースのステータスは **Ready** に移動します。

8.3.5. 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8.1 最適化プロポーザルに含まれるプロパティーの概要

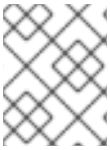
JSON プロパティー	説明
numIntraBrokerReplicaMovements	ディスクとクラスターのブローカーとの間で転送されるパーティションレプリカの合計数。 **リバランス操作中のパフォーマンスへの影響度**: 比較的高いが、**numReplicaMovements** よりも低い。

JSON プロパティー	説明
excludedBrokersForLeadership	サポートされていません。空のリストが返されます。
numReplicaMovements	個別のブローカー間で移動されるパーティションレプリカの数。 **リバランス操作中のパフォーマンスへの影響度**: 比較的高い。
onDemandBalancednessScoreBefore, onDemandBalancednessScoreAfter	最適化プロポーザルの生成前および生成後における、Kafka クラスターの全体的な **分散度** (balancedness) の値。 スコアは、違反した各ソフトゴールの **BalancednessScore** の合計を 100 から引いて算出されます。Cruise Control は、複数の要因を基にして **BalancednessScore** を各最適化ゴールに割り当てます。要因には、**default.goals** またはユーザー提供ゴールのリストでゴールの位置を示す優先順位が含まれます。 **Before** スコアは、Kafka クラスターの現在の設定を基にします。**After** スコアは、生成された最適化プロポーザルを基にします。
intraBrokerDataToMoveMB	同じブローカーのディスク間で移動される各パーティションレプリカのサイズの合計 (**numIntraBrokerReplicaMovements** も参照してください。 **リバランス操作中のパフォーマンスへの影響度**: 場合による。値が大きいほど、クラスターのリバランスの完了にかかる時間が長くなります。大量のデータを移動する場合、同じブローカーのディスク間で移動する方が個別のブローカー間で移動するよりも影響度が低くなります (**dataToMoveMB** 参照)。
recentWindows	最適化プロポーザルの基になるメトリクスウインドウの数。
dataToMoveMB	個別のブローカーに移動される各パーティションレプリカのサイズの合計 (**numReplicaMovements** も参照してください。 **リバランス操作中のパフォーマンスへの影響度**: 場合による。値が大きいほど、クラスターのリバランスの完了にかかる時間が長くなります。
monitoredPartitionsPercentage	最適化プロポーザルの対象となる Kafka クラスターのパーティションの割合 (パーセント)。**excludedTopics** の数が影響します。
excludedTopics	**KafkaRebalance** リソースの **spec.excludedTopicsRegex** プロパティーに正規表現を指定した場合、その式と一致するすべてのトピック名がここにリストされます。これらのトピックは、最適化プロポーザルではパーティションレプリカとリーダーの移動の計算からは除外されます。
numLeaderMovements	リーダーが別のレプリカに切り替えられるパーティションの数。ZooKeeper 設定の変更を伴います。 **リバランス操作中のパフォーマンスへの影響度**: 比較的低い。

JSON プロパティー	説明
excludedBrokersForReplicaMove	サポートされていません。空のリストが返されます。

8.3.6. ブローカーのロードプロパティー

ブローカーの負荷は、JSON 形式の文字列として ConfigMap (KafkaRebalance カスタムリソースと同じ名前) に保存されます。この JSON 文字列は、各ブローカーのいくつかのメトリクスにリンクする各ブローカー ID のキーを持つ JSON オブジェクトで設定されます。各メトリクスは 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クスの値です。2 つ目はプロポーザルの適用後に期待される値、3 つ目は、最初の 2 つの値の差 (後の値から前の値を引いた) です。



注記

ConfigMap は、KafkaRebalance リソースが **ProposalReady** 状態にあると表示され、リバランスが完了すると残ります。

ConfigMap 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ーについて説明しています。

JSON プロパティー	説明
leaders	パーティションリーダーであるこのブローカーのレプリカ数。
replicas	このブローカーのレプリカ数。
cpuPercentage	定義された容量の割合をパーセントで表す CPU 使用率。
diskUsedPercentage	定義された容量の割合をパーセントで表す ディスク 使用率。

JSON プロパティー	説明
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。

8.3.7. 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」

8.4. リバランスパフォーマンスチューニングの概要

クラスターリバランスのパフォーマンスチューニングオプションを調整できます。これらのオプションは、リバランスのパーティションレプリカおよびリーダーシップの移動が実行される方法を制御し、また、リバランス操作に割り当てられた帯域幅も制御します。

8.4.1. 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。

8.4.2. 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がある場合、このストラテジーによって他の再割り当ての適用が遅くなる可能性があります。

Cruise Control は、最適化プロポーザルに適用できる代替のレプリカ移動ストラテジーを 4 つ提供します。

- **PrioritizeSmallReplicaMovementStrategy**: サイズの昇順で再割り当てを並べ替えます。

- **PrioritizeLargeReplicaMovementStrategy**: サイズの降順で再割り当ての順序。

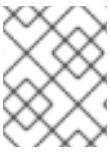
- **PostponeUrpReplicaMovementStrategy**: 非同期レプリカがないパーティションのレプリカの再割り当てを優先します。

- **PrioritizeMinIsrWithOfflineReplicasStrategy**: オフラインレプリカを持つ (At/Under) MinISR パーティションで再割り当てを優先します。この戦略は、**Kafka** カスタムリソースの仕様で **cruiseControl.config.concurrency.adjuster.min.isr.check.enabled** が **true** に設定されている場合にのみ機能します。

これらのストラテジーをシーケンスとして設定できます。最初のストラテジーは、内部ロジックを使用して 2 つのパーティション再割り当ての比較を試みます。再割り当てが同等である場合は、順番を決定するために再割り当てをシーケンスの次のストラテジーに渡します。

8.4.3. 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。

8.4.4. 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。

- クルーズコントロールのサーバー設定は、Kafka のカスタムリソースである **Kafka.spec.cruseControl.config** で設定できます。
- 個々のリバランスのパフォーマンス設定は、**KafkaRebalance.spec** で設定できます。

関連する設定を以下の表にまとめています。

表8.2 リバランスパフォーマンスチューニングの設定

Cruise Control プロパティー	KafkaRebalance プロパティー	デフォルト	説明
num.concurrent.partition.movements.per.broker	**concurrentPartitionMovementsPerBroker**	5	各パーティション再割り当てバッチでの inter-broker パーティション移動の最大数。
num.concurrent.intra.broker.partition.movements	**concurrentIntraBrokerPartitionMovements**	2	各パーティション再割り当てバッチでのブローカー内パーティション移動の最大数。
num.concurrent.leader.movements	**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。
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)。

Cruise Control プロパティー	KafkaRebalance プロパティー	デフォルト	説明
default.replica.movement.strategies	**replicaMovementStrategies**	**BaseReplicaMovementStrategy**	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。
-	**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、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」

8.5. CONFIGURING AND DEPLOYING CRUISE CONTROL WITH KAFKA

Kafka リソースを設定して、Kafka クラスターと共に Cruise Control をデプロイします。**Kafka** リソースの **CruiseControl** プロパティーを使用して、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目標を指定するには、Cruise Control **config** で **goals** 設定を使用します。**brokerCapacity** を使用して、リソース分散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ー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** プロパティーを編集します。
   設定可能なプロパティーは以下の例のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    brokerCapacity: 1
      inboundNetwork: 10000KB/s
      outboundNetwork: 10000KB/s
      overrides: 2
      - brokers: [0]
        inboundNetwork: 20000KiB/s
        outboundNetwork: 20000KiB/s
      - brokers: [1, 2]
        inboundNetwork: 30000KiB/s
        outboundNetwork: 30000KiB/s
      # ...
    config: 3
      default.goals: > 4
        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 (5)
    webserver.http.cors.origin: "*"
    webserver.http.cors.exposeheaders: "User-Task-ID,Content-Type"
    # ...
  resources: (6)
    requests:
      cpu: 1
      memory: 512Mi
    limits:
      cpu: 2
      memory: 2Gi
  logging: (7)
      type: inline
      loggers:
        rootLogger.level: "INFO"
  template: (8)
    pod:
      metadata:
        labels:
          label1: value1
      securityContext:
        runAsUser: 1000001
        fsGroup: 0
      terminationGracePeriodSeconds: 120
  readinessProbe: (9)
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe:
    initialDelaySeconds: 15
    timeoutSeconds: 5
  metricsConfig: (10)
    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) 異種ネットワークリソースを持つノードで実行する場合、特定のブローカーの 設定されたネットワーク容量制限をオーバーライド。

(3) Cruise Control の設定AMQ Streams によって直接管理されないプロパティーに限り、標準 Cruise Control 設定の提供が可能 です。

(4) 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。

5 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。

6 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。

7 ConfigMap を介して直接 (**inline**) または間接 (**external**) に追加された Cruise Control 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。Cruise Control には、**rootLogger.level** という名前の単一のロガーがあります。ログレベルは INFO、ERROR、WARN、TRACE、DEBUG、FATAL、または OFF に設定できます。

8 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ー属性でスケジュールされています。

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

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

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。これらのトピックは、Cruise Control が適切に動作するために必要であるため、削除または変更しないでください。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8.3 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
metric.reporter.topic	**strimzi.cruisecontrol.metrics**	AMQ Streams の Metrics Reporter	Metrics Reporter からの raw メトリクスを各 Kafka ブローカーに格納します。

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
partition.metric.sample.store.topic	**strimzi.cruisecontrol.partitionmetricsamples**	Cruise Control	各パーティションの派生されたメトリクスを格納します。これらは Metric Sample Aggregator によって作成されます。
broker.metric.sample.store.topic	**strimzi.cruisecontrol.modeltrainingsamples**	Cruise Control	クラスターワークロードモデル の作成に使用されるメトリクスサンプルを格納します。

Cruise Control に必要なレコードを削除しないようにするため、自動作成されたトピックではログの圧縮は無効になっています。

注記

自動作成されたトピックの名前が、既に 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

8.6. 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 クラスターで既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** プロパティー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 プロパティーで **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。このプロパティーは必須です。

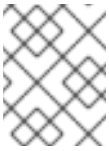
remove-brokers モード

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。このプロパティーは必須です。



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。

2. デフォルトのゴールを使用する代わりに **ユーザー定義の最適化ゴール** を設定するには、**goals** プロパティーを追加し、1 つ以上のゴールを入力します。
以下の例では、ラックアウェアネス (Rack Awareness) およびレプリカの容量はユーザー定義の最適化ゴールとして設定されています。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
```

3. 設定されたハードゴールを無視するには、**skipHardGoalCheck: true** プロパティーを追加します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

4. (オプション) 最適化プロポーザルを自動的に承認するには、**strimzi.io/rebalance-auto-approval** アノテーションを **true** に設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaRebalance
metadata:
  name: my-rebalance
  labels:
    strimzi.io/cluster: my-cluster
  annotations:
    strimzi.io/rebalance-auto-approval: true
spec:
  goals:
    - RackAwareGoal
    - ReplicaCapacityGoal
  skipHardGoalCheck: true
```

5. 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。

6. 自動承認メカニズムを使用した場合は、最適化プロポーザルのステータスが **Ready** に変わるまで待ちます。自動承認メカニズムを有効にしていない場合は、最適化プロポーザルのステータスが **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。

7. 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** プロパティーに含まれます。

```
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ーには、保留クラスターリバランス操作の詳細が表示されます。各プロパティーの説明は、最適化プロポーザルの内容 を参照してください。

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」

8.7. 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 クラスターに適用して、パーティションをブローカーに再割り当てし、パーティションのリーダーを変更します。

注意

これはドライランではありません。最適化プロポーザルを承認する前に、以下を行う必要があります。

- 最新でない可能性があるため、プロポーザルを更新します。
- プロポーザルの内容 を注意して確認します。

前提条件

- Cruise Control から 最適化プロポーザルを生成済み である。
- **KafkaRebalance** カスタムリソースの状態が **ProposalReady** である必要があります。

手順

承認する最適化プロポーザルに対して、以下の手順を実行します。

1. 最適化プロポーザルが新規生成された場合を除き、プロポーザルが Kafka クラスターの状態に関する現在の情報を基にしていることを確認します。これには、最適化プロポーザルを更新し、必ず最新のクラスターメトリクスを使用するようにします。
   a. OpenShift の **KafkaRebalance** リソースに **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
```

4. 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。

5. 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」

8.8. 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の再割り当てはすで適用されています。そのため、Kafka クラスターの状態は、リバランス操作の開始前とは異なります。さらなるリバランスが必要な場合は、新しい最適化プロポーザルを生成してください。

注記

中間 (停止) 状態の Kafka クラスターのパフォーマンスは、初期状態の場合よりも悪くなる可能性があります。

前提条件

- **KafkaRebalance** カスタムリソースに **approve** アノテーションを付けて 最適化プロポーザルが承認済み である必要があります。
- **KafkaRebalance** カスタムリソースの状態が **Rebalancing** である必要があります。

手順

1. OpenShift の **KafkaRebalance** リソースにアノテーションを付けます。

```
oc annotate kafkarebalance rebalance-cr-name strimzi.io/rebalance=stop
```

2. **KafkaRebalance** リソースの状態をチェックします。

```
oc describe kafkarebalance rebalance-cr-name
```

3. 状態が **Stopped** に変わるまで待ちます。

関連情報

- 「最適化プロポーザルの概要」

8.9. 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** プロパティー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9章 APICURIO REGISTRY の RED HAT ビルドを使用したスキーマの検証

AMQ Streams で Apicurio Registry の Red Hat ビルドを使用できます。

Apicurio 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。Apicurio 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。

Apicurio Registry では、メッセージをシリアライズおよびデシリアライズするために使用されるスキーマが保存されます。その後、クライアントアプリケーションからスキーマを参照して、送受信されるメッセージとこれらのスキーマの互換性を維持するようにします。Apicurio Registry によって、Kafka プロデューサーおよびコンシューマーアプリケーションの Kafka クライアントシリアライザーおよびデシリアライザーが提供されます。Kafka プロデューサーアプリケーションは、シリアライザーを使用して、特定のイベントスキーマに準拠するメッセージをエンコードします。Kafka コンシューマーアプリケーションはデシリアライザーを使用して、特定のスキーマ ID に基づいてメッセージが適切なスキーマを使用してシリアライズされたことを検証します。

アプリケーションがレジストリーからスキーマを使用できるようにすることができます。これにより、スキーマが一貫して使用されるようにし、実行時にデータエラーが発生しないようにします。

関連情報

- Apicurio Registry の Red Hat ビルドに関するドキュメント
- Apicurio Registry の Red Hat ビルドは、GitHub の Apicurio/apicurio-registry で利用可能な Apicurio Registry オープンソースコミュニティープロジェクトをもとに構築されています。

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用の 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 mTLS 認証を有効にする場合、他の TLS 証明書も設定されます。

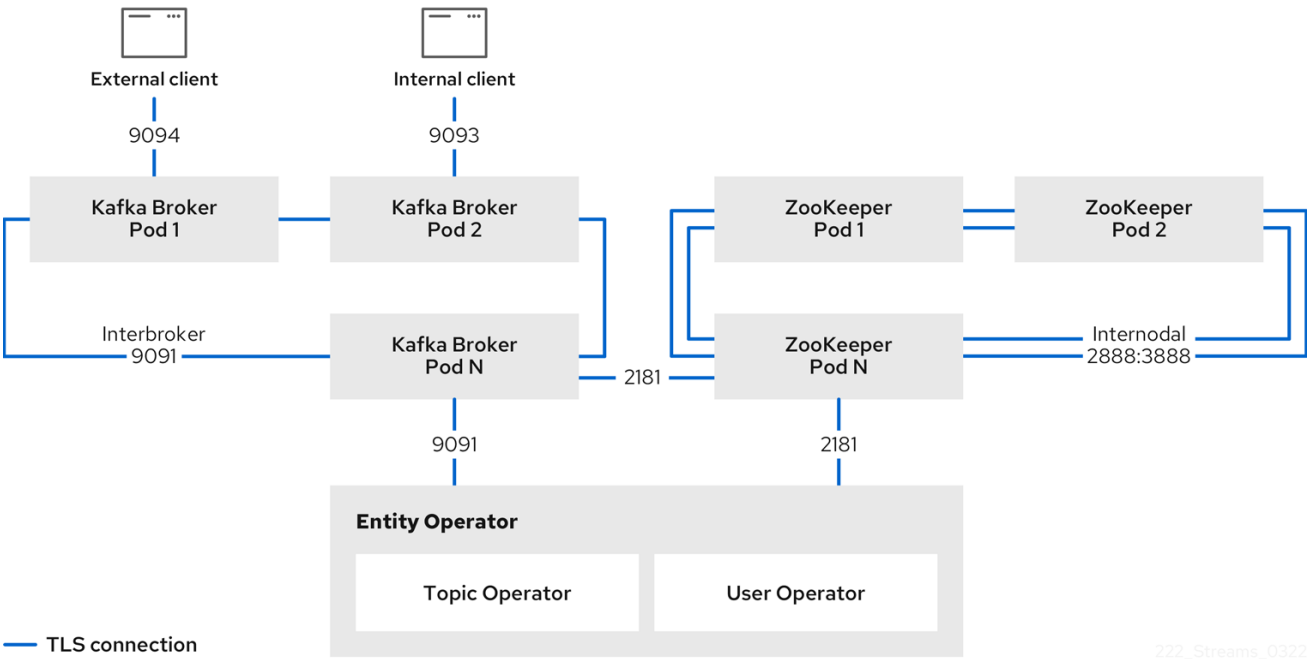
認証局 (CA) 証明書は、コンポーネントとクライアントの ID 検証にクラスター Operator によって生成されます。クラスター Operator によって生成された CA を使用しない場合は 独自のクラスターおよびクライアント CA 証明書をインストールできます。

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クテクチャー例



10.1. 内部クラスター CA とクライアント CA

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、mTLS を使用して AMQ ストリームに接続する各 Kafka クライアントアプリケーションは、秘密鍵と証明書を使用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。

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、mTLS 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。

クラスター CA またはクライアント CA の CA 証明書を独自のものに置き換えることができます。詳細は、「独自の CA 証明書と秘密鍵のインストール」 を参照してください。独自の CA 証明書を提供する場合は、有効期限が切れる前に更新する必要があります。

10.2. OPERATOR によって生成されたシークレット

シークレットは、**Kafka** や **KafkaUser** などのカスタムリソースがデプロイされるときに作成されます。AMQ Streams はこれらの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。これらは mTLS 認証にも使用されます。

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 または PKCS #12 形式の鍵と証明書を使用した TLS 認証

AMQ Streams によって作成されたシークレットは、PEM (Privacy Enhanced Mail) および PKCS #12 (Public-Key Cryptography Standards) 形式の秘密鍵と証明書を提供します。PEM および PKCS #12 は、SSL プロトコルを使用した TLS 通信用に OpenSSL で生成されたキー形式です。

Kafka クラスターおよびユーザー用に生成されたシークレットに含まれる認証情報を使用する相互 TLS (mTLS) 認証を設定できます。

mTLS をセットアップするには、まず次のことを行う必要があります。

- mTLS を使用するリスナーを使用して Kafka クラスターを設定する
- mTL のクライアント認証情報を提供する **KafkaUser** を作成する

Kafka クラスターをデプロイすると、クラスターを検証するために公開鍵を使用して **<cluster_name>-cluster-ca-cert** シークレットが作成されます。公開鍵を使用して、クライアントのトラストストアを設定します。

KafkaUser を作成すると、ユーザー (クライアント) を検証するためのキーと証明書を使用して **<kafka_user_name>** シークレットが作成されます。これらの資格証明を使用して、クライアントのキーストアを設定します。

mTLS を使用するように Kafka クラスターとクライアントをセットアップしたら、シークレットから認証情報を抽出し、それらをクライアント設定に追加します。

PEM キーと証明書

PEM の場合、クライアント設定に以下を追加します。

Truststore

- **<cluster_name>-cluster-ca-cert** シークレットからの **ca.crt**。これは、クラスターの CA 証明書です。

キーストア

- ユーザーの公開証明書である **<kafka_user_name>** シークレットからの **user.crt**。
- ユーザーの公開鍵である **<kafka_user_name>** シークレットの **user.key**。

PKCS #12 キーと証明書

PKCS #12 の場合、クライアント設定に以下を追加します。

Truststore

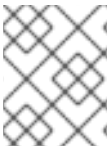
- **<cluster_name>-cluster-ca-cert** シークレットからの **ca.p12**。これは、クラスターの CA 証明書です。
- **<cluster_name>-cluster-ca-cert** シークレットの **ca.password**。これは、パブリッククラスター CA 証明書にアクセスするためのパスワードです。

キーストア

- **<kafka_user_name>** シークレットからの **user.p12**。これは、ユーザーの公開鍵証明書です。
- **<kafka_user_name>** シークレットの **user.password**。これは、Kafka ユーザーの公開鍵証明書にアクセスするためのパスワードです。

PKCS #12 は Java でサポートされているため、証明書の値を Java クライアント設定に直接追加できます。安全な保管場所から証明書を参照することもできます。PEM ファイルを使用する場合、証明書を

単一行形式でクライアント設定に直接追加する必要があります。Kafka クラスターとクライアント間の TLS 接続の確立に適した形式を選択します。PEM に慣れていない場合は、PKCS #12 を使用してください。



注記

すべてのキーのサイズは 2048 ビットで、既定では、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 ブローカーに接続するときに mTLS 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。mTLS 認証が使用されている場合、Kafka ブローカーは公開鍵を使用して、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

クルーズコントロールと 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>-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	証明書とキーを格納するための PKCS #12 ストア。
ca.password	PKCS #12 ストアを保護するためのパスワード。
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	証明書とキーを格納するための PKCS #12 ストア。
<cluster_name>-kafka-<num>.password	PKCS #12 ストアを保護するためのパスワード。

フィールド	説明
<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	証明書とキーを格納するための PKCS #12 ストア。
<cluster_name>-zookeeper-<num>.password	PKCS #12 ストアを保護するためのパスワード。
<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	証明書とキーを格納するための PKCS #12 ストア。
cluster-operator.password	PKCS #12 ストアを保護するためのパスワード。
cluster-operator.crt	クラスター Operator と Kafka または ZooKeeper との間の mTLS 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 mTLS 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	証明書とキーを格納するための PKCS #12 ストア。

フィールド	説明
entity-operator.password	PKCS #12 ストアを保護するためのパスワード。
entity-operator.crt	トピック Operator と Kafka または ZooKeeper との間の mTLS 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 mTLS 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	証明書とキーを格納するための PKCS #12 ストア。
entity-operator.password	PKCS #12 ストアを保護するためのパスワード。
entity-operator.crt	ユーザー Operator と Kafka または ZooKeeper との間の mTLS 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 mTLS 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	証明書とキーを格納するための PKCS #12 ストア。
cruise-control.password	PKCS #12 ストアを保護するためのパスワード。
cruise-control.crt	Cruise Control と Kafka または ZooKeeper との間の mTLS 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 mTLS 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	証明書とキーを格納するための PKCS #12 ストア。

フィールド	説明
kafka-exporter.password	PKCS #12 ストアを保護するためのパスワード。
kafka-exporter.crt	Kafka Exporter と Kafka または ZooKeeper との間の mTLS 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 mTLS 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、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	証明書とキーを格納するための PKCS #12 ストア。
ca.password	PKCS #12 ストアを保護するためのパスワード。
ca.crt	クライアント CA の現在の証明書。

10.2.5. User Operator によって生成されたユーザーシークレット

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**	証明書とキーを格納するための PKCS #12 ストア。
	user.password	PKCS #12 ストアを保護するためのパスワード。
	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ーを設定することで、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ーを設定します。

ラベルおよびアノテーションを Secret に追加するテンプレートのカスタマイズ例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    template:
      clusterCaCert:
        metadata:
          labels:
            label1: value1
            label2: value2
          annotations:
            annotation1: value1
            annotation2: value2
    # ...
```

テンプレートプロパティーの設定に関する詳細は、「OpenShift リソースのカスタマイズ」 を参照してください。

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** プロパティーで作成されます。つまり、**Kafka** カスタムリソースが削除されると、OpenShift によって CA シークレットも削除 (ガベッジコレクション) されます。

新しいクラスターで CA を再利用する場合は、**Kafka**設定でクラスターおよびクライアント CA シークレットの **generateSecretOwnerReference** プロパティーを **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。

- **Kafka.spec.clusterCa.validityDays** のクラスター CA 証明書
- **Kafka.spec.clientsCa.validityDays** のクライアント CA 証明書

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接続を受け入れません。代わりに、コンポーネントおよびクライアントは **新しい** CA 証明書を信頼する必要があります。

サービスを中断せずに CA 証明書を更新できるようにするため、Cluster Operator は古い CA 証明書が期限切れになる前に証明書の更新を開始します。

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

- **Kafka.spec.clusterCa.renewalDays** のクラスター CA 証明書
- **Kafka.spec.clientsCa.renewalDays** のクライアント CA 証明書

デフォルトの更新期間は、両方の証明書とも 30 日です。

更新期間は、現在の証明書の有効期日から逆算されます。

更新期間に対する有効期間

```
Not Before                    Not After
   |                             |
```

```
    |<--------------- validityDays --------------->|
                          <--- renewalDays --->|
```

Kafka クラスターの作成後に有効期間と更新期間の変更を行うには、**Kafka** カスタムリソースの設定と適用、およびmanually renew the CA certificatesを行います。証明書を手動で更新しないと、証明書が次回自動更新される際に新しい期間が使用されます。

証明書の有効および更新期間の Kafka 設定例

```
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 Cluster Operator の動作は、クラスター CA およびクライアント CA の **generateCertificateAuthority** 証明書生成プロパティーの設定によって異なります。

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、CA 証明書を更新する時に以下のプロセスをこの順序で実行します。

1. 新しい CA 証明書を生成しますが、既存のキーは保持します。
   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。

ユーザー証明書はクライアント CA により署名されます。User Operator によって生成されるユーザー証明書は、クライアント CA の更新時に更新されます。

10.3.2. クライアント証明書の更新

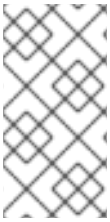
Cluster Operator は、Kafka クラスターを使用するクライアントアプリケーションを認識しません。

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。



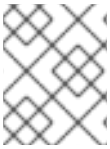
注記

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、これらの証明書の一方または両方を手動で更新することができます。セキュリティー上の理由や、証明書の更新または有効期間を変更した 場合などに、自動更新を行うことがあります。

更新された証明書は、更新前の証明書と同じ秘密鍵を使用します。



注記

独自の CA 証明書を使用している場合は、**force-renew** アノテーションは使用できません。代わりに、独自の CA 証明書を更新する 手順に従ってください。

前提条件

- Cluster Operator が稼働中です。
- CA 証明書と秘密鍵がインストールされている Kafka クラスターが必要です。

手順

1. **strimzi.io/force-renew** アノテーションを、更新対象の CA 証明書が含まれる **Secret** に適用します。

   表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**

   次回の調整で、アノテーションを付けた **Secret** の新規 CA 証明書が Cluster Operator によって生成されます。メンテナー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ーンス時間枠内で新規 CA 証明書が生成されます。

   Cluster Operator によって更新されたクラスターおよびクライアント CA 証明書をクライアントアプリケーションがリロードする必要があります。

2. CA 証明書が有効である期間を確認します。
   たとえば、**openssl** コマンドを使用します。

   ```
   oc get secret CA-CERTIFICATE-SECRET -o 'jsonpath={.data.CA-CERTIFICATE}' | base64 -d | openssl x509 -subject -issuer -startdate -enddate -noout
   ```

   CA-CERTIFICATE-SECRETは **Secret** の名前で、クラスター 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

- 「Operator によって生成されたシークレット」
- 「ローリング更新のメンテナーンス時間枠」
- 「**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 が稼働中です。
- CA 証明書と秘密鍵がインストールされている Kafka クラスターが必要です。

手順

- 更新対象の秘密鍵が含まれる **Secret** に **strimzi.io/force-replace** アノテーションを適用します。

  表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**

次回の調整時に、Cluster Operator は以下を生成します。

- アノテーションを付けた **Secret** の新しい秘密鍵
- 新規 CA 証明書

メンテナー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ー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

- 「Operator によって生成されたシークレット」
- 「ローリング更新のメンテナーンス時間枠」

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。Kafka コントローラーとブローカー間の接続は、ポート 9090 で内部 **コントロールプレーンリスナー** を使用します。ブローカー間のデータのレプリケーション、および AMQ Streams Operator、Cruise Control、または Kafka Exporter からの内部接続では、ポート 9091 で **レプリケーションリスナー** 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** プロパティー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ーで設定します。
    - セキュリティープロトコルのオプション:

- **security.protocol: SSL** (mTLS 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. 抽出した証明書を使用して、X.509 形式の証明書を使用するクライアントで TLS 接続を設定します。

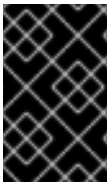
10.6. クラスター CA を信頼する外部クライアントの設定

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** シークレットに複数の 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** 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。

2. Kafka クライアントを以下のプロパティーで設定します。
   - セキュリティープロトコルのオプション:
     - **security.protocol: TLS** を使用する場合の SSL。
     - **security.protocol: SASL_SSL** (TLS 経由で SCRAM-SHA 認証を使用する場合)。
   - **ssl.truststore.location** (証明書がインポートされたトラストストアを指定)。
   - **ssl.truststore.password** (トラストストアにアクセスするためのパスワードを指定)。このプロパティーは、トラストストアで必要なければ省略できます。
   - **ssl.truststore.type=PKCS12** (トラストストアのタイプを識別)。

PEM 形式の使用 (.crt)

1. Kafka クラスターの **<cluster_name>-cluster-ca-cert** シークレットからクラスター 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. 抽出した証明書を使用して、X.509 形式の証明書を使用するクライアントで TLS 接続を設定します。

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ー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。

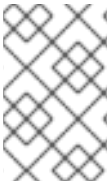
Kafka リスナー証明書の更新が必要な場合は、手作業で更新する必要があります。

10.7.1. TLS 暗号化用の独自の Kafka リスナー証明書を提供する

リスナーは、Kafka ブローカーへのクライアントアクセスを提供します。TLS を使用したクライアントアクセスに必要な設定を含め、**Kafka** リソースでリスナーを設定します。

デフォルトでは、リスナーは、AMQ Streams によって生成された内部 CA (認証局) 証明書によって署名された証明書を使用します。CA 証明書は、Kafka クラスターを作成するときに Cluster Operator によって生成されます。クライアントを TLS 用に設定するときは、CA 証明書をそのトラストストア設定に追加して、Kafka クラスターを検証します。独自の CA 証明書をインストールして使用する こともできます。または、**brokerCertChainAndKey** プロパティーを使用してリスナーを設定し、カスタムサーバー証明書を使用することもできます。

brokerCertChainAndKey プロパティーを使用すると、リスナーレベルで独自のカスタム証明書を使用して Kafka ブローカーにアクセスできます。独自のプライベートキーとサーバー証明書を使用してシークレットを作成し、リスナーの **brokerCertChainAndKey** 設定でキーと証明書を指定します。パブリック (外部) CA またはプライベート CA によって署名された証明書を使用できます。パブリック CA によって署名されている場合、通常、それをクライアントのトラストストア設定に追加する必要はありません。カスタム証明書は AMQ Streams によって管理されないため、手動で更新する必要があります。



注記

リスナー証明書は、TLS 暗号化とサーバー認証のみに使用されます。これらは TLS クライアント認証には使用されません。TLS クライアント認証にも独自の証明書を使用する場合は、独自のクライアント CA をインストールして使用する 必要があります。

前提条件

- Cluster Operator が稼働中です。
- 各リスナーには次のものが必要です。
  - 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。(X.509 証明書を PEM 形式で提供します。)
    複数のリスナーに対して 1 つのリスナー証明書を使用できます。
  - サブジェクト代替名 (SAN) は、各リスナーの証明書で指定されます。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。

自己署名証明書を使用していない場合は、証明書に CA チェーン全体を含む証明書を提供できます。

リスナーに対して TLS 暗号化 (**tls: true**) が設定されている場合は、**brokerCertChainAndKey** プロパティーのみを使用できます。

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
    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
    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

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

- *6章Kafka へのセキュアなアクセスの管理*

- 「Kafka リスナーのサーバー証明書の SAN」
- 「**Generic KafkaListener** スキーマ参照」

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

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

- 「TLS 暗号化用の独自の Kafka リスナー証明書を提供する」

10.8. 独自の CA 証明書と秘密鍵を使用する

Cluster Operator によって生成されたデフォルト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して使用します。クラスターとクライアントの CA 証明書および秘密鍵を置き換えることができます。

次の方法で、独自の CA 証明書と秘密鍵を使用するように切り替えることができます。

- Kafka クラスターをデプロイする前に、独自の CA 証明書と秘密鍵をインストールします。
- Kafka クラスターをデプロイした後、デフォルトの CA 証明書と秘密鍵を独自のものに置き換えます。

Kafka クラスターをデプロイした後にデフォルトの CA 証明書と秘密鍵を置き換える手順は、独自の CA 証明書と秘密鍵を更新するために使用する手順と同じです。

独自の証明書を使用する場合、証明書は自動的に更新されません。有効期限が切れる前に、CA 証明書と秘密鍵を更新する必要があります。

更新オプション:

- CA 証明書のみを更新する
- CA 証明書と秘密鍵を更新する (またはデフォルトを置き換える)

10.8.1. 独自の CA 証明書と秘密鍵のインストール

Cluster Operator によって生成されたクラスターおよびクライアントの CA 証明書と秘密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します。

デフォルトでは、AMQ Streams は次の クラスター CA とクライアント CA シークレット を使用します。これらは自動的に更新されます。

- クラスター CA シークレット
  - **<cluster_name>-cluster-ca**
  - **<cluster_name>-cluster-ca-cert**
- クライアント CA シークレット
  - **<cluster_name>-clients-ca**
  - **<cluster_name>-clients-ca-cert**

独自の証明書をインストールするには、同じ名前を使用します。

前提条件

- Cluster Operator が稼働中です。
- Kafka クラスターがデプロイされていない必要があります。
  すでに Kafka クラスターをデプロイしている場合は、デフォルトの CA 証明書を独自の証明書に置き換える ことができます。
- 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

Cluster Operator は、キーと証明書を PEM (Privacy Enhanced Mail) および PKCS #12 (Public-Key Cryptography Standards) 形式で生成します。どちらの形式でも独自の証明書を追加できます。

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。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> を独自のパスワードに置き換えます。

手順

1. CA 証明書を含む新しい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
   ```

   <cluster_name> は、独自の Kafka クラスターの名前に置き換えます。

2. 秘密鍵を含む新しい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。
     独自の CA 証明書の増分値 (**strimzi.io/ca-cert-generation=0**) として 0 (ゼロ) から開始します。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

- 「独自の CA 証明書の更新」
- 「CA 証明書と秘密鍵を独自のものに更新または置換する」
- 「TLS 暗号化用の独自の Kafka リスナー証明書を提供する」

10.8.2. 独自の CA 証明書の更新

独自の CA 証明書を使用している場合は、手動で更新する必要があります。Cluster Operator はそれらを自動的に更新しません。有効期限が切れる前に、更新期間内に CA 証明書を更新します。

CA 証明書を更新し、同じ秘密キーを使用して続行する場合は、この手順のステップを実行します。独自の CA 証明書 **および** 秘密鍵を更新する場合は、「CA 証明書と秘密鍵を独自のものに更新または置換する」 を参照してください。

この手順では、PEM 形式の CA 証明書の更新を説明します。

前提条件

- Cluster Operator が稼働中です。
- 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: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... 1
   metadata:
     annotations:
       strimzi.io/ca-cert-generation: "0" 2
   ```

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

1 現在の base64 でエンコードされた CA 証明書

2 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** プロパティー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 証明書の場合は、より高い増分値で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... 1
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" 2
  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。

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

10.8.3. CA 証明書と秘密鍵を独自のものに更新または置換する

独自の CA 証明書と秘密鍵を使用している場合は、手動で更新する必要があります。Cluster Operator はそれらを自動的に更新しません。有効期限が切れる前に、更新期間内に CA 証明書を更新します。同じ手順を使用して、AMQ Streams Operator によって生成された CA 証明書と秘密鍵を独自のものに置き換えることもできます。

CA 証明書と秘密鍵を更新または置換する場合は、この手順のステップを実行してください。独自の CA 証明書のみを更新する場合は、「独自の CA 証明書の更新」 を参照してください。

この手順では、PEM 形式の CA 証明書と秘密鍵の更新について説明します。

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

- Cluster Operator が稼働中です。
- 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、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... (1)
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "0" (2)
  labels:
    strimzi.io/cluster: my-cluster
    strimzi.io/kind: Kafka
  name: my-cluster-cluster-ca-cert
  #...
type: Opaque
```

1. 現在の base64 でエンコードされた CA 証明書
2. 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** プロパティー名を **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、プロパティーの値を残します。

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** プロパティーを作成し、上の手順から base64 でエンコードされた CA 証明書を **ca.crt** プロパティー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 証明書の場合は、より高い増分値で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... (1)
  ca-2022-01-26T17-32-00Z.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0F... (2)
metadata:
  annotations:
    strimzi.io/ca-cert-generation: "1" (3)
  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) 古い base64 でエンコードされた CA 証明書

(3) 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: SA1cKF1GFDzOIiPOIUQBHDNFGDFS... (1)
metadata:
  annotations:
    strimzi.io/ca-key-generation: "0" (2)
  labels:
    strimzi.io/cluster: my-cluster
```

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

1 現在の base64 でエンコードされた CA キー

2 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** プロパティー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 証明書と新しい CA キーの場合は、より高い増分値で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: AB0cKF1GFDzOIiPOIUQWERZJQ0F... 1
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。

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

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 クラスターを取得します。

リソースタイプを参照する際には、単数形と複数形の両方の名前を使うことができます。**oc get kafkas** は **oc get kafka** と同じ結果になります。

リソースの **短縮名** を使用することもできます。短縮名を理解すると、AMQ Streams を管理する時間を節約できます。**Kafka** のショートネームは **k** なので、**oc get k** を実行してすべての Kafka クラスターをリストアップすることもできます。

```
oc get k

NAME         DESIRED KAFKA REPLICAS   DESIRED ZK REPLICAS
my-cluster   3                        3
```

表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一覧表示されます。

```
oc get strimzi

NAME                                DESIRED KAFKA REPLICAS DESIRED ZK REPLICAS
kafka.kafka.strimzi.io/my-cluster      3                      3

NAME                                PARTITIONS REPLICATION FACTOR
kafkatopic.kafka.strimzi.io/kafka-apps 3          3

NAME                                AUTHENTICATION AUTHORIZATION
kafkauser.kafka.strimzi.io/my-user     tls            simple
```

oc get strimzi -o name コマンドは、すべてのリソースタイプとリソース名を返します。**-o name** オプションは **type/name** 形式で出力を取得します。

```
oc get strimzi -o name

kafka.kafka.strimzi.io/my-cluster
kafkatopic.kafka.strimzi.io/kafka-apps
kafkauser.kafka.strimzi.io/my-user
```

この **strimzi** コマンドを他のコマンドと組み合わせることができます。たとえば、これを **oc delete** コマンドに渡して、単一のコマンドですべてのリソースを削除できます。

```
oc delete $(oc get strimzi -o name)

kafka.kafka.strimzi.io "my-cluster" deleted
kafkatopic.kafka.strimzi.io "kafka-apps" deleted
kafkauser.kafka.strimzi.io "my-user" deleted
```

1 つの操作ですべてのリソースを削除することは、AMQ Streams の新機能をテストする場合などに役立ちます。

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ーまたはプロパティー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** プロパティーがあります。

表11.2 カスタムリソースの status プロパティー

AMQ Streams リソース	スキーマ参照	ステータス情報がパブリッシュされる場所
Kafka	「**KafkaStatus** スキーマ参照」	Kafka クラスター。
KafkaConnect	「**KafkaConnectStatus** スキーマ参照」	デプロイされている場合は Kafka Connect クラスター。
KafkaConnector	「**KafkaConnectorStatus** スキーマ参照」	デプロイされている場合は KafkaConnector リソース。
KafkaMirrorMaker	「**KafkaMirrorMakerStatus** スキーマ参照」	デプロイされている場合は Kafka MirrorMaker ツール。
KafkaTopic	「**KafkaTopicStatus** スキーマ参照」	Kafka クラスターの Kafka トピック
KafkaUser	「**KafkaUserStatus** スキーマ参照」	Kafka クラスターの Kafka ユーザー。
KafkaBridge	「**KafkaBridgeStatus** スキーマ参照」	デプロイされている場合は AMQ Streams の Kafka Bridge。

リソースの **status** プロパティーによって、リソースの下記項目の情報が提供されます。

- **status.conditions** プロパティーの **Current state** (現在の状態)。
- **status.observedGeneration** プロパティーの **Last observed generation** (最後に確認された生成)。

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

- **KafkaStatus** によって、リスナーアドレスに関する情報と Kafka クラスターの ID が提供されます。

- **KafkaConnectStatus** によって、Kafka Connect コネクターの REST API エンドポイントが提供されます。
- **KafkaUserStatus** によって、Kafka ユーザーの名前と、ユーザーのクレデンシャルが保存される **Secret** が提供されます。
- **KafkaBridgeStatus** によって、外部クライアントアプリケーションが Bridge サービスにアクセスできる HTTP アドレスが提供されます。

リソースの **Current state** (現在の状態) は、**spec** プロパティーによって定義される **Desired state** (望ましい状態) を実現するリソースに関する進捗を追跡するのに便利です。ステータス条件によって、リソースの状態が変更された時間および理由が提供され、Operator によるリソースの望ましい状態の実現を妨げたり遅らせたりしたイベントの詳細が提供されます。

Last observed generation (最後に確認された生成) は、Cluster Operator によって最後に照合されたリソースの生成です。**observedGeneration** の値が **metadata.generation** の値と異なる場合、リソースの最新の更新が Operator によって処理されていません。これらの値が同じである場合、リソースの最新の変更がステータス情報に反映されます。

AMQ Streams によってカスタムリソースのステータスが作成および維持されます。定期的にカスタムリソースの現在の状態が評価され、その結果に応じてステータスが更新されます。くださいーたとえば、**oc edit** を使用してカスタムリソースで更新を行う場合、その **status** は編集不可能です。さらに、**status** の変更は Kafka クラスターステータスの設定に影響しません。

以下では、Kafka カスタムリソースに **status** プロパティーが指定されています。

Kafka カスタムリソースとステータス

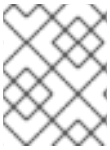
```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
spec:
  # ...
status:
  conditions: 1
  - lastTransitionTime: 2021-07-23T23:46:57+0000
    status: "True"
    type: Ready 2
  observedGeneration: 4 3
  listeners: 4
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9092
    type: plain
  - addresses:
    - host: my-cluster-kafka-bootstrap.myproject.svc
      port: 9093
    certificates:
    - |
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    type: tls
  - addresses:
    - host: 172.29.49.180
      port: 9094
```

```
    certificates:
    - |
      -----BEGIN CERTIFICATE-----
      ...
      -----END CERTIFICATE-----
    type: external
  clusterId: CLUSTER-ID 5
# ...
```

1. status の **conditions** は、既存のリソース情報から推測できないステータスに関連する基準や、リソースのインスタンスに固有する基準を記述します。
2. **Ready** 条件は、Cluster Operator が現在 Kafka クラスターでトラフィックの処理が可能であると判断するかどうかを示しています。
3. **observedGeneration** は、最後に Cluster Operator によって照合された **Kafka** カスタムリソースの生成を示しています。
4. **listeners** は、現在の Kafka ブートストラップアドレスをタイプ別に示しています。
5. Kafka クラスター ID。

重要

タイプが **nodeport** の外部リスナーのカスタムリソースステータスにおけるアドレスは、現在サポートされていません。



注記

Kafka ブートストラップアドレスがステータスに一覧表示されても、それらのエンドポイントまたは Kafka クラスターが準備状態であるとは限りません。

ステータス情報のアクセス

リソースのステータス情報はコマンドラインから取得できます。詳細は、「カスタムリソースのステータスの検出」 を参照してください。

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 クラスター
- Cluster Operator が稼働中です。

手順

- カスタムリソースを指定し、**-o jsonpath** オプションを使用して標準の JSONPath 式を適用して **status** プロパティーを選択します。

```
oc get kafka <kafka_resource_name> -o jsonpath='{.status}'
```

この式は、指定されたカスタムリソースのすべてのステータス情報を返します。**status.listeners** または **status.observedGeneration** などのドット表記を使用すると、表示するステータス情報を微調整できます。

関連情報

- 「AMQ Streams カスタムリソースのステータス情報」
- JSONPath の使用に関する詳細は、JSONPath support を参照してください。

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

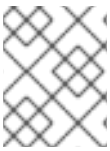
- OpenShift リソースのカスタマイズ
- カスタムリソースのステータスの検出

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、メンテナーンス、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時に稼働を継続できます。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** プロパティーは、HTTPS 通信を検証する証明書チェーンを指定します。証明書は Base64 でエンコードされます。

Pod エビクション通知の Webhook 設定

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
# ...
webhooks:
  - name: strimzi-drain-cleaner.strimzi.io
    rules:
      - apiGroups:   [""]
        apiVersions: ["v1"]
        operations:  ["CREATE"]
        resources:   ["pods/eviction"]
        scope:       "Namespaced"
    clientConfig:
      service:
        namespace: "strimzi-drain-cleaner"
        name: "strimzi-drain-cleaner"
        path: /drainer
        port: 443
        caBundle: Cg==
    # ...
```

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" (1)
          # ...
```

(1) 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 (停止状態の予算) の指定

   ```
   apiVersion: kafka.strimzi.io/v1beta2
   kind: Kafka
   metadata:
     name: my-cluster
     namespace: myproject
   spec:
     kafka:
       template:
         podDisruptionBudget:
           maxUnavailable: 0
   ```

```
# ...
zookeeper:
  template:
    podDisruptionBudget:
      maxUnavailable: 0
# ...
```

最大 Pod 中断バジェットをゼロに減らすと、自発的な中断の場合に OpenShift が Pod を自動的に削除するのを防ぎ、AMQ Streams Drain Cleaner と 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。

- OpenShift で Drain Cleaner を実行するには、**/install/drain-cleaner/openshift** ディレクトリーにあるリソースを適用します。

```
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
```

3. 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ーの復元

Kafka クラスターは、永続ボリューム (PV) が存在していれば、そこから復元できます。

たとえば、以下の場合に行います。

- namespace が意図せずに削除された後。
- OpenShift クラスター全体が失われた後でも PV がインフラストラクチャーに残っている場合。

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 にマウントされます。

PV の回収 (reclaim) ポリシーは、namespace が削除されるときにクラスターに動作方法を指示します。以下に、回収 (reclaim) ポリシーの設定とその結果を示します。

- **Delete** (デフォルト) に設定すると、PVC が namespace 内で削除されるときに PV が削除されます。
- **Retain** に設定すると、namespace の削除時に PV は削除されません。

namespace が意図せず削除された場合に PV から復旧できるようにするには、PV 仕様で **persistentVolumeReclaimPolicy** プロパティーを使用してポリシーを **Delete** から **Retain** にリセットする必要があります。

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
  # ...
  persistentVolumeReclaimPolicy: Retain
```

または、PV は、関連付けられたストレージクラスの回収 (reclaim) ポリシーを継承できます。ストレージクラスは、動的ボリュームの割り当てに使用されます。

ストレージクラスの **reclaimPolicy** プロパティーを設定することで、ストレージクラスを使用する PV が適切な回収 (reclaim) ポリシー で作成されます。ストレージクラスは、**storageClassName** プロパティーを使用して PV に対して設定されます。

```
apiVersion: v1
kind: StorageClass
metadata:
  name: gp2-retain
parameters:
  # ...
# ...
reclaimPolicy: Retain
```

```
apiVersion: v1
kind: PersistentVolume
# ...
spec:
  # ...
  storageClassName: gp2-retain
```

注記

Retain を回収 (reclaim) ポリシーとして使用しながら、クラスター全体を削除する場合は、PV を手動で削除する必要があります。そうしないと、PV は削除されず、リソースに不要な経費がかかる原因になります。

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

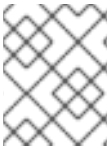
この手順では、削除されたクラスターを永続ボリューム (PV) から復元する方法を説明します。

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** プロパティーを削除します。
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、以下のプロパティーが削除されます。

```
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: data-0-my-cluster-kafka-2
    namespace: myproject
    resourceVersion: "39113"
    uid: 54be1c60-3319-11ea-97b0-0aef8816c7ea
```

5. Cluster Operator をデプロイします。

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

6. クラスターを再作成します。
   クラスターの再作成に必要なすべての **KafkaTopic** リソースがあるかどうかに応じて、以下の手順を実行します。

オプション 1: クラスターを失う前に存在した **KafkaTopic** リソースが **すべて** ある場合 (**__consumer_offsets** からコミットされたオフセットなどの内部トピックを含む)。

1. すべての **KafkaTopic** リソースを再作成します。
クラスターをデプロイする前にリソースを再作成する必要があります。そうでないと、Topic Operator によってトピックが削除されます。

2. Kafka クラスターをデプロイします。
以下に例を示します。

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

オプション 2: クラスターを失う前に存在したすべての **KafkaTopic** リソースがない場合。

1. オプション 1 と同様に Kafka クラスターをデプロイしますが、デプロイ前に Kafka リソースから **topicOperator** プロパティーを削除して、Topic Operator がない状態でデプロイします。
デプロイメントに Topic Operator が含まれると、Topic Operator によってすべてのトピックが削除されます。

2. Kafka クラスターから内部トピックストアのトピックを削除します。

```
oc run kafka-admin -ti --image=registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

このコマンドは、Kafka クラスターへのアクセスに使用されるリスナーおよび認証のタイプに対応している必要があります。

3. Kafka クラスターを **topicOperator** プロパティーで再デプロイして TopicOperator を有効にし、**KafkaTopic** リソースを再作成します。
以下に例を示します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  #...
  entityOperator:
    topicOperator: {} 1
    #...
```

1 ここで示すデフォルト設定には、追加のプロパティーはありません。「**EntityTopicOperatorSpec** スキーマ参照」に説明されているプロパティーを使用して、必要な設定を指定します。

7. **KafkaTopic** リソースのリストを表示して、復旧を確認します。

```
oc get KafkaTopic
```

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 ををプロデューサーに設定できます。2 つのプロデューサーが実行されている場合、それぞれのスループットは 20MBps に制限されます。

ストレージクォータは、Kafka ディスクストレージの制限をソフト制限とハード制限間で調整します。この制限は、利用可能なすべてのディスク容量に適用されます。プロデューサーは、ソフト制限とハード制限の間で徐々に遅くなります。制限により、ディスクの使用量が急激に増加しないようにし、容量を超えないようにします。ディスクがいっぱいになると、修正が難しい問題が発生する可能性があります。ハード制限は、ストレージの上限です。

注記

JBOD ストレージの場合、制限はすべてのディスクに適用されます。ブローカーが 2 つの 1 TB ディスクを使用し、クォータが 1.1 TB の場合は、1 つのディスクにいっぱいになり、別のディスクがほぼ空になることがあります。

前提条件

- Kafka クラスターを管理する Cluster Operator が稼働している。

手順

1. **Kafka** リソースの **config** にプラグインのプロパティーを追加します。
   プラグインプロパティーは、この設定例のとおりです。

   Kafka Static Quota プラグインの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    config:
      client.quota.callback.class: io.strimzi.kafka.quotas.StaticQuotaCallback (1)
      client.quota.callback.static.produce: 1000000 (2)
      client.quota.callback.static.fetch: 1000000 (3)
      client.quota.callback.static.storage.soft: 400000000000 (4)
      client.quota.callback.static.storage.hard: 500000000000 (5)
      client.quota.callback.static.storage.check-interval: 5 (6)
```

(1) Kafka Static Quota プラグインを読み込みます。

2 プロデューサーのバイトレートしきい値を設定します。この例では 1 MBps です。

3 コンシューマーのバイトレートしきい値を設定します。この例では 1 MBps です。

4 ストレージのソフト制限の下限を設定します。この例では 400 GB です。

5 ストレージのハード制限の上限を設定します。この例では 500 GB です。

6 ストレージのチェックの間隔 (秒単位) を設定します。この例では 5 秒です。これを 0 に設定するとチェックを無効にできます。

2. リソースを更新します。

```
oc apply -f <kafka_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ーゾーンなどの実行中の **Node** についての情報を取得できなければなりません。**Node** はクラスタースコープのリソースであるため、アクセスは namespace スコープの **RoleBinding** ではなく、**ClusterRoleBinding** を介してのみ許可できます。

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。

プロパティーファイルを使用してクライアントを設定する場合は、以下のように実行できます。

```
ssl.endpoint.identification.algorithm=
```

Java でクライアントを直接設定する場合は、設定オプションを空の文字列に設定します。

```
props.put("ssl.endpoint.identification.algorithm", "");
```

第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ー

共通設定プロパティーは複数のリソースに適用されます。

12.1.1. replicas

replicas プロパティーを使用してレプリカを設定します。

レプリケーションのタイプはリソースによって異なります。

- **KafkaTopic** はレプリケーション係数を使用して、Kafka クラスター内で各パーティションのレプリカ数を設定します。
- Kafka コンポーネントはレプリカを使用してデプロイメントの Pod 数を設定し、可用性とスケーラビリティーを向上します。

注記

OpenShift で Kafka コンポーネントを実行している場合、高可用性のために複数のレプリカを実行する必要がない場合があります。コンポーネントがデプロイされたノードがクラッシュすると、OpenShift によって自動的に Kafka コンポーネント Pod が別のノードに再スケジュールされます。ただし、複数のレプリカで Kafka コンポーネントを実行すると、他のノードが稼働しているため、フェイルオーバー時間が短縮されます。

12.1.2. bootstrapServers

bootstrapServers プロパティーを使用してブートストラップサーバーのリストを設定します。

ブートストラップサーバーリストは、同じ OpenShift クラスターにデプロイされていない Kafka クラスターを参照できます。AMQ Streams によってデプロイされた Kafka クラスターを参照することもできます。

同じ OpenShift クラスターである場合、各リストに **CLUSTER-NAME-kafka-bootstrap** という名前の Kafka クラスターブートストラップサービスとポート番号が含まれる必要があります。AMQ Streams によって異なる OpenShift クラスターにデプロイされた場合、リストの内容はクラスターを公開するために使用された方法によって異なります (route、ingress、nodeport、または loadbalancer)。

AMQ Streams によって管理されない Kafka クラスターで Kafka を使用する場合は、指定のクラスターの設定に応じてブートストラップサーバーのリストを指定できます。

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 プロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

SSL の設定例

```
# ...
spec:
  config:
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384" (1)
    ssl.enabled.protocols: "TLSv1.2" (2)
    ssl.protocol: "TLSv1.2" (3)
    ssl.endpoint.identification.algorithm: HTTPS (4)
# ...
```

1. TLS の暗号スイートは、**ECDHE** 鍵交換メカニズム、**RSA** 認証アルゴリズム、**AES** 一括暗号化アルゴリズム、および **SHA384** MAC アルゴリズムの組み合わせを使用します。
2. SSl プロトコル **TLSv1.2** は有効になります。
3. **TLSv1.2** プロトコルを指定し、SSL コンテキスト生成します。許可される値は **TLSv1.1** および **TLSv1.2** です。
4. ホスト名の検証は、**HTTPS** に設定して有効化されます。空の文字列を指定すると検証が無効になります。

12.1.4. trustedCertificates

tls を設定して TLS 暗号化を設定する場合は、**trustedCertificates** プロパティーを使用して、証明書が X.509 形式で保存されるキー名にシークレットの一覧を提供します。

Kafka クラスターの Cluster Operator によって作成されるシークレットを使用するか、独自の TLS 証明書ファイルを作成してから、ファイルから **Secret** を作成できます。

```
oc create secret generic MY-SECRET \
--from-file=MY-TLS-CERTIFICATE-FILE.crt
```

TLS による暗号化の設定例

```
tls:
  trustedCertificates:
    - secretName: my-cluster-cluster-cert
      certificate: ca.crt
    - secretName: my-cluster-cluster-cert
      certificate: ca2.crt
```

複数の証明書が同じシークレットに保存されている場合は、複数回リストできます。

TLS 暗号化を有効にし、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: []
```

mTLS 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 コンテナー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。

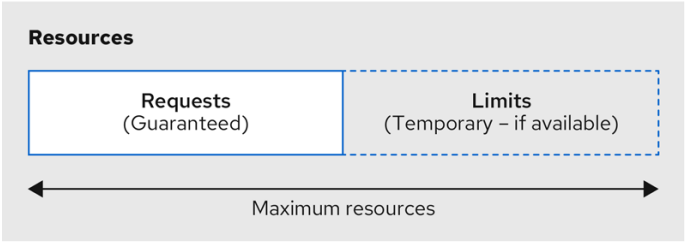
実稼働環境でリソースを設定する方法は、さまざまな要因によって異なります。たとえば、アプリケーションは OpenShift クラスターでリソースを共有する可能性があります。

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、常にコンテナーで利用可能になります。リソース制限によって、指定のコンテナー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。コンテナーは、リソースが利用できる場合のみ、制限以下のリソースを使用できます。リソースの制限は一時的で、再割り当てが可能です。

リソース要求および制限



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ー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。

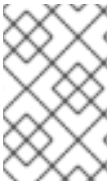
リソース設定の例

```
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 クラスターで利用可能な空きリソースを超える場合、Pod はスケジュールされません。



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** プロパティーを設定 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 プロパティーを使用して、コンポーネントによって使用されるコンテナーイメージを設定します。

コンテナーイメージのオーバーライドは、別のコンテナーレジストリーやカスタマイズされたイメージを使用する必要がある特別な状況でのみ推奨されます。

たとえば、ネットワークで AMQ Streams によって使用されるコンテナーリポジトリーへのアクセスが許可されない場合、AMQ Streams イメージのコピーまたはソースからのビルドを行うことができます。しかし、設定したイメージが AMQ Streams イメージと互換性のない場合は、適切に機能しない可能性があります。

コンテナーイメージのコピーはカスタマイズでき、デバッグに使用されることもあります。

以下のリソースの **image** プロパティーを使用すると、コンポーネントに使用するコンテナーイメージを指定できます。

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

- **KafkaMirrorMaker.spec**
- **KafkaMirrorMaker2.spec**
- **KafkaBridge.spec**

Kafka、Kafka Connect、および Kafka MirrorMaker のimage プロパティーの設定

Kafka、Kafka Connect、および Kafka MirrorMaker では、複数の Kafka バージョンがサポートされます。各コンポーネントには独自のイメージが必要です。異なる Kafka バージョンのデフォルトイメージは、以下の環境変数で設定されます。

- **STRIMZI_KAFKA_IMAGES**
- **STRIMZI_KAFKA_CONNECT_IMAGES**
- **STRIMZI_KAFKA_MIRROR_MAKER_IMAGES**

これらの環境変数には、Kafka バージョンと対応するイメージ間のマッピングが含まれます。マッピングは、**image** および **version** プロパティーとともに使用されます。

- **image** と **version** のどちらもカスタムリソースに指定されていない場合、**version** は Cluster Operator のデフォルトの Kafka バージョンに設定され、環境変数のこのバージョンに対応するイメージが指定されます。
- **image** が指定されていても **version** が指定されていない場合、指定されたイメージが使用され、Cluster Operator のデフォルトの Kafka バージョンが **version** であると想定されます。
- **version** が指定されていても **image** が指定されていない場合、環境変数の指定されたバージョンに対応するイメージが使用されます。
- **version** と **image** の両方を指定すると、指定されたイメージが使用されます。このイメージには、指定のバージョンの Kafka イメージが含まれると想定されます。

異なるコンポーネントの **image** および **version** は、以下のプロパティーで設定できます。

- Kafka の場合は **spec.kafka.image** および **spec.kafka.version**。
- **spec.image** および **spec.version** の Kafka Connect および Kafka MirrorMaker の場合。

警告

version のみを提供し、**image** プロパティーを未指定のままにしておくことが推奨されます。これにより、カスタムリソースの設定時に間違いが発生する可能性が低減されます。異なるバージョンの Kafka に使用されるイメージを変更する必要がある場合は、Cluster Operator の環境変数を設定することが推奨されます。

他のリソースでの image プロパティーの設定

他のカスタムリソースの **image** プロパティーでは、デプロイメント中に指定の値が使用されます。**image** プロパティーがない場合、Cluster Operator 設定に指定された **image** が使用されます。**image** 名が Cluster Operator 設定に定義されていない場合、デフォルト値が使用されます。

- Topic Operator の場合:
  1. Cluster Operator 設定から **STRIMZI_DEFAULT_TOPIC_OPERATOR_IMAGE** 環境変数に指定されたコンテナーイメージ。
  2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0** container image.
- User Operator の場合:
  1. Cluster Operator 設定から **STRIMZI_DEFAULT_USER_OPERATOR_IMAGE** 環境変数に指定されたコンテナーイメージ。
  2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0** container image.
- Entity Operator TLS サイドカーの場合:
  1. Cluster Operator 設定から **STRIMZI_DEFAULT_TLS_SIDECAR_ENTITY_OPERATOR_IMAGE** 環境変数に指定されたコンテナーイメージ。
  2. **registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0** container image.
- Kafka Exporter の場合:
  1. Cluster Operator 設定から **STRIMZI_DEFAULT_KAFKA_EXPORTER_IMAGE** 環境変数に指定されたコンテナーイメージ。
  2. **registry.redhat.io/amq7/amq-streams-kafka-33-rhel8:2.3.0** container image.
- Kafka Bridge の場合:
  1. Cluster Operator 設定から **STRIMZI_DEFAULT_KAFKA_BRIDGE_IMAGE** 環境変数に指定されたコンテナーイメージ。
  2. **registry.redhat.io/amq7/amq-streams-bridge-rhel8:2.3.0** container image.
- Kafka ブローカーイニシャライザーの場合:
  1. Cluster Operator 設定から **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** 環境変数に指定されたコンテナーイメージ。
  2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0** container image.

コンテナーイメージ設定の例

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

12.1.7. livenessProbe および readinessProbe healthcheck

livenessProbe および **readinessProbe** プロパティーを使用して、AMQ Streams でサポートされる healthcheck プローブを設定します。

Healthcheck は、アプリケーションの健全性を検証する定期的なテストです。ヘルスチェックプローブが失敗すると、OpenShift によってアプリケーションが正常でないと見なされ、その修正が試行されます。

プローブの詳細は、Configure Liveness and Readiness Probes を参照してください。

livenessProbe および **readinessProbe** の両方で以下のオプションがサポートされます。

- **initialDelaySeconds**
- **timeoutSeconds**
- **periodSeconds**
- **successThreshold**
- **failureThreshold**

Liveness および Readiness プローブの設定例

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

livenessProbe および **readinessProbe** のオプションに関する詳細は、Probe スキーマ参照 を参照してください。

12.1.8. metricsConfig

metricsConfig プロパティーを使用して、Prometheus メトリクスを有効化および設定します。

metricsConfig プロパティーには、Prometheus JMX Exporter の追加設定が含まれる ConfigMap への参照が含まれます。AMQ Streams では、Apache Kafka および ZooKeeper によってサポートされる JMX メトリクスを Prometheus メトリクスに変換するために、Prometheus JMX エクスポーターを使用した Prometheus メトリクスがサポートされます。

追加設定なしで Prometheus メトリクスのエクスポートを有効にするには、**metricsConfig.valueFrom.configMapKeyRef.key** 配下に空のファイルが含まれる ConfigMap を参照します。空のファイルを参照する場合、名前が変更されていない限り、すべてのメトリクスが公開されます。

Kafka のメトリクス設定が含まれる ConfigMap の例

```
kind: ConfigMap
apiVersion: v1
```

```
metadata:
  name: my-configmap
data:
  my-key: |
    lowercaseOutputName: true
    rules:
    # Special cases and very specific rules
    - pattern: kafka.server<type=(.+), name=(.+), clientId=(.+), topic=(.+), partition=(.*)><>Value
      name: kafka_server_$1_$2
      type: GAUGE
      labels:
       clientId: "$3"
       topic: "$4"
       partition: "$5"
    # further configuration
```

Kafka のメトリクス設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: my-key
    # ...
  zookeeper:
    # ...
```

有効になったメトリクスは、9404 番ポートで公開されます。

metricsConfig (または非推奨の **metrics**) プロパティーがリソースに定義されていない場合、Prometheus メトリクスは無効になります。

Prometheus および Grafana の設定およびデプロイに関する詳細は、**OpenShift での AMQ Streams のデプロイおよびアップグレード**のKafka へのメトリクスの導入を参照してください。

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** プロパティーを設定します。

- **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。

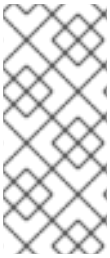
-Xms
: JVM の起動時に最初に割り当てられる最小ヒープサイズ。

-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ー

gcLoggingEnabled
: ガベッジコレクターのロギングを有効にします



注記

-Xmx や **-Xms** などの JVM 設定で使用できる単位は、対応するイメージの JDK **java** バイナリーで使用できる単位と同じです。そのため、**1g** または **1G** は 1,073,741,824 バイトを意味し、**Gi** は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

コンテナー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、コンテナーの 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。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 はコンテナーを強制終了します。
- OpenShift がメモリー不足のノードにコンテナー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、コンテナーはすぐにクラッシュし、存在しない場合、コンテナー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ーなど、多数のディスク I/O を実行するコンテナーには、オペレーティングシステムのページキャッシュとして使用できるメモリーが必要です。このようなコンテナーの場合、要求されるメモリーは、JVM が使用するメモリー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 の詳細については、**JvmOptions** スキーマリファレンス を参照してください。

12.1.10. ガベッジコレクターのロギング

jvmOptions プロパティーでは、ガベージコレクター (GC) のロギングを有効または無効にすることもできます。GC ロギングはデフォルトで無効になっています。これを有効にするには、以下のように **gcLoggingEnabled** プロパティーを設定します。

GC ロギングの設定例

```
# ...
jvmOptions:
  gcLoggingEnabled: true
# ...
```

12.2. スキーマプロパティー

12.2.1. Kafka スキーマ参照

プロパティー	説明
spec	Kafka および ZooKeeper クラスター、Topic Operator の仕様。
KafkaSpec	
status	Kafka および ZooKeeper クラスター、Topic Operator のステータス。
KafkaStatus	

12.2.2. KafkaSpec スキーマ参照

Kafka で使用

プロパティー	説明
kafka	Kafka クラスターの設定。
KafkaClusterSpec	
zookeeper	ZooKeeper クラスターの設定。
ZookeeperClusterSpec	
entityOperator	Entity Operator の設定。
EntityOperatorSpec	
clusterCa	クラスター認証局の設定。
CertificateAuthority	
clientsCa	クライアント認証局の設定。
CertificateAuthority	

プロパティー	説明
cruiseControl	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 は追加のメトリクスを提供できます (例: トピック/パーティションでのコンシューマーグループのラグなど)。
KafkaExporterSpec	
maintenanceTimeWindows	メンテナーンスタスク (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ーの完全リスト

Kafka クラスターを設定します。

12.2.3.1. listeners

listeners プロパティーを使用して、Kafka ブローカーへのアクセスを提供するようにリスナーを設定します。

認証のないプレーン (暗号化されていない) リスナーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  kafka:
    # ...
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
    # ...
  zookeeper:
    # ...
```

12.2.3.2. config

config プロパティーを使用して、Kafka ブローカーオプションをキーとして設定します。

標準の Apache Kafka 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。

以下に関連する設定オプションは設定できません。

- セキュリティー (暗号化、認証、および承認)
- リスナーの設定
- Broker ID の設定
- ログデータディレクトリーの設定
- ブローカー間の通信
- ZooKeeper の接続

値は以下の JSON タイプのいずれかになります。

- 文字列
- 数値
- ブール値

AMQ Streams で直接管理されるオプションを除き、Apache Kafka ドキュメント に記載されているオプションを指定および設定できます。以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはすべて禁止されています。

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

禁止されているオプションが **config** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。サポートされるその他すべてのオプションは Kafka に渡されます。

禁止されているオプションには例外があります。TLS バージョンの特定の **暗号スイート** を使用するクライアント接続に、許可された **ssl** プロパティーを設定 することができます。**zookeeper.connection.timeout.ms** プロパティーを設定して、ZooKeeper 接続の確立に許可される最大時間を設定することもできます。

Kafka ブローカーの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    config:
      num.partitions: 1
      num.recovery.threads.per.data.dir: 1
      default.replication.factor: 3
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 1
      log.retention.hours: 168
      log.segment.bytes: 1073741824
      log.retention.check.interval.ms: 300000
      num.network.threads: 3
      num.io.threads: 8
      socket.send.buffer.bytes: 102400
      socket.receive.buffer.bytes: 102400
      socket.request.max.bytes: 104857600
      group.initial.rebalance.delay.ms: 0
      ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
      ssl.enabled.protocols: "TLSv1.2"
      ssl.protocol: "TLSv1.2"
      zookeeper.connection.timeout.ms: 6000
    # ...
```

12.2.3.3. brokerRackInitImage

ラックアウェアネス (Rack Awareness) が有効である場合、Kafka ブローカー Pod は init コンテナーを使用して OpenShift クラスターノードからラベルを収集します。このコンテナーに使用されるコンテナーイメージは、**brokerRackInitImage** プロパティーを使用して設定できます。**brokerRackInitImage** フィールドがない場合、以下のイメージが優先度順に使用されます。

1. Cluster Operator 設定の **STRIMZI_DEFAULT_KAFKA_INIT_IMAGE** 環境変数に指定されたコンテナーイメージ。
2. **registry.redhat.io/amq7/amq-streams-rhel8-operator:2.3.0** container image.

brokerRackInitImage の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
```

```
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    brokerRackInitImage: my-org/my-image:latest
    # ...
```

注記

コンテナーイメージのオーバーライドは、別のコンテナーレジストリーを使用する必要がある特別な状況でのみ推奨されます。たとえば、AMQ Streams によって使用されるコンテナーレジストリーにネットワークがアクセスできない場合などがこれに該当します。この場合は、AMQ Streams イメージをコピーするか、ソースからビルドする必要があります。設定したイメージが AMQ Streams イメージと互換性のない場合は、適切に機能しない可能性があります。

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.I0Itec.zkclient.ZkClient**
- **log4j.logger.org.apache.zookeeper**
- **log4j.logger.kafka**
- **log4j.logger.org.apache.kafka**
- **log4j.logger.kafka.request.logger**
- **log4j.logger.kafka.network.Processor**
- **log4j.logger.kafka.server.KafkaApis**
- **log4j.logger.kafka.network.RequestChannel$**
- **log4j.logger.kafka.controller**
- **log4j.logger.kafka.log.LogCleaner**
- **log4j.logger.state.change.logger**
- **log4j.logger.kafka.authorizer.logger**

Kafka では Apache **log4j** ロガー実装が使用されます。

logging プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は **log4j.properties** を使用して記述されま

す。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、Apache logging services を参照してください。

ここで、**inline** および **external** ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    # ...
    logging:
      type: inline
      loggers:
        kafka.root.logger.level: "INFO"
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: kafka-log4j.properties
  # ...
```

設定されていない利用可能なロガーのレベルは **OFF** に設定されています。

Cluster Operator を使用して Kafka がデプロイされた場合、Kafka のロギングレベルの変更は動的に適用されます。

外部ロギングを使用する場合は、ロギングアペンダーが変更されるとローリング更新がトリガーされます。

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.3.5. KafkaClusterSpec スキーマプロパティー

プロパティー	説明
version string	Kafka ブローカーのバージョン。デフォルトは 3.3.1 です。バージョンのアップグレードまたはダウングレードに必要なプロセスを理解するには、ユーザードキュメントを参照してください。
replicas integer	クラスター内の Pod 数。
image string	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。
listeners **GenericKafkaListener** 配列	Kafka ブローカーのリスナーを設定します。
config map	次の接頭辞のある Kafka ブローカーの config プロパティーは設定できません: listeners, advertised., broker., listener., host.name, port, inter.broker.listener.name, sasl., ssl., security., password., log.dir, zookeeper.connect, zookeeper.set.acl, zookeeper.ssl, zookeeper.clientCnxnSocket, authorizer., super.user, cruise.control.metrics.topic, cruise.control.metrics.reporter.bootstrap.servers,node.id, process.roles, controller. (次は対象外: zookeeper.connection.timeout.ms, ssl.cipher.suites, ssl.protocol, ssl.enabled.protocols, sasl.server.max.receive.size,cruise.control.metrics.topic.num.partitions, cruise.control.metrics.topic.replication.factor, cruise.control.metrics.topic.retention.ms,cruise.control.metrics.topic.auto.create.retries, cruise.control.metrics.topic.auto.create.timeout.ms,cruise.control.metrics.topic.min.insync.replicas,controller.quorum.election.backoff.max.ms, controller.quorum.election.timeout.ms, controller.quorum.fetch.timeout.ms).
storage **EphemeralStorage**、**PersistentClaimStorage**、**JbodStorage**	ストレージの設定 (ディスク)。更新はできません。タイプは、指定のオブジェクト内の **storage.type** プロパティーの値によって異なり、[ephemeral、persistent-claim、jbod] のいずれかでなければなりません。

プロパティー	説明
認可 **KafkaAuthorizationSimple**、**KafkaAuthorizationOpa**、**KafkaAuthorizationKeycloak**、**KafkaAuthorizationCustom**	Kafka ブローカーの承認設定。タイプは、指定のオブジェクト内の **authorization.type** プロパティーの値によって異なり、[simple、opa、keycloak、custom] のいずれかでなければなりません。
rack **Rack**	**broker.rack** ブローカー設定の設定
brokerRackInitImage string	**broker.rack** の初期化に使用される init コンテナーのイメージ。
livenessProbe **Probe**	Pod の liveness チェック。
readinessProbe **Probe**	Pod の readiness チェック。
jvmOptions **JvmOptions**	Pod の JVM オプション。
jmxOptions **KafkaJmxOptions**	Kafka ブローカーの JMX オプション。
resources ResourceRequirements	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
metricsConfig **JmxPrometheusExporterMetrics**	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
logging **InlineLogging**、**ExternalLogging**	Kafka のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。

プロパティー	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ーの完全リスト

OpenShift 内外の Kafka ブローカーに接続するようにリスナーを設定します。

Kafka リソースでリスナーを設定します。

リスナー設定を示す Kafka リソースの例

```
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** プロパティーを使用して **Kafka** ブローカーリスナーを設定します。リスナーは配列として定義されます。

リスナーの設定例

```
listeners:
  - name: plain
    port: 9092
    type: internal
    tls: false
```

名前およびポートは Kafka クラスター内で一意である必要があります。名前は最大 25 文字で、小文字と数字で設定されます。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** として設定されるか、外部リスナーの場合は **route**、**loadbalancer**、**nodeport**、**ingress** または **cluster-ip** として設定されます。また、カスタムアクセスメカニズムの構築に使用できる内部リスナーの一種である **cluster-ip** リスナーを設定することもできます。

internal

tls プロパティーを使用して、暗号化の有無に関わらず内部リスナーを設定できます。

internal リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
        authentication:
          type: tls
    #...
```

route

OpenShift **Routes** および HAProxy ルーターを使用して Kafka を公開するように外部リスナーを設定します。
Kafka ブローカー 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

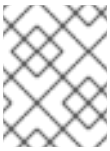
Kubernetes **Ingress** および Ingress NGINX Controller for Kubernetes を使用して、Kafka を公開するように外部リスナーを設定します。
各 Kafka ブローカー Pod に専用の **Ingress** リソースが作成されます。追加の **Ingress** リソースが作成され、Kafka ブートストラップアドレスとして提供されます。これらの **Ingress** リソースを使用すると、Kafka クライアントを 443 番ポートで Kafka に接続することができます。クライアントはデフォルトのコントローラーポートであるポート 443 に接続しますが、トラフィックは設定するポート (以下の例では **9095** にルーティングされます)。

GenericKafkaListenerConfigurationBootstrap および **GenericKafkaListenerConfigurationBroker** プロパティーを使用して、ブートストラップおよびブローカーごとのサービスによって使用されるホスト名を指定する必要があります。

Ingress リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      #...
      - name: external2
        port: 9095
        type: ingress
        tls: true
        authentication:
          type: tls
        configuration:
          bootstrap:
            host: bootstrap.myingress.com
          brokers:
```

```
        - broker: 0
          host: broker-0.myingress.com
        - broker: 1
          host: broker-1.myingress.com
        - broker: 2
          host: broker-2.myingress.com
  #...
```



注記

Ingress を使用する外部リスナーは、現在 Ingress NGINX Controller for Kubernetes でのみテストされています。

loadbalancer

Loadbalancer タイプの **Service** 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 ごとに新しいロードバランサーサービスが作成されます。追加のロードバランサーが作成され、Kafka の **ブートストラップ** アドレスとして提供されます。ロードバランサーは指定のポート番号をリッスンします。以下の例ではポート **9094** です。

loadBalancerSourceRanges プロパティーを使用して、指定された IP アドレスへのアクセスを制限する ソース範囲 を設定できます。

loadbalancer リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      - name: external3
        port: 9094
        type: loadbalancer
        tls: true
        configuration:
          loadBalancerSourceRanges:
            - 10.0.0.0/8
            - 88.208.76.87/32
    #...
```

nodeport

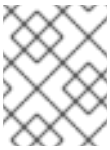
NodePort タイプの **Services** を使用して Kafka を公開するように外部リスナーを設定します。Kafka クライアントは OpenShift のノードに直接接続します。追加の **NodePort** タイプのサービスが作成され、Kafka ブートストラップアドレスとして提供されます。

Kafka ブローカー Pod にアドバタイズされたアドレスを設定する場合、AMQ Stremas では該当の Pod が稼働しているノードのアドレスが使用されます。**preferredNodePortAddressType** プロパティーを使用して、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。

cluster-ip

ブローカーごとの **ClusterIP** タイプ **Service** を使用して Kafka を公開するように内部リスナーを設定します。
リスナーは、ヘッドレスサービスとその DNS 名を使用してトラフィックを Kafka ブローカーにルーティングしません。ヘッドレスサービスの使用が不適切な場合は、このタイプのリスナーを使用して Kafka クラスターを公開できます。特定の Ingress コントローラーや OpenShift Gateway API を使用するものなど、カスタムアクセスメカニズムで使用できます。

Kafka ブローカー Pod ごとに新しい **ClusterIP** サービスが作成されます。このサービスには、ブローカーごとのポート番号を持つ Kafka **ブートストラップ** アドレスとして機能する **ClusterIP** アドレスが割り当てられます。たとえば、TCP ポート設定を使用して、Nginx Ingress Controller を介して Kafka クラスターを公開するようにリスナーを設定できます。

cluster-ip リスナーの設定例

```
#...
spec:
  kafka:
    #...
    listeners:
      - name: external-cluster-ip
        type: cluster-ip
        tls: false
        port: 9096
    #...
```

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、指定されたポート番号を使用します。**internal** および **cluster-ip** リスナーも同様です。
- **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) およびメトリクス (9404) 用に確保されたポートを使用するようにリスナーを設定することはできません。

12.2.4.4. tls

TLS プロパティーが必要です。

デフォルトでは、TLS による暗号化は有効になっていません。これを有効にするには、**tls** プロパティーを **true** に設定します。

ingress および **ingress** タイプのリスナーの場合、TLS 暗号化を有効にする必要があります。

12.2.4.5. authentication

リスナーの認証は以下のように指定できます。

- mTLS (**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。

以下の例では、下記の点を前提としています。

- 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 プロパティーの構文は、**NetworkPolicy** リソースの **from** プロパティーと同じです。

ネットワークポリシー設定の例

```
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
# ...
```

12.2.4.7. GenericKafkaListener スキーマプロパティー

プロパティー	説明
name string	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。
port integer	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。

プロパティー	説明
type 文字列 (ingress、internal、route、loadbalancer、cluster-ip、nodeport のいずれか)	リスナーのタイプ。現在サポートされているタイプは、**internal**、**route**、**loadbalancer**、**nodeport**、**ingress** です。 ● **internal** タイプは、OpenShift クラスター内でのみ 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って Kafka を公開します。 ● **ingress** タイプは、OpenShift Nginx Ingress を使用して、TLS パススルーで Kafka を公開します。 ● **cluster-ip** タイプは、ブローカーごとの **ClusterIP** サービスを使用します。
tls boolean	リスナーで TLS による暗号化を有効にします。これは必須プロパティーです。
authentication **KafkaListenerAuthenticationTls**、**KafkaListenerAuthenticationScramSha512**、**KafkaListenerAuthenticationOAuth**、**KafkaListenerAuthenticationCustom**	このリスナーの認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls、scram-sha-512、oauth、custom] のいずれかでなければなりません。
configuration **GenericKafkaListenerConfiguration**	追加のリスナー設定。
networkPolicyPeers NetworkPolicyPeer アレイ	このリスナーに接続できるピアの一覧。この一覧のピアは、論理演算子 OR を使用して組み合わせます。このフィールドが空であるか、または存在しない場合、このリスナーのすべてのコネクションが許可されます。このフィールドが存在し、1 つ以上の項目が含まれる場合、リスナーはこの一覧の少なくとも 1 つの項目と一致するトラフィックのみを許可します。詳細は、networking.k8s.io/v1 networkpolicypeer の外部ドキュメント を参照してください。

12.2.5. KafkaListenerAuthenticationTls スキーマ参照

GenericKafkaListener で使用されています。

typeプロパティーは、**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。

プロパティー	説明
type string	**tls** でなければなりません。

12.2.6. KafkaListenerAuthenticationScramSha512 スキーマ参照

GenericKafkaListener で使用されています。

type プロパティーは、**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。

プロパティー	説明
type string	**scram-sha-512** でなければなりません。

12.2.7. KafkaListenerAuthenticationOAuth スキーマ参照

GenericKafkaListener で使用されています。

type プロパティーは、**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。

プロパティー	説明
accessTokenIsJwt boolean	アクセストークンを JWT として処理するかどうかを設定します。承認サーバーが不透明なトークンを返す場合は、**false** に設定する必要があります。デフォルトは **true** です。
checkAccessTokenType boolean	アクセストークンタイプのチェックを行うかどうかを設定します。承認サーバーの JWT トークンに 'typ' 要求が含まれない場合は、**false** に設定する必要があります。デフォルトは **true** です。

プロパティー	説明
checkAudience boolean	オーディエンスのチェックを有効または無効にします。オーディエンスのチェックによって、トークンの受信者が特定されます。オーディエンスチェックが有効な場合、OAuth クライアント ID も **clientId** プロパティーで設定する必要があります。Kafka ブローカは、**aud**(オーディエンス) クレームに **clientId** がないトークンを拒否します。デフォルト値は **false** です。
checkIssuer boolean	発行元のチェックを有効または無効にします。デフォルトでは、**validIssuerUri** によって設定された値を使用して発行元がチェックされます。デフォルト値は **true** です。
clientAudience string	承認サーバーのトークンエンドポイントにリクエストを送信するときに使用するオーディエンス。ブローカー間の認証や、**clientId** と **secret** メソッドを用いた PLAIN 上の OAuth 2.0 の設定に使用されます。
clientId string	Kafka ブローカーは、OAuth クライアント ID を使用して承認サーバーに対して認証し、イントロスペクションエンドポイント URI を使用することができます。
clientScope string	承認サーバーのトークンエンドポイントにリクエストを送信するときに使用するスコープ。ブローカー間の認証や、**clientId** と **secret** メソッドを用いた PLAIN 上の OAuth 2.0 の設定に使用されます。
clientSecret **GenericSecretSource**	OAuth クライアントシークレットが含まれる OpenShift シークレットへのリンク。Kafka ブローカーは、OAuth クライアントシークレットを使用して承認サーバーに対して認証し、イントロスペクションエンドポイント URI を使用することができます。
connectTimeoutSeconds integer	承認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。
customClaimCheck string	JWT トークンに適用される JSONPath フィルタークエリー、または追加のトークン検証のイントロスペクションエンドポイントの応答に適用される JSONPath フィルタークエリー。デフォルトでは設定されません。

プロパティー	説明
disableTlsHostnameVerification boolean	TLS ホスト名の検証を有効または無効にします。デフォルト値は **false** です。
enableECDSA boolean	**enableECDSA プロパティーは非推奨となりました。**BouncyCastle 暗号プロバイダーをインストールして、ECDSA サポートを有効または無効にします。ECDSA サポートが常に有効になります。BouncyCastle ライブラリーは、AMQ Streams とパッケージ化されなくなりました。値は無視されます。
enableMetrics boolean	OAuth メトリックを有効または無効にします。デフォルト値は **false** です。
enableOauthBearer boolean	SASL_OAUTHBEARER での OAuth 認証を有効または無効にします。デフォルト値は **true** です。
enablePlain boolean	SASL_PLAIN で OAuth 認証を有効または無効にします。このメカニズムが使用される場合、再認証はサポートされません。デフォルト値は **false** です。
failFast boolean	起動時に回復可能な実行時エラーが発生する可能性があるため、Kafka ブローカープロセスの終了を有効または無効にします。デフォルト値は **true** です。
fallbackUserNameClaim string	**userNameClaim** によって指定された要求が存在しない場合に、ユーザー ID に使用するフォールバックユーザー名要求。これは、**client_credentials** 認証によってクライアント ID が別の要求のみに提供される場合に便利です。**userNameClaim** が設定されている場合のみ有効です。
fallbackUserNamePrefix string	ユーザー ID を設定するために **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。
groupsClaim string	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。
groupsClaimDelimiter	

プロパティー	説明
	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	
jwksEndpointUri	ローカルの JWT 検証に使用できる JWKS 証明書エンドポイントの URI。
string	
jwksExpirySeconds	JWKS 証明書が有効とみなされる頻度を設定します。期限切れの間隔は、**jwksRefreshSeconds** で指定される更新間隔よりも 60 秒以上長くする必要があります。デフォルトは 360 秒です。
integer	
jwksIgnoreKeyUse	JWKS エンドポイント応答の **key** 宣言の use 属性を無視するフラグ。デフォルト値は **false** です。
boolean	
jwksMinRefreshPauseSeconds	連続する 2 回の更新の間に適用される最小の一時停止期間。不明な署名鍵が検出されると、更新は即座にスケジュールされますが、この最小一時停止の期間は待機します。デフォルトは 1 秒です。
integer	
jwksRefreshSeconds	JWKS 証明書が更新される頻度を設定します。更新間隔は、**jwksExpirySeconds** で指定される期限切れの間隔よりも 60 秒以上短くする必要があります。デフォルトは 300 秒です。
integer	
maxSecondsWithoutReauthentication	再認証せずに認証されたセッションが有効な状態でいられる最大期間 (秒単位)。これにより、Apache Kafka の再認証機能が有効になり、アクセストークンの有効期限が切れるとセッションが期限切れになります。最大期間の前または最大期間の到達時にアクセストークンが期限切れになると、クライアントは再認証する必要があります。そうでないと、サーバーは接続を切断します。デフォルトでは設定されません。アクセストークンが期限切れになっても認証されたセッションは期限切れになりません。このオプションは、SASL_OAUTHBEARER 認証メカニズムにのみ適用されます (**enableOauthBearer** が **true** の場合)。
integer	
readTimeoutSeconds	承認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。

プロパティー	説明
integer	
tlsTrustedCertificates	OAuth サーバーへの TLS 接続の信頼済み証明書。
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 client**secret** に設定するか、または **username** をアカウントのユーザー名に、**password** を **$accessToken:** の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

プロパティー	説明
key	OpenShift シークレットでシークレット値が保存されるキー。
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

プロパティー	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 で使用されています。

KafkaListenerAuthenticationCustom スキーマプロパティーの完全リスト

カスタム認証を設定するには、**type** プロパティーを **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ーの/**opt/kafka/custom-authn-secrets/custom-listener-<listener_name>-<port>/<secret_name>** にマウントされます。

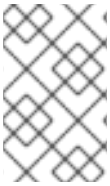
たとえば、設定例ではマウントされたシークレット (**example**) は /**opt/kafka/custom-authn-secrets/custom-listener-oauth-bespoke-9093/example** にあります。

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 プロパティーは **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。

プロパティー	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	

プロパティー	説明
type	**custom** である必要があります。
string	

12.2.11. GenericKafkaListenerConfiguration スキーマ参照

GenericKafkaListener で使用されています。

GenericKafkaListenerConfiguration スキーマプロパティーの全リスト

Kafka リスナーの設定。

12.2.11.1. brokerCertChainAndKey

brokerCertChainAndKey プロパティーは、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。プロパティーを使用して、独自の Kafka リスナー証明書を提供できます。

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 プロパティーは、**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 プロパティーは、**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 プロパティーは、**ingress** リスナーでのみ使用されます。**Ingress** クラスの設定は **class** プロパティーで行います。

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 プロパティーは、**nodeport** リスナーでのみ使用されます。

リスナーの設定で **preferredNodePortAddressType** プロパティーを使用して、ノードアドレスとしてチェックされる最初のアドレスタイプを指定します。たとえば、デプロイメントに DNS サポートがない場合や、内部 DNS または IP アドレスを介してブローカーを内部でのみ公開する場合、このプロパティーは便利です。該当タイプのアドレスが見つかった場合はそのアドレスが使用されます。アドレスタイプが見つからなかった場合、AMQ Streams は標準の優先順位でタイプの検索を続行します。

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

優先ノードポートアドレスタイプで設定された外部リスナーの例

```
listeners:
  #...
  - name: external
    port: 9094
    type: nodeport
    tls: false
    configuration:
      preferredNodePortAddressType: InternalDNS
      # ...
# ...
```

12.2.11.6. useServiceDnsDomain

useServiceDnsDomain プロパティーは、**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 「環境変数を使用した 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ー

プロパティー	説明
brokerCertChainAndKey **CertAndKeySecretSource**	このリスナーに使用される証明書とプライベートキーのペアを保持する **Secret** への参照。証明書には、任意でチェーン全体を含めることができます。このフィールドは、TLS による暗号化が有効なリスナーでのみ使用できます。

プロパティー	説明
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。
loadBalancerSourceRanges string array	クライアントがロードバランサータイプのリスナーに接続できる CIDR 形式による範囲 (例: **10.0.0.0/8**、**130.211.204.1/32**) の一覧。プラットフォームでサポートされる場合、ロードバランサー経由のトラフィックは指定された CIDR 範囲に制限されます。このフィールドは、ロードバランサータイプのサービスのみに適用され、クラウドプロバイダーがこの機能をサポートしない場合は無視されます。このフィールドは、**loadbalancer** のリスナーでのみ使用できます。
bootstrap **GenericKafkaListenerConfigurationBootstrap**	ブートストラップの設定。
brokers **GenericKafkaListenerConfigurationBroker** アレイ	ブローカーごとの設定。
ipFamilyPolicy string ([RequireDualStack、SingleStack、PreferDualStack] のいずれか)	サービスによって使用される IP Family Policy を指定します。利用可能なオプションは、**SingleStack**、**PreferDualStack**、**RequireDualStack** です。**SingleStack** は単一の IP ファミリー用です。**PreferDualStack** は、デュアルスタック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。**RequireDualStack** 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。

プロパティー	説明
ipFamilies string ([IPv6, IPv4] の 1 つ以上) array	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。
createBootstrapService boolean	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す。
class string	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。
finalizers string array	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。
maxConnectionCreationRate integer	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。
maxConnections integer	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。

プロパティー	説明
preferredNodePortAddressType string ([ExternalDNS、ExternalIP、Hostname、InternalIP、InternalDNS] のいずれか)	ノードアドレスとして使用するアドレスタイプを定義します。使用できるタイプ: **ExternalDNS**、**ExternalIP**、**InternalDNS**、**InternalIP**、**Hostname**デフォルトでは、アドレスは以下の順序で使用されます (最初に見つかったアドレスが使用されます): • **ExternalDNS** • **ExternalIP** • **InternalDNS** • **InternalIP** • **Hostname** このフィールドは、最初にチェックされる優先アドレスタイプの選択に使用されます。このアドレスタイプのアドレスが見つからない場合、他のタイプがデフォルトの順序でチェックされます。このフィールドは、**nodeport** タイプのリスナーでのみ使用できます。
useServiceDnsDomain boolean	OpenShift サービス DNS ドメインを使用するべきかどうかを設定します。**true** に設定すると、生成されるアドレスにはサービスの DNS ドメインの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。

12.2.12. CertAndKeySecretSource スキーマ参照

GenericKafkaListenerConfiguration、**KafkaClientAuthenticationTls** で使用されます。

プロパティー	説明
certificate string	Secret のファイル証明書の名前。
key string	Secret の秘密鍵の名前。
secretName string	証明書が含まれる Secret の名前。

12.2.13. GenericKafkaListenerConfigurationBootstrap スキーマ参照

GenericKafkaListenerConfiguration で使用されます。

GenericKafkaListenerConfigurationBootstrap スキーマプロパティーの全リスト

nodePort、**host**、**loadBalancerIP**、**annotations** プロパティー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** プロパティーは、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 プロパティーは、**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ブ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** のプロパティー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** プロパティーを使用します。特定の IP アドレスでロードバランサーを使用する必要がある場合は、このプロパティーを使用します。クラウドプロバイダーがこの機能に対応していない場合、**loadBalancerIP** フィールドは無視されます。

特定のロードバランサー IP アドレスリクエストのある loadbalancer タイプの外部リスナーの例

```
# ...
listeners:
  #...
  - name: external
    port: 9094
    type: loadbalancer
    tls: true
    authentication:
      type: tls
    configuration:
      bootstrap:
        loadBalancerIP: 172.29.3.10
      brokers:
      - broker: 0
        loadBalancerIP: 172.29.3.1
      - broker: 1
        loadBalancerIP: 172.29.3.2
      - broker: 2
        loadBalancerIP: 172.29.3.3
# ...
```

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ー

プロパティー	説明
alternativeNames string array	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。
host string	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。
nodePort integer	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。

プロパティー	説明
loadBalancerIP string	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。
annotations map	**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。
labels map	**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。

12.2.14. GenericKafkaListenerConfigurationBroker スキーマ参照

GenericKafkaListenerConfiguration で使用されます。

GenericKafkaListenerConfigurationBroker スキーマプロパティーの全リスト

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** プロパティー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** プロパティー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ー

プロパティー	説明
broker integer	Kafka ブローカーの ID (ブローカー識別子)。ブローカー ID は 0 から始まり、ブローカーレプリカの数に対応します。
advertisedHost string	ブローカーの **advertised.brokers** で使用されるホスト名。
advertisedPort integer	ブローカーの **advertised.brokers** で使用されるポート番号。
host string	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。
nodePort integer	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。

プロパティー	説明
loadBalancerIP string	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。
annotations map	**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。
labels map	**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。

12.2.15. EphemeralStorage スキーマ参照

JbodStorage、**KafkaClusterSpec**、**ZookeeperClusterSpec** で使用

type プロパティーは、**EphemeralStorage** タイプの使用を、**PersistentClaimStorage**から区別する識別子です。**EphemeralStorage** タイプには **ephemeral** の値が必要です。

プロパティー	説明
id integer	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。
sizeLimit string	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)。
type string	**ephemeral** でなければなりません。

12.2.16. PersistentClaimStorage スキーマ参照

JbodStorage、**KafkaClusterSpec**、**ZookeeperClusterSpec** で使用

type プロパティーは、**PersistentClaimStorage** タイプの使用を、EphemeralStorageから区別する識別子です。**PersistentClaimStorage** タイプには **persistent-claim** の値が必要です。

プロパティー	説明
type	**persistent-claim** でなければなりません。
string	
size	type=persistent-claim の場合、永続ボリューム要求のサイズを定義します (例: 1Gi)。。type=persistent-claim の場合には必須です。
string	
selector	使用する特定の永続ボリュームを指定します。このようなボリュームを選択するラベルを表す key:value ペアが含まれます。
map	
deleteClaim	クラスターのアンデプロイ時に永続ボリューム要求を削除する必要があるかどうかを指定します。
boolean	
class	動的ボリュームの割り当てに使用するストレージクラス。
string	
id	ストレージ ID 番号。これは、'jbod' タイプのストレージで定義されるストレージボリュームのみで必須です。
integer	
overrides	個々のブローカーを上書きします。**overrides** フィールドでは、異なるブローカーに異なる設定を指定できます。
PersistentClaimStorageOverride array	

12.2.17. PersistentClaimStorageOverride スキーマ参照

PersistentClaimStorage で使用

プロパティー	説明
class	このブローカーの動的ボリュームの割り当てに使用するストレージクラス。
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 プロパティーは、**JbodStorage** タイプの使用を **EphemeralStorage**と **PersistentClaimStorage** から区別する識別子です。**JbodStorage** タイプには **jbod** の値が必要です。

プロパティー	説明
type	**jbod** でなければなりません。
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一覧。
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ーの全リスト

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** プロパティーに **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。

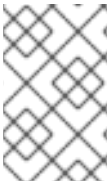
12.2.19.1. superUsers

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。

12.2.19.2. KafkaAuthorizationSimple スキーマのプロパティー

type プロパティーは、**KafkaAuthorizationSimple** タイプの使用を**KafkaAuthorizationOpa**および**KafkaAuthorizationKeycloak**、**KafkaAuthorizationCustom**と区別するための識別子です。**KafkaAuthorizationSimple** タイプには **simple** の値が必要です。

プロパティー	説明
type	**simple** でなければなりません。
string	
superUsers	スーパーユーザーの一覧。無制限のアクセス権を取得する必要のあるユーザープリンシパルの一覧が含まれなければなりません。
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ーの全リスト

Open Policy Agent の認証を使用するには、**authorization** セクションの **type** プロパティーに **opa** という値を設定し、必要に応じて OPA のプロパティーを設定します。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ー

type プロパティーは、**KafkaAuthorizationOpa** タイプの使用を **KafkaAuthorizationSimple**、**KafkaAuthorizationKeycloak**、**KafkaAuthorizationCustom** と区別するための識別子です。**KafkaAuthorizationOpa** タイプには **opa** の値が必要です。

プロパティー	説明
type	**opa** でなければなりません。

プロパティー	説明
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	
superUsers	スーパーユーザーのリスト。これは、無制限のアクセス権限を持つユーザープリンシパルのリストです。
string array	
enableMetrics	Open Policy Agent オーソライザープラグインでメトリクス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 プロパティーは、**KafkaAuthorizationKeycloak** タイプの使用を**KafkaAuthorizationSimple**、**KafkaAuthorizationOpa**、**KafkaAuthorizationCustom**と区別するための識別子です。**KafkaAuthorizationKeycloak** タイプには **keycloak** の値が必要です。

プロパティー	説明
type string	**keycloak** でなければなりません。
clientId string	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。
tokenEndpointUri string	承認サーバートークンエンドポイント URI。
tlsTrustedCertificates **CertSecretSource** array	OAuth サーバーへの TLS 接続の信頼済み証明書。
disableTlsHostnameVerification boolean	TLS ホスト名の検証を有効または無効にします。デフォルト値は **false** です。
delegateToKafkaAcls boolean	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。
grantsRefreshPeriodSeconds integer	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。
grantsRefreshPoolSize integer	アクティブなセッションの付与 (Grants) の更新に使用するスレッドの数。スレッドが多いほど並列処理多くなるため、ジョブがより早く完了します。ただし、使用するスレッドが多いほど、承認サーバーの負荷が大きくなります。デフォルト値は 5 です。
superUsers string array	スーパーユーザーの一覧。無制限のアクセス権を取得する必要のあるユーザープリンシパルの一覧が含まれなければなりません。
connectTimeoutSeconds integer	承認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。
readTimeoutSeconds integer	承認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。

プロパティー	説明
enableMetrics	OAuth メトリックを有効または無効にします。デフォルト値は **false** です。
boolean	

12.2.22. KafkaAuthorizationCustom スキーマリファレンス

KafkaClusterSpec で使用

KafkaAuthorizationCustom スキーマプロパティーの全リスト

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** プロパティーに値 **custom** を設定し、以下のプロパティーを設定します。

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 設定プロパティーを使用して **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 を使って、カスタムオーサライザーを初期化するための設定を追加することができます。

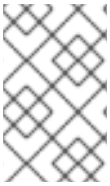
Kafka.spec でのカスタム認証設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    authorization:
      type: custom
      authorizerClass: io.mycompany.CustomAuthorizer
```

```
    superUsers:
      - CN=client_1
      - user_2
      - CN=client_3
  # ...
  config:
    authorization.custom.property1=value1
    authorization.custom.property2=value2
  # ...
```

Kafka カスタムリソースの設定に加えて、カスタムオーソライザークラスとその依存関係を含む JAR ファイルが Kafka ブローカーのクラスパス上で利用可能である必要があります。

AMQ Streams の Maven ビルドプロセスでは、**docker-images/kafka/kafka-thirdparty-libs** ディレクトリーの下にある **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ー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ー

type プロパティーは、**KafkaAuthorizationCustom** タイプの使用を **KafkaAuthorizationSimple**、**KafkaAuthorizationOpa**、**KafkaAuthorizationKeycloak**と区別する識別子です。タイプ **KafkaAuthorizationCustom** の値が **custom** である必要があります。

プロパティー	説明
type	**custom** である必要があります。

プロパティー	説明
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限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

使用先: **KafkaBridgeSpec**、**KafkaClusterSpec**、**KafkaConnectSpec**、**KafkaMirrorMaker2Spec**

Rack スキーマプロパティーの全リスト

rack オプションは、ラックの認識を設定します。**ラック**は、アベイラビリティーゾーン、データセンター、またはデータセンターの実際のラックを表すことができます。rackの設定は、**topologyKey** で行います。**topologyKey** は、OpenShift ノード上のラベルを識別するもので、その値にはトポロジーの名前が含まれています。このようなラベルの例としては、**topology.kubernetes.io/zone**(古い OpenShift バージョンでは **failure-domain.beta.kubernetes.io/zone**) があり、これには OpenShift ノードが実行されているアベイラビリティゾーンの名前が含まれています。Kafka クラスターが実行する**ラック**を認識するように設定し、パーティションレプリカを異なるラックに分散したり、最も近いレプリカからのメッセージの消費したりするなどの追加機能を有効にできます。

OpenShift ノードラベルの詳細は、Well-Known Labels, Annotations and Taints を参照してください。ノードがデプロイされたゾーンやラックを表すノードラベルについては、OpenShift 管理者に相談します。

12.2.23.1. ラック間でのパーティションレプリカの分散

ラックアウェアネスを設定すると、AMQ Streams は各 Kafka ブローカーの **broker.rack** 設定を行います。**broker.rack** の 設定では、各ブローカにラック ID を割り当てます。**broker.rack** を設定すると、Kafka ブローカーはパーティションレプリカをできるだけ多くの異なるラックに分散して配置します。レプリカが複数のラックに分散されている場合、複数のレプリカが同時に失敗する可能性は、同じラックにある場合よりも低くなります。レプリカを分散すると回復性が向上し、可用性と信頼性にとっても重要です。Kafka でラックアウェアネスを有効にするには、以下の例のように、**Kafka** のカスタムリソースの **.spec.kafka** セクションに **rack** オプションを追加します。

Kafka の rack 設定例

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

```
spec:
  kafka:
    # ...
    rack:
      topologyKey: topology.kubernetes.io/zone
    # ...
```

注記

Pod が削除または再起動すると、ブローカーが実行されている**ラック**は、変更されることがあります。その結果、異なるラックで実行しているレプリカが、同じラックを共有する可能性があります。**RackAwareGoal** で Cruise Control と **KafkaRebalance** リソースを使用して、レプリカが異なるラックに分散していることを確認します。

Kafka カスタムリソースでラックアウェアネスが有効になっている場合、AMQ Streams は自動的に OpenShift の **preferredDuringSchedulingIgnoredDuringExecution** アフィニティールールを追加して、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

ラックアウェアネスをコンシューマーで使用して、最も近いレプリカからデータを取得することもできます。これは、Kafka クラスターが複数のデータセンターにまたがる場合に、ネットワークの負荷を軽減するのに役立ちます。また、パブリッククラウドで Kafka を実行する場合にコストを削減することもできます。ただし、レイテンシーが増加する可能性があります。

最も近いレプリカから利用するためには、Kafka クラスターでラックアウェアが設定されており、**RackAwareReplicaSelector** が有効になっている必要があります。レプリカセレクタープラグインは、クライアントが最も近いレプリカから消費できるようにするロジックを提供します。デフォルトの実装では、**LeaderSelector** を使って、常にクライアントのリーダーレプリカを選択します。**replica.selector.class** に **RackAwareReplicaSelector** を指定すると、デフォルトの実装から切り替わります。

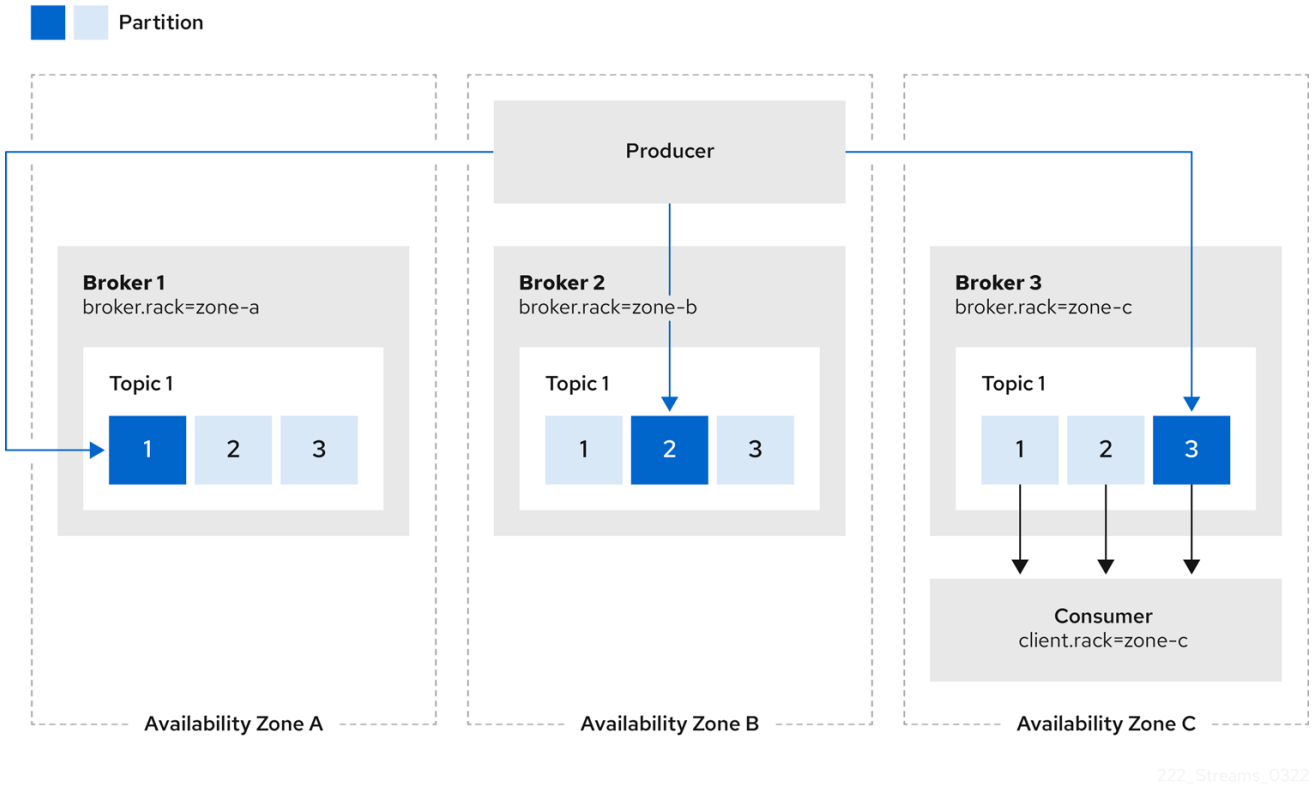
レプリカ対応セレクターを有効にした rack 設定例

```
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

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ーゾーンのレプリカから消費するクライアントの例



コネクター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、および Kafka Bridge を設定することもできます。**KafkaConnect**、**KafkaMirrorMaker2**、および **KafkaBridge** 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ー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 を展開する場合、**KafkaConnect** カスタムリソースの **rack** セクションを使用して、**client.rack** オプションを自動的に設定することができます。

Kafka Connect の rack 設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
# ...
spec:
  # ...
  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
  # ...
```

AMQ Streams を使用して Kafka Bridge をデプロイする場合、**KafkaBridge** 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 Bridge の rack 設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
# ...
spec:
  # ...
  rack:
    topologyKey: topology.kubernetes.io/zone
  # ...
```

12.2.23.3. Rack スキーマのプロパティー

プロパティー	説明
topologyKey string	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。

12.2.24. Probe スキーマ参照

CruiseControlSpec、**EntityTopicOperatorSpec**、**EntityUserOperatorSpec**、**KafkaBridgeSpec**、**KafkaClusterSpec**、**KafkaConnectSpec**、**KafkaExporterSpec**、**KafkaMirrorMaker2Spec**、**KafkaMirrorMakerSpec**、**TlsSidecar**、**ZookeeperClusterSpec** で使用されています。

プロパティー	説明
failureThreshold integer	正常に実行された後に失敗とみなされるプローブの連続失敗回数の最小値。デフォルトは 3 です。最小値は 1 です。
initialDelaySeconds integer	最初に健全性をチェックするまでの初期の遅延。デフォルトは 15 秒です。最小値は 0 です。

プロパティー	説明
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** で使用されています。

プロパティー	説明
-XX	JVM への -XX オプションのマップ。
map	
-Xms	JVM への -Xms オプション。
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ーのマップ。
SystemProperty array	

12.2.26. SystemProperty スキーマ参照

JvmOptions で使用

プロパティー	説明
name	システムプロパティー名。
string	
value	システムプロパティーの値。
string	

12.2.27. KafkaJmxOptions スキーマ参照

KafkaClusterSpec、**KafkaConnectSpec**、**KafkaMirrorMaker2Spec**、**ZookeeperClusterSpec** で使用されます。

KafkaJmxOptions スキーマプロパティーの全リスト

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 メトリクスを取得します。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** プロパティーを使用します。パスワードで保護すると、未許可の Pod によるポートへの不正アクセスを防ぐことができます。

その後、コンポーネントに関するメトリクスを取得できます。

たとえば、Kafka ブローカーごとに、クライアントからのバイト/秒の使用度データや、ブローカーのネットワークの要求レートを取得することができます。

JMX ポートのセキュリティ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 メトリクスを取得できます。

たとえば、ブローカー 0 から JMX メトリクスを取得するには、以下を指定します。

> "**CLUSTER-NAME**-kafka-0.**CLUSTER-NAME**-kafka-brokers"

CLUSTER-NAME-kafka-0 はブローカー 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、メトリクスを取得できますが、この場合はどの Pod も JMX ポートから読み取ることができます。

Kafka ブローカーと ZooKeeper ノードのオープンポート JMX 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クスの詳細は、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ー

プロパティー	説明
authentication	JMX ポートに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[password] の 1 つでなければなりません。
KafkaJmxAuthenticationPassword	

12.2.28. KafkaJmxAuthenticationPassword スキーマ参照

KafkaJmxOptions で使用

type プロパティーは、**KafkaJmxAuthenticationPassword** タイプの使用と、今後追加される可能性のある他のサブタイプとを区別するための識別情報です。**KafkaJmxAuthenticationPassword** タイプには **password** の値が必要です。

プロパティー	説明
type	**password** でなければなりません。
string	

12.2.29. JmxPrometheusExporterMetrics スキーマ参照

CruiseControlSpec、**KafkaClusterSpec**、**KafkaConnectSpec**、**KafkaMirrorMaker2Spec**、**KafkaMirrorMakerSpec**、**ZookeeperClusterSpec** で使用されています。

type プロパティーは、**JmxPrometheusExporterMetrics** タイプの使用と、将来追加される可能性のある他のサブタイプとを区別する識別子です。**JmxPrometheusExporterMetrics** タイプの値 **jmxPrometheusExporter** を持つ必要があります。

プロパティー	説明
type	**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 エントリー。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 で使用されています。

プロパティー	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** で使用されています。

type プロパティーは、**InlineLogging** タイプの使用と、**ExternalLogging**.を区別するための識別子です。**InlineLogging** タイプには **inline** の値が必要です。

プロパティー	説明
type	**inline** でなければなりません。
string	
loggers	ロガー名からロガーレベルへのマップ。
map	

12.2.32. ExternalLogging スキーマ参照

CruiseControlSpec、**EntityTopicOperatorSpec**、**EntityUserOperatorSpec**、**KafkaBridgeSpec**、**KafkaClusterSpec**、**KafkaConnectSpec**、**KafkaMirrorMaker2Spec**、**KafkaMirrorMakerSpec**、**ZookeeperClusterSpec** で使用されています。

type プロパティーは、**ExternalLogging** タイプの使用を、**InlineLogging**と区別するための識別子です。**ExternalLogging** タイプには **external** の値が必要です。

プロパティー	説明
type	**external** でなければなりません。
string	
valueFrom	ロギング設定が保存される **ConfigMap** エントリーです。
ExternalConfigurationReference	

12.2.33. KafkaClusterTemplate スキーマ参照

KafkaClusterSpec で使用

プロパティー	説明
statefulset	Kafka **StatefulSet** のテンプレート。
StatefulSetTemplate	
Pod	Kafka **Pod** のテンプレート。
PodTemplate	
bootstrapService	Kafka ブートストラップ **Service** のテンプレート。
InternalServiceTemplate	

プロパティー	説明
brokersService **InternalServiceTemplate**	Kafka ブローカー **Service** のテンプレート。
externalBootstrapService **ResourceTemplate**	Kafka 外部ブートストラップ **Service** のテンプレート。
perPodService **ResourceTemplate**	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。
externalBootstrapRoute **ResourceTemplate**	Kafka 外部ブートストラップ **Route** のテンプレート。
perPodRoute **ResourceTemplate**	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。
externalBootstrapIngress **ResourceTemplate**	Kafka 外部ブートストラップ **Ingress** のテンプレート。
perPodIngress **ResourceTemplate**	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。
persistentVolumeClaim **ResourceTemplate**	すべての Kafka **PersistentVolumeClaims** のテンプレート。
podDisruptionBudget **PodDisruptionBudgetTemplate**	Kafka **PodDisruptionBudget** のテンプレート。
kafkaContainer **ContainerTemplate**	Kafka ブローカーコンテナーのテンプレート。
initContainer **ContainerTemplate**	Kafka init コンテナーのテンプレート。

プロパティー	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

プロパティー	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	
podManagementPolicy	この StatefulSet に使用される PodManagementPolicy。有効な値は **Parallel** および **OrderedReady** です。デフォルトは **Parallel** です。
string ([OrderedReady、Parallel] のいずれか)	

12.2.35. MetadataTemplate スキーマ参照

以下で使用: **BuildConfigTemplate**, **DeploymentTemplate**, **InternalServiceTemplate**, **PodDisruptionBudgetTemplate**, **PodTemplate**, **ResourceTemplate**, **StatefulSetTemplate**

MetadataTemplate スキーマプロパティーの全リスト

Labels および **Annotations** は、リソースの識別および整理に使用され、**metadata** プロパティーで設定されます。

以下に例を示します。

```
# ...
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ー

プロパティー	説明
labels	リソーステンプレートに追加されたラベル。**StatefulSets**、**Deployments**、**Pods**、**Services** などの異なるリソースに適用できます。
map	
annotations	リソーステンプレートに追加されたアノテーション。**StatefulSets**、**Deployments**、**Pods**、**Services** などの異なるリソースに適用できます。
map	

12.2.36. PodTemplate スキーマ参照

CruiseControlTemplate、**EntityOperatorTemplate**、**KafkaBridgeTemplate**、**KafkaClusterTemplate**、**KafkaConnectTemplate**、**KafkaExporterTemplate**、**KafkaMirrorMakerTemplate**、**ZookeeperClusterTemplate** で使用されます。

PodTemplate スキーマプロパティーの全リスト

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 プロパティーを使用して、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ー

プロパティー	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	
imagePullSecrets	この Pod で使用されるイメージのプルに使用する同じ namespace のシークレットへの参照の一覧です。Cluster Operator の環境変数 **STRIMZI_IMAGE_PULL_SECRETS** と **imagePullSecrets** オプションが指定されている場合、**imagePullSecrets** 変数のみが使用され、**STRIMZI_IMAGE_PULL_SECRETS** 変数は無視されます。詳細は、core/v1 localobjectreference の外部ドキュメント を参照してください。
LocalObjectReference アレイ	
securityContext	Pod レベルのセキュリティー属性と共通のコンテナー設定を設定します。詳細は、core/v1 podsecuritycontext の外部ドキュメント を参照してください。
PodSecurityContext	

プロパティー	説明
terminationGracePeriodSeconds integer	猶予期間とは、Pod で実行されているプロセスに終了シグナルが送信されてから、kill シグナルでプロセスを強制的に終了するまでの期間 (秒単位) です。この値は、プロセスの予想されるクリーンアップ時間よりも長く設定します。値は負の値ではない整数にする必要があります。値をゼロにすると、即座に削除されます。非常に大型な Kafka クラスターの場合は、正常終了期間を延長し、Kafka ブローカーの終了前に作業を別のブローカーに転送する時間を十分確保する必要があることがあります。デフォルトは 30 秒です。
affinity 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。
tolerations toleration アレイ	Pod の許容 (Toleration)。詳細は、core/v1 toleration の外部ドキュメント を参照してください。
priorityClassName string	優先順位を Pod に割り当てるために使用される優先順位クラス (Priority Class) の名前。Priority Class (優先順位クラス) の詳細は、Pod Priority and Preemption を参照してください。
schedulerName string	この **Pod** のディスパッチに使用されるスケジューラーの名前。指定されていない場合、デフォルトのスケジューラーが使用されます。
hostAliases HostAlias アレイ	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 hostalias を参照してください。
tmpDirSizeLimit string	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。
enableServiceLinks boolean	サービスについての情報を Pod の環境変数に注入するかどうかを示します。
topologySpreadConstraints TopologySpreadConstraint 配列	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。

12.2.37. InternalServiceTemplate のスキーマ参照

CruiseControlTemplate、**KafkaBridgeTemplate**、**KafkaClusterTemplate**、**KafkaConnectTemplate**、**ZookeeperClusterTemplate** で使用されています。

プロパティー	説明
metadata	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**、**RequireDualStack** です。**SingleStack** は単一の IP ファミリー用です。**PreferDualStack** は、デュアルスタック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。**RequireDualStack** 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 `ipFamilyPolicy** の設定に基づいてデフォルト値を選択します。OpenShift 1.20 以降で利用できます。
string ([IPv6, IPv4] の 1 つ以上) array	

12.2.38. ResourceTemplate スキーマ参照

CruiseControlTemplate, **EntityOperatorTemplate**, **KafkaBridgeTemplate**, **KafkaClusterTemplate**, **KafkaConnectTemplate**, **KafkaExporterTemplate**, **KafkaMirrorMakerTemplate**, **KafkaUserTemplate**, **ZookeeperClusterTemplate** で使用されています。

プロパティー	説明
metadata	リソースに適用済みのメタデータ。
MetadataTemplate	

12.2.39. PodDisruptionBudgetTemplate スキーマ参照

CruiseControlTemplate、**KafkaBridgeTemplate**、**KafkaClusterTemplate**、**KafkaConnectTemplate**、**KafkaMirrorMakerTemplate**、**ZookeeperClusterTemplate** で使用されます。

PodDisruptionBudgetTemplate スキーマプロパティーの全リスト

AMQ Streams は、新しい **StatefulSet** や **Deployment** ごとに **PodDisruptionBudget** を作成します。デフォルトでは、Pod の Disruption Budget (停止状態の予算) は単一の Pod を指定時に利用不可能にすることのみ許可します。**maxUnavailable** プロパティーのデフォルト値を変更して、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

プロパティー	説明
metadata	**PodDisruptionBudgetTemplate** リソースに適用するメタデータ。
MetadataTemplate	
maxUnavailable	自動 Pod エビクションを許可するための利用不可能な Pod の最大数。Pod エビクションは、**maxUnavailable** の Pod 数またはそれより少ない Pod 数がエビクション後に利用できない場合に許可されます。この値を 0 に設定するとすべての自発的なエビクションを阻止するため、Pod を手動でエビクトする必要があります。デフォルトは 1 です。
integer	

12.2.40. ContainerTemplate スキーマ参照

CruiseControlTemplate、**EntityOperatorTemplate**、**KafkaBridgeTemplate**、**KafkaClusterTemplate**、**KafkaConnectTemplate**、**KafkaExporterTemplate**、**KafkaMirrorMakerTemplate**、**ZookeeperClusterTemplate** で使用されます。

ContainerTemplate スキーマプロパティーの全リスト

コンテナーのカスタムのセキュリティーコンテキストおよび環境変数を設定できます。

環境変数は、**env** プロパティーで **name** および **value** フィールドのあるオブジェクトのリストとして定義されます。以下の例は、Kafka ブローカーコンテナーに設定された 2 つのカスタム環境変数と 1 つのセキュリティーコンテキストを示しています。

```
# ...
template:
```

```
  kafkaContainer:
    env:
    - name: EXAMPLE_ENV_1
      value: example.env.one
    - name: EXAMPLE_ENV_2
      value: example.env.two
    securityContext:
      runAsUser: 2000
# ...
```

KAFKA_ で始まる環境変数は AMQ Streams 内部となるため、使用しないようにしてください。AMQ Streams によってすでに使用されているカスタム環境変数を設定すると、その環境変数は無視され、警告がログに記録されます。

12.2.40.1. ContainerTemplate スキーマのプロパティー

プロパティー	説明
env	コンテナーに適用する必要のある環境変数。
ContainerEnvVar array	
securityContext	コンテナーのセキュリティーコンテキスト。詳細は、core/v1 securitycontext の外部ドキュメント を参照してください。
SecurityContext	

12.2.41. ContainerEnvVar スキーマ参照

ContainerTemplate で使用

プロパティー	説明
name	環境変数のキー。
string	
value	環境変数の値。
string	

12.2.42. ZookeeperClusterSpec スキーマ参照

KafkaSpec で使用

ZookeeperClusterSpec スキーマプロパティーの全リスト

ZooKeeper クラスターを設定します。

12.2.42.1. config

config プロパティーを使用して、ZooKeeper のオプションをキーとして設定します。

標準の Apache ZooKeeper 設定が提供されることがあり、AMQ Streams によって直接管理されないプロパティーに限定されます。

以下に関連する設定オプションは設定できません。

- セキュリティー (暗号化、認証、および承認)
- リスナーの設定
- データディレクトリーの設定
- ZooKeeper クラスターの設定

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** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。サポートされるその他すべてのオプションは ZooKeeper に渡されます。

禁止されているオプションには例外があります。TLS バージョンの特定の **暗号スイート** を使用するクライアント接続に、許可された **ssl** プロパティーを設定 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 プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は **log4j.properties** を使用して記述されます。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、Apache logging services を参照してください。

ここで、**inline** および **external** ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  zookeeper:
    # ...
    logging:
      type: inline
      loggers:
        zookeeper.root.logger: "INFO"
    # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  zookeeper:
    # ...
    logging:
      type: external
```

```
      valueFrom:
        configMapKeyRef:
          name: customConfigMap
          key: zookeeper-log4j.properties
# ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.42.3. ZookeeperClusterSpec スキーマプロパティー

プロパティー	説明
replicas	クラスター内の Pod 数。
integer	
image	Pod の Docker イメージ。
string	
storage	ストレージの設定 (ディスク)。更新はできません。タイプは、指定のオブジェクト内の **storage.type** プロパティーの値によって異なり、[ephemeral、persistent-claim] のいずれかでなければなりません。
EphemeralStorage、**PersistentClaimStorage**	
config	ZooKeeper ブローカーの設定。次の接頭辞のあるプロパティーは設定できません: server.、 dataDir、dataLogDir、clientPort、authProvider、quorum.auth、requireClientAuthScheme、snapshot.trust.empty、standaloneEnabled、reconfigEnabled、4lw.commands.whitelist、secureClientPort、ssl、serverCnxnFactory、sslQuorum (次の例外を除く: ssl.protocol、ssl.quorum.protocol、ssl.enabledProtocols、ssl.quorum.enabledProtocols、ssl.ciphersuites、ssl.quorum.ciphersuites、ssl.hostnameVerification、ssl.quorum.hostnameVerification)
map	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	
jvmOptions	Pod の JVM オプション。

プロパティー	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	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
JmxPrometheusExporterMetrics	
logging	ZooKeeper のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
template	ZooKeeper クラスターリソースのテンプレート。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

プロパティー	説明
statefulset	ZooKeeper **StatefulSet** のテンプレート。
StatefulSetTemplate	
Pod	ZooKeeper **Pod** のテンプレート。
PodTemplate	
clientService	ZooKeeper クライアント **Service** のテンプレート。
InternalServiceTemplate	
nodesService	ZooKeeper ノード **Service** のテンプレート。

プロパティー	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 コンテナーのテンプレート。
ContainerTemplate	
serviceAccount	ZooKeeper サービスアカウントのテンプレート。
ResourceTemplate	
jmxSecret	Zookeeper Cluster JMX 認証の Secret のテンプレート。
ResourceTemplate	
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

プロパティー	説明
topicOperator	Topic Operator の設定。
EntityTopicOperatorSpec	
userOperator	User Operator の設定。
EntityUserOperatorSpec	
tlsSidecar	TLS サイドカーの設定。
TlsSidecar	

プロパティー	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ーの全リスト

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** プロパティーを使用して、ロガーとロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は **log4j2.properties** を使用して記述されます。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、Apache logging services を参照してください。

ここで、**inline** および **external** ロギングの例を示します。

inline ロギング

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

```
    reconciliationIntervalSeconds: 60
    logging:
      type: inline
      loggers:
        rootLogger.level: INFO
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    topicOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: external
        valueFrom:
          configMapKeyRef:
            name: customConfigMap
            key: topic-operator-log4j2.properties
  # ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.45.2. EntityTopicOperatorSpec スキーマプロパティー

プロパティー	説明
watchedNamespace	Topic Operator が監視する必要のある namespace。
string	
image	Topic Operator に使用するイメージ。
string	
reconciliationIntervalSeconds	定期的な調整の間隔。
integer	

プロパティー	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** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
jvmOptions	Pod の JVM オプション。
JvmOptions	

12.2.46. EntityUserOperatorSpec スキーマ参照

EntityOperatorSpec で使用

EntityUserOperatorSpec スキーマプロパティーの全リスト

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** プロパティーを使用して、ロガーとロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は **log4j2.properties** を使用して記述されます。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、Apache logging services を参照してください。

ここで、**inline** および **external** ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    # ...
  zookeeper:
    # ...
  entityOperator:
    # ...
    userOperator:
      watchedNamespace: my-topic-namespace
      reconciliationIntervalSeconds: 60
      logging:
        type: inline
        loggers:
          rootLogger.level: INFO
  # ...
```

外部ロギング

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

```
    reconciliationIntervalSeconds: 60
    logging:
      type: external
      valueFrom:
        configMapKeyRef:
          name: customConfigMap
          key: user-operator-log4j2.properties
  # ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.46.2. EntityUserOperatorSpec スキーマプロパティー

プロパティー	説明
watchedNamespace	User Operator が監視する必要のある namespace。
string	
image	User Operator に使用するイメージ。
string	
reconciliationIntervalSeconds	定期的な調整の間隔。
integer	
zookeeperSessionTimeoutSeconds	**zookeeperSessionTimeoutSeconds プロパティー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、このプロパティー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	

プロパティー	説明
resources ResourceRequirements	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
logging **InlineLogging**、**ExternalLogging**	ロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
jvmOptions **JvmOptions**	Pod の JVM オプション。

12.2.47. TlsSidecar スキーマ参照

CruiseControlSpec、**EntityOperatorSpec** で使用されています。

TlsSidecar スキーマプロパティーの全リスト

Pod で実行されるコンテナーである TLS サイドカーを設定しますが、サポートの目的で提供されます。。AMQ Streams では、TLS サイドカーは TLS を使用して、コンポーネントと ZooKeeper との間の通信を暗号化および復号化します。

TLS サイドカーは Entity Operator で使用されます。

TLS サイドカーは、**Kafka.spec.entityOperator** の **tlsSidecar** プロパティー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 プロパティーは、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 プロパティーは、使用されるコンテナーイメージを設定します。

readinessProbe プロパティーと **livenessProbe** プロパティーは、TLS サイドカーのhealthcheck プローブを設定します。

logLevel プロパティーは、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ー

プロパティー	説明
image	コンテナーの 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)	

プロパティー	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

プロパティー	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 コンテナーのテンプレート。
ContainerTemplate	
userOperatorContainer	Entity User Operator コンテナーのテンプレート。
ContainerTemplate	
tlsSidecarContainer	Entity Operator TLS サイドカーコンテナーのテンプレート。
ContainerTemplate	
serviceAccount	Entity Operator サービスアカウントのテンプレート。
ResourceTemplate	

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。

プロパティー	説明
generateCertificateAuthority boolean	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。
generateSecretOwnerReference boolean	**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。
validityDays integer	生成される証明書の有効日数。デフォルトは 365 です。
renewalDays integer	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。
certificateExpirationPolicy string ([replace-key、renew-certificate] のいずれか)	**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。

12.2.50. CruiseControlSpec スキーマ参照

KafkaSpec で使用

CruiseControlSpec スキーマプロパティーの完全なリスト

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 プロパティーを使用して、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ーに限り、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** プロパティー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** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。サポートされ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** プロパティーを設定 することができます。**webserver** プロパティーを設定して、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** プロパティーを使用して、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 クラスター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、現在のクラスター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 を使用する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 Basic 認証および SSL でセキュリティー保護され、Kafka ブローカーの停止などの破壊的な Cruise Control 操作からクラスターを保護します。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**ことをお勧めします。

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

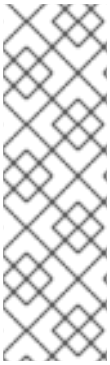
Cruise Control は容量制限を使用して、リソース分散の最適化ゴールが破損しているかどうかを判断します。このタイプには 4 つのゴールがあります。

- **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** プロパティーに指定します。これらはデフォルトで有効になっており、デフォルト値を変更できます。容量制限は、以下のブローカーリソースに設定できます。

- **cpu** - ミリコアまたは CPU コアの CPU リソース (デフォルト: 1)
- **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ビバイト (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。CPU ゴールを使用するときに正確なリバランスの提案を保証するために、**Kafka.spec.kafka.resources** で CPU リクエストを CPU 制限と同じに設定できます。これにより、すべての CPU リソースが事前に予約され、常に利用できます。この設定を使用すると、CPU ゴールに基づきリバランスプロポーザルを準備する際に、Cruise Control は CPU 使用率を適切に評価できます。**Kafka.spec.kafka.resources** の CPU 制限と同じ CPU 要求を設定できない場合は、CPU 容量を手動で設定して同じ精度にすることができます。

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:
      cpu: "2"
      inboundNetwork: 10000KiB/s
      outboundNetwork: 10000KiB/s
    # ...
```

12.2.50.5. 容量の上書き

ブローカーは、異種ネットワークまたは CPU リソースを持つノードで実行されている可能性があります。その場合は、ブローカーごとにネットワーク容量と CPU 制限を設定する **overrides** を指定します。オーバーライドにより、ブローカー間の正確な再調整が保証されます。次のブローカリソースに対してオーバーライド容量制限を設定できます。

- **cpu** - ミリコアまたは CPU コアの CPU リソース (デフォルト: 1)
- **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)

バイバイト単位を使用した Cruise Control 容量オーバーライド設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  cruiseControl:
    # ...
    brokerCapacity:
      cpu: "1"
      inboundNetwork: 10000KiB/s
```

```
    outboundNetwork: 10000KiB/s
    overrides:
    - brokers: [0]
      cpu: "2.755"
      inboundNetwork: 20000KiB/s
      outboundNetwork: 20000KiB/s
    - brokers: [1, 2]
      cpu: 3000m
      inboundNetwork: 30000KiB/s
      outboundNetwork: 30000KiB/s
```

詳細は、BrokerCapacity スキーマリファレンス を参照してください。

12.2.50.6. 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 プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は **log4j.properties** を使用して記述されます。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ここで、**inline** および **external** ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
  cruiseControl:
    # ...
    logging:
      type: inline
      loggers:
        rootLogger.level: "INFO"
    # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
# ...
spec:
  cruiseControl:
```

```
# ...
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: customConfigMap
      key: cruise-control-log4j.properties
# ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.50.7. CruiseControlSpec スキーマプロパティー

プロパティー	説明
image	Pod の Docker イメージ。
string	
tlsSidecar	**tlsSidecar プロパティーは廃止されました。**TLS サイドカーの設定。
TlsSidecar	
resources	Cruise Control コンテナー用に予約された CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
livenessProbe	Cruise Control コンテナーの Pod liveness チェック
Probe	
readinessProbe	Cruise Control コンテナーの Pod readiness チェック
Probe	
jvmOptions	Cruise Control コンテナーの 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** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
InlineLogging、**ExternalLogging**	
template	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。

プロパティー	説明
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ーは設定できません: 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	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
JmxPrometheusExporterMetrics	

12.2.51. CruiseControlTemplate スキーマ参照

CruiseControlSpec で使用されます。

プロパティー	説明
deployment	Cruise Control **Deployment** のテンプレート。
ResourceTemplate	
Pod	Cruise Control **Pods** のテンプレート。
PodTemplate	
apiService	Cruise Control API **Service** のテンプレート。
InternalServiceTemplate	

プロパティー	説明
podDisruptionBudget	Cruise Control **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	
cruiseControlContainer	Cruise Control コンテナーのテンプレート。
ContainerTemplate	
tlsSidecarContainer	**tlsSidecarContainer プロパティーは廃止されました。** Cruise Control TLS サイドカーコンテナーのテンプレート。
ContainerTemplate	
serviceAccount	Cruise Control サービスアカウントのテンプレート。
ResourceTemplate	

12.2.52. BrokerCapacity スキーマー参照

CruiseControlSpec で使用されます。

プロパティー	説明
disk	**disk プロパティー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)、それらのビバイト (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	
cpuUtilization	**cpuUtilization プロパティー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	
cpu	コアまたはミリコア単位の CPU リソースのブローカー容量。たとえば、1、1.500、1500m などです。有効な CPU リソースユニットの詳細については、https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu を参照してください。
string	

プロパティー	説明
inboundNetwork string	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ビバイト (Ki、Mi、Gi)/秒を使用します。たとえば、10000KiB/s です。
outboundNetwork string	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ビバイト (Ki、Mi、Gi)/秒を使用します。たとえば、10000KiB/s です。
overrides **BrokerCapacityOverride** アレイ	個々のブローカーを上書きします。**overrides** プロパティーを使用すると、ブローカーごとに異なる容量設定を指定できます。

12.2.53. BrokerCapacityOverride スキーマリファレンス

使用先: **BrokerCapacity**

プロパティー	説明
brokers 整数配列	Kafka ブローカー (ブローカー識別子) のリスト。
cpu string	コアまたはミリコア単位の CPU リソースのブローカー容量。たとえば、1、1.500、1500m などです。有効な CPU リソースユニットの詳細については、https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu を参照してください。
inboundNetwork string	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ビバイト (Ki、Mi、Gi)/秒を使用します。たとえば、10000KiB/s です。
outboundNetwork string	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ビバイト (Ki、Mi、Gi)/秒を使用します。たとえば、10000KiB/s です。

12.2.54. KafkaExporterSpec スキーマ参照

KafkaSpec で使用

プロパティー	説明
image	Pod の Docker イメージ。
string	
groupRegex	収集するコンシューマーグループを指定する正規表現。デフォルト値は **.*** です。
string	
topicRegex	収集するトピックを指定する正規表現。デフォルト値は **.*** です。
string	
resources	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
ResourceRequirements	
logging	指定の重大度以上のログメッセージのみ。有効なレベル: [**info**、 debug、**trace**]デフォルトのログレベルは **info** です。
string	
enableSaramaLogging	Kafka Exporter によって使用される Go クライアントライブラリーである Sarama ロギングを有効にします。
boolean	
template	デプロイメントテンプレートおよび Pod のカスタマイズ。
KafkaExporterTemplate	
livenessProbe	Pod の liveness チェック。
Probe	
readinessProbe	Pod の readiness チェック。
Probe	

12.2.55. KafkaExporterTemplate スキーマ参照

KafkaExporterSpec で使用

プロパティー	説明
deployment **ResourceTemplate**	Kafka Exporter **Deployment** のテンプレート。
Pod **PodTemplate**	Kafka Exporter **Pod** のテンプレート。
サービス **ResourceTemplate**	**service プロパティー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。
container **ContainerTemplate**	Kafka Exporter コンテナーのテンプレート。
serviceAccount **ResourceTemplate**	Kafka Exporter サービスアカウントのテンプレート。

12.2.56. KafkaStatus スキーマ参照

Kafka で使用

プロパティー	説明
conditions **Condition** array	ステータス条件の一覧。
observedGeneration integer	最後に Operator によって調整された CRD の生成。
listeners **ListenerStatus** array	内部リスナーおよび外部リスナーのアドレス。
clusterId string	Kafka クラスター ID。

12.2.57. Condition スキーマ参照

使用先:
KafkaBridgeStatus、**KafkaConnectorStatus**、**KafkaConnectStatus**、**KafkaMirrorMaker2Status**、**KafkaMirrorMakerStatus**、**KafkaRebalanceStatus**、**KafkaStatus**、**KafkaTopicStatus**、**KafkaUserStatus**

プロパティー	説明
type	リソース内の他の条件と区別するために使用される条件の固有識別子。
string	
status	条件のステータス (True、False、または Unknown のいずれか)。
string	
lastTransitionTime	タイプの条件がある状態から別の状態へと最後に変更した時間。必須形式は、UTC タイムゾーンの 'yyyy-MM-ddTHH:mm:ssZ' です。
string	
reason	条件の最後の遷移の理由 (CamelCase の単一の単語)。
string	
message	条件の最後の遷移の詳細を示す、人間が判読できるメッセージ。
string	

12.2.58. ListenerStatus スキーマ参照

KafkaStatus で使用

プロパティー	説明
type	**type プロパティーは非推奨となり、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一覧。
ListenerAddress array	
bootstrapServers	このリスナーを使用して Kafka クラスターに接続するための **host:port** ペアのコンマ区切りリスト。

プロパティー	説明
string	
certificates string array	指定のリスナーへの接続時に、サーバーのアイデンティティーを検証するために使用できる TLS 証明書の一覧。**tls** リスナーと **external** リスナーに対してのみ設定します。

12.2.59. ListenerAddress スキーマ参照

ListenerStatus で使用

プロパティー	説明
host string	Kafka ブートストラップサービスの DNS 名または IP アドレス。
port integer	Kafka ブートストラップサービスのポート。

12.2.60. KafkaConnect スキーマ参照

プロパティー	説明
spec **KafkaConnectSpec**	Kafka Connect クラスターの仕様。
status **KafkaConnectStatus**	Kafka Connect クラスターのステータス。

12.2.61. KafkaConnectSpec スキーマ参照

KafkaConnect で使用

KafkaConnectSpec スキーマプロパティーの全リスト

Kafka Connect クラスターを設定します。

12.2.61.1. config

Kafka のオプションをキーとして設定するには、**config** プロパティーを使用します。

標準の Apache Kafka Connect 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。

以下に関連する設定オプションは設定できません。

- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- リスナー / REST インターフェイスの設定
- 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** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka Connect に渡されます。

重要

提供された **config** オブジェクトのキーまたは値は Cluster Operator によって検証されません。無効な設定を指定すると、Kafka Connect クラスターが起動しなかったり、不安定になる可能性があります。この状況で、**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** プロパティーにない場合に自動的に設定されます。

禁止されているオプションには例外があります。TLS バージョンの特定の **暗号スイート** を使用して、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** プロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

Kafka Connect の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  config:
    group.id: my-connect-cluster
    offset.storage.topic: my-connect-cluster-offsets
    config.storage.topic: my-connect-cluster-configs
    status.storage.topic: my-connect-cluster-status
    key.converter: org.apache.kafka.connect.json.JsonConverter
    value.converter: org.apache.kafka.connect.json.JsonConverter
    key.converter.schemas.enable: true
    value.converter.schemas.enable: true
    config.storage.replication.factor: 3
    offset.storage.replication.factor: 3
    status.storage.replication.factor: 3
    ssl.cipher.suites: "TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384"
    ssl.enabled.protocols: "TLSv1.2"
    ssl.protocol: "TLSv1.2"
    ssl.endpoint.identification.algorithm: HTTPS
  # ...
```

TLS バージョンの特定の **暗号スイート** を使用するクライアント接続に、許可された **ssl** プロパティーを設定 することができます。また、 **ssl.endpoint.identification.algorithm** プロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

12.2.61.2. 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 プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は **log4j.properties** を使用して記述されます。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、Apache logging services を参照してください。

ここで、**inline** および **external** ロギングの例を示します。

inline ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
spec:
  # ...
  logging:
    type: inline
    loggers:
      connect.root.logger.level: "INFO"
  # ...
```

外部ロギング

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
        name: customConfigMap
        key: connect-logging.log4j
  # ...
```

設定されていない利用可能なロガーのレベルは **OFF** に設定されています。

Cluster Operator を使用して Kafka Connect がデプロイされた場合、Kafka Connect のロギングレベルの変更は動的に適用されます。

外部ロギングを使用する場合は、ロギングアペンダーが変更されるとローリング更新がトリガーされます。

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.61.3. KafkaConnectSpec スキーマプロパティー

プロパティー	説明
version string	Kafka Connect のバージョン。デフォルトは 3.3.1 です。バージョンのアップグレードまたはダウングレードに必要なプロセスを理解するには、ユーザードキュメントを参照してください。
replicas integer	Kafka Connect グループの Pod 数。
image string	Pod の Docker イメージ。
bootstrapServers string	接続するブートストラップサーバー。これは **<hostname>**:_<port>_ pairs のコンマ区切りリストとして指定する必要があります。
tls **ClientTls**	TLS 設定。
authentication **KafkaClientAuthenticationTls**, **KafkaClientAuthenticationScramSha256**, **KafkaClientAuthenticationScramSha512**, **KafkaClientAuthenticationPlain**, **KafkaClientAuthenticationOAuth**	Kafka Connect の認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls, scram-sha-256, scram-sha-512, plain, oauth] のいずれかでなければなりません。
config map	Kafka Connect の設定。次の接頭辞を持つプロパティーは設定できません: ssl.、sasl.、security.、listeners、plugin.path、rest.、bootstrap.servers、consumer.interceptor.classes、producer.interceptor.classes (ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols を除く)。

プロパティー	説明
resources ResourceRequirements	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。
livenessProbe **Probe**	Pod の liveness チェック。
readinessProbe **Probe**	Pod の readiness チェック。
jvmOptions **JvmOptions**	Pod の JVM オプション。
jmxOptions **KafkaJmxOptions**	JMX オプション。
logging **InlineLogging**、**ExternalLogging**	Kafka Connect のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
clientRackInitImage string	**client.rack** の初期化に使用される init コンテナーのイメージです。
rack **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。
tracing **JaegerTracing**、**OpenTelemetryTracing**	Kafka Connect でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger, opentelemetry] の 1 つでなければなりません。
template **KafkaConnectTemplate**	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。
externalConfiguration	Secret または ConfigMap から Kafka Connect Pod にデータを渡し、これを使用してコネクターを設定します。

プロパティー	説明
ExternalConfiguration	
build	Connect コンテナーイメージを構築する方法を設定します。オプション:
ビルド	
metricsConfig	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
JmxPrometheusExporterMetrics	

12.2.62. 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ーの完全リスト

KafkaConnect、KafkaBridge、KafkaMirror、KafkaMirrorMaker2 をクラスターに接続するための TLS 信頼証明書を設定します。

12.2.62.1. trustedCertificates

trustedCertificates プロパティーを使ってシークレットのリストを提供する。

12.2.62.2. ClientTls スキーマプロパティー

プロパティー	説明
trustedCertificates	TLS 接続の信頼済み証明書。
CertSecretSource array	

12.2.63. 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ーの全リスト

mTLS 認証を設定するには、**type** プロパティーを値 **tls** に設定します。mTLS は TLS 証明書を使用して認証します。

12.2.63.1. certificateAndKey

証明書は **certificateAndKey** プロパティーで指定され、常に OpenShift シークレットからロードされます。シークレットでは、公開鍵と秘密鍵の 2 つの鍵を使用して証明書を X509 形式で保存する必要があります。

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

mTLS 認証は、TLS 接続でのみ使用できます。

mTLS 設定の例

```
authentication:
  type: tls
  certificateAndKey:
    secretName: my-secret
    certificate: my-public-tls-certificate-file.crt
    key: private.key
```

12.2.63.2. KafkaClientAuthenticationTls スキーマプロパティー

typeプロパティーは、**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。

プロパティー	説明
certificateAndKey	証明書と秘密鍵のペアを保持する **Secret** への参照。
CertAndKeySecretSource	
type	**tls** でなければなりません。
string	

12.2.64. 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ーの全リスト

SASL ベースの SCRAM-SHA-256 認証を設定するには、**type** プロパティーを **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。

12.2.64.1. username

username プロパティーでユーザー名を指定します。

12.2.64.2. passwordSecret

passwordSecret プロパティーで、パスワードが含まれる **Secret** へのリンクを指定します。

User Operator によって作成されたシークレットを使用できます。

必要に応じて、認証に使用するクリアテキストのパスワードが含まれるテキストファイルを作成できます。

```
echo -n PASSWORD > MY-PASSWORD.txt
```

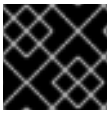
次に、テキストファイルから **Secret** を作成し、パスワードに独自のフィールド名 (鍵) を設定できます。

```
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: LFTIyFRFlMmU2N2Tm
```

secretName プロパティーには **Secret** の名前が含まれ、**password** プロパティーには **Secret** 内にパスワードが格納されるキーの名前が含まれます。



重要

password プロパティーには、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
```

12.2.64.3. KafkaClientAuthenticationScramSha256 スキーマプロパティー

プロパティー	説明
passwordSecret	パスワードを保持する **Secret** への参照。
PasswordSecretSource	
type	**scram-sha-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	

12.2.65. PasswordSecretSource スキーマ参照

使用先:
KafkaClientAuthenticationOAuth、**KafkaClientAuthenticationPlain**、**KafkaClientAuthenticationScramSha256**、**KafkaClientAuthenticationScramSha512**

プロパティー	説明
password	パスワードが保存される Secret のキーの名前。
string	
secretName	パスワードを含むシークレットの名前。
string	

12.2.66. 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ーの全リスト

SASL ベースの SCRAM-SHA-512 認証を設定するには、**type** プロパティーを **scram-sha-512** に設定します。SCRAM-SHA-512 認証メカニズムには、ユーザー名とパスワードが必要です。

12.2.66.1. username

username プロパティーでユーザー名を指定します。

12.2.66.2. passwordSecret

passwordSecret プロパティーで、パスワードが含まれる **Secret** へのリンクを指定します。

User Operator によって作成されたシークレットを使用できます。

必要に応じて、認証に使用するクリアテキストのパスワードが含まれるテキストファイルを作成できます。

```
echo -n PASSWORD > MY-PASSWORD.txt
```

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: LFTIyFRFlMmU2N2Tm
```

secretName プロパティーには **Secret** の名前が含まれ、**password** プロパティーには **Secret** 内にパスワードが格納されるキーの名前が含まれます。

重要

password プロパティーには、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
```

12.2.66.3. KafkaClientAuthenticationScramSha512 スキーマプロパティー

プロパティー	説明
passwordSecret	パスワードを保持する **Secret** への参照。
PasswordSecretSource	
type	**scram-sha-512** でなければなりません。
string	

プロパティー	説明
username	認証に使用されるユーザー名。
string	

12.2.67. 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ーの全リスト

SASL ベースの PLAIN 認証を設定するには、**type** プロパティーを **plain** に設定します。SASL PLAIN 認証メカニズムには、ユーザー名とパスワードが必要です。

警告

SASL PLAIN メカニズムは、クリアテキストでユーザー名とパスワードをネットワーク全体に転送します。TLS による暗号化が有効になっている場合にのみ SASL PLAIN 認証を使用します。

12.2.67.1. username

username プロパティーでユーザー名を指定します。

12.2.67.2. passwordSecret

passwordSecret プロパティーで、パスワードが含まれる **Secret** へのリンクを指定します。

User Operator によって作成されたシークレットを使用できます。

必要に応じて、認証に使用するクリアテキストのパスワードが含まれるテキストファイルを作成します。

```
echo -n PASSWORD > MY-PASSWORD.txt
```

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: LFTIyFRFlMmU2N2Tm
```

secretName プロパティーには **Secret** の名前が含まれ、**password** プロパティーには **Secret** 内にパスワードが格納されるキーの名前が含まれます。

重要

password プロパティーには、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
```

12.2.67.3. KafkaClientAuthenticationPlain スキーマプロパティー

type プロパティーは、**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。

プロパティー	説明
passwordSecret	パスワードを保持する **Secret** への参照。
PasswordSecretSource	
type	**plain** でなければなりません。
string	
username	認証に使用されるユーザー名。
string	

12.2.68. 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ーの全リスト

OAuth クライアント認証を設定するには、**type** プロパティーを **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
- ユーザー名およびパスワード
- TLS

クライアント ID およびシークレット

認証で使用されるクライアント ID およびクライアントシークレットとともに、**tokenEndpointUri** プロパティーで承認サーバーのアドレスを設定できます。OAuth クライアントは OAuth サーバーに接続し、クライアント ID およびシークレットを使用して認証し、Kafka ブローカーとの認証に使用するアクセストークンを取得します。**clientSecret** プロパティーで、クライアントシークレットを含む **Secret** へのリンクを指定します。

クライアント ID およびクライアントシークレットを使用した OAuth クライアント認証の例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  clientSecret:
    secretName: my-client-oauth-secret
    key: client-secret
```

必要に応じて、**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** プロパティーで OAuth サーバーのアドレスを設定できます。OAuth クライアントは OAuth サーバーに接続し、クライアント ID と更新トークンを使用して認証し、Kafka ブローカーとの認証に使用するアクセストークンを取得します。**refreshToken** プロパティーで、更新トークンが含まれる **Secret** へのリンクを指定します。

クライアント ID と更新トークンを使用した OAuth クライアント認証の例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
```

アクセストークン

Kafka ブローカーとの認証に使用されるアクセストークンを直接設定できます。この場合、**tokenEndpointUri** は指定しません。**accessToken** プロパティーで、アクセストークンが含まれる **Secret** へのリンクを指定します。

アクセストークンのみを使用した OAuth クライアント認証の例

```
authentication:
  type: oauth
  accessToken:
    secretName: my-access-token-secret
    key: access-token
```

ユーザー名およびパスワード

OAuth のユーザー名とパスワードの設定では、OAuth **リソースオーナーのパスワード付与** メカニズムを使用します。このメカニズムは非推奨であり、クライアント認証情報 (ID とシークレット) を使用できない環境での統合を有効にするためにのみサポートされています。アクセス管理システムが別のアプローチをサポートしていない場合、または認証にユーザーアカウントが必要な場合は、ユーザーアカウントの使用が必要になることがあります。

典型的なアプローチは、クライアントアプリケーションを表す認可サーバーに特別なユーザーアカウントを作成することです。次に、ランダムに生成された長いパスワードと非常に限られた権限セットをアカウントに与えます。たとえば、アカウントは Kafka クラスターにのみ接続できますが、他のサービスを使用したり、ユーザーインターフェイスにログインしたりすることはできません。

最初にリフレッシュトークンメカニズムの使用を検討してください。

tokenEndpointUri プロパティーで、認証に使用されるクライアント ID、ユーザー名、およびパスワードと共に、承認サーバーのアドレスを設定できます。OAuth クライアントは OAuth サーバーに接続し、ユーザー名、パスワード、クライアント ID、およびオプションでクライアントシークレットを使用して認証し、Kafka ブローカーでの認証に使用するアクセストークンを取得します。

passwordSecret プロパティーで、パスワードが含まれる **Secret** へのリンクを指定します。

通常、パブリック OAuth クライアントを使用して **clientId** も設定する必要があります。機密 OAuth クライアントを使用している場合は、**clientSecret** も設定する必要があります。

パブリッククライアントでのユーザー名とパスワードを使用した 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
  username: my-username
  passwordSecret:
    secretName: my-password-secret-name
    password: my-password-field-name
  clientId: my-public-client-id
```

機密クライアントでのユーザー名とパスワードを使用した 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
  username: my-username
  passwordSecret:
```

```
    secretName: my-password-secret-name
    password: my-password-field-name
  clientId: my-confidential-client-id
  clientSecret:
    secretName: my-confidential-client-oauth-secret
    key: client-secret
```

必要に応じて、**scope** と **audience** を指定できます。

TLS

HTTPS プロトコルを使用して OAuth サーバーにアクセスする場合、信頼される認証局によって署名された証明書を使用し、そのホスト名が証明書に記載されている限り、追加の設定は必要ありません。

OAuth サーバーが自己署名証明書を使用している場合、または信頼されていない認証局によって署名されている場合は、カスタムリソースで信頼済み証明書の一覧を設定できます。**tlsTrustedCertificates** プロパティーには、証明書が保存されるキーの名前を持つシークレットの一覧が含まれます。証明書は X509 形式で保存する必要があります。

提供される TLS 証明書の例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
  tlsTrustedCertificates:
    - secretName: oauth-server-ca
      certificate: tls.crt
```

OAuth クライアントはデフォルトで、OAuth サーバーのホスト名が、証明書サブジェクトまたは別の DNS 名のいずれかと一致することを確認します。必要でない場合は、ホスト名の検証を無効にできます。

無効にされた TLS ホスト名の検証例

```
authentication:
  type: oauth
  tokenEndpointUri: https://sso.myproject.svc:8443/auth/realms/internal/protocol/openid-connect/token
  clientId: my-client-id
  refreshToken:
    secretName: my-refresh-token-secret
    key: refresh-token
  disableTlsHostnameVerification: true
```

12.2.68.1. KafkaClientAuthenticationOAuth スキーマプロパティー

type プロパティーは、**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。

プロパティー	説明
accessToken **GenericSecretSource**	承認サーバーから取得したアクセストークンが含まれる OpenShift シークレットへのリンク。
accessTokenIsJwt boolean	アクセストークンを JWT として処理すべきかどうかを設定します。承認サーバーが不透明なトークンを返す場合は、**false** に設定する必要があります。デフォルトは **true** です。
audience string	承認サーバーに対して認証を行うときに使用する OAuth オーディエンス。一部の承認サーバーでは、オーディエンスを明示的に設定する必要があります。許可される値は、承認サーバーの設定によります。デフォルトでは、トークンエンドポイントリクエストを実行する場合は **audience** は指定されません。
clientId string	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。
clientSecret **GenericSecretSource**	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。
connectTimeoutSeconds integer	承認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。
disableTlsHostnameVerification boolean	TLS ホスト名の検証を有効または無効にします。デフォルト値は **false** です。
enableMetrics boolean	OAuth メトリックを有効または無効にします。デフォルト値は **false** です。
maxTokenExpirySeconds integer	アクセストークンの有効期間を指定の秒数に設定または制限します。これは、承認サーバーが不透明なトークンを返す場合に設定する必要があります。
passwordSecret **PasswordSecretSource**	パスワードを保持する **Secret** への参照。

プロパティー	説明
readTimeoutSeconds integer	承認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。
refreshToken **GenericSecretSource**	承認サーバーからアクセストークンを取得するために使用できる更新トークンが含まれる OpenShift シークレットへのリンク。
scope string	承認サーバーに対して認証を行うときに使用する OAuth スコープ。一部の承認サーバーでこれを設定する必要があります。許可される値は、承認サーバーの設定によります。デフォルトでは、トークンエンドポイントリクエストを実行する場合は **scope** は指定されません。
tlsTrustedCertificates **CertSecretSource** array	OAuth サーバーへの TLS 接続の信頼済み証明書。
tokenEndpointUri string	承認サーバートークンエンドポイント URI。
type string	**oauth** でなければなりません。
username string	認証に使用されるユーザー名。

12.2.69. JaegerTracing スキーマ参照

タイプ JaegerTracing は非推奨になりました。

使用先: **KafkaBridgeSpec**、**KafkaConnectSpec**、**KafkaMirrorMaker2Spec**、**KafkaMirrorMakerSpec**

type プロパティーは、**JaegerTracing** タイプの使用を **OpenTelemetryTracing** と区別する識別子です。**JaegerTracing** タイプには **jaeger** の値が必要です。

プロパティー	説明
type	**jaeger** でなければなりません。

プロパティー	説明
string	

12.2.70. OpenTelemetryTracing スキーマ参照

使用先: KafkaBridgeSpec、KafkaConnectSpec、KafkaMirrorMaker2Spec、KafkaMirrorMakerSpec

type プロパティーは、**OpenTelemetryTracing** タイプの使用をJaegerTracing と区別する識別子です。タイプ **OpenTelemetryTracing** の値が **opentelemetry** である必要があります。

プロパティー	説明
type	**opentelemetry** でなければなりません。
string	

12.2.71. KafkaConnectTemplate スキーマ参照

使用先: KafkaConnectSpec、KafkaMirrorMaker2Spec

プロパティー	説明
deployment	Kafka Connect **Deployment** のテンプレート。
DeploymentTemplate	
Pod	Kafka Connect **Pod** のテンプレート。
PodTemplate	
apiService	Kafka Connect API **Service** のテンプレート。
InternalServiceTemplate	
connectContainer	Kafka Connect コンテナーのテンプレート。
ContainerTemplate	
initContainer	Kafka init コンテナーのテンプレート。
ContainerTemplate	
podDisruptionBudget	Kafka Connect **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	

プロパティー	説明
serviceAccount **ResourceTemplate**	Kafka Connect サービスアカウントのテンプレート。
clusterRoleBinding **ResourceTemplate**	Kafka Connect ClusterRoleBinding のテンプレート。
buildPod **PodTemplate**	Kafka Connect Build **Pods** のテンプレート。build Pod は OpenShift でのみ使用されます。
buildContainer **ContainerTemplate**	Kafka Connect Build コンテナーのテンプレート。build コンテナーは OpenShift でのみ使用されます。
buildConfig **BuildConfigTemplate**	新しいコンテナーイメージをビルドするために使用される Kafka Connect BuildConfig のテンプレート。BuildConfig は OpenShift でのみ使用されます。
buildServiceAccount **ResourceTemplate**	Kafka Connect Build サービスアカウントのテンプレート。
jmxSecret **ResourceTemplate**	Kafka Connect Cluster JMX 認証の Secret のテンプレートです。

12.2.72. DeploymentTemplate スキーマ参照

KafkaBridgeTemplate、**KafkaConnectTemplate**、**KafkaMirrorMakerTemplate** で使用

プロパティー	説明
metadata **MetadataTemplate**	リソースに適用済みのメタデータ。
deploymentStrategy string ([RollingUpdate、Recreate] のいずれか)	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。

12.2.73. BuildConfigTemplate スキーマ参照

使用先: **KafkaConnectTemplate**

プロパティー	説明
metadata	**PodDisruptionBudgetTemplate** リソースに適用するメタデータ。
MetadataTemplate	
pullSecret	ベースイメージをプルするためのクレデンシャルが含まれる Container Registry Secret。
string	

12.2.74. ExternalConfiguration スキーマ参照

使用先: **KafkaConnectSpec**、**KafkaMirrorMaker2Spec**

ExternalConfiguration スキーマプロパティーの完全リスト

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ーを設定します。

ConfigMap またはシークレットを環境変数またはボリュームとして Kafka Connect Pod にマウントできます。ボリュームおよび環境変数は、**KafkaConnect.spec** の **externalConfiguration** プロパティーで設定されます。

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。

12.2.74.1. env

env プロパティーを使用して 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: QUtJQVhYWFhYWFhYWFhYWFg=
  awsSecretAccessKey: Ylhsd1lYTnpkMjl5WkE=
```

注記

ユーザー定義の環境変数に、**KAFKA_** または **STRIMZI_** で始まる名前を付けることはできません。

シークレットから環境変数に値をマウントするには、**valueFrom** プロパティーおよび **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、コネクターが Amazon AWS と通信するためのものです。コネクターは **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** プロパティーで **configMapKeyRef** を使用します。

ConfigMap からの値に設定された環境変数の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect
spec:
  # ...
  externalConfiguration:
    env:
      - name: MY_ENVIRONMENT_VARIABLE
        valueFrom:
          configMapKeyRef:
            name: my-config-map
            key: my-key
```

12.2.74.2. volumes

ボリュームを使用して ConfigMap またはシークレットを Kafka Connect Pod にマウントします。

以下の場合、環境変数の代わりにボリュームを使用すると便利です。

- Kafka Connect コネクターの設定に使用されるプロパティーファイルのマウント
- TLS 証明書でのトラストストアまたはキーストアのマウント

ボリュームは、パス **/opt/kafka/external-configuration/<volume-name>** の Kafka Connect コンテナー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 ConnectREST 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ー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ー値の読み込み

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ープロパティーが含まれています。

データベースプロパティーのある Secret の例

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  connector.properties: |- 1
    dbUsername: my-username 2
    dbPassword: my-password
```

1 プロパティーファイル形式のコネクター設定。

2 設定で使用されるデータベースのユーザー名およびパスワードプロパティー。

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 1
    config.providers.file.class: org.apache.kafka.common.config.provider.FileConfigProvider 2
  #...
  externalConfiguration:
    volumes:
      - name: connector-config 3
        secret:
          secretName: mysecret 4
```

-

1. 設定プロバイダーのエイリアスは、他の設定パラメーターを定義するために使用されます。
2. **FileConfigProvider** はプロパティー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** の形式を取ります。
3. Secret が含まれるボリュームの名前。各ボリュームは **name** プロパティーに名前を指定し、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 のプロパティー値のプレースホルダーは、コネクター設定で参照されます。プレースホルダー構造は、**configmaps:PATH-AND-FILE-NAME:PROPERTY** です。**FileConfigProvider** は、コネクター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** プロパティーの値を読み取りおよび展開します。

外部値のプレースホルダーを示すコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    database.hostname: 192.168.99.1
    database.port: "3306"
    database.user: "${file:/opt/kafka/external-configuration/connector-config/mysecret:dbUsername}"
    database.password: "${file:/opt/kafka/external-configuration/connector-
config/mysecret:dbPassword}"
    database.server.id: "184054"
    #...
```

DirectoryConfigProvider を使用した個別ファイルからのプロパティー値のロード

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:
  ca.crt: <public_key> # Public key of the clients CA
  user.crt: <user_certificate> # Public key of the user
```

```
user.key: <user_private_key> # Private key of the user
user.p12: <store> # PKCS #12 store for user certificates and keys
user.password: <password_for_store> # Protects the PKCS #12 store
```

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
(1)
  #...
  externalConfiguration:
    volumes:
      - name: cluster-ca
        secret:
          secretName: my-cluster-cluster-ca-cert
      - name: my-user
        secret:
          secretName: my-user
```

(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ー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、コネクター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展開します。

外部値のプレースホルダーを示すコネクター設定の例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector
  labels:
    strimzi.io/cluster: my-connect-cluster
spec:
  class: io.debezium.connector.mysql.MySqlConnector
  tasksMax: 2
  config:
    # ...
    database.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}"
    #...
```

12.2.74.3. ExternalConfiguration スキーマプロパティー

プロパティー	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	

12.2.75. ExternalConfigurationEnv スキーマ参照

ExternalConfiguration で使用

プロパティー	説明
name	Kafka Connect Pod に渡される環境変数の名前。環境変数に、**KAFKA_** または **STRIMZI_** で始まる名前を付けることはできません。
string	
valueFrom	Kafka Connect Pod に渡される環境変数の値。Secret または ConfigMap フィールドのいずれかへ参照として渡すことができます。このフィールドでは、Secret または ConfigMap を 1 つだけ指定する必要があります。
ExternalConfigurationEnvVarSource	

12.2.76. ExternalConfigurationEnvVarSource スキーマ参照

ExternalConfigurationEnv で使用

プロパティー	説明
configMapKeyRef	ConfigMap のキーへの参照。詳細は、core/v1 configmapkeyselector の外部ドキュメント を参照してください。
ConfigMapKeySelector	

プロパティー	説明
secretKeyRef SecretKeySelector	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。

12.2.77. ExternalConfigurationVolumeSource スキーマ参照

ExternalConfiguration で使用

プロパティー	説明
configMap ConfigMapVolumeSource	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。
name string	Kafka Connect Pod に追加されるボリュームの名前。
secret SecretVolumeSource	Secret のキーへの参照。Secret または ConfigMap を 1 つだけ指定する必要があります。詳細は、core/v1 secretvolumesource の外部ドキュメント を参照してください。

12.2.78. Build スキーマ参照

使用先: **KafkaConnectSpec**

Build スキーマプロパティーの全リスト

Kafka Connect デプロイメントの追加コネクターを設定します。

12.2.78.1. 出力

追加のコネクタープラグインで新しいコンテナーイメージをビルドするには、イメージをプッシュ、保存、およびプルできるコンテナーレジストリーが AMQ Streams に必要です。AMQ Streams は独自のコンテナーレジストリーを実行しないため、レジストリーを指定する必要があります。AMQ Streams は、プライベートコンテナーレジストリーだけでなく、Quay や Docker Hub などのパブリックレジストリーもサポートします。コンテナーレジストリーは、**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ーイメージのフルネームを指定する必要があります。フルネームには以下が含まれる必要があります。

- レジストリーのアドレス
- ポート番号 (標準以外のポートでリッスンしている場合)
- 新しいコンテナーイメージのタグ

有効なコンテナーイメージ名の例:

- **docker.io/my-org/my-image/my-tag**
- **quay.io/my-org/my-image/my-tag**
- **image-registry.image-registry.svc:5000/myproject/kafka-connect-build:latest**

Kafka Connect デプロイメントごとに個別のイメージを使用する必要があります。これは、最も基本的なレベルで異なるタグを使用する可能性があることを意味します。

レジストリ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 1
      image: my-registry.io/my-org/my-connect-cluster:latest 2
      pushSecret: my-registry-credentials 3
  #...
```

1 (必須) AMQ Streams によって使用される出力のタイプ。

2 (必須) リポジトリーとタグを含む、使用されるイメージのフルネーム。

3 (任意) コンテナーレジストリーのクレデンシャルが含まれるシークレットの名前。

OpenShift ImageStream の使用

Docker の代わりに OpenShift ImageStream を使用して、新しいコンテナーイメージを保存できます。Kafka Connect をデプロイする前に、ImageStream を手動で作成する必要があります。ImageStream を使用するには、**type** を **imagestream** に設定し、**image** プロパティーを使用して 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 (1)
      image: my-connect-build:latest (2)
  #...
```

(1) (必須) AMQ Streams によって使用される出力のタイプ。

(2) (必須) ImageStream およびタグの名前。

12.2.78.2. plugins

コネクタープラグインは、特定タイプの外部システムへの接続に必要な実装を定義するファイルのセットです。コンテナーイメージに必要なコネクタープラグインは、**KafkaConnect** カスタムリソースの **.spec.build.plugins** プロパティーを使用して設定する必要があります。各コネクタープラグインには、Kafka Connect デプロイメント内で一意となる名前が必要です。さらに、プラグインアーティファクトもリストする必要があります。これらのアーティファクトは AMQ Streams によってダウンロードされ、新しいコンテナーイメージに追加され、Kafka Connect デプロイメントで使用されます。コネクタープラグインアーティファクトには、シリアライザーやデシリアライザーなどの追加のコンポーネントを含めることもできます。各コネクタープラグインは、異なるコネクターとそれらの依存関係が適切に **サンドボックス化** されるように、個別のディレクトリーにダウンロードされます。各プラグインは、1つ以上の **artifact** で設定する必要があります。

2つのコネクタープラグインを持つ 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) コネクタープラグインおよびそれらのアーティファクトの一覧。

AMQ Streams 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、コンテナーイメージにダウンロードされ、追加された JAR ファイルを表します。JAR アーティファクトを使用するには、**type** プロパティーを **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 は新しいコンテナーイメージのビルド中にアーティファクトのチェックサムを検証します。

JAR アーティファクトの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: jar 1
            url: https://my-domain.tld/my-jar.jar 2
            sha512sum: 589...ab4 3
```

```
      - type: jar
        url: https://my-domain.tld/my-jar2.jar
  #...
```

1. (必須) アーティファクトのタイプ。
2. (必須) アーティファクトのダウンロード元 URL。
3. (任意) アーティファクトを検証する SHA-512 チェックサム。

TGZ アーティファクトの使用

TGZ アーティファクトは、Gzip 圧縮を使用して圧縮された TAR アーカイブをダウンロードするために使用されます。複数の異なるファイルで設定される場合でも、TGZ アーティファクトに Kafka Connect コネクター全体を含めることができます。TGZ アーティファクトは、新しいコンテナーイメージのビルド時に AMQ Streams によって自動的にダウンロードおよび展開されます。TGZ アーティファクトを使用するには、**type** プロパティーを **tgz** 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、展開して新しいコンテナーイメージをビルドする前に、チェックサムが AMQ Streams によって検証されます。

TGZ アーティファクトの例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec:
  #...
  build:
    output:
      #...
    plugins:
      - name: my-plugin
        artifacts:
          - type: tgz (1)
            url: https://my-domain.tld/my-connector-archive.tgz (2)
            sha512sum: 158...jg10 (3)
  #...
```

1. (必須) アーティファクトのタイプ。
2. (必須) アーカイブのダウンロード元 URL。
3. (任意) アーティファクトを検証する SHA-512 チェックサム。

ZIP アーティファクトの使用

ZIP アーティファクトは ZIP 圧縮アーカイブのダウンロードに使用されます。前のセクションで説明した TGZ アーティファクトと同じ方法で ZIP アーティファクトを使用します。唯一の違いは、**type: tgz** ではなく **type: zip** を指定することです。

Maven アーティファクトの使用

Maven アーティファクトは、コネクタープラグインアーティファクトを Maven コーディネートとして指定するために使用されます。Maven コーディネートは、プラグインアーティファクトおよび依存関係を特定し、Maven リポジトリーから検索および取得できるようにします。

注記

コネクタービルドプロセスがアーティファクトをコンテナーイメージに追加するには、Maven リポジトリー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 (1)
            repository: https://mvnrepository.com (2)
            group: org.apache.camel.kafkaconnector (3)
            artifact: camel-kafka-connector (4)
            version: 0.11.0 (5)
  #...
```

(1) (必須) アーティファクトのタイプ。

(2) (任意) アーティファクトのダウンロード元となる Maven リポジトリー。リポジトリーを指定しないと、デフォルトで Maven Central リポジトリー が使用されます。

(3) (必須) Maven グループ ID。

(4) (必須) Maven アーティファクトタイプ。

(5) (必須) Maven バージョン番号。

other アーティファクトの使用

other アーティファクトは、コンテナー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ー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 は新しいコンテナーイメージのビルド中にアーティファクトのチェックサムを検証します。

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 (1)
            url: https://my-domain.tld/my-other-file.ext (2)
            sha512sum: 589...ab4 (3)
            fileName: name-the-file.ext (4)
  #...
```

(1) (必須) アーティファクトのタイプ。

(2) (必須) アーティファクトのダウンロード元 URL。

(3) (任意) アーティファクトを検証する SHA-512 チェックサム。

(4) (任意) 結果となるコンテナーイメージに保存されるファイルの名前。

12.2.78.3. Build スキーマのプロパティー

プロパティー	説明
output **DockerOutput**, **ImageStreamOutput**	新たにビルドされたイメージの保存先を設定します。必須。タイプは、指定のオブジェクト内の **output.type** プロパティー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。
resources ResourceRequirements	ビルド用に予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
plugins **Plugin** アレイ	Kafka Connect に追加する必要のあるコネクタープラグインのリスト。必須。

12.2.79. DockerOutput スキーマ参照

Build で使用

type プロパティーは、**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。

プロパティー	説明
image string	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。
pushSecret string	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。
additionalKanikoOptions string array	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。
type string	**docker** でなければなりません。

12.2.80. ImageStreamOutput スキーマ参照

Build で使用

type プロパティーは、**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。

プロパティー	説明
image string	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。
type string	**ImageStream** の例

12.2.81. Plugin スキーマ参照

Build で使用

プロパティー	説明
name string	コネクタープラグインの一意名。コネクターアーティファクトが保存されるパスの生成に使用されます。名前は KafkaConnect リソース内で一意である必要があります。この名前は、**^[a-z][-_a-z0-9]*[a-z]$** のパターンに従う必要があります。必須。
artifacts **JarArtifact**, **TgzArtifact**, **ZipArtifact**, **MavenArtifact**, **OtherArtifact** アレイ	このコネクタープラグインに属するアーティファクトの一覧。必須。

12.2.82. JarArtifact スキーマ参照

Plugin で使用されます。

プロパティー	説明
url string	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。**jar**, **zip**, **tgz** 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。
sha512sum string	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、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。**maven** アーティファクトタイプには該当しません。
insecure boolean	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。
type	**jar** でなければなりません。

プロパティー	説明
string	

12.2.83. TgzArtifact スキーマ参照

Plugin で使用されます。

プロパティー	説明
url string	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。**jar**, **zip**, **tgz** 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。
sha512sum string	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、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。**maven** アーティファクトタイプには該当しません。
insecure boolean	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。
type string	**tgz** でなければなりません。

12.2.84. ZipArtifact スキーマ参照

Plugin で使用されます。

プロパティー	説明

プロパティー	説明
url string	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。**jar**, **zip**, **tgz** 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。
sha512sum string	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、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。**maven** アーティファクトタイプには該当しません。
insecure boolean	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。
type string	**zip** でなければなりません。

12.2.85. MavenArtifact スキーマ参照

Plugin で使用されます。

type プロパティーは、**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。

プロパティー	説明
repository string	アーティファクトのダウンロード元となる Maven リポジトリー。**maven** アーティファクトタイプにのみ適用されます。
group string	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。
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。

プロパティー	説明
string	
version	Maven のバージョン番号。**maven** アーティファクトタイプにのみ適用されます。
string	
type	**maven** でなければなりません。
string	

12.2.86. OtherArtifact スキーマ参照

Plugin で使用されます。

プロパティー	説明
url	ダウンロードされるアーティファクトの URL。AMQ Streams では、ダウンロードしたアーティファクトのセキュリティースキャンは行いません。セキュリティー上の理由から、最初にアーティファクトを手動で検証し、チェックサムの検証を設定して、自動ビルドで同じアーティファクトが使用されるようにする必要があります。**jar**, **zip**, **tgz** 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、新しいコンテナーのビルド時にチェックサムが検証されます。指定のない場合は、ダウンロードしたアーティファクトは検証されません。**maven** アーティファクトタイプには該当しません。
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。

プロパティー	説明
string	

12.2.87. KafkaConnectStatus スキーマ参照

KafkaConnect で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
url	Kafka Connect コネクターの管理および監視用の REST API エンドポイントの URL。
string	
connectorPlugins	この Kafka Connect デプロイメントで使用できるコネクタープラグインの一覧。
ConnectorPlugin array	
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	

12.2.88. ConnectorPlugin スキーマ参照

使用先: **KafkaConnectStatus**、**KafkaMirrorMaker2Status**

プロパティー	説明
type	コネクタープラグインのタイプ。**sink** タイプと **source** タイプを利用できます。
string	
version	コネクタープラグインのバージョン。

プロパティー	説明
string	
class	コネクタープラグインのクラス。
string	

12.2.89. KafkaTopic スキーマ参照

プロパティー	説明
spec	トピックの仕様。
KafkaTopicSpec	
status	トピックのステータス。
KafkaTopicStatus	

12.2.90. KafkaTopicSpec スキーマ参照

KafkaTopic で使用

プロパティー	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 metadata.name に設定されます。トピック名が有効な OpenShift リソース名ではない場合を除き、これを設定しないことが推奨されます。
string	

12.2.91. KafkaTopicStatus スキーマ参照

KafkaTopic で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
topicName	トピック名。
string	

12.2.92. KafkaUser スキーマ参照

プロパティー	説明
spec	ユーザーの仕様。
KafkaUserSpec	
status	Kafka User のステータス。
KafkaUserStatus	

12.2.93. KafkaUserSpec スキーマ参照

KafkaUser で使用

プロパティー	説明

プロパティー	説明
authentication **KafkaUserTlsClientAuthentication**, **KafkaUserTlsExternalClientAuthentication**, **KafkaUserScramSha512ClientAuthentication**	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。 • **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。 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** プロパティー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。
認可 **KafkaUserAuthorizationSimple**	この Kafka ユーザーの承認ルール。タイプは、指定のオブジェクト内の **authorization.type** プロパティーの値によって異なり、[simple] の 1 つでなければなりません。
quotas **KafkaUserQuotas**	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。
template **KafkaUserTemplate**	Kafka User **Secrets** の生成方法を指定するテンプレート。

12.2.94. KafkaUserTlsClientAuthentication スキーマ参照

KafkaUserSpec で使用

type プロパティーは **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。

プロパティー	説明
type	**tls** でなければなりません。
string	

12.2.95. KafkaUserTlsExternalClientAuthentication スキーマ参照

KafkaUserSpec で使用

type プロパティーは **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。

プロパティー	説明
type	**tls-external** でなければなりません。
string	

12.2.96. KafkaUserScramSha512ClientAuthentication スキーマ参照

KafkaUserSpec で使用

type プロパティーは **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。

プロパティー	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	

12.2.97. Password スキーマ参照

使用先: KafkaUserScramSha512ClientAuthentication

プロパティー	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	

12.2.98. PasswordSource スキーマ参照

使用先: **Password**

プロパティー	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	

12.2.99. KafkaUserAuthorizationSimple スキーマ参照

KafkaUserSpec で使用

type プロパティーは、**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。

プロパティー	説明
type	**simple** でなければなりません。
string	
ACL	このユーザーに適用される必要のある ACL ルールの一覧。
AclRule array	

12.2.100. AclRule スキーマ参照

KafkaUserAuthorizationSimple で使用

AclRule スキーマプロパティーの全リスト

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
        operations:
          - Read
          - Describe
      - resource:
          type: group
          name: my-group
          patternType: prefix
        operations:
          - Read
```

12.2.100.1. resource

resource プロパティーを使用して、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** プロパティーに指定される、以下の 4 つのリソースタイプをサポートします。

- トピック (**topic**)
- コンシューマーグループ (**group**)
- クラスター (**cluster**)
- トランザクション ID (**transactionalId**)

Topic、Group、および Transactional ID リソースでは、**name** プロパティーでルールが適用されるリソースの名前を指定できます。

クラスタータイプのリソースには名前がありません。

名前は、**patternType** プロパティーを使用して **literal** または **prefix** として指定されます。

- リテラル (literal) 名には、**name** フィールドに指定された名前がそのまま使われます。
- 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
  operations:
    - Read
```

12.2.100.2. 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。

12.2.100.3. operations

ルールが許可または拒否する **operation** のリストを指定します。

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。

12.2.100.4. host

host プロパティーを使用して、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。

12.2.100.5. AclRule スキーマのプロパティー

プロパティー	説明
host	ACL ルールに記述されているアクションを許可または拒否するホスト。
string	
operation	**operation プロパティーは廃止されたため、spec.authorization.acls*.operations を使用して設定する必要があります。**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)	
operations	許可または拒否される操作の一覧。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 1 つ以上) array	
resource	指定の ACL ルールが適用されるリソースを示します。タイプは、指定のオブジェクト内の **resource.type** プロパティーの値によって異なり、[topic、group、cluster、transactionalId] のいずれかでなければなりません。
AclRuleTopicResource、**AclRuleGroupResource**、**AclRuleClusterResource**、**AclRuleTransactionalIdResource**	
type	ルールのタイプ。現在サポートされているタイプは **allow** のみです。**allow** タイプの ACL ルールを使用すると、ユーザーは指定した操作を実行できます。デフォルト値は **allow** です。
string ([allow、deny] のいずれか)	

12.2.101. AclRuleTopicResource スキーマ参照

AclRule で使用されます。

type プロパティーは、**AclRuleTopicResource** タイプを使用する際に **AclRuleGroupResource**、**AclRuleClusterResource**、**AclRuleTransactionalIdResource** タイプと区別する識別子です。**AclRuleTopicResource** タイプには **topic** の値が必要です。

プロパティー	説明
type	**topic** でなければなりません。
string	
name	指定の ACL ルールが適用されるリソースの名前。**patternType** フィールドと組み合わせて、接頭辞のパターンを使用できます。

プロパティー	説明
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)	

12.2.102. AclRuleGroupResource スキーマ参照

AclRule で使用されます。

type プロパティーは、**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。

プロパティー	説明
type	**group** でなければなりません。
string	
name	指定の ACL ルールが適用されるリソースの名前。**patternType** フィールドと組み合わせて、接頭辞のパターンを使用できます。
string	
patternType	リソースフィールドで使用されるパターンを指定します。サポートされるタイプは **literal** と **prefix** です。**literal** パターンタイプでは、リソースフィールドは完全なトピック名の定義として使用されます。**prefix** パターンタイプでは、リソース名は接頭辞としてのみ使用されます。デフォルト値は **literal** です。
string ([prefix、literal] のいずれか)	

12.2.103. AclRuleClusterResource スキーマ参照

AclRule で使用されます。

type プロパティーは、**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。

プロパティー	説明

プロパティー	説明
type	**cluster** でなければなりません。
string	

12.2.104. AclRuleTransactionalIdResource スキーマ参照

AclRule で使用されます。

type プロパティーは、**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。

プロパティー	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)	

12.2.105. KafkaUserQuotas スキーマ参照

KafkaUserSpec で使用

KafkaUserQuotas スキーマプロパティーの全リスト

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。

12.2.105.1. quotas

クライアントを設定して、以下のタイプのクォータを使用できます。

- **ネットワーク使用率** クォータは、クォータを共有するクライアントの各グループのバイトレートしきい値を指定します。
- **CPU 使用率**クォータは、クライアントからのブローカー要求のウィンドウを指定します。ウィンドウは、クライアントが要求を行う時間の割合 (パーセント) です。クライアントはブローカーの I/O スレッドおよびネットワークスレッドで要求を行います。

- **パーティション変更**クォータは、クライアントが 1 秒ごとに実行できるパーティション変更の数を制限します。

パーティション変更クォータにより、Kafka クラスターが同時にトピック操作に圧倒されないようにします。パーティション変更は、次のタイプのユーザー要求に応答して発生します。

- 新しいトピック用のパーティションの作成
- 既存のトピックへのパーティションの追加
- トピックからのパーティションの削除

パーティション変更クオータを設定して、ユーザー要求に対して変更が許可されるレートを制御できます。

Kafka クライアントにクォータを使用することは、さまざまな状況で役に立つ場合があります。レートが高すぎる要求を送信する Kafka プロデューサーを誤って設定したとします。このように設定が間違っていると、他のクライアントにサービス拒否を引き起こす可能性があるため、問題のあるクライアントはブロックする必要があります。ネットワーク制限クォータを使用すると、他のクライアントがこの状況の著しい影響を受けないようにすることが可能です。

AMQ Streams はユーザーレベルのクォータをサポートしますが、クライアントレベルのクォータはサポートしません。

Kafka ユーザークォータの設定例

```
spec:
  quotas:
    producerByteRate: 1048576
    consumerByteRate: 2097152
    requestPercentage: 55
    controllerMutationRate: 10
```

Kafka ユーザークォータの詳細は Apache Kafka ドキュメント を参照してください。

12.2.105.2. KafkaUserQuotas スキーマ参照

プロパティー	説明
consumerByteRate integer	グループのクライアントにスロットリングが適用される前に、各クライアントグループがブローカーから取得できる最大 bps (ビット毎秒) のクオータ。ブローカーごとに定義されます。
controllerMutationRate number	トピックの作成リクエスト、パーティションの作成リクエスト、トピックの削除リクエストで変更が受け入れられるレートのクオータ。レートは、作成または削除されたパーティション数で累積されます。
producerByteRate integer	グループのクライアントにスロットリングが適用される前に、各クライアントグループがブローカーにパブリッシュできる最大 bps (ビット毎秒) のクオータ。ブローカーごとに定義されます。

プロパティー	説明
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	

12.2.106. KafkaUserTemplate スキーマ参照

KafkaUserSpec で使用

KafkaUserTemplate スキーマプロパティーの全リスト

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
  # ...
```

12.2.106.1. KafkaUserTemplate スキーマのプロパティー

プロパティー	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	

12.2.107. KafkaUserStatus スキーマ参照

KafkaUser で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
username	ユーザー名。
string	
secret	認証情報が保存される **Secret** の名前。
string	

12.2.108. 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。

プロパティー	説明
spec	Kafka MirrorMaker の仕様。
KafkaMirrorMakerSpec	
status	Kafka MirrorMaker のステータス。
KafkaMirrorMakerStatus	

12.2.109. KafkaMirrorMakerSpec スキーマ参照

KafkaMirrorMaker で使用

KafkaMirrorMakerSpec スキーマプロパティーの完全リスト

Kafka MirrorMaker を設定します。

12.2.109.1. include

include プロパティーを使用して、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。

このプロパティーでは、簡単な単一のトピック名から複雑なパターンまですべての正規表現が許可されます。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。

12.2.109.2. 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。

12.2.109.3. 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 プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。ConfigMap 内では、ロギング設定は **log4j.properties** を使用して記述されます。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティーはいずれも必須です。Cluster Operator の実行時に、指定された正確なロギング設定を使用する ConfigMap がカスタムリソースを使用して作成され、その後は調整のたびに再作成されます。カスタム ConfigMap を指定しない場合、デフォルトのロギング設定が使用されます。特定のロガー値が設定されていない場合、上位レベルのロガー設定がそのロガーに継承されます。ログレベルの詳細は、Apache logging services を参照してください。

inline および **external** ロギングの例は次のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: inline
    loggers:
      mirrormaker.root.logger: "INFO"
  # ...
```

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaMirrorMaker
spec:
  # ...
  logging:
    type: external
    valueFrom:
      configMapKeyRef:
```

```
      name: customConfigMap
      key: mirror-maker-log4j.properties
# ...
```

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.109.4. KafkaMirrorMakerSpec スキーマプロパティー

プロパティー	説明
version	Kafka MirrorMaker のバージョン。デフォルトは 3.3.1 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	
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 プロパティーは非推奨となり、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一覧。このオプションは、Java スタイルの正規表現を使用するあらゆる正規表現を許可します。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	

プロパティー	説明
include string	ミラーリングに含まれるトピックの一覧。このオプションは、Java スタイルの正規表現を使用するあらゆる正規表現を許可します。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。
jvmOptions **JvmOptions**	Pod の JVM オプション。
logging **InlineLogging**、**ExternalLogging**	MirrorMaker のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
metricsConfig **JmxPrometheusExporterMetrics**	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。
tracing **JaegerTracing**、**OpenTelemetryTracing**	Kafka MirrorMaker でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger, opentelemetry] の 1 つでなければなりません。
template **KafkaMirrorMakerTemplate**	Kafka MirrorMaker のリソースである **Deployments** および **Pods** の生成方法を指定するテンプレート。
livenessProbe **Probe**	Pod の liveness チェック。
readinessProbe **Probe**	Pod の readiness チェック。

12.2.110. KafkaMirrorMakerConsumerSpec スキーマ参照

KafkaMirrorMakerSpec で使用

KafkaMirrorMakerConsumerSpec スキーマプロパティーの完全リスト

MirrorMaker コンシューマーを設定します。

12.2.110.1. numStreams

consumer.numStreams プロパティーを使用して、コンシューマーのストリームの数を設定します。

コンシューマースレッドの数を増やすと、ミラーリングトピックのスループットを増やすことができます。コンシューマースレッドは、Kafka MirrorMaker に指定されたコンシューマーグループに属します。トピックパーティションはコンシューマースレッド全体に割り当てられ、メッセージが並行して消費されます。

12.2.110.2. offsetCommitInterval

consumer.offsetCommitInterval プロパティーを使用して、コンシューマーのオフセット自動コミット間隔を設定します。

Kafka MirrorMaker によってソース Kafka クラスターのデータが消費された後に、オフセットがコミットされる通常の間隔を指定できます。間隔はミリ秒単位で設定され、デフォルト値は 60,000 です。

12.2.110.3. config

consumer.config プロパティーを使用して、コンシューマーの Kafka オプションを設定します。

config プロパティーには、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** プロパティーを設定 することができます。また、 **ssl.endpoint.identification.algorithm** プロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

例外

コンシューマー向けの Apache Kafka 設定ドキュメント に記載されているオプションを指定および設定できます。

しかし、以下に関連する AMQ Streams によって自動的に設定され、直接管理されるオプションには例外があります。

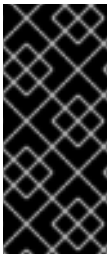
- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- コンシューマーグループ ID
- インターセプター

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** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka MirrorMaker に渡されます。



重要

Cluster Operator では、提供された **config** オブジェクトのキーまたは値は検証されません。無効な設定が指定されると、Kafka MirrorMaker が起動しなかったり、不安定になったりする場合があります。このような場合、**KafkaMirrorMaker.spec.consumer.config** オブジェクトの設定を修正し、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。

12.2.110.4. groupId

consumer.groupId プロパティーを使用して、コンシューマーにコンシューマーグループ ID を設定します。

Kafka MirrorMaker は Kafka コンシューマーを使用してメッセージを消費し、他の Kafka コンシューマークライアントと同様に動作します。ソース Kafka クラスターから消費されるメッセージは、ターゲット Kafka クラスターにミラーリングされます。パーティションの割り当てには、コンシューマーがコンシューマーグループの一部である必要があるため、グループ ID が必要です。

12.2.110.5. KafkaMirrorMakerConsumerSpec スキーマプロパティー

プロパティー	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 ペアの一覧。
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	

プロパティー	説明
authentication	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls, scram-sha-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 コンシューマーの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、bootstrap.servers、group.id、sasl.、security.、interceptor.classes (次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols).
map	
tls	MirrorMaker をクラスターに接続するための TLS 設定。
ClientTls	

12.2.111. KafkaMirrorMakerProducerSpec スキーマ参照

KafkaMirrorMakerSpec で使用

KafkaMirrorMakerProducerSpec スキーマプロパティーの完全リスト

MirrorMaker プロデューサーを設定します。

12.2.111.1. abortOnSendFailure

producer.abortOnSendFailure プロパティーを使用して、プロデューサーからメッセージ送信の失敗を処理する方法を設定します。

デフォルトでは、メッセージを Kafka MirrorMaker から Kafka クラスターに送信する際にエラーが発生した場合、以下が行われます。

- Kafka MirrorMaker コンテナーが OpenShift で終了します。
- その後、コンテナーが再作成されます。

abortOnSendFailure オプションを **false** に設定した場合、メッセージ送信エラーは無視されます。

12.2.111.2. config

producer.config プロパティーを使用して、プロデューサーの Kafka オプションを設定します。

config プロパティーには、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** プロパティーを設定 することができます。また、 **ssl.endpoint.identification.algorithm** プロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

例外

プロデューサー向けの Apache Kafka 設定ドキュメント に記載されているオプションを指定および設定できます。

しかし、以下に関連する AMQ Streams によって自動的に設定され、直接管理されるオプションには例外があります。

- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- インターセプター

以下の文字列の 1 つと同じキーまたは以下の文字列の 1 つで始まるキーを持つ設定オプションはすべて禁止されています。

- **bootstrap.servers**
- **interceptor.classes**
- **ssl.** (特定の例外は除外)
- **sasl.**
- **security.**

禁止されているオプションが **config** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka MirrorMaker に渡されます。

重要

Cluster Operator では、提供された **config** オブジェクトのキーまたは値は検証されません。無効な設定が指定されると、Kafka MirrorMaker が起動しなかったり、不安定になったりする場合があります。このような場合、**KafkaMirrorMaker.spec.producer.config** オブジェクトの設定を修正し、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。

12.2.111.3. KafkaMirrorMakerProducerSpec スキーマプロパティー

プロパティー	説明
bootstrapServers	Kafka クラスターへの最初の接続を確立するための host:port ペアの一覧。
string	

プロパティー	説明
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	
authentication	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls, scram-sha-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ーは設定できません: ssl.、bootstrap.servers、sasl.、security.、interceptor.classes (次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols).
map	
tls	MirrorMaker をクラスターに接続するための TLS 設定。
ClientTls	

12.2.112. KafkaMirrorMakerTemplate スキーマ参照

KafkaMirrorMakerSpec で使用

プロパティー	説明
deployment	Kafka MirrorMaker **Deployment** のテンプレート。
DeploymentTemplate	
Pod	Kafka MirrorMaker **Pods** のテンプレート。
PodTemplate	
podDisruptionBudget	Kafka MirrorMaker **PodDisruptionBudget** のテンプレート。
PodDisruptionBudgetTemplate	
mirrorMakerContainer	Kafka MirrorMaker コンテナーのテンプレート。
ContainerTemplate	

プロパティー	説明
serviceAccount	Kafka MirrorMaker サービスアカウントのテンプレート。
ResourceTemplate	

12.2.113. KafkaMirrorMakerStatus スキーマ参照

KafkaMirrorMaker で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	
labelSelector	このリソースを提供する Pod のラベルセレクター。
string	
replicas	このリソースを提供するために現在使用されている Pod の数。
integer	

12.2.114. KafkaBridge スキーマ参照

プロパティー	説明
spec	Kafka Bridge の仕様。
KafkaBridgeSpec	
status	Kafka Bridge のステータス。
KafkaBridgeStatus	

12.2.115. KafkaBridgeSpec スキーマ参照

KafkaBridge で使用

KafkaBridgeSpec スキーマプロパティーの全リスト

Kafka Bridge クラスターを設定します。

設定オプションは以下に関連しています。

- Kafka クラスターブートストラップアドレス
- セキュリティー (暗号化、認証、および承認)
- コンシューマー設定
- プロデューサーの設定
- HTTP の設定

12.2.115.1. 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 プロパティーを使用してロガーおよびロガーレベルを設定します。

ログレベルを設定するには、ロガーとレベルを直接指定 (インライン) するか、またはカスタム (外部) ConfigMap を使用します。ConfigMap を使用する場合、**logging.valueFrom.configMapKeyRef.name** プロパティーを外部ロギング設定が含まれる ConfigMap の名前に設定します。**logging.valueFrom.configMapKeyRef.name** および **logging.valueFrom.configMapKeyRef.key** プロパティー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。

ガベッジコレクター (GC)

ガベッジコレクターのロギングは **jvmOptions** プロパティーを使用して 有効 (または無効) にすることもできます。

12.2.115.2. KafkaBridgeSpec スキーマプロパティー

プロパティー	説明
replicas	**Deployment** の Pod 数。
integer	
image	Pod の Docker イメージ。
string	
bootstrapServers	Kafka クラスターへの最初の接続を確立するための host:port ペアの一覧。
string	
tls	Kafka Bridge をクラスターに接続するための TLS 設定。
ClientTls	
authentication	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls, scram-sha-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。

プロパティー	説明
KafkaBridgeAdminClientSpec	
consumer **KafkaBridgeConsumerSpec**	Kafka コンシューマーに関連する設定。
producer **KafkaBridgeProducerSpec**	Kafka プロデューサーに関連する設定。
resources ResourceRequirements	予約する CPU およびメモリーリソース。詳細は、core/v1 resourcerequirements の外部ドキュメント を参照してください。
jvmOptions **JvmOptions**	**現時点でサポートされていない** Pod の JVM オプション。
logging **InlineLogging**、**ExternalLogging**	Kafka Bridge のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
clientRackInitImage string	**client.rack** の初期化に使用される init コンテナーのイメージです。
rack **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。
enableMetrics boolean	Kafka Bridge のメトリクスを有効にします。デフォルトは false です。
livenessProbe **Probe**	Pod の liveness チェック。
readinessProbe **Probe**	Pod の readiness チェック。

プロパティー	説明
template **KafkaBridgeTemplate**	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。
tracing **JaegerTracing**、**OpenTelemetryTracing**	Kafka Bridge でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger, opentelemetry] の 1 つでなければなりません。

12.2.116. KafkaBridgeHttpConfig スキーマ参照

KafkaBridgeSpec で使用

KafkaBridgeHttpConfig スキーマプロパティーの完全リスト

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。

12.2.116.1. cors

HTTP プロパティーは、Kafka クラスターへの HTTP アクセスを有効にする他に、CPRS (Cross-Origin Resource Sharing) により Kafka Bridge のアクセス制御を有効化または定義する機能を提供します。CORS は、複数のオリジンから指定のリソースにブラウザーでアクセスできるようにする HTTP メカニズムです。CORS を設定するには、許可されるリソースオリジンのリストと、HTTP のアクセス方法を定義します。オリジンには、URL または Java 正規表現を使用できます。

Kafka Bridge HTTP の設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  http:
    port: 8080
    cors:
      allowedOrigins: "https://strimzi.io"
      allowedMethods: "GET,POST,PUT,DELETE,OPTIONS,PATCH"
  # ...
```

12.2.116.2. KafkaBridgeHttpConfig スキーマプロパティー

プロパティー	説明
port integer	サーバーがリッスンするポート。
cors **KafkaBridgeHttpCors**	HTTP Bridge の CORS 設定。

12.2.117. KafkaBridgeHttpCors スキーマ参照

KafkaBridgeHttpConfig で使用されます。

プロパティー	説明
allowedOrigins string array	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。
allowedMethods string array	許可される HTTP メソッドのリスト。

12.2.118. KafkaBridgeAdminClientSpec スキーマプロパティー

KafkaBridgeSpec で使用

プロパティー	説明
config map	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。

12.2.119. KafkaBridgeConsumerSpec スキーマ参照

KafkaBridgeSpec で使用

KafkaBridgeConsumerSpec スキーマプロパティーの完全リスト

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。

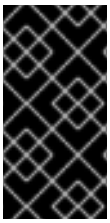
- 文字列
- 数値

- 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** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka 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** プロパティーを設定 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
    # ...
```

12.2.119.1. KafkaBridgeConsumerSpec スキーマプロパティー

プロパティー	説明

プロパティー	説明
config map	ブリッジによって作成されたコンシューマーインスタンスに使用される Kafka コンシューマーの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、bootstrap.servers、group.id、sasl.、security.(次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols).

12.2.120. KafkaBridgeProducerSpec スキーマ参照

KafkaBridgeSpec で使用

KafkaBridgeProducerSpec スキーマプロパティーの完全リスト

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。

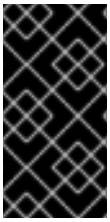
値は以下の JSON タイプのいずれかになります。

- 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** プロパティーにある場合、そのオプションは無視され、警告メッセージが Cluster Operator ログファイルに出力されます。その他のオプションはすべて Kafka 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** プロパティーを設定 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
    # ...
```

12.2.120.1. KafkaBridgeProducerSpec スキーマプロパティー

プロパティー	説明
config	ブリッジによって作成されたプロデューサーインスタンスに使用される Kafka プロデューサーの設定。次の接頭辞を持つプロパティーは設定できません: ssl.、bootstrap.servers、sasl.、security. (次の例外を除く: ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols).
map	

12.2.121. KafkaBridgeTemplate スキーマ参照

KafkaBridgeSpec で使用

プロパティー	説明
deployment	Kafka Bridge **Deployment** のテンプレート。
DeploymentTemplate	
Pod	Kafka Bridge **Pod** のテンプレート。
PodTemplate	
apiService	Kafka Bridge API **Service** のテンプレート。
InternalServiceTemplate	

プロパティー	説明
podDisruptionBudget **PodDisruptionBudgetTemplate**	Kafka Bridge **PodDisruptionBudget** のテンプレート。
bridgeContainer **ContainerTemplate**	Kafka Bridge コンテナーのテンプレート。
serviceAccount **ResourceTemplate**	Kafka Bridge サービスアカウントのテンプレート。
initContainer **ContainerTemplate**	Kafka Bridge init コンテナーのテンプレート。

12.2.122. KafkaBridgeStatus スキーマ参照

KafkaBridge で使用

プロパティー	説明
conditions **Condition** array	ステータス条件の一覧。
observedGeneration integer	最後に Operator によって調整された CRD の生成。
url string	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。
labelSelector string	このリソースを提供する Pod のラベルセレクター。
replicas integer	このリソースを提供するために現在使用されている Pod の数。

12.2.123. KafkaConnector スキーマ参照

プロパティー	説明
spec	Kafka Connector の仕様。
KafkaConnectorSpec	
status	Kafka Connector のステータス。
KafkaConnectorStatus	

12.2.124. KafkaConnectorSpec スキーマ参照

KafkaConnector で使用

プロパティー	説明
class	Kafka Connector のクラス。
string	
tasksMax	Kafka Connector のタスクの最大数。
integer	
config	Kafka Connector の設定。次のプロパティーは設定できません: connector.class、tasks.max
map	
pause	コネクターを一時停止すべきかどうか。デフォルトは false です。
boolean	

12.2.125. KafkaConnectorStatus スキーマ参照

KafkaConnector で使用

プロパティー	説明
conditions	ステータス条件の一覧。
Condition array	
observedGeneration	最後に Operator によって調整された CRD の生成。
integer	

プロパティー	説明
connectorStatus	Kafka Connect REST API によって報告されるコネクターのステータス。
map	
tasksMax	Kafka Connector のタスクの最大数。
integer	
topics	Kafka Connector によって使用されるトピックのリスト。
string array	

12.2.126. KafkaMirrorMaker2 スキーマ参照

プロパティー	説明
spec	Kafka MirrorMaker 2.0 クラスターの仕様。
KafkaMirrorMaker2Spec	
status	Kafka MirrorMaker 2.0 クラスターのステータス。
KafkaMirrorMaker2Status	

12.2.127. KafkaMirrorMaker2Spec スキーマ参照

KafkaMirrorMaker2 で使用

プロパティー	説明
version	Kafka Connect のバージョン。デフォルトは 3.3.1 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	

プロパティー	説明
connectCluster string	Kafka Connect に使用されるクラスターエイリアス。エイリアスは **spec.clusters** にある一覧のクラスターと一致する必要があります。
clusters **KafkaMirrorMaker2ClusterSpec** array	ミラーリング用の Kafka クラスター。
mirrors **KafkaMirrorMaker2MirrorSpec** array	MirrorMaker 2.0 コネクターの設定。
resources ResourceRequirements	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。
livenessProbe **Probe**	Pod の liveness チェック。
readinessProbe **Probe**	Pod の readiness チェック。
jvmOptions **JvmOptions**	Pod の JVM オプション。
jmxOptions **KafkaJmxOptions**	JMX オプション。
logging **InlineLogging**、**ExternalLogging**	Kafka Connect のロギング設定。タイプは、指定のオブジェクト内の **logging.type** プロパティーの値によって異なり、[inline、external] のいずれかでなければなりません。
clientRackInitImage string	**client.rack** の初期化に使用される init コンテナーのイメージです。
rack **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。

プロパティー	説明
tracing **JaegerTracing**、**OpenTelemetryTracing**	Kafka Connect でのトレースの設定。タイプは、指定のオブジェクト内の **tracing.type** プロパティーの値によって異なり、[jaeger, opentelemetry] の 1 つでなければなりません。
template **KafkaConnectTemplate**	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。
externalConfiguration **ExternalConfiguration**	Secret または ConfigMap から Kafka Connect Pod にデータを渡し、これを使用してコネクターを設定します。
metricsConfig **JmxPrometheusExporterMetrics**	メトリクスの設定。タイプは、指定のオブジェクト内の **metricsConfig.type** プロパティーの値によって異なり、[jmxPrometheusExporter] のいずれかでなければなりません。

12.2.128. KafkaMirrorMaker2ClusterSpec スキーマ参照

KafkaMirrorMaker2Spec で使用

KafkaMirrorMaker2ClusterSpec スキーマプロパティーの完全リスト

ミラーリング用の Kafka クラスターを設定します。

12.2.128.1. config

Kafka のオプションを設定するには、**config** プロパティーを使用します。

標準の Apache Kafka 設定が提供されることがありますが、AMQ Streams によって直接管理されないプロパティーに限定されます。

TLS バージョンの特定の **暗号スイート** を使用するクライアント接続に、許可された **ssl** プロパティーを設定 することができます。また、 **ssl.endpoint.identification.algorithm** プロパティーを設定して、ホスト名の検証を有効または無効にすることもできます。

12.2.128.2. KafkaMirrorMaker2ClusterSpec スキーマプロパティー

プロパティー	説明
alias string	Kafka クラスターの参照に使用されるエイリアス。
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。

プロパティー	説明
string	
tls	MirrorMaker 2.0 コネクターをクラスターに接続するための TLS 設定。
ClientTls	
authentication	クラスターに接続するための認証設定。タイプは、指定のオブジェクト内の **authentication.type** プロパティーの値によって異なり、[tls, scram-sha-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ーは設定できません: ssl.、sasl.、security.、listeners、plugin.path、rest.、bootstrap.servers、consumer.interceptor.classes、producer.interceptor.classes (ssl.endpoint.identification.algorithm、ssl.cipher.suites、ssl.protocol、ssl.enabled.protocols を除く)
map	

12.2.129. KafkaMirrorMaker2MirrorSpec スキーマ参照

KafkaMirrorMaker2Spec で使用

プロパティー	説明
sourceCluster	Kafka MirrorMaker 2.0 コネクターによって使用されるソースクラスターのエイリアス。エイリアスは **spec.clusters** にある一覧のクラスターと一致する必要があります。
string	
targetCluster	Kafka MirrorMaker 2.0 コネクターによって使用されるターゲットクラスターのエイリアス。エイリアスは **spec.clusters** にある一覧のクラスターと一致する必要があります。
string	
sourceConnector	Kafka MirrorMaker 2.0 ソースコネクターの仕様。
KafkaMirrorMaker2ConnectorSpec	
heartbeatConnector	Kafka MirrorMaker 2.0 ハートビートコネクターの仕様。
KafkaMirrorMaker2ConnectorSpec	

プロパティー	説明
checkpointConnector	Kafka MirrorMaker 2.0 チェックポイントコネクターの仕様。
KafkaMirrorMaker2ConnectorSpec	
topicsPattern	ミラーリングするトピックに一致する正規表現 (例: "topic1|topic2|topic3")。コンマ区切りリストもサポートされます。
string	
topicsBlacklistPattern	**topicsBlacklistPattern プロパティーは非推奨となり、.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 プロパティーは非推奨となり、.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	

12.2.130. KafkaMirrorMaker2ConnectorSpec スキーマ参照

KafkaMirrorMaker2MirrorSpec で使用

プロパティー	説明
tasksMax	Kafka Connector のタスクの最大数。
integer	
config	Kafka Connector の設定。次のプロパティーは設定できません: connector.class、tasks.max
map	

プロパティー	説明
pause boolean	コネクターを一時停止すべきかどうか。デフォルトは false です。

12.2.131. KafkaMirrorMaker2Status スキーマ参照

KafkaMirrorMaker2 で使用

プロパティー	説明
conditions **Condition** array	ステータス条件の一覧。
observedGeneration integer	最後に Operator によって調整された CRD の生成。
url string	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。
connectorPlugins **ConnectorPlugin** array	この Kafka Connect デプロイメントで使用できるコネクタープラグインの一覧。
connectors map array	Kafka Connect REST API によって報告される MirrorMaker 2.0 コネクターステータスの一覧。
labelSelector string	このリソースを提供する Pod のラベルセレクター。
replicas integer	このリソースを提供するために現在使用されている Pod の数。

12.2.132. KafkaRebalance スキーマ参照

プロパティー	説明
spec	Kafka のリバランス (再分散) の仕様。

プロパティー	説明
KafkaRebalanceSpec	
status	Kafka のリバランス (再分散) のステータス。
KafkaRebalanceStatus	

12.2.133. KafkaRebalanceSpec スキーマ参照

KafkaRebalance で使用されます。

プロパティー	説明
mode 文字列 (remove-brokers、full、add-brokers のいずれか)	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。 ● **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。
brokers 整数配列	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。
goals string array	リバランスプロポーザルの生成および実行に使用されるゴールのリスト (優先度順)。サポートされるゴールは https://github.com/linkedin/cruise-control#goals を参照してください。空のゴールリストを指定すると、default.goals Cruise Control 設定パラメーターに宣言されたゴールが使用されます。
skipHardGoalCheck boolean	最適化プロポーザルの生成で、Kafka CR に指定されたハードゴールのスキップを許可するかどうか。これは、これらのハードゴールの一部が原因で分散ソリューションが検索できない場合に便利です。デフォルトは false です。

プロパティー	説明
rebalanceDisk boolean	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。
excludedTopics string	一致するトピックが最適化プロポーザルの計算から除外される正規表現。この正規表現は java.util.regex.Pattern クラスによって解析されます。サポートされる形式の詳細は、このクラスのドキュメントを参照してください。
concurrentPartitionMovementsPerBroker integer	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。
concurrentIntraBrokerPartitionMovements integer	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。
concurrentLeaderMovements integer	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。
replicationThrottle integer	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。
replicaMovementStrategies string array	生成された最適化プロポーザルでのレプリカ移動の実行順序を決定するために使用されるストラテジークラス名のリスト。デフォルトでは、生成された順序でレプリカの移動が実行される BaseReplicaMovementStrategy が使用されます。

12.2.134. KafkaRebalanceStatus スキーマ参照

KafkaRebalance で使用されます。

プロパティー	説明
conditions **Condition** array	ステータス条件の一覧。

プロパティー	説明
observedGeneration integer	最後に Operator によって調整された CRD の生成。
sessionId string	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。
optimizationResult map	最適化の結果を示す JSON オブジェクト。

付録A サブスクリプションの使用

AMQ Streams は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **サブスクリプション** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合は、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **製品のダウンロード** ページにログインします。
2. **インテグレーションおよび自動化** カテゴリーで、**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 **ダウンロード** 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: 2023-04-06