



Red Hat AMQ Streams 2.2

AMQ Streams on OpenShift のデプロイおよび アップグレード

OpenShift Container Platform での AMQ Streams 2.2 のデプロイ

Red Hat AMQ Streams 2.2 AMQ Streams on OpenShift のデプロイおよびアップグレード

OpenShift Container Platform での AMQ Streams 2.2 のデプロイ

法律上の通知

Copyright © 2023 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

概要

OperatorHub またはインストールアーティファクトを使用して、AMQ Streams を OpenShift クラスタにデプロイします。Kafka コンポーネントをデプロイおよび管理する AMQ Streams Cluster Operator を使用します。AMQ Streams をアップグレードして新機能を活用します。アップグレードの一環として、Kafka を最新のサポート対象バージョンにアップグレードします。

目次

多様性を受け入れるオープンソースの強化	4
第1章 デプロイメントの概要	5
1.1. デプロイメントの設定	5
1.2. AMQ STREAMS のカスタムリソース	6
1.3. KAFKA BRIDGE を使用した KAFKA クラスターへの接続	9
1.4. 本書の表記慣例	9
1.5. 関連情報	9
第2章 AMQ STREAMS のインストール方法	10
第3章 AMQ STREAMS でデプロイされる内容	11
3.1. デプロイメントの順序	11
第4章 AMQ STREAMS デプロイメントの準備	12
4.1. デプロイメントの前提条件	12
4.2. AMQ STREAMS リリースアーティファクトのダウンロード	12
4.3. 設定ファイルとデプロイメントファイルの例	12
4.4. コンテナイメージの独自のレジストリーへのプッシュ	14
4.5. AMQ STREAMS の管理者の指定	15
第5章 WEB コンソールを使用した OPERATORHUB からの AMQ STREAMS のインストール	17
5.1. RED HAT INTEGRATION OPERATOR を使用した AMQ STREAMS OPERATOR のインストール	17
5.2. OPERATORHUB からの AMQ STREAMS OPERATOR のインストール	17
5.3. AMQ STREAMS OPERATOR を使用した KAFKA コンポーネントのデプロイ	19
第6章 インストールアーティファクトを使用した AMQ STREAMS のデプロイ	21
6.1. KAFKA クラスターの作成	21
6.2. KAFKA CONNECT のデプロイ	38
6.3. KAFKA MIRRORMAKER のデプロイ	53
6.4. KAFKA BRIDGE のデプロイ	54
第7章 KAFKA クラスターへのクライアントアクセスの設定	57
7.1. サンプルクライアントのデプロイ	57
7.2. OPENSIFT 外クライアントのアクセスの設定	57
第8章 AMQ STREAMS のメトリクスおよびダッシュボードの設定	64
8.1. KAFKA EXPORTER でのコンシューマーラグの監視	65
8.2. CRUISE CONTROL 操作の監視	66
8.3. メトリクスファイルの例	68
8.4. PROMETHEUS メトリクス設定のデプロイ	71
8.5. OPENSIFT での KAFKA メトリックおよびダッシュボードの表示	75
第9章 AMQ STREAMS のアップグレード	83
9.1. AMQ STREAMS のアップグレードパス	83
9.2. 必要なアップグレードシーケンス	84
9.3. 最小限のダウンタイムでの OPENSIFT のアップグレード	84
9.4. CLUSTER OPERATOR のアップグレード	86
9.5. KAFKA のアップグレード	89
9.6. コンシューマーの COOPERATIVE REBALANCING へのアップグレード	97
第10章 AMQ STREAMS のダウングレード	99
10.1. CLUSTER OPERATOR の以前のバージョンへのダウングレード	99
10.2. KAFKA のダウングレード	100

第11章 AMQ STREAMS のアンインストール	104
11.1. WEB コンソールを使用した OPERATORHUB からの AMQ STREAMS のアンインストール	104
11.2. CLI を使用した AMQ STREAMS のアンインストール	105
第12章 AMQ STREAMS でのメータリングの使用	107
12.1. メータリングリソース	107
12.2. AMQ STREAMS のメータリングラベル	107
付録A サブスクリプションの使用	110
アカウントへのアクセス	110
サブスクリプションのアクティベート	110
Zip および Tar ファイルのダウンロード	110
DNF を使用したパッケージのインストール	110

多様性を受け入れるオープンソースの強化

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

第1章 デプロイメントの概要

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 のデプロイとアップグレードに使用できるすべてのオプションについて取り上げ、さらに、デプロイメントの対象や、OpenShift クラスターで Apache Kafka を実行するために必要なデプロイメントの順序について説明します。

デプロイメントの手順を説明する他に、デプロイメントを準備および検証するためのデプロイメントの前および後の手順についても説明します。本ガイドでは、メトリクスを導入するための追加のデプロイメントオプションについても説明しています。

アップグレードの手順は、AMQ Streams および Kafka のアップグレードを参照してください。

AMQ Streams は、パブリックおよびプライベートクラウドから開発を目的とするローカルデプロイメントまで、ディストリビューションに関係なく、すべてのタイプの OpenShift クラスターで動作するように設計されています。

1.1. デプロイメントの設定

本ガイドのデプロイメント手順は、デプロイメントの初期構造の設定に役立つように設計されています。構造の設定後、カスタムリソースを使用して、正確なニーズに合わせてデプロイメントを設定できます。デプロイメント手順では、AMQ Streams に同梱されているインストールファイルのサンプルを使用します。この手順では、設定に関する重要な考慮事項について主に触れていますが、使用可能なすべての設定オプションについて説明するわけではありません。

AMQ Streams をデプロイする前に、Kafka コンポーネントに使用できる設定オプションを確認することを推奨します。設定オプションの詳細は、[OpenShift での AMQ Streams の設定](#)を参照してください。

1.1.1. Kafka のセキュリティー

Cluster Operator はデプロイ時に、クラスター内のデータ暗号化と認証に対して TLS 証明書を自動設定します。

AMQ Streams では、**暗号化**、**認証**、および **承認** の追加の設定オプションが提供されます。

- [Kafka へのセキュアなアクセスの管理](#) を行い、Kafka クラスターとクライアント間のデータ交換のセキュリティーを確保します。
- 承認サーバーが [OAuth 2.0 認証](#) および [OAuth 2.0 承認](#) を使用するように、デプロイメントを設定します。
- [独自の証明書を使用して Kafka をセキュア](#) にします。

1.1.2. デプロイメントの監視

AMQ Streams は、デプロイメントを監視する追加のデプロイメントオプションをサポートします。

- [Prometheus](#) および [Grafana](#) を [Kafka クラスターでデプロイ](#) し、メトリックを抽出して、Kafka コンポーネントを監視します。
- [Kafka Exporter](#) を [Kafka クラスターでデプロイ](#) し、特にコンシューマーラグの監視に関する追加のメトリクスを抽出します。

- [分散トレーシングを設定](#) し、エンドツーエンドのメッセージ追跡を行います。

1.1.3. CPU およびメモリーのリソース制限および要求

デフォルトでは、AMQ Streams Cluster Operator はデプロイするオペランドの CPU およびメモリーリソースの要求および制限を指定しません。

Kafka などのアプリケーションでリソースの安定性を確保してパフォーマンスを向上させるには、十分なリソースを確保する必要があります。

使用する適切なリソース量は、特定の要件やユースケースによって異なります。

CPU およびメモリーリソースの設定を検討してください。[AMQ Streams カスタムリソース](#) の各コンテナのリソース要求および制限を設定できます。

1.2. AMQ STREAMS のカスタムリソース

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

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

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

1.2.1. AMQ Streams カスタムリソースの例

AMQ Streams 固有のリソースのインスタンス化と管理に使用されるスキーマを定義するには、CRD をクラスターに1回だけインストールする必要があります。

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

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



注記

カスタムリソースの管理は、AMQ Streams 管理者のみが行えます。詳細は、[AMQ Streams 管理者の指定](#) を参照してください。

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

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



警告

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

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

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

Kafka トピックの CRD

```

apiVersion: kafka.strimzi.io/v1beta2
kind: CustomResourceDefinition
metadata: ❶
  name: kafkatopics.kafka.strimzi.io
  labels:
    app: strimzi
spec: ❷
  group: kafka.strimzi.io
  versions:
    v1beta2
  scope: Namespaced
  names:
    # ...
    singular: kafkatopic
    plural: kafkatopics
    shortNames:
      - kt ❸
  additionalPrinterColumns: ❹
    # ...
  subresources:
    status: {} ❺
  validation: ❻
  openAPIV3Schema:
    properties:
      spec:
        type: object
        properties:
          partitions:
            type: integer
            minimum: 1
          replicas:
            type: integer
            minimum: 1
            maximum: 32767
    # ...

```

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



注記

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

KafkaTopic カスタムリソースに該当する例は次のとおりです。

Kafka トピックカスタムリソース

```

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

```

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

トファイルサイズが指定されています。

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

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

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

関連情報

- [Extend the Kubernetes API with CustomResourceDefinitions](#)
- [設定ファイルとデプロイメントファイルの例](#)

1.3. KAFKA BRIDGE を使用した KAFKA クラスターへの接続

AMQ Streams Kafka Bridge API を使用して、コンシューマーを作成および管理し、ネイティブ Kafka プロトコルではなく HTTP を介してレコードを送受信できます。

Kafka Bridge を設定する場合、Kafka クラスターへの HTTP アクセスを設定します。その後、Kafka Bridge を使用して、クラスターからのメッセージを生成および消費したり、REST インターフェイスを介して他の操作を実行することができます。

関連情報

- Kafka Bridge のインストールおよび使用に関する詳細は、[AMQ Streams Kafka Bridge の使用](#) を参照してください。

1.4. 本書の表記慣例

ユーザー置換値

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

たとえば、以下のコードでは `<my_namespace>` を namespace の名前に置き換えます。

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

1.5. 関連情報

- [AMQ Streams の概要](#)
- [AMQ Streams の設定](#)
- [AMQ Streams Kafka Bridge の使用](#)

第2章 AMQ STREAMS のインストール方法

AMQ Streams を OpenShift 4.8 から 4.11 にインストールする方法は 2 つあります。

インストール方法	説明
インストールアーティファクト (YAML ファイル)	<p>AMQ Streams ソフトウェアダウンロードページ から、Red Hat AMQ Streams 2.2 OpenShift インストールおよびサンプルファイル をダウンロードします。oc を使用して YAML インストールアーティファクトを OpenShift クラスタにデプロイします。最初に、Cluster Operator を install/cluster-operator から単一、複数、またはすべての namespace にデプロイします。</p> <p>install/ アーティファクトを使用して以下をデプロイすることもできます。</p> <ul style="list-style-type: none"> ● AMQ Streams 管理者ロール (strimzi-admin) ● スタンドアロン Topic Operator (topic-operator) ● スタンドアロン User Operator (user-operator) ● AMQ Streams Drain Cleaner (drain-cleaner)
OperatorHub	<p>OperatorHub で Red Hat Integration - AMQ Streams を使用し、AMQ Streams を単一の namespace またはすべての namespace にデプロイします。</p>

できるだけ柔軟性を確保するには、アーティファクトのインストール方法を選択してください。OperatorHub メソッドは標準的な設定を提供し、自動更新を活用できるようにします。

第3章 AMQ STREAMS でデプロイされる内容

Apache Kafka コンポーネントは、AMQ Streams ディストリビューションを使用して OpenShift にデプロイする場合に提供されます。Kafka コンポーネントは通常、クラスターとして実行され、可用性を確保します。

Kafka コンポーネントが組み込まれた通常のデプロイメントには以下が含まれます。

- ブローカーノードの **Kafka** クラスター
- レプリケートされた ZooKeeper インスタンスの **zookeeper** クラスター
- 外部データ接続用の **Kafka Connect** クラスター
- セカンダリークラスターで Kafka クラスターをミラーリングする **Kafka MirrorMaker** クラスター
- 監視用に追加の Kafka メトリックデータを抽出する **Kafka Exporter**
- Kafka クラスターに対して HTTP ベースの要求を行う **Kafka Bridge**

少なくとも Kafka および ZooKeeper は必要ですが、上記のコンポーネントがすべて必須なわけではありません。MirrorMaker や Kafka Connect など、一部のコンポーネントでは Kafka なしでデプロイできます。

3.1. デプロイメントの順序

OpenShift クラスターへのデプロイで必要とされる順序は、次のとおりです。

1. Cluster Operator をデプロイし、Kafka クラスターを管理します。
2. ZooKeeper クラスターとともに Kafka クラスターをデプロイし、Topic Operator および User Operator がデプロイメントに含まれるようにします。
3. 任意で以下をデプロイします。
 - Topic Operator および User Operator (Kafka クラスターとともにデプロイしなかった場合)
 - Kafka Connect
 - Kafka MirrorMaker
 - Kafka Bridge
 - メトリクスを監視するためのコンポーネント

Cluster Operator は、**Deployment**、**Service**、および **Pod** リソースなど、コンポーネントの OpenShift リソースを作成します。OpenShift リソース名には、デプロイ時にコンポーネントに指定された名前が追加されます。たとえば、**my-kafka-cluster** という名前の Kafka クラスターには、**my-kafka-cluster-kafka** という名前のサービスがあります。

第4章 AMQ STREAMS デプロイメントの準備

ここでは、AMQ Streams デプロイメントを準備する方法を説明します。

- [AMQ Streams をデプロイする前に必要となる前提条件](#)
- [デプロイメントで使用する AMQ Streams リリースアーティファクトのダウンロード方法](#)
- [AMQ Streams コンテナイメージを独自のレジストリーにプッシュする方法 \(必要な場合\)](#)
- [デプロイメントで使用されるカスタムリソースの設定に `admin` ロールを設定する方法](#)



注記

本ガイドのコマンドを実行するには、クラスターユーザーに RBAC (ロールベースアクセス制御) および CRD を管理する権限を付与する必要があります。

4.1. デプロイメントの前提条件

AMQ Streams をデプロイするには、以下が必要です。

- OpenShift 4.8 から 4.11 クラスター。
AMQ Streams は AMQ Streams Strimzi 0.29.x をベースとしている。
- `oc` コマンドラインツールがインストールされ、稼働中のクラスターに接続するように設定されている。

4.2. AMQ STREAMS リリースアーティファクトのダウンロード

デプロイメントファイルを使用して AMQ Streams をインストールするには、[AMQ Streams ソフトウェアダウンロードページ](#) からファイルをダウンロードしてデプロイメントします。

AMQ Streams のリリースアーティファクトには、YAML ファイルが含まれています。これらのファイルは、AMQ Streams コンポーネントの OpenShift へのデプロイ、共通の操作の実行、および Kafka クラスターの設定に便利です。

`oc` を使用して、ダウンロードした ZIP ファイルの `install/cluster-operator` フォルダーから Cluster Operator をデプロイします。Cluster Operator のデプロイメントおよび設定に関する詳細は、「[Cluster Operator のデプロイ](#)」を参照してください。

また、AMQ Streams Cluster Operator によって管理されない Kafka クラスターをトピックおよび User Operator のスタンドアロンインストールと共に使用する場合は、`install/topic-operator` および `install/user-operator` フォルダーからデプロイできます。



注記

AMQ Streams コンテナイメージは、[Red Hat Ecosystem Catalog](#) から使用することもできます。ただし、指定の YAML ファイルを使用して AMQ Streams をデプロイすることを推奨します。

4.3. 設定ファイルとデプロイメントファイルの例

AMQ Streams で提供される設定およびデプロイメントファイルの例を使用して、異なる設定で Kafka コンポーネントをデプロイし、デプロイメントを監視します。カスタムリソースの設定ファイルの例に

は、重要なプロパティおよび値が含まれています。これは、独自のデプロイメントでサポートされる追加の設定プロパティで拡張できます。

4.3.1. ファイルの場所の例

サンプルファイルは、[AMQ Streams ソフトウェアダウンロードページ](#) からダウンロード可能なリリースアーティファクトとともに提供されます。

oc コマンドラインツールを使用してサンプルをダウンロードおよび適用できます。これらの例は、デプロイメントに独自の Kafka コンポーネント設定を構築する際の開始点として使用できます。



注記

Operator を使用して AMQ Streams をインストールした場合でも、サンプルファイルをダウンロードして、そのファイルを使用して設定をアップロードできます。

4.3.2. AMQ Streams で提供されるサンプルファイル

リリースアーティファクトには、**examples** ディレクトリーがあり、そこに設定例が含まれています。

Examples ディレクトリー

```
examples
├── user ①
├── topic ②
├── security ③
│   ├── tls-auth
│   ├── scram-sha-512-auth
│   └── keycloak-authorization
├── mirror-maker ④
├── metrics ⑤
├── kafka ⑥
├── cruise-control ⑦
├── connect ⑧
└── bridge ⑨
```

- ① User Operator によって管理される **KafkaUser** カスタムリソース設定。
- ② Topic Operator によって管理される **KafkaTopic** カスタムリソースの設定。
- ③ Kafka コンポーネントの認証および承認設定。TLS および SCRAM-SHA-512 認証の設定例が含まれています。Red Hat Single Sign-On の例には、**Kafka** カスタムリソース設定および Red Hat Single Sign-On レルム仕様が含まれています。この例を使用して、Red Hat Single Sign-On 承認サービスを試すことができます。
- ④ Mirror Maker のデプロイメント用の **Kafka** カスタムリソース設定。レプリケーションポリシーおよび同期頻度の設定例が含まれます。
- ⑤ Prometheus インストールおよび Grafana ダッシュボードファイルが含まれる **メトリック設定**。
- ⑥ Kafka のデプロイメント用の **Kafka** カスタムリソース設定。一時的または永続的なシングルまたはマルチノードデプロイメントの設定例が含まれています。

- 7 Cruise Control のデプロイ設定を含む **Kafka** カスタムリソース。デフォルトまたはユーザー最適化ゴールを使用する設定の例とともに、Cruise Control から最適化プロポーザルを生成するための
- 8 Kafka Connect をデプロイするための **KafkaConnect** および **KafkaConnector** カスタムリソース設定。シングルまたはマルチノードデプロイメントの設定例が含まれています。
- 9 Kafka Bridge をデプロイするための **KafkaBridge** カスタムリソース設定。

関連情報

- [AMQ Streams デプロイメントの設定](#)

4.4. コンテナイメージの独自のレジストリーへのプッシュ

AMQ Streams のコンテナイメージは [Red Hat Ecosystem Catalog](#) にあります。AMQ Streams によって提供されるインストール YAML ファイルは、直接 [Red Hat Ecosystem Catalog](#) からイメージをプルします。

[Red Hat Ecosystem Catalog](#) にアクセスできない場合や独自のコンテナリポジトリーを使用する場合は以下を行います。

1. リストにある **すべての** コンテナイメージをプルします。
2. 独自のレジストリーにプッシュします。
3. インストール YAML ファイルのイメージ名を更新します。



注記

リリースに対してサポートされる各 Kafka バージョンには別のイメージがあります。

コンテナイメージ	namespace/リポジトリー	説明
Kafka	<ul style="list-style-type: none"> • registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.2 • registry.redhat.io/amq7/amq-streams-kafka-31-rhel8:2.2.2 	<p>次を含む、Kafka を実行するための AMQ Streams イメージ</p> <ul style="list-style-type: none"> • Kafka Broker • Kafka Connect • Kafka MirrorMaker • ZooKeeper • TLS Sidecars

コンテナイメージ	namespace/リポジトリ	説明
Operator	<ul style="list-style-type: none"> registry.redhat.io/amq7/amq-streams-rhel8-operator:2.2.2 	Operator を実行するための AMQ Streams イメージ <ul style="list-style-type: none"> Cluster Operator Topic Operator User Operator Kafka Initializer
Kafka Bridge	<ul style="list-style-type: none"> registry.redhat.io/amq7/amq-streams-bridge-rhel8:2.2.2 	AMQ Streams Kafka Bridge を稼働するための AMQ Streams イメージ
AMQ Streams Drain Cleaner	<ul style="list-style-type: none"> registry.redhat.io/amq7/amq-streams-drain-cleaner-rhel8:2.2.2 	AMQ Streams Drain Cleaner を実行するための AMQ Streams イメージ

4.5. AMQ STREAMS の管理者の指定

AMQ Streams では、デプロイメントの設定にカスタムリソースが提供されます。デフォルトでは、これらのリソースの表示、作成、編集、および削除権限は OpenShift クラスター管理者に限定されます。AMQ Streams には、これらの権利を他のユーザーに割り当てるために使用できる 2 つのクラスターロールが用意されています。

- **strimzi-view** ロールを指定すると、ユーザーは AMQ Streams リソースを表示できます。
- **strimzi-admin** ロールを指定すると、ユーザーは AMQ Streams リソースを作成、編集、または削除することもできます。

これらのロールをインストールすると、これらの権限が自動的にデフォルトの OpenShift クラスターロールに集約 (追加) されます。**strimzi-view** は **view** ロールに、**strimzi-admin** は **edit** および **admin** ロールに集約されます。このように集約することで、すでに同様の権限を持つユーザーに、これらのロールを割り当てる必要がなくなる可能性があります。

以下の手順では、クラスター管理者でないユーザーが AMQ Streams リソースを管理できるようにする **strimzi-admin** ロールの割り当て方法を説明します。

システム管理者は、Cluster Operator のデプロイ後に AMQ Streams の管理者を指定できます。

前提条件

- [Cluster Operator](#) でデプロイとデプロイされた CRD (カスタムリソース定義) を管理する AMQ Streams の CRD リソースおよび RBAC (ロールベースアクセス制御) リソース。

手順

1. OpenShift で **strimzi-view** および **strimzi-admin** クラスターロールを作成します。

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

2. 必要な場合は、ユーザーに必要なアクセス権限を付与するロールを割り当てます。

```
oc create clusterrolebinding strimzi-admin --clusterrole=strimzi-admin --user=user1 --  
user=user2
```

第5章 WEB コンソールを使用した OPERATORHUB からの AMQ STREAMS のインストール

OpenShift Container Platform Web コンソールの OperatorHub から Red Hat Integration - AMQ Streams Operator をインストールします。

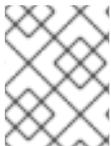
本セクションの手順では以下の方法を説明します。

- [OperatorHub からの AMQ Streams Operator のインストール](#)
- [AMQ Streams Operator を使用した Kafka コンポーネントのデプロイ](#)

5.1. RED HAT INTEGRATION OPERATOR を使用した AMQ STREAMS OPERATOR のインストール

Red Hat Integration Operator (非推奨) を使用すると、Red Hat Integration コンポーネントを管理する Operator を選択およびインストールできます。複数の Red Hat Integration サブスクリプションがある場合、Red Hat Integration Operator を使用して、AMQ Streams Operator およびサブスクライブしている Red Hat Integration コンポーネントのすべての Operator をインストールおよび更新できます。

AMQ Streams Operator の場合は、Operator Lifecycle Manager (OLM) を使用して、OCP コンソールの OperatorHub から OpenShift Container Platform (OCP) クラスターに Red Hat Integration Operator をインストールできます。



注記

Red Hat Integration Operator は非推奨となり、今後削除予定です。OpenShift 4.6 から 4.10 では、OperatorHub で利用できる予定です。

関連情報

Red Hat Integration Operator のインストールおよび使用に関する詳細は、[Red Hat Integration Operator のインストール](#)を参照してください。

5.2. OPERATORHUB からの AMQ STREAMS OPERATOR のインストール

OpenShift Container Platform Web コンソールの OperatorHub を使用して、AMQ Streams Operator をインストールしてサブスクライブできます。

この手順では、プロジェクトを作成し、そのプロジェクトに AMQ Streams Operator をインストールする方法を説明します。プロジェクトは namespace を表します。namespace を使用して機能を分離することで管理性を確保することを推奨します。



警告

適切な更新チャンネルを使用するようにしてください。サポート対象の OpenShift のバージョンを使用している場合は、デフォルトの stable チャンネルから安全に AMQ Streams をインストールできます。ただし、stable チャンネルで自動更新を有効にすることは推奨されません。自動アップグレードでは、アップグレード前の必要手順がスキップされます。バージョン固有のチャンネルでのみ自動アップグレードを使用します。

前提条件

- **cluster-admin** または **strimzi-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform Web コンソールにアクセスできる。

手順

1. OpenShift Web コンソールで **Home > Projects** ページに移動し、インストール用のプロジェクト (namespace) を作成します。
この例では、**amq-streams-kafka** という名前のプロジェクトを使用します。
2. **Operators > OperatorHub** ページに移動します。
3. スクロール、または **Filter by keyword** ボックスにキーワードを入力して、**Red Hat Integration - AMQ Streams Operator** を見つけます。
Operator は、**Streaming & Messaging** カテゴリーにあります。
4. **Red Hat Integration - AMQ Streams** をクリックして、Operator 情報を表示します。
5. Operator に関する情報を確認し、**Install** をクリックします。
6. **Install Operator** ページで、次のインストールおよび更新オプションから選択します。
 - **Update Channel:** Operator の更新チャンネルを選択します。
 - **stable** チャンネル (デフォルト) には最新の更新とリリースがすべて含まれます。これには、十分なテストを行った上、安定していることが想定される、メジャー、マイナー、およびマイクロリリースが含まれます。
 - **amq-streams-X.x** チャンネルには、メジャーリリースのマイナーリリースの更新およびマイクロリリースの更新が含まれます。X は、メジャーリリースのバージョン番号に置き換えてください。
 - **amq-streams-X.Y.x** チャンネルには、マイナーリリースのマイクロリリースの更新が含まれます。X はメジャーリリースのバージョン番号、Y はマイナーリリースのバージョン番号に置き換えてください。
 - **Installation Mode:** 作成したプロジェクトを選択して、特定の namespace に Operator をインストールします。
AMQ Streams Operator をクラスターのすべての namespace (デフォルトのオプション) にインストールするか、特定の namespace にインストールするかを選択できます。特定の namespace を Kafka クラスターおよびその他の AMQ Streams コンポーネント専用とすることが推奨されます。

- **Update approval:** デフォルトでは、OLM (Operator Lifecycle Manager) によって、AMQ Streams Operator が自動的に最新の AMQ Streams バージョンにアップグレードされます。今後のアップグレードを手動で承認する場合は、**Manual** を選択します。詳細は、OpenShift ドキュメントの [Operators](#) ガイドを参照してください。
7. **Install** をクリックして、選択した namespace に Operator をインストールします。
AMQ Streams Operator によって、Cluster Operator、CRD、およびロールベースアクセス制御 (RBAC) リソースは選択された namespace にデプロイされます。
 8. Operator を使用する準備ができたなら、**Operators > Installed Operators** に移動して、Operator が選択した namespace にインストールされていることを確認します。
ステータスは **Succeeded** と表示されます。

これで、AMQ Streams Operator を使用して、Kafka クラスタから順に Kafka コンポーネントをデプロイできるようになりました。



注記

Workloads > Deployments に移動すると、Cluster Operator および Entity Operator のデプロイメントの詳細を確認できます。Cluster Operator の名前には、バージョン番号 **amq-streams-cluster-operator-<version>** が含まれます。AMQ Streams インストールアーティファクトを使用して Cluster Operator をデプロイする場合、名前は異なります。この場合、名前は **strimzi-cluster-operator** です。

5.3. AMQ STREAMS OPERATOR を使用した KAFKA コンポーネントのデプロイ

Openshift にインストールすると、AMQ Streams Operator は、ユーザーインターフェイスから Kafka コンポーネントをインストールできるようにします。

次の Kafka コンポーネントをインストールできます。

- Kafka
- Kafka Connect
- Kafka MirrorMaker
- Kafka MirrorMaker 2
- Kafka Topic
- Kafka User
- Kafka Bridge
- Kafka Connector
- Kafka Rebalance

コンポーネントを選択して、インスタンスを作成します。少なくとも、Kafka インスタンスを作成します。この手順では、デフォルト設定を使用して Kafka インスタンスを作成する方法を説明します。インストールを実行する前に、デフォルトのインストール仕様を設定できます。

プロセスは、他の Kafka コンポーネントのインスタンスを作成する場合と同じです。

前提条件

- AMQ Streams Operator が [OpenShift クラスターにインストールされている](#)。

手順

1. Web コンソールで **Operators > Installed Operators** ページに移動し、**Red Hat Integration - AMQ Streams** をクリックして、Operator の詳細を表示します。
提供されている API から、Kafka コンポーネントのインスタンスを作成できます。
2. **Kafka** の下の **Create instance** をクリックして、Kafka インスタンスを作成します。
デフォルトでは、3 つの Kafka ブローカーノードと 3 つの ZooKeeper ノードを持つ **my-cluster** という名の Kafka クラスターを作成します。クラスターはエフェメラルストレージを使用します。
3. **Create** をクリックして、Kafka のインストールを開始します。
ステータスが **Ready** に変わるまで待ちます。

第6章 インストールアーティファクトを使用した AMQ STREAMS のデプロイ

AMQ Streams のデプロイメント環境を準備したら、AMQ Streams を OpenShift クラスターにデプロイできます。リリースアーティファクトで提供されるデプロイメントファイルを使用できます。

デプロイメントファイルを使用して [Kafka クラスター](#) を作成します。

任意で、要件に応じて以下の Kafka コンポーネントをデプロイできます。

- [Kafka Connect](#)
- [Kafka MirrorMaker](#)
- [Kafka Bridge](#)

AMQ Streams は Strimzi 0.29.x をベースとしています。AMQ Streams 2.2 は、OpenShift 4.8 から 4.11 にデプロイできます。



注記

本ガイドのコマンドを実行するには、クラスターユーザーに RBAC (ロールベースアクセス制御) および CRD を管理する権限を付与する必要があります。

6.1. KAFKA クラスターの作成

Cluster Operator で Kafka クラスターを管理できるようにするには、これを **Kafka** リソースとしてデプロイする必要があります。AMQ Streams では、この目的のために、デプロイメントファイルのサンプルが同梱されています。これらのファイルを使用して、Topic Operator および User Operator を同時にデプロイできます。

Kafka クラスターを **Kafka** リソースとしてデプロイしていない場合は、Cluster Operator を使用してこのクラスターを管理できません。これには、OpenShift 外で実行されている Kafka クラスターなどが該当します。ただし、Topic Operator および User Operator は、スタンドアロンコンポーネントとしてデプロイし、使用することができます。



注記

Cluster Operator は、OpenShift クラスターの1つ、複数、またはすべての namespace を監視できます。Topic Operator および User Operator は、単一の namespace で **KafkaTopic** および **KafkaUser** リソースを監視します。詳細は、[AMQ Streams Operator を使用した namespace の監視](#) を参照してください。

6.1.1. Topic Operator および User Operator と Kafka クラスターのデプロイ

AMQ Streams によって管理される Kafka クラスターを Topic Operator および User Operator と使用する場合は、このデプロイメント手順を実行します。

1. [Cluster Operator](#) をデプロイします。
2. Cluster Operator を使用して以下をデプロイします。
 - a. [Kafka クラスター](#)

b. [Topic Operator](#)

c. [User Operator](#)

6.1.2. スタンドアロン Topic Operator および User Operator のデプロイ

AMQ Streams によって管理されない Kafka クラスターを Topic Operator および User Operator と使用する場合は、このデプロイメント手順を実行します。

1. [スタンドアロン Topic Operator のデプロイ](#)
2. [スタンドアロン User Operator のデプロイ](#)

6.1.3. Cluster Operator のデプロイ

Cluster Operator は、OpenShift クラスター内で Apache Kafka クラスターのデプロイおよび管理を行います。

本セクションの手順では、以下のいずれかを監視するように Cluster Operator をデプロイする方法を説明します。

- [単一の namespace](#)
- [複数の namespace](#)
- [すべての namespace](#)

6.1.3.1. Cluster Operator デプロイメントの監視オプション

Cluster Operator の稼働中に、Kafka リソースの更新に対する監視が開始されます。

Cluster Operator をデプロイして、以下からの Kafka リソースの監視を選択できます。

- 単一の namespace (Cluster Operator が含まれる同じ namespace)
- 複数の namespace
- すべての namespace



注記

AMQ Streams では、デプロイメントの処理を簡単にするため、YAML ファイルのサンプルが提供されます。

Cluster Operator では、以下のリソースの変更が監視されます。

- Kafka クラスターの **Kafka**。
- Kafka Connect クラスターの **KafkaConnect**。
- Kafka Connect クラスターでコネクタを作成および管理するための **KafkaConnector**。
- Kafka MirrorMaker インスタンスの **KafkaMirrorMaker**。
- Kafka MirrorMaker 2.0 インスタンスの **KafkaMirrorMaker2**。

- Kafka Bridge インスタンスの **KafkaBridge**。
- Cruise Control の最適化リクエストの **KafkaRebalance**。

OpenShift クラスターでこれらのリソースの1つが作成されると、Operator がクラスターの詳細をリソースから取得します。さらに、StatefulSet、Service、および ConfigMap などの必要な OpenShift リソースが作成され、リソースの新しいクラスターの作成が開始されます。

Kafka リソースが更新されるたびに、リソースのクラスターを設定する OpenShift リソースで該当する更新が Operator によって実行されます。

リソースは、パッチを適用するか削除してから、再作成して、目的とするクラスターの状態を、リソースのクラスターに反映させます。この操作は、サービスの中断を引き起こすローリング更新の原因となる可能性があります。

リソースが削除されると、Operator によってクラスターがアンデプロイされ、関連する OpenShift リソースがすべて削除されます。

6.1.3.2. 単一の namespace を監視対象とする Cluster Operator のデプロイメント

この手順では、OpenShift クラスターの単一の namespace で AMQ Streams リソースを監視する Cluster Operator をデプロイする方法を説明します。

前提条件

- この手順では、**CustomResourceDefinitions**、**ClusterRoles**、および **ClusterRoleBindings** を作成できる OpenShift ユーザーアカウントを使用する必要があります。通常、OpenShift クラスターでロールベースアクセス制御 (RBAC) を使用する場合、これらのリソースを作成、編集、および削除する権限を持つユーザーは **system:admin** などの OpenShift クラスター管理者に限定されます。

手順

1. Cluster Operator のインストール先の namespace を使用するように、AMQ Streams インストールファイルを編集します。
たとえば、この手順では Cluster Operator は **<my_cluster_operator_namespace>** という namespace にインストールされます。

Linux の場合は、以下を使用します。

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

MacOS の場合は、以下を使用します。

```
sed -i "s/namespace: ./namespace: <my_cluster_operator_namespace>/' install/cluster-operator/*RoleBinding*.yaml
```

2. Cluster Operator をデプロイします。

```
oc create -f install/cluster-operator -n <my_cluster_operator_namespace>
```

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
strimzi-cluster-operator 1/1    1          1
```

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。

6.1.3.3. 複数の namespace を監視対象とする Cluster Operator のデプロイメント

この手順では、OpenShift クラスターの複数の namespace で AMQ Streams リソースを監視する Cluster Operator をデプロイする方法を説明します。

前提条件

- この手順では、**CustomResourceDefinitions**、**ClusterRoles**、および **ClusterRoleBindings** を作成できる OpenShift ユーザーアカウントを使用する必要があります。通常、OpenShift クラスターでロールベースアクセス制御 (RBAC) を使用する場合、これらのリソースを作成、編集、および削除する権限を持つユーザーは **system:admin** などの OpenShift クラスター管理者に限定されます。

手順

- Cluster Operator のインストール先の namespace を使用するように、AMQ Streams インストールファイルを編集します。
たとえば、この手順では Cluster Operator は **<my_cluster_operator_namespace>** という namespace にインストールされます。

Linux の場合は、以下を使用します。

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

MacOS の場合は、以下を使用します。

```
sed -i "s/namespace: ./namespace: <my_cluster_operator_namespace>/" install/cluster-operator/*RoleBinding*.yaml
```

- install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** ファイルを編集し、Cluster Operator が監視するすべての namespace のリストを **STRIMZI_NAMESPACE** 環境変数に追加します。
たとえば、この手順では Cluster Operator は **watched-namespace-1**、**watched-namespace-2**、および **watched-namespace-3** という namespace を監視します。

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

```
containers:
- name: strimzi-cluster-operator
  image: registry.redhat.io/amq7/amq-streams-rhel8-operator:2.2.2
  imagePullPolicy: IfNotPresent
env:
- name: STRIMZI_NAMESPACE
  value: watched-namespace-1,watched-namespace-2,watched-namespace-3
```

- リストした各 namespace に **RoleBindings** をインストールします。
この例では、コマンドの **watched-namespace** を前述のステップでリストした namespace に置き換えます。**watched-namespace-1**、**watched-namespace-2**、および **watched-namespace-3** にも、繰り返し同様の操作を実行します。

```
oc create -f install/cluster-operator/020-RoleBinding-strimzi-cluster-operator.yaml -n <watched_namespace>
oc create -f install/cluster-operator/031-RoleBinding-strimzi-cluster-operator-entity-operator-delegation.yaml -n <watched_namespace>
```

- Cluster Operator をデプロイします。

```
oc create -f install/cluster-operator -n <my_cluster_operator_namespace>
```

- 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
strimzi-cluster-operator 1/1    1          1
```

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。

6.1.3.4. すべての namespace を対象とする Cluster Operator のデプロイメント

この手順では、OpenShift クラスターのすべての namespace で AMQ Streams リソースを監視する Cluster Operator をデプロイする方法を説明します。

このモードで実行している場合、Cluster Operator は、新規作成された namespace でクラスターを自動的に管理します。

前提条件

- この手順では、**CustomResourceDefinitions**、**ClusterRoles**、および **ClusterRoleBindings** を作成できる OpenShift ユーザーアカウントを使用する必要があります。通常、OpenShift クラスターでロールベースアクセス制御 (RBAC) を使用する場合、これらのリソースを作成、編集、および削除する権限を持つユーザーは **system:admin** などの OpenShift クラスター管理者に限定されます。

手順

- Cluster Operator のインストール先の namespace を使用するように、AMQ Streams インストールファイルを編集します。
たとえば、この手順では Cluster Operator は `<my_cluster_operator_namespace>` という namespace にインストールされます。

Linux の場合は、以下を使用します。

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

MacOS の場合は、以下を使用します。

```
sed -i " 's/namespace: */namespace: <my_cluster_operator_namespace>/' install/cluster-operator/*RoleBinding*.yaml
```

- `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml` ファイルを編集し、`STRIMZI_NAMESPACE` 環境変数の値を * に設定します。

```
apiVersion: apps/v1
kind: Deployment
spec:
  # ...
  template:
    spec:
      # ...
      serviceAccountName: strimzi-cluster-operator
      containers:
      - name: strimzi-cluster-operator
        image: registry.redhat.io/amq7/amq-streams-rhel8-operator:2.2.2
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: "*"
      # ...
```

- クラスター全体ですべての namespace にアクセスできる権限を Cluster Operator に付与する **ClusterRoleBindings** を作成します。

```
oc create clusterrolebinding strimzi-cluster-operator-namespaced --clusterrole=strimzi-cluster-operator-namespaced --serviceaccount
<my_cluster_operator_namespace>:strimzi-cluster-operator
oc create clusterrolebinding strimzi-cluster-operator-entity-operator-delegation --clusterrole=strimzi-entity-operator --serviceaccount
<my_cluster_operator_namespace>:strimzi-cluster-operator
```

`<my_cluster_operator_namespace>` は、Cluster Operator をインストールする namespace に置き換えます。

- Cluster Operator を OpenShift クラスターにデプロイします。

```
oc create -f install/cluster-operator -n <my_cluster_operator_namespace>
```

- 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
strimzi-cluster-operator 1/1    1            1
```

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。

6.1.4. Kafka のデプロイ

Apache Kafka は、耐障害性のリアルタイムデータフィードを実現する、オープンソースの分散型 publish/subscribe メッセージングシステムです。

本セクションの手順では、以下を説明します。

- Cluster Operator を使用して以下をデプロイする方法
 - [一時または永続 Kafka クラスタ](#)
 - Topic Operator および User Operator (**Kafka** カスタムリソースを設定してデプロイする)
 - [Topic Operator](#)
 - [User Operator](#)
- Topic Operator および User Operator の代替のスタンドアロンデプロイメント手順
 - [スタンドアロン Topic Operator のデプロイ](#)
 - [スタンドアロン User Operator のデプロイ](#)

Kafka をインストールする場合、AMQ Streams によって ZooKeeper クラスタもインストールされ、Kafka と ZooKeeper との接続に必要な設定が追加されます。

6.1.4.1. Kafka クラスタのデプロイメント

この手順では、Cluster Operator を使用して Kafka クラスタを OpenShift クラスタにデプロイする方法を説明します。

デプロイメントでは、YAML ファイルの仕様を使用して **Kafka** リソースが作成されます。

AMQ Streams には、[設定ファイルのサンプル](#) が含まれています。Kafka デプロイメントでは、以下の例が提供されます。

kafka-persistent.yaml

3つの Zookeeper ノードと3つの Kafka ノードを使用して永続クラスタをデプロイします。

kafka-jbod.yaml

それぞれが複数の永続ボリュームを使用する、3つの ZooKeeper ノードと3つの Kafka ノードを使用して、永続クラスタをデプロイします。

kafka-persistent-single.yaml

1つの ZooKeeper ノードと1つの Kafka ノードを使用して、永続クラスタをデプロイします。

kafka-ephemeral.yaml

3 つの ZooKeeper ノードと 3 つの Kafka ノードを使用して、一時クラスターをデプロイします。

kafka-ephemeral-single.yaml

3 つの ZooKeeper ノードと 1 つの Kafka ノードを使用して、一時クラスターをデプロイします。

この手順では、一時 および 永続 Kafka クラスターデプロイメントの例を使用します。

一時クラスター

通常、Kafka の一時クラスターは開発およびテスト環境での使用に適していますが、本番環境での使用には適していません。このデプロイメントでは、ブローカー情報 (ZooKeeper) と、トピックまたはパーティション (Kafka) を格納するための **emptyDir** ボリュームが使用されます。**emptyDir** ボリュームを使用すると、その内容は Pod のライフサイクルと厳密な関係を持つため、Pod がダウンすると削除されます。

永続クラスター

Kafka の永続クラスターでは、永続ボリュームを使用して ZooKeeper および Kafka データを格納します。**PersistentVolumeClaim** を使用して **PersistentVolume** が取得され、**PersistentVolume** の実際のタイプには依存しません。**PersistentVolumeClaim** で **StorageClass** を使用し、自動ボリュームプロビジョニングをトリガーすることができます。**StorageClass** が指定されていない場合、OpenShift はデフォルトの **StorageClass** を使用しようとしています。

次の例では、一般的なタイプの永続ボリュームを一部紹介しています。

- OpenShift クラスターが Amazon AWS で実行されている場合、OpenShift は Amazon EBS ボリュームをプロビジョニングできます。
- OpenShift クラスターが Microsoft Azure で実行されている場合、OpenShift は Azure Disk Storage ボリュームをプロビジョニングできます。
- OpenShift クラスターが Google Cloud で実行されている場合、OpenShift は永続ディスクボリュームをプロビジョニングできます
- OpenShift クラスターがベアメタルで実行されている場合、OpenShift はローカル永続ボリュームをプロビジョニングできます

このサンプル YAML ファイルは、最新のサポート対象 Kafka バージョン、サポート対象のログメッセージ形式バージョンとブローカー間のプロトコルバージョンの設定を指定します。Kafka **config** の **inter.broker.protocol.version** プロパティは、指定された Kafka バージョン (**spec.kafka.version**) によってサポートされるバージョンである必要があります。このプロパティは、Kafka クラスターで使用する 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 のアップグレード](#) 時に、**inter.broker.protocol.version** への更新が必要です。

サンプルクラスターの名前はデフォルトで **my-cluster** になります。クラスター名はリソースの名前によって定義され、クラスターがデプロイされた後に変更できません。クラスターをデプロイする前にクラスター名を変更するには、関連する YAML ファイルにある **Kafka** リソースの **Kafka.metadata.name** プロパティを編集します。

デフォルトのクラスター名および指定された Kafka バージョン

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

```

metadata:
  name: my-cluster
spec:
  kafka:
    version: 3.2.3
    #...
  config:
    #...
    log.message.format.version: "3.2"
    inter.broker.protocol.version: "3.2"
  # ...

```

前提条件

- [Cluster Operator](#) がデプロイされている。

手順

1. 一時または 永続 クラスターを作成およびデプロイします。
開発またはテストでは、一時クラスターの使用が適している可能性があります。永続クラスターはどのような状況でも使用することができます。

- 一時 クラスターを作成およびデプロイするには、以下を実行します。

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

- 永続 クラスターを作成およびデプロイするには、以下を実行します。

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

2. デプロイメントのステータスを確認します。

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

Pod 名および readiness が表示される出力

NAME	READY	STATUS	RESTARTS
my-cluster-entity-operator	3/3	Running	0
my-cluster-kafka-0	1/1	Running	0
my-cluster-kafka-1	1/1	Running	0
my-cluster-kafka-2	1/1	Running	0
my-cluster-zookeeper-0	1/1	Running	0
my-cluster-zookeeper-1	1/1	Running	0
my-cluster-zookeeper-2	1/1	Running	0

my-cluster は Kafka クラスターの名前です。

デフォルトのデプロイメントでは、Entity Operator クラスター、3つの Kafka Pod、および3つの ZooKeeper Pod をインストールします。

READY は、Ready/expected 状態のレプリカ数を表示します。**STATUS** が **Running** と表示されると、デプロイメントは成功しています。

関連情報

Kafka クラスターの設定

6.1.4.2. Cluster Operator を使用した Topic Operator のデプロイ

この手順では、Cluster Operator を使用して Topic Operator をデプロイする方法を説明します。

Kafka リソースの **entityOperator** プロパティを設定し、**topicOperator** が含まれるようにします。デフォルトでは、Topic Operator は Cluster Operator によってデプロイされた Kafka クラスターの namespace で **KafkaTopic** リソースを監視します。Topic Operator **spec** で **watchedNamespace** を使用して namespace を指定することもできます。1つの Topic Operator が監視できるのは、namespace 1つです。1つの namespace を監視するのは、Topic Operator 1つのみとします。

AMQ Streams を使用して複数の Kafka クラスターを同じ namespace にデプロイする場合は、1つの Kafka クラスターに対してのみ Topic Operator を有効にするか、**watchedNamespace** プロパティを使用して Topic Operators が他の namespace を監視するように設定します。

AMQ Streams によって管理されない Kafka クラスターを Topic Operator と使用する場合は、[Topic Operator をスタンドアロンコンポーネントとしてデプロイ](#) する必要があります。

entityOperator および **topicOperator** プロパティの設定に関する詳細は、[エンティティ Operator の設定](#) を参照してください。

前提条件

- [Cluster Operator がデプロイされている](#)。

手順

1. **Kafka** リソースの **entityOperator** プロパティを編集し、**topicOperator** が含まれるようにします。

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

2. [EntityTopicOperatorSpec スキーマ参照](#)に記載されているプロパティを使用して、Topic Operator の **spec** を設定します。
すべてのプロパティにデフォルト値を使用する場合は、空のオブジェクト ({}) を使用します。
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 <kafka_configuration_file>
```

4. デプロイメントのステータスを確認します。

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

Pod 名と準備状況が表示される出力

```
NAME                READY STATUS RESTARTS
my-cluster-entity-operator 3/3   Running 0
# ...
```

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。**STATUS** が **Running** と表示されると、デプロイメントは成功しています。

6.1.4.3. Cluster Operator を使用した User Operator のデプロイ

この手順では、Cluster Operator を使用して User Operator をデプロイする方法を説明します。

Kafka リソースの **entityOperator** プロパティを設定し、**userOperator** が含まれるようにします。デフォルトでは、User Operator は Kafka クラスターデプロイメントの namespace で **KafkaUser** リソースを監視します。User Operator **spec** で **watchedNamespace** を使用して namespace を指定することもできます。1つの User Operator が監視できるのは、namespace 1つです。1つの namespace を監視するのは、User Operator 1つのみとします。

AMQ Streams によって管理されない Kafka クラスターを User Operator と使用する場合は、[User Operator をスタンドアロンコンポーネントとしてデプロイ](#) する必要があります。

entityOperator および **userOperator** プロパティの設定に関する詳細は、[エンティティ Operator の設定](#) を参照してください。

前提条件

- [Cluster Operator がデプロイされている](#)。

手順

1. **Kafka** リソースの **entityOperator** プロパティを編集し、**userOperator** が含まれるようにします。

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

2. [EntityUserOperatorSpec スキーマ参照](#) に記載されているプロパティを使用して、User Operator の **spec** を設定します。
すべてのプロパティにデフォルト値を使用する場合は、空のオブジェクト ({}) を使用します。
3. リソースを作成または更新します。

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

4. デプロイメントのステータスを確認します。

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

Pod 名と準備状況が表示される出力

```
NAME                READY STATUS RESTARTS
my-cluster-entity-operator 3/3   Running 0
# ...
```

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。**STATUS** が **Running** と表示されると、デプロイメントは成功しています。

6.1.5. AMQ Streams Operator の代替のスタンドアロンデプロイメントオプション

Topic Operator および User Operator のスタンドアロンデプロイメントを実行できます。Cluster Operator によって管理されない Kafka クラスターを使用している場合は、これらの Operator のスタンドアロンデプロイメントを検討してください。

Operator を OpenShift にデプロイします。Kafka は OpenShift 外で実行できます。たとえば、Kafka をマネージドサービスとして使用する場合があります。スタンドアロン Operator のデプロイメント設定を調整し、Kafka クラスターのアドレスと一致するようにします。

6.1.5.1. スタンドアロン Topic Operator のデプロイ

この手順では、Topic Operator をトピック管理のスタンドアロンコンポーネントとしてデプロイする方法を説明します。スタンドアロン Topic Operator を Cluster Operator によって管理されない Kafka クラスターと使用できます。

スタンドアロンデプロイメントは、任意の Kafka クラスターで操作できます。

スタンドアロンデプロイメントファイルは AMQ Streams で提供されます。**05-Deployment-strimzi-topic-operator.yaml** デプロイメントファイルを使用して、Topic Operator をデプロイします。Kafka クラスターへの接続に必要な環境変数を追加または設定します。

Topic Operator は、単一の namespace で **KafkaTopic** リソースを監視します。Topic Operator 設定で、監視する namespace と Kafka クラスターへの接続を指定します。1つの Topic Operator が監視できるのは、namespace 1つです。1つの namespace を監視するのは、Topic Operator 1つのみとします。複数の Topic Operator を使用する場合は、それぞれが異なる namespace を監視するように設定します。このようにして、Topic Operator を複数の Kafka クラスターで使用できます。

前提条件

- Topic Operator の接続先となる Kafka クラスターを実行している。
スタンドアロンの Topic Operator が接続用に正しく設定されている限り、Kafka クラスターはベアメタル環境、仮想マシン、またはマネージドクラウドアプリケーションサービスで実行できます。

手順

1. `install/topic-operator/05-Deployment-strimzi-topic-operator.yaml` スタンドアロンデプロイメントファイルの `env` プロパティを編集します。

スタンドアロンの Topic Operator デプロイメント設定の例

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-topic-operator
  labels:
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-topic-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE 1
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS 2
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_RESOURCE_LABELS 3
              value: "strimzi.io/cluster=my-cluster"
            - name: STRIMZI_ZOOKEEPER_CONNECT 4
              value: my-cluster-zookeeper-client:2181
            - name: STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS 5
              value: "18000"
            - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS 6
              value: "120000"
            - name: STRIMZI_TOPIC_METADATA_MAX_ATTEMPTS 7
              value: "6"
            - name: STRIMZI_LOG_LEVEL 8
              value: INFO
            - name: STRIMZI_TLS_ENABLED 9
              value: "false"
            - name: STRIMZI_JAVA_OPTS 10
              value: "-Xmx=512M -Xms=256M"
            - name: STRIMZI_JAVA_SYSTEM_PROPERTIES 11
              value: "-Djavax.net.debug=verbose -DpropertyName=value"
            - name: STRIMZI_PUBLIC_CA 12
              value: "false"
            - name: STRIMZI_TLS_AUTH_ENABLED 13
              value: "false"
            - name: STRIMZI_SASL_ENABLED 14
              value: "false"
            - name: STRIMZI_SASL_USERNAME 15
              value: "admin"
            - name: STRIMZI_SASL_PASSWORD 16

```

```

value: "password"
- name: STRIMZI_SASL_MECHANISM 17
  value: "scram-sha-512"
- name: STRIMZI_SECURITY_PROTOCOL 18
  value: "SSL"

```

- 1** **KafkaTopic** リソースを監視する Topic Operator の OpenShift namespace。Kafka クラスターの namespace を指定します。
- 2** Kafka クラスターのすべてのブローカーを検出し、接続するブートストラップブローカーアドレスのホストとポートのペア。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して2つまたは3つのブローカーアドレスを指定します。
- 3** Topic Operator によって管理される **KafkaTopic** リソースを識別するラベル。これは、Kafka クラスターの名前である必要はありません。**KafkaTopic** リソースに割り当てられたラベルにすることができます。複数の Topic Operator をデプロイする場合、ラベルはそれぞれに一意である必要があります。つまり、Operator は同じリソースを管理できません。
- 4** ZooKeeper クラスターに接続するためのアドレスのホストおよびポートのペア。これは、Kafka クラスターが使用する ZooKeeper クラスターと同じである必要があります。
- 5** ZooKeeper セッションのタイムアウト (秒単位)。デフォルトは **18000** (18 秒) です。
- 6** 定期的な調整の間隔 (秒単位)。デフォルトは **120000** (2 分) です。
- 7** 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で原因で、トピックの作成に時間がかかる場合は、この値を大きくすることを検討してください。デフォルトの試行回数は **6** 回です。
- 8** ロギングメッセージの出力レベル。レベルを、**ERROR**、**WARNING**、**INFO**、**DEBUG**、または **TRACE** に設定できます。
- 9** Kafka ブローカーとの暗号化された通信の TLS サポートを有効にします。
- 10** (任意) Topic Operator を実行する JVM に使用される Java オプション。
- 11** (任意) Topic Operator に設定されたデバッグ (**-D**) オプション。
- 12** (オプション) TLS が **STRIMZI_TLS_ENABLED** によって有効になっている場合、トラストストア証明書の生成を省略します。この環境変数が有効になっている場合、ブローカーは TLS 証明書に公的に信頼できる認証局を使用する必要があります。デフォルトは **false** です。
- 13** (オプション) 相互 TLS 認証用のキーストア証明書を生成します。これを **false** に設定すると、TLS を使用した Kafka ブローカーへのクライアント認証が無効になります。デフォルトは **true** です。
- 14** (オプション) Kafka ブローカーに接続するときにクライアント認証の SASL サポートを有効にします。デフォルトは **false** です。
- 15** (任意) クライアント認証用の SASL ユーザー名。SASL が **STRIMZI_SASL_ENABLED** によって有効化された場合のみ必須です。
- 16** (任意) クライアント認証用の SASL パスワード。SASL が **STRIMZI_SASL_ENABLED** によって有効化された場合のみ必須です。

- 17 (任意) クライアント認証用の SASL メカニズム。SASL が **STRIMZI_SASL_ENABLED** によって有効化された場合のみ必須です。この値は **plain**、**scram-sha-256**、または **scram-sha-512** です。
 - 18 (任意) Kafka ブローカーとの通信に使用されるセキュリティープロトコル。デフォルト値は **PLAINTEXT** です。値は **PLAINTEXT**、**SSL**、**SASL_PLAINTEXT**、または **SASL_SSL** に設定できます。
2. 公開認証局から証明書を使用している Kafka ブローカーに接続する場合は、**STRIMZI_PUBLIC_CA** を **true** に設定します。たとえば、Amazon AWS MSK サービスを使用している場合は、このプロパティを **true** に設定します。
 3. **STRIMZI_TLS_ENABLED** 環境変数で TLS を有効にした場合、Kafka クラスターへの接続を認証するために使用されるキーストアおよびトラストストアを指定します。

TLS 設定の例

```
# ...
env:
  - name: STRIMZI_TRUSTSTORE_LOCATION 1
    value: "/path/to/truststore.p12"
  - name: STRIMZI_TRUSTSTORE_PASSWORD 2
    value: "TRUSTSTORE-PASSWORD"
  - name: STRIMZI_KEYSTORE_LOCATION 3
    value: "/path/to/keystore.p12"
  - name: STRIMZI_KEYSTORE_PASSWORD 4
    value: "KEYSTORE-PASSWORD"
# ...
```

- 1 トラストストアには、Kafka および ZooKeeper サーバー証明書の署名に使用される認証局の公開鍵が含まれます。
 - 2 トラストストアにアクセスするためのパスワード。
 - 3 キーストアには、TLS クライアント認証の秘密鍵が含まれます。
 - 4 キーストアにアクセスするためのパスワード。
4. Topic Operator をデプロイします。

```
oc create -f install/topic-operator
```

5. デプロイメントのステータスを確認します。

```
oc get deployments
```

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-topic-operator 1/1    1          1
```

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。

6.1.5.2. スタンドアロン User Operator のデプロイ

この手順では、ユーザー管理のスタンドアロンコンポーネントとして User Operator をデプロイする方法を説明します。Cluster Operator の管理対象外となっている Kafka クラスターでは、スタンドアロンの User Operator を使用します。

スタンドアロンデプロイメントは、任意の Kafka クラスターで操作できます。

スタンドアロンデプロイメントファイルは AMQ Streams で提供されます。**05-Deployment-strimzi-user-operator.yaml** デプロイメントファイルを使用して、User Operator をデプロイします。Kafka クラスターへの接続に必要な環境変数を追加または設定します。

User Operator は、単一の namespace で **KafkaUser** リソースを監視します。User Operator 設定で、監視する namespace と Kafka クラスターへの接続を指定します。1つの User Operator が監視できるのは、namespace 1つです。1つの namespace を監視するのは、User Operator 1つのみとします。複数の User Operator を使用する場合は、それぞれが異なる namespace を監視するように設定します。このようにして、User Operator を複数の Kafka クラスターで使用できます。

前提条件

- User Operator の接続先となる Kafka クラスターを実行している。
スタンドアロンの User Operator が接続用に正しく設定されている限り、Kafka クラスターはベアメタル環境、仮想マシン、またはマネージドクラウドアプリケーションサービスで実行できます。

手順

1. 以下の **env** プロパティを **install/user-operator/05-Deployment-strimzi-user-operator.yaml** スタンドアロンデプロイメントファイルで編集します。

スタンドアロン User Operator デプロイメント設定の例

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: strimzi-user-operator
  labels:
    app: strimzi
spec:
  # ...
  template:
    # ...
    spec:
      # ...
      containers:
        - name: strimzi-user-operator
          # ...
          env:
            - name: STRIMZI_NAMESPACE 1
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: STRIMZI_KAFKA_BOOTSTRAP_SERVERS 2
              value: my-kafka-bootstrap-address:9092
            - name: STRIMZI_CA_CERT_NAME 3
```

```

value: my-cluster-clients-ca-cert
- name: STRIMZI_CA_KEY_NAME ④
  value: my-cluster-clients-ca
- name: STRIMZI_LABELS ⑤
  value: "strimzi.io/cluster=my-cluster"
- name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS ⑥
  value: "120000"
- name: STRIMZI_LOG_LEVEL ⑦
  value: INFO
- name: STRIMZI_GC_LOG_ENABLED ⑧
  value: "true"
- name: STRIMZI_CA_VALIDITY ⑨
  value: "365"
- name: STRIMZI_CA_RENEWAL ⑩
  value: "30"
- name: STRIMZI_JAVA_OPTS ⑪
  value: "-Xmx=512M -Xms=256M"
- name: STRIMZI_JAVA_SYSTEM_PROPERTIES ⑫
  value: "-Djavax.net.debug=verbose -DpropertyName=value"
- name: STRIMZI_SECRET_PREFIX ⑬
  value: "kafka-"
- name: STRIMZI_ACLS_ADMIN_API_SUPPORTED ⑭
  value: "true"
- name: STRIMZI_MAINTENANCE_TIME_WINDOWS ⑮
  value: '* * 8-10 * * ?; * * 14-15 * * ?'

```

- ① **KafkaUser** リソースを監視する User Operator の OpenShift namespace。指定できる namespace は1つだけです。
- ② Kafka クラスターのすべてのブローカーを検出し、接続するブートストラップブローカーアドレスのホストとポートのペア。サーバーがダウンした場合に備えて、コンマ区切りリストを使用して2つまたは3つのブローカーアドレスを指定します。
- ③ TLS クライアント認証に対して新しいユーザー証明書に署名する認証局の公開鍵 (**ca.crt**) の値が含まれる OpenShift **Secret**。
- ④ TLS クライアント認証に対して新しいユーザー証明書に署名する認証局の秘密鍵 (**ca.key**) の値が含まれる OpenShift **Secret**。
- ⑤ User Operator によって管理される **KafkaUser** リソースを識別するラベル。これは、Kafka クラスターの名前である必要はありません。**KafkaUser** リソースに割り当てられたラベルにすることができます。複数の User Operator をデプロイする場合、ラベルはそれぞれに一意である必要があります。つまり、Operator は同じリソースを管理できません。
- ⑥ 定期的な調整の間隔 (秒単位)。デフォルトは **120000** (2分) です。
- ⑦ ロギングメッセージの出力レベル。レベルを、**ERROR**、**WARNING**、**INFO**、**DEBUG**、または **TRACE** に設定できます。
- ⑧ ガベージコレクション (GC) ロギングを有効にします。デフォルトは **true** です。
- ⑨ 認証局の有効期間。デフォルトは **365** 日です。
- ⑩ 認証局の更新期間。更新期間は、現在の証明書の有効期日から逆算されます。デフォルトでは、古い証明書が期限切れになる前の証明書の更新期間は **30** 日です。

- 11 (任意) User Operator を実行する JVM に使用される Java オプション。
 - 12 (任意) User Operator に設定されたデバッグ (-D) オプション。
 - 13 (オプション) User Operator によって作成される OpenShift シークレットの名前の接頭辞。
 - 14 (任意) Kafka クラスターが Kafka Admin API を使用した承認 ACL ルールの管理をサポートするかどうかを示します。 **false** に設定すると、User Operator は **simple** 承認 ACL ルールを持つすべてのリソースを拒否します。これは、Kafka クラスターログで不要な例外を回避するのに役立ちます。デフォルトは **true** です。
 - 15 (オプション) 期限切れのユーザー証明書が更新されるメンテナンス時間枠を定義する Cron 式のセミコロンで区切られたリスト。
2. TLS を使用して Kafka クラスターに接続する場合は、接続の認証に使用されるシークレットを指定します。それ以外の場合は、次のステップに進みます。

TLS 設定の例

```
# ....
env:
  - name: STRIMZI_CLUSTER_CA_CERT_SECRET_NAME 1
    value: my-cluster-cluster-ca-cert
  - name: STRIMZI_EO_KEY_SECRET_NAME 2
    value: my-cluster-entity-operator-certs
# ..."
```

- 1 TLS クライアント認証に対して Kafka ブローカー証明書に署名する認証局の公開鍵 (**ca.crt**) の値が含まれる OpenShift **Secret**。
 - 2 Kafka クラスターに対する TLS 認証の秘密鍵と証明書が含まれるキーストア (**entity-operator.p12**) が含まれる OpenShift **Secret**。 **Secret** には、キーストアにアクセスするためのパスワード (**entity-operator.password**) も含まれる必要があります。
3. User Operator をデプロイします。

```
oc create -f install/user-operator
```

4. デプロイメントのステータスを確認します。

```
oc get deployments
```

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-user-operator 1/1    1            1
```

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。

6.2. KAFKA CONNECT のデプロイ

[Kafka Connect](#) は、Apache Kafka と外部システムとの間でデータをストリーミングするためのツールです。

AMQ Streams では、Kafka Connect は分散 (distributed) モードでデプロイされます。Kafka Connect はスタンドアロンモードでも動作しますが、AMQ Streams ではサポートされません。

Kafka Connect は、**コネクタ** の概念を使用し、スケーラビリティと信頼性を維持しながら Kafka クラスタで大量のデータを移動するフレームワークを提供します。

Kafka Connect は通常、Kafka を外部データベース、ストレージシステム、およびメッセージングシステムと統合するために使用されます。

Cluster Operator は、**KafkaConnect** リソースを使用してデプロイされた Kafka Connect クラスタと、**KafkaConnector** リソースを使用して作成されたコネクタを管理します。

次の手順は、Kafka Connect をデプロイし、ストリーミングデータ用のコネクタを設定する方法を示しています。

- [「Kafka Connect の OpenShift クラスタへのデプロイ」](#)
- [「複数インスタンスの Kafka Connect 設定」](#)
- [「コネクタプラグインでの Kafka Connect の拡張」](#)
- [「コネクタの作成および管理」](#)
- [「サンプル KafkaConnector リソースのデプロイ」](#)



注記

コネクタ という用語は、Kafka Connect クラスタ内で実行されているコネクタインスタンスや、コネクタクラスと同じ意味で使用されます。本ガイドでは、本文の内容で意味が明確である場合に **コネクタ** という用語を使用します。

6.2.1. Kafka Connect の OpenShift クラスタへのデプロイ

この手順では、Cluster Operator を使用して Kafka Connect クラスタを OpenShift クラスタにデプロイする方法を説明します。

Kafka Connect クラスタは、設定可能なノード数 (このノードの別称: **ワーカー**) を指定して **Deployment** として実装されます。このノードは、メッセージフローのスケーラビリティと信頼性が高くなるように、コネクタのワークロードを **タスク** として分散します。

デプロイメントでは、YAML ファイルの仕様を使用して **KafkaConnect** リソースが作成されます。

AMQ Streams には、[設定ファイルのサンプル](#) が用意されています。この手順では、以下のサンプルファイルを使用します。

- [examples/connect/kafka-connect.yaml](#)

前提条件

- [Cluster Operator](#) がデプロイされている。
- [Kafka](#) クラスタが稼働している。

手順

1. Kafka Connect を OpenShift クラスターにデプロイします。**examples/connect/kafka-connect.yaml** ファイルを使用して Kafka Connect をデプロイします。

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

2. 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-connect-cluster 1/1    1          1
```

my-connect-cluster は、Kafka Connect 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。

関連情報

[Kafka Connect クラスターの設定](#)

6.2.2. 複数インスタンスの Kafka Connect 設定

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

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

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



注記

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

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

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

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

6.2.3. コネクタプラグインでの Kafka Connect の拡張

Kafka Connect は、コネクタインスタンスを使用して他のシステムと統合し、データをストリーミングします。コネクタは、次のいずれかのタイプにすることができます。

- データを Kafka にプッシュするソースコネクタ
- Kafka からデータを抽出するシンクコネクタ

このセクションの手順では、次のいずれかを実行してコネクタを追加する方法について説明します。

- [「AMQ Streams を使用した新しいコンテナイメージの自動作成」](#)
- [「Kafka Connect ベースイメージからの Docker イメージの作成」](#) (手動または継続的インテグレーションを使用)



重要

Kafka Connect REST API または **KafkaConnector** カスタムリソースを使用して、直接コネクタの設定を作成します。

独自のコネクタを使用するか、サンプルの **FileStreamSourceConnector** および **FileStreamSinkConnector** コネクタを試して、ファイルベースのデータを Kafka クラスターに出し入れすることができます。サンプルファイルコネクタを **KafkaConnector** リソースとしてデプロイする方法は、[「サンプル KafkaConnector リソースのデプロイ」](#) を参照してください。



注記

Apache Kafka 3.1.0 までは、Kafka Connect の AMQ Streams コンテナイメージにサンプルファイルコネクタが含まれていました。Apache Kafka 3.1.1 および 3.2.0 以降、これらのコネクタは含まれなくなり、他のコネクタと同様にデプロイする必要があります。

6.2.3.1. AMQ Streams を使用した新しいコンテナイメージの自動作成

この手順では、AMQ Streams が追加のコネクタで新しいコンテナイメージを自動的にビルドするように Kafka Connect を設定する方法を説明します。コネクタプラグインは、**KafkaConnect** カスタムリソースの **.spec.build.plugins** プロパティを使用して定義します。AMQ Streams はコネクタプ

ラグインを自動的にダウンロードし、新しいコンテナイメージに追加します。コンテナは、**.spec.build.output** に指定されたコンテナリポジトリにプッシュされ、Kafka Connect デプロイメントで自動的に使用されます。

前提条件

- [Cluster Operator](#) がデプロイされている。
- コンテナレジストリー。

イメージをプッシュ、保存、およびプルできる独自のコンテナレジストリーを提供する必要があります。AMQ Streams は、プライベートコンテナレジストリーだけでなく、[Quay](#) や [Docker Hub](#) などのパブリックレジストリーもサポートします。

手順

1. **.spec.build.output** でコンテナレジストリーを、**.spec.build.plugins** で追加のコネクタを指定して、**KafkaConnect** カスタムリソースを設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ①
  #...
  build:
    output: ②
      type: docker
      image: my-registry.io/my-org/my-connect-cluster:latest
      pushSecret: my-registry-credentials
    plugins: ③
      - name: debezium-postgres-connector
        artifacts:
          - type: tgz
            url: https://repo1.maven.org/maven2/io/debezium/debezium-connector-
postgres/1.3.1.Final/debezium-connector-postgres-1.3.1.Final-plugin.tar.gz
            sha512sum:
962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb54804
5e115e33a9e69b1b2a352bee24df035a0447cb820077af00c03
          - name: camel-telegram
            artifacts:
              - type: tgz
                url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-
telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
                sha512sum:
a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e
8020bf4187870699819f54ef5859c7846ee4081507f48873479
            #...
```

- ① [Kafka Connect クラスターの仕様](#)。
- ② (必須) 新しいイメージがプッシュされるコンテナレジストリーの設定。
- ③ (必須) 新しいコンテナイメージに追加するコネクタプラグインとそれらのアーティファクトのリスト。各プラグインは、1つ以上の **artifact** を使用して設定する必要があります。

- リソースを作成または更新します。

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

- 新しいコンテナイメージがビルドされ、Kafka Connect クラスターがデプロイされるまで待ちます。
- Kafka Connect REST API または KafkaConnector カスタムリソースを使用して、追加したコネクタプラグインを使用します。

関連情報

詳細は、[Using Strimzi ガイド](#)を参照してください。

- [Kafka Connect Build スキーマ参照](#)

6.2.3.2. Kafka Connect ベースイメージからの Docker イメージの作成

この手順では、カスタムイメージを作成し、`/opt/kafka/plugins` ディレクトリーに追加する方法を説明します。

[Red Hat Ecosystem Catalog](#) の Kafka コンテナイメージは、追加のコネクタプラグインで独自のカスタムイメージを作成するためのベースイメージとして使用できます。

AMQ Stream バージョンの Kafka Connect は起動時に、`/opt/kafka/plugins` ディレクトリーに含まれるサードパーティーのコネクタプラグインをロードします。

前提条件

- [Cluster Operator](#) がデプロイされている。

手順

- [registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.2](#) をベースイメージとして使用して、新規の `Dockerfile` を作成します。

```
FROM registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.2
USER root:root
COPY ./my-plugins/ /opt/kafka/plugins/
USER 1001
```

プラグインファイルの例

```
$ tree ./my-plugins/
./my-plugins/
├── debezium-connector-mongodb
│   ├── bson-3.4.2.jar
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mongodb-0.7.1.jar
│   └── debezium-core-0.7.1.jar
```

```

├── LICENSE.txt
├── mongodb-driver-3.4.2.jar
├── mongodb-driver-core-3.4.2.jar
├── README.md
├── debezium-connector-mysql
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-mysql-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── mysql-binlog-connector-java-0.13.0.jar
│   ├── mysql-connector-java-5.1.40.jar
│   ├── README.md
│   └── wkb-1.0.2.jar
├── debezium-connector-postgres
│   ├── CHANGELOG.md
│   ├── CONTRIBUTE.md
│   ├── COPYRIGHT.txt
│   ├── debezium-connector-postgres-0.7.1.jar
│   ├── debezium-core-0.7.1.jar
│   ├── LICENSE.txt
│   ├── postgresql-42.0.0.jar
│   ├── protobuf-java-2.6.1.jar
│   └── README.md

```



注記

この例では、MongoDB、MySQL、および PostgreSQL 用の Debezium コネクターを使用します。Kafka Connect で実行されている Debezium は、他の Kafka Connect タスクと同じように表示されます。

2. コンテナイメージをビルドします。
3. カスタムイメージをコンテナレジストリーにプッシュします。
4. 新しいコンテナイメージを示します。
以下のいずれかを行います。
 - **KafkaConnect** カスタムリソースの **KafkaConnect.spec.image** プロパティを編集します。
設定された場合、このプロパティによって Cluster Operator の **STRIMZI_KAFKA_CONNECT_IMAGES** 変数がオーバーライドされます。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
spec: ①
  #...
  image: my-new-container-image ②
  config: ③
  #...

```

- 1 Kafka Connect クラスターの仕様。
- 2 Pod の Docker イメージ。
- 3 Kafka Connect ワーカー (コネクタではない) の設定。

または

- `install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml` ファイルの `STRIMZI_KAFKA_CONNECT_IMAGES` 変数を編集して新しいコンテナイメージを参照するようにし、Cluster Operator を再インストールします。

関連情報

- [コンテナイメージの設定および `KafkaConnect.spec.image` プロパティ](#)
- [Cluster Operator 設定および `STRIMZI_KAFKA_CONNECT_IMAGES` 変数](#)

6.2.4. コネクタの作成および管理

コネクタプラグインのコンテナイメージを作成したら、Kafka Connect クラスターにコネクタインスタンスを作成する必要があります。その後、稼働中のコネクタインスタンスを設定、監視、および管理できます。

コネクタは特定の `コネクタクラス` のインスタンスで、関連のある外部システムとメッセージについて通信する方法を認識しています。コネクタは多くの外部システムで使用でき、独自のコネクタを作成することもできます。

ソース および シンク タイプのコネクタを作成できます。

ソースコネクタ

ソースコネクタは、外部システムからデータを取得し、それをメッセージとして Kafka に提供するランタイムエンティティです。

シンクコネクタ

シンクコネクタは、Kafka トピックからメッセージを取得し、外部システムに提供するランタイムエンティティです。

6.2.4.1. コネクタの作成および管理用の API

AMQ Streams では、コネクタの作成および管理に 2 つの API が提供されます。

- **KafkaConnector** カスタムリソース (別称: `KafkaConnectors`)
- Kafka Connect REST API

API を使用すると、以下を行うことができます。

- コネクタインスタンスのステータスの確認。
- 稼働中のコネクタの再設定。
- コネクタインスタンスのコネクタタスク数の増減。
- コネクタの再起動。

- 失敗したタスクを含むコネクタータスクの再起動。
- コネクタースタンスの一時停止。
- 一時停止したコネクタースタンスの再開。
- コネクタースタンスの削除。

KafkaConnector カスタムリソース

KafkaConnectors を使用すると、Kafka Connect のコネクタースタンスを OpenShift ネイティブに作成および管理できるため、cURL などの HTTP クライアントは必要ありません。他の Kafka リソースと同様に、**KafkaConnector** YAML ファイルで、コネクターの目的の状態を宣言して、このファイルを OpenShift クラスターにデプロイしてコネクタースタンスを作成します。**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。

該当する **KafkaConnector** リソースを更新して稼働中のコネクタースタンスを管理した後、更新を適用します。該当する **KafkaConnector** を削除して、コネクタースタンスを削除します。

下位バージョンの AMQ Streams との互換性を維持するため、KafkaConnectors はデフォルトで無効になっています。Kafka Connect クラスターの KafkaConnectors を有効にするには、**KafkaConnect** リソースで **strimzi.io/use-connector-resources** アノテーションを **true** に設定します。手順は、[Kafka Connect の設定](#) を参照してください。

KafkaConnectors が有効になると、Cluster Operator によって監視が開始されます。KafkaConnectors に定義された設定と一致するよう、稼働中のコネクタースタンスの設定を更新します。

AMQ Streams には **KafkaConnector** 設定ファイルの例が含まれます。これをもとに、[Kafka Connector のデプロイ](#) を使用して **FileStreamSourceConnector** と **FileStreamSinkConnector** を作成して管理できます。



注記

KafkaConnector リソースにアノテーションを付けて、[コネクターの再起動](#) または [コネクタータスクの再起動](#) ができます。

Kafka Connect API

Kafka Connect REST API でサポートされる操作は、[Apache Kafka Connect API のドキュメント](#) で説明されています。

Kafka Connect API から KafkaConnectors の使用への切り替え

Kafka Connect API から KafkaConnectors の使用に切り替えると、コネクタースタンスを管理できます。スイッチの作成は、次の作業を以下の順序で行います。

1. 設定で **KafkaConnector** リソースをデプロイし、コネクタースタンスを作成します。
2. **strimzi.io/use-connector-resources** アノテーションを **true** に設定して、Kafka Connect 設定で KafkaConnectors を有効にします。



警告

リソースを作成する前に KafkaConnectors を有効にすると、すべてのコネクタが削除されます。

KafkaConnectors から Kafka Connect API の使用に切り替えるには、まず Kafka Connect 設定から KafkaConnectors を有効にするアノテーションを削除します。それ以外の場合、Kafka Connect REST API を使用して直接行われた手動による変更は、Cluster Operator によって元に戻されます。

6.2.4.2. サンプル KafkaConnector リソースのデプロイ

KafkaConnector リソースは、Cluster Operator によるコネクタの管理に Kubernetes ネイティブのアプローチを提供します。AMQ Streams には、[設定ファイルのサンプル](#) が用意されています。この手順では、`examples/connect/source-connector.yaml` ファイルを使用して、次のコネクタインスタンスを **KafkaConnector** リソースとして作成します。

- Kafka ライセンスファイル (ソース) から各行を読み取り、データをメッセージとして単一の Kafka トピックに書き込む **FileStreamSourceConnector** インスタンス。
- Kafka トピックからメッセージを読み取り、メッセージを一時ファイル (シンク) に書き込む **FileStreamSinkConnector** インスタンス。

または、`examples/connect/kafka-connect-build.yaml` ファイルを使用して、ファイルコネクタを使用して新しい Kafka Connect イメージを構築することもできます。

Apache Kafka 3.1.0 までは、サンプルファイルコネクタプラグインが Apache Kafka に含まれていました。Apache Kafka の 3.1.1 および 3.2.0 リリースから、例を他のコネクタと同様にプラグインパスに追加する必要があります。詳細は、[コネクタプラグインを使用した Kafka Connect の拡張](#) を参照してください。



注記

「[コネクタプラグインでの Kafka Connect の拡張](#)」で説明されているように、実稼働環境では、必要な Kafka Connect コネクタを使用してコンテナイメージを準備します。

FileStreamSourceConnector および **FileStreamSinkConnector** が例として提供されています。ここで記載されているようなコンテナで上記のコネクタを実行することは、実稼働のユースケースとして適切ではありません。

前提条件

- Kafka Connect デプロイメント。
- [Kafka Connect デプロイメント](#)で `KafkaConnectors` が有効である。
- Cluster Operator が稼働している。

手順

1. `examples/connect/source-connector.yaml` ファイルを編集します。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-source-connector ❶
  labels:
    strimzi.io/cluster: my-connect-cluster ❷
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector ❸
  tasksMax: 2 ❹
  config: ❺
    file: "/opt/kafka/LICENSE" ❻
    topic: my-topic ❼
  # ...

```

- ❶ コネクタの名前として使用される **KafkaConnector** リソースの名前。OpenShift リソースで有効な名前を使用します。
- ❷ コネクタインスタンスを作成する Kafka Connect クラスターの名前。コネクタは、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。
- ❸ コネクタクラスのフルネームまたはエイリアス。これは、Kafka Connect クラスターによって使用されているイメージに存在するはずです。
- ❹ コネクタが作成できる Kafka Connect **Tasks** の最大数。
- ❺ キーと値のペアとしての [コネクタ設定](#)。
- ❻ このサンプルソースコネクタ設定では、`/opt/kafka/LICENSE` ファイルからデータが読み取られます。
- ❼ ソースデータのパブリッシュ先となる Kafka トピック。

2. OpenShift クラスターでソース **KafkaConnector** を作成します。

```
oc apply -f examples/connect/source-connector.yaml
```

3. `examples/connect/sink-connector.yaml` ファイルを作成します。

```
touch examples/connect/sink-connector.yaml
```

4. 以下の YAML を `sink-connector.yaml` ファイルに貼り付けます。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  name: my-sink-connector
  labels:
    strimzi.io/cluster: my-connect
spec:
  class: org.apache.kafka.connect.file.FileStreamSinkConnector ❶
  tasksMax: 2

```

```
config: 2
file: "/tmp/my-file" 3
topics: my-topic 4
```

- 1 コネクタークラスのフルネームまたはエイリアス。これは、Kafka Connect クラスターによって使用されているイメージに存在するはずです。
- 2 キーと値のペアとしての [コネクタ設定](#)。
- 3 ソースデータのパブリッシュ先となる一時ファイル。
- 4 ソースデータの読み取り元となる Kafka トピック。

5. OpenShift クラスターにシンク **KafkaConnector** を作成します。

```
oc apply -f examples/connect/sink-connector.yaml
```

6. コネクタリソースが作成されたことを確認します。

```
oc get kctr --selector strimzi.io/cluster=MY-CONNECT-CLUSTER -o name
my-source-connector
my-sink-connector
```

MY-CONNECT-CLUSTER を Kafka Connect クラスターに置き換えます。

7. コンテナで、**kafka-console-consumer.sh** を実行して、ソースコネクタによってトピックに書き込まれたメッセージを読み取ります。

```
oc exec MY-CLUSTER-kafka-0 -i -t -- bin/kafka-console-consumer.sh --bootstrap-server MY-CLUSTER-kafka-bootstrap.NAMESPACE.svc:9092 --topic my-topic --from-beginning
```

ソースおよびシンクコネクタの設定オプション

コネクタ設定は、**KafkaConnector** リソースの **spec.config** プロパティで定義されます。

FileStreamSourceConnector クラスおよび **FileStreamSinkConnector** クラスは、Kafka Connect REST API と同じ設定オプションをサポートします。他のコネクタは異なる設定オプションをサポートします。

表6.1 FileStreamSource コネクタークラスの設定オプション

名前	タイプ	デフォルト値	説明
file	文字列	Null	メッセージを書き込むソースファイル。指定のない場合は、標準入力を使用されます。
topic	List	Null	データのパブリッシュ先となる Kafka トピック。

表6.2 FileStreamSinkConnector クラスの設定オプション

名前	タイプ	デフォルト値	説明
file	文字列	Null	メッセージを書き込む宛先ファイル。指定のない場合は標準出力が使用されます。
topics	List	Null	データの読み取り元となる1つ以上の Kafka トピック。
topics.regex	文字列	Null	データの読み取り元となる1つ以上の Kafka トピックと一致する正規表現。

6.2.4.3. Kafka コネクターの再起動の実行

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

前提条件

- Cluster Operator が稼働中である。

手順

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

```
oc get KafkaConnector
```

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

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

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

6.2.4.4. Kafka コネクタータスクの再起動の実行

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

前提条件

- Cluster Operator が稼働中である。

手順

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

```
oc get KafkaConnector
```

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

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

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

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

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

6.2.4.5. Kafka Connect API の公開

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



注記

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

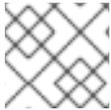
コネクタ設定を JSON オブジェクトとして追加できます。

コネクタ設定を追加するための curl 要求の例

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

API には OpenShift クラスター内でのみアクセスできます。OpenShift クラスター外部で実行しているアプリケーションに Kafka Connect API がアクセスできるようにする場合は、以下の機能のいずれかを使用して Kafka Connect API を手動で公開できます。

- **LoadBalancer** または **NodePort** タイプのサービス
- **Ingress** リソース
- OpenShift ルート



注記

接続は安全ではないため、外部からのアクセスはよく考えてから許可してください。

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

サービスのセレクター設定

```
# ...
selector:
  strimzi.io/cluster: my-connect-cluster ❶
  strimzi.io/kind: KafkaConnect
  strimzi.io/name: my-connect-cluster-connect ❷
#...
```

- ❶ OpenShift クラスターでの Kafka Connect カスタムリソースの名前。
- ❷ Cluster Operator によって作成された Kafka Connect デプロイメントの名前。

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

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

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

```
strimzi.io/name: my-connect-cluster-connect
policyTypes:
- Ingress
```

- 1 API への接続が許可される Pod のラベル。

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

6.3. KAFKA MIRRORMAKER のデプロイ

Cluster Operator によって、1つ以上の Kafka MirrorMaker のレプリカがデプロイされ、Kafka クラスターの間でデータが複製されます。このプロセスは、Kafka パーティションのレプリケーションの概念と混同しないように、ミラーリングと呼ばれます。MirrorMaker は、ソースクラスターからメッセージを消費し、これらのメッセージをターゲットクラスターにパブリッシュします。

6.3.1. Kafka MirrorMaker の OpenShift クラスターへのデプロイ

この手順では、Cluster Operator を使用して Kafka MirrorMaker クラスターを OpenShift クラスターにデプロイする方法を説明します。

デプロイメントでは、YAML ファイルの仕様を使用して、デプロイされた MirrorMaker のバージョンに応じて **KafkaMirrorMaker** または **KafkaMirrorMaker2** リソースが作成されます。



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。

AMQ Streams には、[設定ファイルのサンプル](#) が用意されています。この手順では、以下のサンプルファイルを使用します。

- `examples/mirror-maker/kafka-mirror-maker.yaml`
- `examples/mirror-maker/kafka-mirror-maker-2.yaml`

前提条件

- [Cluster Operator](#) がデプロイされている。

手順

1. Kafka MirrorMaker を OpenShift クラスターにデプロイします。
MirrorMaker の場合

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

MirrorMaker 2.0 の場合

■

```
oc apply -f examples/mirror-maker/kafka-mirror-maker-2.yaml
```

2. 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-mirror-maker-mirror-maker	1/1	1	1
my-mm2-cluster-mirrormaker2	1/1	1	1

my-mirror-maker は、Kafka MirrorMaker クラスターの名前です。**my-mm2-cluster** は Kafka MirrorMaker 2.0 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。

関連情報

- [Kafka MirrorMaker クラスターの設定](#)

6.4. KAFKA BRIDGE のデプロイ

Cluster Operator によって、1つ以上の Kafka Bridge のレプリカがデプロイされ、HTTP API 経由で Kafka クラスターとクライアントの間でデータが送信されます。

6.4.1. Kafka Bridge を OpenShift クラスターへデプロイ

この手順では、Cluster Operator を使用して Kafka Bridge クラスターを OpenShift クラスターにデプロイする方法を説明します。

デプロイメントでは、YAML ファイルの仕様を使用して **KafkaBridge** リソースが作成されます。

AMQ Streams には、[設定ファイルのサンプル](#) が用意されています。この手順では、以下のサンプルファイルを使用します。

- **examples/bridge/kafka-bridge.yaml**

前提条件

- [Cluster Operator](#) がデプロイされている。

手順

1. Kafka Bridge を OpenShift クラスターにデプロイします。

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

2. 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-bridge-bridge 1/1    1          1
```

my-bridge は、Kafka Bridge 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。

関連情報

- [Kafka Bridge クラスターの設定](#)
- [AMQ Streams Kafka Bridge の使用](#)

6.4.2. Kafka Bridge サービスのローカルマシンへの公開

ポート転送を使用して AMQ Streams の Kafka Bridge サービスを <http://localhost:8080> 上でローカルマシンに公開します。



注記

ポート転送は、開発およびテストの目的でのみ適切です。

手順

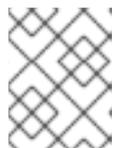
1. OpenShift クラスターの Pod の名前をリストします。

```
oc get pods -o name

pod/kafka-consumer
# ...
pod/quickstart-bridge-589d78784d-9jcnr
pod/strimzi-cluster-operator-76bcf9bc76-8dnfm
```

2. ポート **8080** で Kafka Bridge Pod に接続します。

```
oc port-forward pod/quickstart-bridge-589d78784d-9jcnr 8080:8080 &
```



注記

ローカルマシンのポート 8080 がすでに使用中の場合は、代替の HTTP ポート (**8008** など) を使用します。

これで、API リクエストがローカルマシンのポート 8080 から Kafka Bridge Pod のポート 8080 に転送されるようになります。

6.4.3. OpenShift 外部の Kafka Bridge へのアクセス

デプロイメント後、AMQ Streams Kafka Bridge には同じ OpenShift クラスターで実行しているアプリケーションのみがアクセスできます。これらのアプリケーションは、<kafka_bridge_name>-bridge-service サービスを使用して API にアクセスします。

OpenShift クラスター外部で実行しているアプリケーションに Kafka Bridge がアクセスできるようにする場合は、以下の機能のいずれかを作成して Kafka Bridge を手動で公開できます。

- **LoadBalancer** または **NodePort** タイプのサービス
- **Ingress** リソース
- OpenShift ルート

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

```
# ...
selector:
  strimzi.io/cluster: kafka-bridge-name 1
  strimzi.io/kind: KafkaBridge
#...
```

- 1 OpenShift クラスターでの Kafka Bridge カスタムリソースの名前。

第7章 KAFKA クラスターへのクライアントアクセスの設定

AMQ Streams のデプロイ 後、本章では以下の操作を行う方法について説明します。

- サンプルプロデューサーおよびコンシューマクライアントをデプロイし、これを使用してデプロイメントを検証する
- Kafka クラスターへの外部クライアントアクセスを設定する
OpenShift 外部のクライアントに Kafka クラスターへのアクセスを設定する手順はより複雑で、[Kafka コンポーネントの設定手順](#) に精通している必要があります。

7.1. サンプルクライアントのデプロイ

この手順では、ユーザーが作成した Kafka クラスターを使用してメッセージを送受信するプロデューサーおよびコンシューマクライアントの例をデプロイする方法を説明します。

前提条件

- クライアントが Kafka クラスターを使用できる。

手順

1. Kafka プロデューサーをデプロイします。

```
oc run kafka-producer -ti --image=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.2
--rm=true --restart=Never -- bin/kafka-console-producer.sh --bootstrap-server cluster-name-
kafka-bootstrap:9092 --topic my-topic
```

2. プロデューサーが稼働しているコンソールにメッセージを入力します。
3. **Enter** を押してメッセージを送信します。
4. Kafka コンシューマーをデプロイします。

```
oc run kafka-consumer -ti --image=registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.2
--rm=true --restart=Never -- bin/kafka-console-consumer.sh --bootstrap-server cluster-
name-kafka-bootstrap:9092 --topic my-topic --from-beginning
```

5. コンシューマーコンソールに受信メッセージが表示されることを確認します。

7.2. OPENSIFT 外クライアントのアクセスの設定

以下の手順では、OpenShift 外部からの Kafka クラスターへのクライアントアクセスを設定する方法を説明します。

Kafka クラスターのアドレスを使用して、異なる OpenShift namespace または完全に OpenShift 外のクライアントに外部アクセスを提供できます。

アクセスを提供するために、外部 Kafka リスナーを設定します。

以下のタイプの外部リスナーがサポートされます。

- OpenShift **Route** およびデフォルトの HAProxy ルーターを使用する **route**

- ロードバランサーサービスを使用する **loadbalancer**
- OpenShift ノードのポートを使用する **nodeport**
- OpenShift **Ingress** と **NGINX Ingress Controller for Kubernetes** を使用する **ingress**

要件ならびにお使いの環境およびインフラストラクチャーに応じて、選択するタイプは異なります。たとえば、ロードバランサーは、ベアメタルなどの特定のインフラストラクチャーには適さない場合があります。ベアメタルでは、ノードポートがより適したオプションを提供します。

以下の手順では、

1. TLS 暗号化および認証、ならびに Kafka **簡易承認** を有効にして、Kafka クラスターに外部リスナーが設定されます。
2. **簡易承認** 用に TLS 認証およびアクセス制御リスト (ACL) を定義して、クライアントに **KafkaUser** が作成されます。

TLS、SCRAM-SHA-512、または OAuth 2.0 認証を使用するようにリスナーを設定できます。TLS は常に暗号化を使用しますが、SCRAM-SHA-512 および OAuth 2.0 認証でも暗号化を使用することが推奨されます。

Kafka ブローカーにシンプルな、OAuth 2.0、OPA、またはカスタム承認を設定できます。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。

KafkaUser 認証および承認メカニズムを設定する場合は、必ず同等の Kafka 設定と一致させてください。

- **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。

KafkaUser に使用する認証をサポートするリスナーが少なくとも1つ必要です。



注記

Kafka ユーザーと Kafka ブローカー間の認証は、それぞれの認証設定によって異なります。たとえば、TLS が Kafka 設定で有効になっていない場合は、TLS でユーザーを認証できません。

AMQ Streams Operator により設定プロセスが自動されます。

- 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。
- 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。

この手順では、Cluster Operator によって生成された証明書が使用されますが、[独自の証明書をインストール](#)してそれらを置き換えることができます。[外部認証局によって管理される Kafka リスナー証明書を使用するようにリスナーを設定](#)することもできます。

PKCS #12 (.p12) 形式および PEM (.crt) 形式の証明書を利用できます。この手順では、PKCS #12 証明書を説明します。

前提条件

- クライアントが Kafka クラスターを使用できる。
- Cluster Operator および User Operator がクラスターで実行されている必要があります。
- OpenShift クラスター外のクライアントが Kafka クラスターに接続できる。

手順

1. **external** Kafka リスナーと共に Kafka クラスターを設定します。

- リスナーを通じて Kafka ブローカーにアクセスするのに必要な認証を定義します。
- Kafka ブローカーで承認を有効にします。
以下に例を示します。

```

apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
  namespace: myproject
spec:
  kafka:
    # ...
    listeners: 1
    - name: external 2
      port: 9094 3
      type: LISTENER-TYPE 4
      tls: true 5
      authentication:
        type: tls 6
      configuration:
        preferredNodePortAddressType: InternalDNS 7
        bootstrap and broker service overrides 8
    #...
    authorization: 9
      type: simple
      superUsers:
        - super-user-name 10
    # ...

```

- 1 外部リスナーを有効にする設定オプションは、[汎用 Kafka リスナースキーマ参照](#)に記載されています。
- 2 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。
- 3 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。
- 4 **route**、**loadbalancer**、**nodeport**、または **ingress** として指定される外部リスナータイプ。内部リスナーは **internal** として指定されます。

- 5 リスナーで TLS による暗号化を有効にします。デフォルトは **false** です。 **route** リスナーには TLS 暗号化は必要ありません。
- 6 認証は **tls** として指定されます。
- 7 (任意設定: **nodeport** リスナーのみ) ノードアドレスとして AMQ Streams により使用される最初のアドレスタイプの希望を指定します。
- 8 (任意設定) AMQ Streams はクライアントに公開するアドレスを自動的に決定します。アドレスは OpenShift によって自動的に割り当てられます。AMQ Streams を実行しているインフラストラクチャーが正しいアドレスを提供しない場合、**ブートストラップおよびブローカーサービス** アドレスを上書きできます。検証はオーバーライドに対しては実行されません。オーバーライド設定はリスナーのタイプによって異なります。たとえば、**route** の場合はホストを、**loadbalancer** の場合は DNS 名または IP アドレスを、また **nodeport** の場合はノードポートを、それぞれ上書きすることができます。
- 9 **simple** と指定された承認 (**AclAuthorizer** Kafka プラグインを使用する)。
- 10 (任意設定) スーパーユーザーは、ACL で定義されたアクセス制限に関係なく、すべてのブローカーにアクセスできます。



警告

OpenShift Route アドレスは、Kafka クラスターの名前、リスナーの名前、および作成される namespace の名前を設定されます。たとえば、**my-cluster-kafka-listener1-bootstrap-myproject** (**CLUSTER-NAME-kafka-LISTENER-NAME-bootstrap-NAMESPACE**) となります。**route** リスナータイプを使用している場合、アドレス全体の長さが上限の 63 文字を超えないように注意してください。

2. 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、TLS 認証を使用する Kafka ブローカーリスナーと共に設定されます。

Kafka ブローカー Pod ごとにサービスが作成されます。

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

サービスは、**nodeport** リスナーを使用した Kafka クラスターへの外部接続用 **外部ブートストラップアドレス** としても作成されます。

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



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。

3. **Kafka** リソースのステータスからブートストラップアドレスおよびポートを見つけます。

```
oc get kafka KAFKA-CLUSTER-NAME -o jsonpath='{.status.listeners[?(@.name=="external")].bootstrapServers}'
```

Kafka クライアントのブートストラップアドレスを使用して、Kafka クラスターに接続します。

4. Kafka クラスターにアクセスする必要があるクライアントに対応するユーザーを作成または変更します。

- **Kafka** リスナーと同じ認証タイプを指定します。
- 簡易承認に承認 ACL を指定します。
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 ❶
spec:
  authentication:
    type: tls ❷
  authorization:
    type: simple
  acls: ❸
    - resource:
        type: topic
        name: my-topic
        patternType: literal
        operation: Read
    - resource:
        type: topic
        name: my-topic
        patternType: literal
        operation: Describe
    - resource:
        type: group
        name: my-group
        patternType: literal
        operation: Read
```

- ❶ ラベルは、作成するユーザーの Kafka クラスターのラベルと一致する必要があります。
- ❷ 認証は **tls** として指定されます。
- ❸ 簡易承認には、ユーザーに適用する ACL ルールのリストが必要です。ルールは、**ユーザー名 (my-user)** を基に Kafka リソースで許可される操作を定義します。

ユーザ (`my-user`) を含む `KafkaUser` リソースを作成し、必要に応じて既存の探検に追加します。

5. **KafkaUser** リソースを作成または変更します。

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

KafkaUser リソースと同じ名前の Secret と共に、ユーザーが作成されます。Secret には、TLS クライアント認証の秘密鍵と公開鍵が含まれます。

以下に例を示します。

```
apiVersion: v1
kind: Secret
metadata:
  name: my-user
labels:
  strimzi.io/kind: KafkaUser
  strimzi.io/cluster: my-cluster
type: Opaque
data:
  ca.crt: PUBLIC-KEY-OF-THE-CLIENT-CA
  user.crt: USER-CERTIFICATE-CONTAINING-PUBLIC-KEY-OF-USER
  user.key: PRIVATE-KEY-OF-USER
  user.p12: P12-ARCHIVE-FILE-STORING-CERTIFICATES-AND-KEYS
  user.password: PASSWORD-PROTECTING-P12-ARCHIVE
```

6. パブリッククラスター CA 証明書を必要な証明書形式にデプロイメントします。

```
oc get secret KAFKA-CLUSTER-NAME-cluster-ca-cert -o jsonpath='{.data.ca.p12}' | base64
-d > ca.p12
```

7. パスワードファイルからパスワードを抽出します。

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

8. パブリッククラスター証明書の認証情報でクライアントを設定します。

クライアントコードのサンプル

```
properties.put("security.protocol","SSL"); 1
properties.put(SslConfigs.SSL_TRUSTSTORE_LOCATION_CONFIG,"/path/to/ca.p12"); 2
properties.put(SslConfigs.SSL_TRUSTSTORE_PASSWORD_CONFIG,CA-PASSWORD);
3
properties.put(SslConfigs.SSL_TRUSTSTORE_TYPE_CONFIG,"PKCS12"); 4
```

- 1** (TLS クライアント認証ありまたはなしで) TLS による暗号化を有効にします。
- 2** 証明書がインポートされたトラストストアの場所を指定します。
- 3** 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します。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。

- 4 トラストストアのタイプを識別します。



注記

TLS 経由で SCRAM-SHA 認証を使用する場合は、**security.protocol: SASL_SSL** を使用します。

9. ユーザー Secret から必要な証明書形式にユーザー CA 証明書を抽出します。

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

10. パスワードファイルからパスワードを抽出します。

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

11. ユーザー CA 証明書の認証情報でクライアントを設定します。

クライアントコードのサンプル

```
properties.put(SslConfigs.SSL_KEYSTORE_LOCATION_CONFIG, "/path/to/user.p12"); 1
properties.put(SslConfigs.SSL_KEYSTORE_PASSWORD_CONFIG, "<user.password>"); 2
properties.put(SslConfigs.SSL_KEYSTORE_TYPE_CONFIG, "PKCS12"); 3
```

- 1 証明書がインポートされたキーストアの場所を指定します。
- 2 キーストアにアクセスするためのパスワードを指定します。このプロパティは、キーストアで必要なければ省略できます。パブリックユーザー証明書は、作成時にクライアント CA により署名されます。
- 3 キーストアのタイプを識別します。

12. 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およびポートを追加します。

```
bootstrap.servers: BOOTSTRAP-ADDRESS:PORT
```

関連情報

- [リスナー認証オプション](#)
- [Kafka 承認オプション](#)
- 承認サーバーを使用している場合は、トークンベースの [OAuth 2.0 認証](#) および [OAuth 2.0 承認](#) を使用できます。

第8章 AMQ STREAMS のメトリクスおよびダッシュボードの設定

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。

ダッシュボードでキーメトリクスを表示し、特定の条件下でトリガーされるアラートを設定すると、AMQ Streams デプロイメントを監視できます。メトリックは、AMQ Streams の各コンポーネントで利用できます。

AMQ Streams は、メトリクス情報を提供するために、Prometheus ルールと Grafana ダッシュボードを使用します。

Prometheus に AMQ Streams の各コンポーネントのルールセットが設定されている場合、Prometheus はクラスターで稼働している Pod からキーメトリックを使用します。次に、Grafana はこれらのメトリックをダッシュボードで可視化します。AMQ Streams には、デプロイメントに合わせてカスタマイズできる Grafana ダッシュボードのサンプルが含まれています。

AMQ Streams は、**ユーザー定義プロジェクトのモニタリング** (OpenShift 機能) を使用して、Prometheus の設定プロセスを単純化します。

要件に応じて以下を行うことができます。

- [メトリクスを公開するための Prometheus の設定およびデプロイ](#)
- [追加のメトリクスを提供するための Kafka Exporter のデプロイ](#)
- [Grafana を使用した Prometheus メトリクスの表示](#)

Prometheus および Grafana が設定されると、監視に AMQ Streams が提供する Grafana ダッシュボードのサンプルを使用できます。

さらに、[分散トレーシングを設定](#) して、エンドツーエンドのメッセージ追跡を行うようにデプロイメントを設定することもできます。



注記

AMQ Streams は、Prometheus と Grafana のインストールファイルの例を提供します。AMQ Streams の監視を試みる際に、このファイルを開始点として使用できます。さらにサポートするには、Prometheus および Grafana 開発者コミュニティに参加してみてください。

メトリクスおよびモニタリングツールのサポートドキュメント

メトリクスおよびモニタリングツールの詳細は、サポートドキュメントを参照してください。

- [Prometheus](#)
- [Prometheus の設定](#)
- [Kafka Exporter](#)
- [Grafana Labs](#)
- [Apache Kafka Monitoring](#) では、Apache Kafka により公開される JMX メトリクスについて解説しています。

- [ZooKeeper JMX](#) では、Apache Zookeeper により公開される JMX メトリックについて解説しています。

8.1. KAFKA EXPORTER でのコンシューマーラグの監視

[Kafka Exporter](#) は、Apache Kafka ブローカーおよびクライアントの監視を強化するオープンソースプロジェクトです。[Kafka クラスタで Kafka Exporter をデプロイ](#) するように、**Kafka** リソースを設定できます。Kafka Exporter は、オフセット、コンシューマーグループ、コンシューマーラグ、およびトピックに関連する Kafka ブローカーから追加のメトリックデータを抽出します。一例として、メトリクスデータを使用すると、低速なコンシューマーの識別に役立ちます。ラグデータは Prometheus メトリクスとして公開され、解析のために Grafana で使用できます。



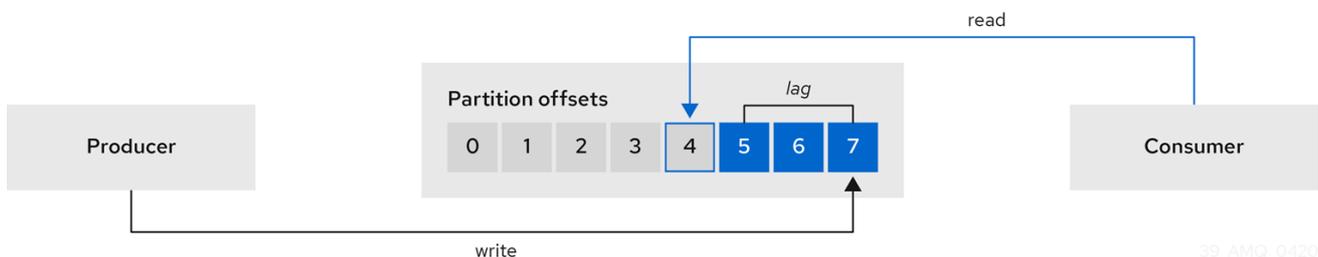
重要

Kafka Exporter は、コンシューマーラグおよびコンシューマーオフセットに関連する追加のメトリックのみを提供します。通常の Kafka メトリクスでは、[Kafka ブローカー](#) で、Prometheus メトリクスを設定する必要があります。

コンシューマーラグは、メッセージの生成と消費の差を示しています。具体的には、指定のコンシューマーグループのコンシューマーラグは、パーティションの最後のメッセージと、そのコンシューマーが現在ピックアップしているメッセージとの時間差を示しています。

ラグには、パーティションログの最後を基準とする、コンシューマーオフセットの相対的な位置が反映されます。

プロデューサーおよびコンシューマーオフセット間のコンシューマーラグ



39 AMQ_0420

この差は、Kafka ブローカートピックパーティションの読み取りと書き込みの場所である、プロデューサーオフセットとコンシューマーオフセットの間の **デルタ** とも呼ばれます。

あるトピックで毎秒 100 個のメッセージがストリーミングされる場合を考えてみましょう。プロデューサーオフセット (トピックパーティションの先頭) と、コンシューマーが読み取った最後のオフセットとの間のラグが 1000 個のメッセージであれば、10 秒の遅延があることを意味します。

コンシューマーラグ監視の重要性

可能な限りリアルタイムのデータの処理に依存するアプリケーションでは、コンシューマーラグを監視して、ラグが過度に大きくならないようにチェックする必要があります。ラグが大きくなるほど、プロセスはリアルタイム処理の目的から遠ざかります。

たとえば、コンシューマーラグは、ページされていない古いデータを大量に消費したり、計画外のシャットダウンが原因である可能性があります。

コンシューマーラグの削減

Grafana のチャートを使用して、ラグを分析し、ラグ削減の方法が対象のコンシューマーグループに影響しているかどうかを確認します。たとえば、ラグを減らすように Kafka ブローカーを調整すると、ダッシュボードには **コンシューマーグループ別のラグ** のチャートが下降し **毎分のメッセージ消費** の

チャートが上昇する状況が示されます。

通常、ラグを削減するには以下を行います。

- 新規コンシューマーを追加してコンシューマーグループをスケールアップします。
- メッセージがトピックに留まる保持時間を延長します。
- ディスク容量を追加してメッセージバッファを増やします。

コンシューマーラグを減らす方法は、基礎となるインフラストラクチャーや、AMQ Streams によりサポートされるユースケースによって異なります。たとえば、ラグが生じているコンシューマーでは、ディスクキャッシュからフェッチリクエストに対応できるブローカーを活用できる可能性は低いでしょう。場合によっては、コンシューマーの状態が改善されるまで、自動的にメッセージをドロップすることが許容されることがあります。

8.2. CRUISE CONTROL 操作の監視

Cruise Control は、ブローカー、トピック、およびパーティションの使用状況を追跡するために Kafka ブローカーを監視します。Cruise Control は、独自のパフォーマンスを監視するためのメトリックのセットも提供します。

Cruise Control メトリックレポーターは、Kafka ブローカーから未加工のメトリックデータを収集します。データは、Cruise Control によって自動的に作成されるトピックに生成されます。メトリックは、[Kafka クラスターの最適化提案の生成](#) に使用されます。

Cruise Control メトリックは、Cruise Control 操作のリアルタイム監視で利用できます。たとえば、Cruise Control メトリックを使用して、実行中のリバランス操作のステータスを監視したり、操作のパフォーマンスで検出された異常についてアラートを提供したりできます。

Cruise Control 設定で [Prometheus JMX Exporter](#) を有効にして Cruise Control メトリクスを公開します。



注記

センサーとして知られる利用可能な Cruise Control メトリクスの完全なリストは、[Cruise Control のドキュメント](#)を参照してください。

8.2.1. Cruise Control メトリックの公開

Cruise Control 操作でメトリクスを公開する場合は、**Kafka** リソースを設定して、[Cruise Control](#) をデプロイし、[デプロイメント](#)で [Prometheus メトリクスを有効にします](#)。独自の設定を使用するか、AMQ Streams によって提供される `kafka-cruise-control-metrics.yaml` ファイルのサンプルを使用できます。

設定を **Kafka** リソースの **CruiseControl** プロパティの `metricsConfig` に追加します。この設定により、[Prometheus JMX Exporter](#) が有効化され、HTTP エンドポイント経由で Cruise Control メトリクスが公開されます。HTTP エンドポイントは Prometheus サーバーによってスクレイプされます。

Cruise Control のメトリック設定例

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

```

Spec:
# ...
cruiseControl:
# ...
metricsConfig:
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: cruise-control-metrics
      key: metrics-config.yml
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: cruise-control-metrics
labels:
  app: strimzi
data:
  metrics-config.yml: |
# metrics configuration...

```

8.2.2. Cruise Control メトリックの表示

Cruise Control メトリクスを公開したら、Prometheus または別の適切なモニタリングシステムを使用して、メトリクスデータの情報を表示できます。Streams for Apache Kafka は、Cruise Control メトリクスの視覚化を表示する [Grafana ダッシュボードのサンプル](#) を提供します。ダッシュボードは **strimzi-cruise-control.json** という名前の JSON ファイルです。公開されるメトリクスは、[Grafana ダッシュボードを有効にする](#) 際に監視データを提供します。

8.2.2.1. 分散スコアの監視

Cruise Control メトリックには、分散スコアが含まれます。分散度は、Kafka クラスタ内でワークロードがどの程度均等に分散されているかを示す尺度です。

分散スコア (**balancedness-score**) の Cruise Control メトリクスは、**KafkaRebalance** リソースの分散スコアとは異なる可能性があります。Cruise Control は **anomaly.detection.goals** を使用して各スコアを計算します。これは、**KafkaRebalance** リソースで使用される **default.goals** と同じでない可能性があります。**anomaly.detection.goals** は、**Kafka** カスタムリソースの **spec.cruiseControl.config** に指定されます。

注記

KafkaRebalance リソースを更新すると、最適化プロポーザルをフェッチします。以下の条件のいずれかが適用されると、キャッシュされた最新の最適化プロポーザルがフェッチされます。

- **KafkaRebalance goals** は、**Kafka** リソースの **default.goals** セクションに設定されたゴールと一致する。
- **KafkaRebalance goals** は指定されていない。

これ以外の場合は、Cruise Control は **KafkaRebalance goals** に基づいて、新しい最適化プロポーザルを生成します。更新ごとに新しいプロポーザルが生成されると、パフォーマンスの監視に影響を及ぼす可能性があります。

8.2.2.2. 異常検出へのアラート

Cruise Control の **異常検出** は、ブローカーの障害などの最適化ゴールの生成をブロックする条件のメトリクスデータを提供します。可視性を高める場合は、異常検出器が提供するメトリックを使用して、アラートを設定し、通知を送信できます。Cruise Control の **異常通知機能** を設定して、指定された通知チャンネルを介してこれらのメトリクスに基づいてアラートをルーティングできます。または、Prometheus を設定して、異常検出器によって提供されるメトリックデータをスクレープし、アラートを生成することもできます。その後、Prometheus Alertmanager は Prometheus で生成されるアラートをルーティングできます。

[Cruise Control ドキュメント](#) には、**AnomalyDetector** メトリクスおよび異常通知機能に関する情報が記載されています。

8.3. メトリクスファイルの例

Grafana ダッシュボードおよびその他のメトリクス設定ファイルの例は、AMQ Streams によって提供される [設定ファイルの例](#) を参照してください。

AMQ Streams で提供されるサンプルメトリクスファイル

```

metrics
├── grafana-dashboards 1
│   ├── strimzi-cruise-control.json
│   ├── strimzi-kafka-bridge.json
│   ├── strimzi-kafka-connect.json
│   ├── strimzi-kafka-exporter.json
│   ├── strimzi-kafka-mirror-maker-2.json
│   ├── strimzi-kafka.json
│   ├── strimzi-operators.json
│   └── strimzi-zookeeper.json
├── grafana-install
│   └── grafana.yaml 2
├── prometheus-additional-properties
│   └── prometheus-additional.yaml 3
├── prometheus-alertmanager-config
│   └── alert-manager-config.yaml 4
├── prometheus-install
│   ├── alert-manager.yaml 5
│   ├── prometheus-rules.yaml 6
│   ├── prometheus.yaml 7
│   └── strimzi-pod-monitor.yaml 8
├── kafka-bridge-metrics.yaml 9
├── kafka-connect-metrics.yaml 10
├── kafka-cruise-control-metrics.yaml 11
├── kafka-metrics.yaml 12
└── kafka-mirror-maker-2-metrics.yaml 13

```

1 異なる AMQ Streams コンポーネントの Grafana ダッシュボードの例。

2 Grafana イメージのインストールファイル。

3

CPU、メモリー、およびディスクボリュームの使用状況についてのメトリックをスクレールする追加の設定。これらのメトリックは、ノード上の OpenShift cAdvisor エージェントおよび kubelet

- 4 Alertmanager による通知送信のためのフック定義。
- 5 Alertmanager をデプロイおよび設定するためのリソース。
- 6 Prometheus Alertmanager と使用するアラートルールの例 (Prometheus とデプロイ)。
- 7 Prometheus イメージのインストーリソースファイル。
- 8 Prometheus Operator によって Prometheus サーバーのジョブに変換される PodMonitor の定義。これにより、Pod から直接メトリックデータをスクレールできます。
- 9 メトリックが有効になっている Kafka Bridge リソース。
- 10 Kafka Connect に対する Prometheus JMX Exporter の再ラベル付けルールを定義するメトリック設定。
- 11 Cruise Control に対する Prometheus JMX Exporter の再ラベル付けルールを定義するメトリック設定。
- 12 Kafka および ZooKeeper に対する Prometheus JMX Exporter の再ラベル付けルールを定義するメトリック設定。
- 13 Kafka Mirror Maker 2.0 に対する Prometheus JMX Exporter の再ラベル付けルールを定義するメトリクス設定。

8.3.1. Prometheus メトリクス設定の例

AMQ Streams は、[Prometheus JMX Exporter](#) を使用して、Prometheus サーバーによってスクレールできる HTTP エンドポイント経由でメトリクスを公開します。

Grafana ダッシュボードが依存する Prometheus JMX Exporter の再ラベル付けルールは、カスタムリソース設定として AMQ Streams コンポーネントに対して定義されます。

ラベルは名前と値のペアです。再ラベル付けは、ラベルを動的に書き込むプロセスです。たとえば、ラベルの値は Kafka サーバーおよびクライアント ID の名前から派生されます。

AMQ Streams では、再ラベル付けルールが含まれるカスタムリソース設定用の YAML ファイルのサンプルが提供されます。Prometheus メトリック設定をデプロイする場合、カスタムリソースのサンプルをデプロイすることや、メトリック設定を独自のカスタムリソース定義にコピーすることができます。

表8.1メトリック設定を含むカスタムリソースの例

コンポーネント	カスタムリソース	サンプル YAML ファイル
Kafka および ZooKeeper	Kafka	kafka-metrics.yaml
Kafka Connect	KafkaConnect	kafka-connect-metrics.yaml
Kafka MirrorMaker 2.0	KafkaMirrorMaker2	kafka-mirror-maker-2-metrics.yaml

コンポーネント	カスタムリソース	サンプル YAML ファイル
Kafka Bridge	KafkaBridge	kafka-bridge-metrics.yaml
Cruise Control	Kafka	kafka-cruise-control-metrics.yaml

8.3.2. アラート通知の Prometheus ルールの例

アラート通知の Prometheus ルールの例は、AMQ Streams によって提供される [メトリクス設定ファイルの例](#) と共に提供されます。ルールは、[Prometheus デプロイメント](#) で使用するための **prometheus-rules.yaml** ファイルのサンプルに指定されています。

アラートルールによって、メトリックで監視される特定条件についての通知が提供されます。ルールは Prometheus サーバーで宣言されますが、アラート通知は Prometheus Alertmanager で対応します。

Prometheus アラートルールでは、継続的に評価される [PromQL](#) 表現を使用して条件が記述されます。

アラート表現が true になると、条件が満たされ、Prometheus サーバーからアラートデータが Alertmanager に送信されます。次に Alertmanager は、そのデプロイメントに設定された通信方法を使用して通知を送信します。

アラートルールの定義に関する一般的な留意点:

- **for** プロパティは、ルールと併用し、アラートがトリガーされるまでに、条件を維持する必要のある期間を決定します。
- ティック (tick) は ZooKeeper の基本的な時間単位です。ミリ秒単位で測定され、**Kafka.spec.zookeeper.config** の **tickTime** パラメーターを使用して設定されます。たとえば、ZooKeeper で **tickTime=3000** の場合、3 ティック (3 x 3000) は 9000 ミリ秒と等しくなります。
- **ZookeeperRunningOutOfSpace** メトリックおよびアラートを利用できるかどうかは、使用される OpenShift 設定およびストレージ実装によります。特定のプラットフォームのストレージ実装では、メトリクスによるアラートの提供に必要な利用可能な領域について情報が提供されない場合があります。

Alertmanager は、電子メール、チャットメッセージなどの通知方法を使用するように設定できます。ルールの例に含まれるデフォルト設定は、特定のニーズに合わせて調整してください。

8.3.2.1. ルールの変更例

prometheus-rules.yaml ファイルには、以下のコンポーネントのルールの例が含まれます。

- Kafka
- ZooKeeper
- Entity Operator
- Kafka Connect

- Kafka Bridge
- MirrorMaker
- Kafka Exporter

各ルールの例の説明は、ファイルに記載されています。

8.3.3. Grafana ダッシュボードのサンプル

Prometheus をデプロイしてメトリックを提供する場合は、AMQ Streams で提供される Grafana ダッシュボードのサンプルを使用して、AMQ Streams コンポーネントを監視できます。

ダッシュボードのサンプルは、**examples/metrics/grafana-dashboards** ディレクトリーに JSON ファイルで提供されます。

すべてのダッシュボードは、JVM メトリクスに加えてコンポーネントに固有のメトリクスを提供します。たとえば、AMQ Streams Operator の Grafana ダッシュボードは、調整の数または処理中のカスタムリソースに関する情報を提供します。

ダッシュボードのサンプルには、Kafka でサポートされるすべてのメトリクスは表示されません。ダッシュボードには、監視用の代表的なメトリックのセットが表示されます。

表8.2 Grafana ダッシュボードの例

コンポーネント	JSON ファイルの例:
AMQ Streams の Operator	strimzi-operators.json
Kafka	strimzi-kafka.json
ZooKeeper	strimzi-zookeeper.json
Kafka Connect	strimzi-kafka-connect.json
Kafka MirrorMaker 2.0	strimzi-kafka-mirror-maker-2.json
Kafka Bridge	strimzi-kafka-bridge.json
Cruise Control	strimzi-cruise-control.json
Kafka Exporter	strimzi-kafka-exporter.json

8.4. PROMETHEUS メトリクス設定のデプロイ

Prometheus メトリック設定をデプロイし、AMQ Streams で Prometheus を使用します。 **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。

独自の設定、または [AMQ Streams で提供されるサンプルのカスタムリソース設定ファイル](#) を使用できます。

- **kafka-metrics.yaml**

- `kafka-connect-metrics.yaml`
- `kafka-mirror-maker-2-metrics.yaml`
- `kafka-bridge-metrics.yaml`
- `kafka-cruise-control-metrics.yaml`

設定ファイルのサンプルには、再ラベル付けルールと Prometheus メトリックの有効化に必要な設定があります。Prometheus は、ターゲット HTTP エンドポイントからメトリクスを収集します。サンプルファイルは、AMQ Streams で Prometheus を試すのに適した方法です。

再ラベル付けルールおよびメトリクス設定を適用するには、以下のいずれかを行います。

- 独自のカスタムリソースに設定例をコピーする。
- メトリクス設定でカスタムリソースをデプロイする。

Kafka Exporter メトリックを含める場合は、`kafkaExporter` 設定を **Kafka** リソースに追加します。



重要

Kafka Exporter は、コンシューマーラグおよびコンシューマーオフセットに関連する追加のメトリックのみを提供します。通常の Kafka メトリクスでは、[Kafka ブローカー](#) で、Prometheus メトリクスを設定する必要があります。

この手順では、**Kafka** リソースに Prometheus メトリクス設定をデプロイする方法を説明します。このプロセスは、他のリソースのサンプルファイルを使用する場合と同じです。

手順

1. Prometheus 設定でカスタムリソースのサンプルをデプロイします。
たとえば、**Kafka** リソースごとに `kafka-metrics.yaml` ファイルを適用します。

サンプル設定のデプロイ

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

または、`kafka-metrics.yaml` の設定例を独自の **Kafka** リソースにコピーすることもできます。

サンプル設定のコピー

```
oc edit kafka <kafka-configuration-file>
```

`metricsConfig` プロパティと、**Kafka** リソースを参照する `ConfigMap` をコピーします。

Kafka のメトリクス設定例

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

```
# ...
metricsConfig: ❶
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: my-key
---
kind: ConfigMap ❷
apiVersion: v1
metadata:
  name: kafka-metrics
  labels:
    app: strimzi
data:
  kafka-metrics-config.yml: |
# metrics configuration...
```

- ❶ メトリクス設定が含まれる ConfigMap を参照する **metricsConfig** プロパティをコピーします。
- ❷ メトリクス設定を指定する **ConfigMap** 全体をコピーします。

注記

Kafka Bridge の場合、**enableMetrics** プロパティを指定し、これを **true** に設定します。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaBridge
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    # ...
  enableMetrics: true
  # ...
```

2. Kafka Exporter をデプロイするには、**kafkaExporter** 設定を追加します。**KafkaExporter** 設定は、**Kafka** リソースでのみ指定されます。

Kafka Exporter のデプロイの設定例

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    image: my-registry.io/my-org/my-exporter-cluster:latest ❶
```

```

groupRegex: ".*" 2
topicRegex: ".*" 3
resources: 4
  requests:
    cpu: 200m
    memory: 64Mi
  limits:
    cpu: 500m
    memory: 128Mi
logging: debug 5
enableSaramaLogging: true 6
template: 7
  pod:
    metadata:
      labels:
        label1: value1
    imagePullSecrets:
      - name: my-docker-credentials
    securityContext:
      runAsUser: 1000001
      fsGroup: 0
    terminationGracePeriodSeconds: 120
  readinessProbe: 8
    initialDelaySeconds: 15
    timeoutSeconds: 5
  livenessProbe: 9
    initialDelaySeconds: 15
    timeoutSeconds: 5
# ...

```

- 1 高度な任意設定: 特別な場合のみ推奨される コンテナイメージの設定。
- 2 メトリックに含まれるコンシューマーグループを指定する正規表現。
- 3 メトリックに含まれるトピックを指定する正規表現。
- 4 予約する CPU およびメモリーリソース。
- 5 指定の重大度 (debug、info、warn、error、fatal) 以上でメッセージをログに記録するためのログ設定。
- 6 Sarama ログを有効にするブール値 (Kafka Exporter によって使用される Go クライアントライブラリー)。
- 7 デプロイメントテンプレートおよび Pod のカスタマイズ。
- 8 ヘルスチェックの readiness プローブ。
- 9 ヘルスチェックの liveness プローブ。

関連情報

- [KafkaExporterTemplate スキーマ参照](#)

- [metricsConfig スキーマ参照](#)

8.5. OPENSIFT での KAFKA メトリックおよびダッシュボードの表示

AMQ Streams が OpenShift Container Platform にデプロイされると、**ユーザー定義プロジェクトのモニタリング**によりメトリクスが提供されます。この OpenShift 機能により、開発者は独自のプロジェクト (例: **Kafka** プロジェクト) を監視するために別の Prometheus インスタンスにアクセスできます。

ユーザー定義プロジェクトのモニタリングが有効な場合は、**openshift-user-workload-monitoring** プロジェクトには以下のコンポーネントが含まれます。

- Prometheus Operator
- Prometheus インスタンス (Prometheus Operator によって自動的にデプロイされます)
- Thanos Ruler インスタンス

AMQ Streams は、これらのコンポーネントを使用してメトリックを消費します。

クラスター管理者は、ユーザー定義プロジェクトのモニタリングを有効にし、開発者およびその他のユーザーに自分のプロジェクトに含まれるアプリケーションを監視するパーミッションを付与する必要があります。

Grafana のデプロイメント

Grafana インスタンスを、Kafka クラスターが含まれるプロジェクトにデプロイできます。その後、Grafana ダッシュボードのサンプルを使用して、AMQ Streams の Prometheus メトリックを Grafana ユーザーインターフェイスで可視化できます。



重要

openshift-monitoring プロジェクトはコアプラットフォームコンポーネントをモニタリングできます。このプロジェクトの Prometheus および Grafana コンポーネントを使用して、OpenShift Container Platform 4.x 上の AMQ Streams の監視を設定しないでください。

手順の概要

OpenShift Container Platform で AMQ Streams のモニタリングを設定するには、以下の手順を順番に行います。

1. 前提条件: [Prometheus メトリクス設定のデプロイ](#)
2. [Prometheus リソースのデプロイ](#)
3. [Grafana のサービスアカウントの作成](#)
4. [Prometheus データソースでの Grafana のデプロイ](#)
5. [Grafana サービスへのルートの作成](#)
6. [Grafana ダッシュボードサンプルのインポート](#)

8.5.1. 前提条件

- YAML ファイルのサンプルを使用して、[Prometheus メトリクス設定がデプロイされている](#)。

- ユーザー定義プロジェクトの監視が有効になっている。クラスター管理者が OpenShift クラスターに **cluster-monitoring-config** の Config Map を作成している。
- クラスター管理者は、**monitoring-rules-edit** または **monitoring-edit** ロールを割り当てている。

cluster-monitoring-config の Config Map の作成およびユーザー定義プロジェクトの監視用のパーミッションをユーザーに付与する方法の詳細は、OpenShift Container Platform の [モニタリング](#) を参照してください。

8.5.2. 関連情報

- OpenShift Container Platform [モニタリング](#)

8.5.3. Prometheus リソースのデプロイ

Prometheus を使用して、Kafka クラスターのモニタリングデータを取得します。

独自の Prometheus デプロイメントを使用するか、AMQ Streams によって提供される [メトリクス設定ファイルのサンプル](#) を使用して Prometheus をデプロイできます。サンプルファイルを使用するには、**PodMonitor** リソースを設定し、デプロイします。**PodMonitors** は、Apache Kafka、ZooKeeper、Operator、Kafka Bridge、および Cruise Control から直接データをスクレールします。

次に、Alertmanager のアラートルールのサンプルをデプロイします。

前提条件

- Kafka クラスターが稼働している。
- AMQ Streams で [提供されるアラートルールのサンプル](#) を確認している。

手順

1. ユーザー定義プロジェクトのモニタリングが有効であることを確認します。

```
oc get pods -n openshift-user-workload-monitoring
```

有効であると、モニタリングコンポーネントの Pod が返されます。以下に例を示します。

NAME	READY	STATUS	RESTARTS	AGE
prometheus-operator-5cc59f9bc6-kgcq8	1/1	Running	0	25s
prometheus-user-workload-0	5/5	Running	1	14s
prometheus-user-workload-1	5/5	Running	1	14s
thanos-ruler-user-workload-0	3/3	Running	0	14s
thanos-ruler-user-workload-1	3/3	Running	0	14s

Pod が返されなければ、ユーザー定義プロジェクトのモニタリングは無効になっています。「[OpenShift での Kafka メトリックおよびダッシュボードの表示](#)」の前提条件を参照してください。

2. 複数の **PodMonitor** リソースは、**examples/metrics/prometheus-install/strimzi-pod-monitor.yaml** で定義されます。

PodMonitor リソースごとに **spec.namespaceSelector.matchNames** プロパティを編集します。

■

```

apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cluster-operator-metrics
  labels:
    app: strimzi
spec:
  selector:
    matchLabels:
      strimzi.io/kind: cluster-operator
  namespaceSelector:
    matchNames:
      - <project-name> ①
  podMetricsEndpoints:
    - path: /metrics
      port: http
  # ...

```

① メトリックをスクレールする Pod が実行されているプロジェクト (例: **Kafka**)。

3. **strimzi-pod-monitor.yaml** ファイルを、Kafka クラスターが稼働しているプロジェクトにデプロイします。

```
oc apply -f strimzi-pod-monitor.yaml -n MY-PROJECT
```

4. Prometheus ルールのサンプルを同じプロジェクトにデプロイします。

```
oc apply -f prometheus-rules.yaml -n MY-PROJECT
```

8.5.4. Grafana のサービスアカウントの作成

AMQ Streams の Grafana インスタンスは、**cluster-monitoring-view** ロールが割り当てられたサービスアカウントで実行する必要があります。

Grafana を使用してモニタリングのメトリクスを表示する場合は、サービスアカウントを作成します。

前提条件

- [Prometheus リソースのデプロイ](#)

手順

1. Grafana の **ServiceAccount** を作成します。ここでは、リソースの名前は **grafana-serviceaccount** です。

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: grafana-serviceaccount
  labels:
    app: strimzi

```

2. **ServiceAccount** を、Kafka クラスターが含まれるプロジェクトにデプロイします。

```
oc apply -f GRAFANA-SERVICEACCOUNT -n MY-PROJECT
```

3. **cluster-monitoring-view** ロールを Grafana **ServiceAccount** に割り当てる **ClusterRoleBinding** リソースを作成します。ここでは、リソースの名前は **grafana-cluster-monitoring-binding** です。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: grafana-cluster-monitoring-binding
  labels:
    app: strimzi
subjects:
  - kind: ServiceAccount
    name: grafana-serviceaccount
    namespace: <my-project> ❶
roleRef:
  kind: ClusterRole
  name: cluster-monitoring-view
  apiGroup: rbac.authorization.k8s.io
```

❶ プロジェクトの名前。

4. **ClusterRoleBinding** を、Kafka クラスターが含まれるプロジェクトにデプロイします。

```
oc apply -f <grafana-cluster-monitoring-binding> -n <my-project>
```

8.5.5. Prometheus データソースを使用した Grafana のデプロイ

Grafana をデプロイし、Prometheus メトリックを表示します。Grafana アプリケーションには、OpenShift Container Platform モニタリングスタックの設定が必要です。

OpenShift Container Platform では、**openshift-monitoring** プロジェクトに **Thanos Querier** インスタンスが含まれています。Thanos Querier は、プラットフォームメトリクスを集約するために使用されません。

必要なプラットフォームメトリクスを使用するには、Grafana インスタンスには Thanos Querier に接続できる Prometheus データソースが必要です。この接続を設定するには、トークンを使用し、Thanos Querier と並行して実行される **oauth-proxy** サイドカーに対して認証を行う config map を作成します。**datasource.yaml** ファイルは config map のソースとして使用されます。

最後に、Kafka クラスターが含まれるプロジェクトにボリュームとしてマウントされた config map で Grafana アプリケーションをデプロイします。

前提条件

- [Prometheus リソースのデプロイ](#)
- [Grafana のサービスアカウントの作成](#)

手順

1. Grafana **ServiceAccount** のアクセストークンを取得します。

```
oc serviceaccounts get-token grafana-serviceaccount -n MY-PROJECT
```

次のステップで使用するアクセストークンをコピーします。

2. Grafana の Thanos Querier 設定が含まれる **datasource.yaml** ファイルを作成します。以下に示すように、アクセストークンを **httpHeaderValue1** プロパティに貼り付けます。

```
apiVersion: 1

datasources:
- name: Prometheus
  type: prometheus
  url: https://thanos-querier.openshift-monitoring.svc.cluster.local:9091
  access: proxy
  basicAuth: false
  withCredentials: false
  isDefault: true
  jsonData:
    timeInterval: 5s
    tlsSkipVerify: true
    httpHeaderName1: "Authorization"
  secureJsonData:
    httpHeaderValue1: "Bearer ${GRAFANA-ACCESS-TOKEN}" ❶
  editable: true
```

❶ **GRAFANA-ACCESS-TOKEN**: Grafana **ServiceAccount** のアクセストークンの値。

3. **datasource.yaml** ファイルから **grafana-config** という名前の config map を作成します。

```
oc create configmap grafana-config --from-file=datasource.yaml -n MY-PROJECT
```

4. **Deployment** および **Service** で設定される Grafana アプリケーションを作成します。**grafana-config** config map はデータソース設定のボリュームとしてマウントされます。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
  labels:
    app: strimzi
spec:
  replicas: 1
  selector:
    matchLabels:
      name: grafana
  template:
    metadata:
      labels:
        name: grafana
    spec:
      serviceAccountName: grafana-serviceaccount
      containers:
      - name: grafana
```

```

image: grafana/grafana:7.5.15
ports:
- name: grafana
  containerPort: 3000
  protocol: TCP
volumeMounts:
- name: grafana-data
  mountPath: /var/lib/grafana
- name: grafana-logs
  mountPath: /var/log/grafana
- name: grafana-config
  mountPath: /etc/grafana/provisioning/datasources/datasource.yaml
  readOnly: true
  subPath: datasource.yaml
readinessProbe:
  httpGet:
    path: /api/health
    port: 3000
  initialDelaySeconds: 5
  periodSeconds: 10
livenessProbe:
  httpGet:
    path: /api/health
    port: 3000
  initialDelaySeconds: 15
  periodSeconds: 20
volumes:
- name: grafana-data
  emptyDir: {}
- name: grafana-logs
  emptyDir: {}
- name: grafana-config
  configMap:
    name: grafana-config
---
apiVersion: v1
kind: Service
metadata:
  name: grafana
  labels:
    app: strimzi
spec:
  ports:
  - name: grafana
    port: 3000
    targetPort: 3000
    protocol: TCP
  selector:
    name: grafana
  type: ClusterIP

```

5. Grafana アプリケーションを、Kafka クラスタが含まれるプロジェクトにデプロイします。

```
oc apply -f <grafana-application> -n <my-project>
```

8.5.6. Grafana サービスへのルートの作成

Grafana サービスを公開するルートを紹介して、Grafana ユーザーインターフェイスにアクセスできません。

前提条件

- [Prometheus リソースのデプロイ](#)
- [Grafana のサービスアカウントの作成](#)
- [Prometheus データソースでの Grafana のデプロイ](#)

手順

- **grafana** サービスへのルートを作成します。

```
oc create route edge <my-grafana-route> --service=grafana --namespace=KAFKA-NAMESPACE
```

8.5.7. Grafana ダッシュボードサンプルのインポート

Grafana を使用して、カスタマイズ可能なダッシュボードで Prometheus メトリックを視覚化します。

AMQ Streams は、[Grafana のダッシュボード設定ファイルのサンプル](#) を JSON 形式で提供します。

- **examples/metrics/grafana-dashboards**

この手順では、Grafana ダッシュボードのサンプルを使用します。

ダッシュボードのサンプルは、キーメトリックを監視するを開始点として適していますが、Kafka でサポートされるすべてのメトリックは表示されません。使用するインフラストラクチャーに応じて、ダッシュボードのサンプルの編集や、他のメトリクスの追加を行うことができます。

前提条件

- [Prometheus リソースのデプロイ](#)
- [Grafana のサービスアカウントの作成](#)
- [Prometheus データソースでの Grafana のデプロイ](#)
- [Grafana サービスへのルートの作成](#)

手順

1. Grafana サービスへのルートの詳細を取得します。以下に例を示します。

```
oc get routes

NAME                HOST/PORT                                PATH SERVICES
MY-GRAFANA-ROUTE    MY-GRAFANA-ROUTE-amq-streams.net        grafana
```

2. Web ブラウザーで、Route ホストおよびポートの URL を使用して Grafana ログイン画面にアクセスします。

3. ユーザー名とパスワードを入力し、続いて **Log In** をクリックします。
デフォルトの Grafana ユーザー名およびパスワードは、どちらも **admin** です。初回ログイン後に、パスワードを変更できます。
4. **Configuration > Data Sources** で、**Prometheus** データソースが作成済みであることを確認します。データソースは「[Prometheus データソースを使用した Grafana のデプロイ](#)」に作成されています。
5. + アイコンをクリックしてから、**Import** をクリックします。
6. **examples/metrics/grafana-dashboards** で、インポートするダッシュボードの JSON をコピーします。
7. JSON をテキストボックスに貼り付け、**Load** をクリックします。
8. 他の Grafana ダッシュボードのサンプル用に、ステップ 5-7 を繰り返します。

インポートされた Grafana ダッシュボードは、**Dashboards** ホームページから表示できます。

第9章 AMQ STREAMS のアップグレード

AMQ Streams をバージョン 2.2 にアップグレードすると、新機能および改良された機能、パフォーマンスの向上、およびセキュリティーオプションを利用できます。

このアップグレード中に、Kafka をサポートされる最新バージョンにアップグレードします。各 Kafka リリースによって、AMQ Streams デプロイメントに新機能、改善点、およびバグ修正が導入されません。

新しいバージョンで問題が発生した場合は、AMQ Streams を以前のバージョンに [ダウングレード](#) できます。

リリースされた AMQ Streams バージョンは、[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。

ダウンタイムと可用性

高可用性に対してトピックが設定されている場合、AMQ Streams をアップグレードしても、これらのトピックからデータをパブリッシュおよび読み取るコンシューマーとプロデューサーのダウンタイムは発生しません。高可用性トピックのレプリケーション係数は 3 以上であり、パーティションはブローカー間で均等に分散されます。

AMQ Streams をアップグレードするとローリング更新がトリガーされ、プロセスのさまざまな段階ですべてのブローカーが順に再起動されます。ローリング更新中は、すべてのブローカーがオンラインであるとは限らないため、[クラスター全体の可用性](#)が一時的に低下します。クラスターの可用性が低下すると、ブローカーで障害が発生した場合にメッセージが失われる可能性が高くなります。

9.1. AMQ STREAMS のアップグレードパス

利用できるアップグレードパスには、2 種類あります。

増分アップグレード

AMQ Streams を以前のマイナーバージョンからバージョン 2.2 にアップグレードします。

マルチバージョンのアップグレード

AMQ Streams を 1 回で古いバージョンからバージョン 2.2 にアップグレードします (1 つ以上の中間バージョンを飛ばします)。

たとえば、AMQ Streams 1.8 から直接 AMQ Streams 2.2 にアップグレードします。

9.1.1. サポート対象の Kafka バージョン

AMQ Streams のアップグレードプロセスを開始する前に、アップグレードする Kafka バージョンを決定します。サポートされている Kafka のバージョンは [Red Hat AMQ Streams Supported Configurations](#) で確認できます。

- Kafka 3.2.3 は実稼働環境での使用でサポートされます。
- Kafka 3.1.0 は、AMQ Streams 2.2 にアップグレードする目的でのみサポートされます。

ご使用中の AMQ Streams バージョンでサポートされている Kafka バージョンのみを使用できます。AMQ Streams のバージョンでサポートされている限り、Kafka の上位バージョンにアップグレードできます。場合によっては、サポートされている以前の Kafka バージョンにダウングレードすることもできます。

9.1.2. 1.7 より前の AMQ Streams バージョンからのアップグレード

AMQ Streams を 1.7 以前から 2.2 にアップグレードする場合、カスタムリソースが API バージョン **v1beta2** を使用していることを確認する必要があります。AMQ Streams 1.8 以降にアップグレードする前に、カスタムリソース定義およびカスタムリソースをアップグレードする必要があります。アップグレードを実行するには、AMQ Streams 1.7 で提供される **API 変換ツール** を使用できます。詳細は、[AMQ Streams 1.7 アップグレードドキュメント](#) を参照してください。

すべてのカスタムリソースの **v1beta2** API バージョンが AMQ Streams 1.7 で導入されました。AMQ Streams 1.8 以降では、**KafkaTopic** および **KafkaUser** を除くすべての AMQ Streams カスタムリソースから **v1alpha1** および **v1beta1** API バージョンが削除されました。

バージョン 1.7 より前の AMQ Streams バージョンからアップグレードする場合は、以下を行います。

1. AMQ Streams を 1.7 にアップグレードする
2. カスタムリソースの **v1beta2** への変換
3. AMQ Streams を 1.8 以降にアップグレードする



注記

別の方法として、バージョン 1.7 からカスタムリソースをインストールし、リソースを変換してから 1.8 以降にアップグレードすることもできます。

9.2. 必要なアップグレードシーケンス

ダウンタイムなしでブローカーとクライアントをアップグレードするには、以下の順序でアップグレード手順を **必ず** 完了してください。

1. OpenShift クラスターのバージョンがサポートされていることを確認してください。AMQ Streams 2.2 は、OpenShift 4.8 から 4.11 でサポートされます。

[最小限のダウンタイムで OpenShift をアップグレード](#) できます。

2. AMQ Streams を 1.7 以前からアップグレードする場合は、[既存のカスタムリソースを更新して、v1beta2 API バージョンをサポート](#) します。
3. 新しい AMQ Streams バージョンに [Cluster Operator を更新](#) します。
4. サポートされる最新の Kafka バージョンに、[すべての Kafka ブローカーとクライアントアプリケーションをアップグレード](#) します。
5. オプション: パーティションの再分散に [Incremental Cooperative Rebalance プロトコルを使用](#) するために、コンシューマーと Kafka Streams アプリケーションをアップグレードします。

9.3. 最小限のダウンタイムでの OPENSIFT のアップグレード

OpenShift をアップグレードする場合は、OpenShift アップグレードのドキュメントを参照して、アップグレードパスとノードを正しくアップグレードする手順を確認してください。OpenShift をアップグレードする前に、[お使いの AMQ Streams バージョンでサポートされるバージョン](#) を確認してください。

アップグレードを実行する際に、Kafka クラスターを利用できるようにしておくことを推奨します。

以下のストラテジーのいずれかを使用できます。

1. Pod の Disruption Budget を設定します。
2. 以下の方法の1つで Pod をローリングします。
 - a. AMQ Streams Drain Cleaner の使用
 - b. Pod へのアノテーションの手動適用

Pod のローリング手法のいずれかを使用する前に、Pod の Disruption Budget を設定する必要があります。

Kafka を稼働し続けるには、高可用性のためにトピックも複製する必要があります。これには、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が必要です。

高可用性のためにレプリケートされた Kafka トピック

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    # ...
    min.insync.replicas: 2
    # ...
```

高可用性環境では、Cluster Operator はアップグレードプロセス時にトピックの In-Sync レプリカの最小数を維持し、ダウンタイムが発生しないようにします。

9.3.1. AMQ Streams Drain Cleaner を使用した Pod のローリング

AMQ Streams Drain Cleaner ツールを使用して、アップグレード時にノードをエビクトできます。AMQ Streams Drain Cleaner は、Pod のローリング更新アノテーションを Pod に付けます。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。

Pod の Disruption Budget を使用すると、特定の時点で、指定された数の Pod だけが、利用できなくなります。Kafka ブローカー Pod の計画メンテナンス時に、Pod の Disruption Budget を使用して、Kafka が高可用性環境で引き続き実行されるようにします。

Kafka コンポーネントの **template** のカスタマイズを使用して、Pod の Disruption Budget を指定します。デフォルトでは、Pod の Disruption Budget は、単一の Pod のみを指定時に利用できないようにします。

これを実行するには、**maxUnavailable** を **0** (ゼロ) に設定します。Pod の Disruption Budget の最大値をゼロに減らすと、自発的に中断されないため、Pod を手動でエビクトする必要があります。

Pod の Disruption Budget の指定

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

```

name: my-cluster
namespace: myproject
spec:
  kafka:
    # ...
  template:
    podDisruptionBudget:
      maxUnavailable: 0
  # ...

```

9.3.2. トピックを利用可能な状態に維持しながらの手動での Pod のローリング

アップグレード時に、Cluster Operator 経由で Pod の手動ローリング更新をトリガーできます。**Pod** リソースを使用して、ローリング更新は新規 Pod でリソースの Pod を再起動します。AMQ Streams Drain Cleaner を使用する場合と同様に、Pod の Disruption Budget の **maxUnavailable** の値をゼロに設定する必要があります。

ドレイン (解放) する必要がある Pod を監視する必要があります。次に Pod アノテーションを追加して更新を行います。

ここで、アノテーションは Kafka ブローカーを更新します。

Kafka ブローカー Pod での手動ローリング更新の実行

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

<cluster_name> は、クラスターの名前に置き換えます。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**

関連情報

- [OpenShift ドキュメント](#)
- [AMQ Streams Drain Cleaner を使用した Pod のドレイン](#)
- [高可用性のためのトピックの複製](#)
- [PodDisruptionBudgetTemplate スキーマ参照](#)
- [Pod アノテーションを使用したローリング更新の実行](#)

9.4. CLUSTER OPERATOR のアップグレード

デプロイの最初の方法と同じ方法を使用して、Cluster Operator をアップグレードします。

インストールファイルの使用

インストール用の YAML ファイルを使用して Cluster Operator をデプロイした場合は、[インストールファイルを使用した Cluster Operator のアップグレード](#)の説明に従って、Operator のインストールファイルを変更してアップグレードを実行します。

OperatorHub の使用

OperatorHub から AMQ Streams をデプロイした場合は、Operator Lifecycle Manager (OLM) を使用して AMQ Streams Operator の更新チャンネルを新しい AMQ Streams バージョンに変更します。

チャンネルを更新すると、選択したアップグレード戦略に応じて、次のタイプのアップグレードのいずれかが開始されます。

- 自動アップグレード
- インストール開始前に承認が必要な手動アップグレード



注記

安定したチャンネルに登録すると、チャンネルを変更せずに自動更新を取得できます。ただし、インストール前のアップグレード手順が失われる可能性があるため、自動更新を有効にすることは推奨しません。バージョン固有のチャンネルでのみ自動アップグレードを使用します。

OperatorHub を使用した Operator のアップグレードについての詳細は、[Upgrading installed Operators \(OpenShift documentation\)](#) を参照してください。

9.4.1. Cluster Operator をアップグレードすると Kafka バージョンエラーが返される

Cluster Operator をアップグレードし、サポートされていない Kafka バージョンエラーが発生した場合、Kafka クラスターのデプロイには、新しい Operator バージョンではサポートされていない古い Kafka バージョンがあります。このエラーは、すべてのインストール方法に適用されます。

このエラーが発生した場合は、Kafka をサポートされている Kafka バージョンにアップグレードしてください。Kafka リソースの `spec.kafka.version` をサポートされているバージョンに変更します。

`oc` を使用して、Kafka リソースの `ステータス` でこのようなエラーメッセージを確認できます。

エラーの Kafka ステータスの確認

```
oc get kafka <kafka_cluster_name> -n <namespace> -o jsonpath='{.status.conditions}'
```

<kafka_cluster_name> は、Kafka クラスターの名前に、<namespace> は、Pod が実行されている OpenShift namespace に置き換えます。

9.4.2. OperatorHub を使用した AMQ Streams 1.7 以前からのアップグレード

OperatorHub を使用して AMQ Streams 1.7 以前からアップグレードする場合に必要なアクション

Red Hat Integration - AMQ Streams Operatorは、**v1beta2** カスタムリソースのみをサポートします。OperatorHub で AMQ Streams Operator をバージョン 2.2 にアップグレードする前に、カスタムリソースを **v1beta2** にアップグレードする必要があります。

バージョン 1.7 より前の AMQ Streams バージョンからアップグレードする場合は、以下を行います。

1. AMQ Streams 1.7 にアップグレードします。
2. [AMQ Streams ソフトウェアのダウンロードページ](#) から、AMQ Streams 1.8 で提供される **Red Hat AMQ Streams API Conversion Tool** をダウンロードします。
3. [カスタムリソースおよび CRD を v1beta2 に変換します](#)。
詳細は、[AMQ Streams 1.7 アップグレードドキュメント](#) を参照してください。

4. OperatorHub で、**Red Hat Integration - AMQ Streams Operator**のバージョン 1.7.0 を削除します。
5. 存在する場合は、**Red Hat Integration - AMQ Streams Operator**のバージョン 2.1.0 を削除します。
存在しない場合は、次のステップに進みます。

AMQ Streams Operator の **Approval Strategy** が **Automatic** に設定されている場合、Operator のバージョン 2.1.0 がすでにクラスターに存在する可能性があります。リリース前にカスタムリソースおよび CRD を **v1beta2** API バージョンに **変換しなかった** 場合、Operator が管理するカスタムリソースおよび CRD は古い API バージョンを使用します。その結果、2.1.0 Operator は **Pending** ステータスで停止します。このような場合、**Red Hat Integration - AMQ Streams Operator** のバージョン 2.1.0 およびバージョン 1.7.0 を削除する必要があります。

両方の Operator を削除すると、新しい Operator バージョンがインストールされるまで、調整は一時停止されます。カスタムリソースへの変更が遅延しないように、次の手順を直ちに実行します。

6. OperatorHub で、**Red Hat Integration - AMQ Streams Operator**のバージョン 2.1.0 を即時にインストールします。
インストールされた 2.1.0 Operator はクラスターの監視を開始し、ローリング更新を実行します。このプロセス中に、クラスターのパフォーマンスが一時的に低下する場合があります。

9.4.3. インストールファイルを使用した Cluster Operator のアップグレード

この手順では、AMQ Streams 2.2 を使用するように Cluster Operator デプロイメントをアップグレードする方法を説明します。

インストール YAML ファイルを使用して Cluster Operator をデプロイした場合は、以下の手順に従います。

Cluster Operator によって管理される Kafka クラスターの可用性は、アップグレード操作による影響を受けません。



注記

特定バージョンの AMQ Streams へのアップグレード方法は、そのバージョンをサポートするドキュメントを参照してください。

前提条件

- 既存の Cluster Operator デプロイメントを利用できる。
- [AMQ Streams 2.2 のリリースアーティファクトがダウンロード済みである](#)。

手順

1. 既存の Cluster Operator リソース (`/install/cluster-operator` ディレクトリー内) に追加した設定変更を書留めておきます。すべての変更は、新しいバージョンの Cluster Operator によって上書きされます。
2. カスタムリソースを更新して、AMQ Streams バージョン 2.2 で使用できるサポート対象の設定オプションを反映します。
3. Cluster Operator を更新します。

- a. Cluster Operator を実行している namespace に従い、新しい Cluster Operator ハーソンのインストールファイルを編集します。

Linux の場合は、以下を使用します。

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

MacOS の場合は、以下を使用します。

```
sed -i " 's/namespace: */namespace: <my_cluster_operator_namespace>/'
install/cluster-operator/*RoleBinding*.yaml
```

- b. 既存の Cluster Operator **Deployment** で1つ以上の環境変数を編集した場合、**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** ファイルを編集し、これらの環境変数を使用します。

4. 設定を更新したら、残りのインストールリソースとともにデプロイします。

```
oc replace -f install/cluster-operator
```

ローリング更新が完了するのを待ちます。

5. 新しい Operator バージョンがアップグレード元の Kafka バージョンをサポートしなくなった場合、Cluster Operator はバージョンがサポートされていないことを示すエラーメッセージを返します。そうでない場合は、エラーメッセージは返されません。

- エラーメッセージが返される場合は、新しい Cluster Operator バージョンでサポートされる Kafka バージョンにアップグレードします。
 - a. **Kafka** カスタムリソースを編集します。
 - b. **spec kafka.version** プロパティをサポートされる Kafka バージョンに変更します。
- エラーメッセージが返されない場合は、次のステップに進みます。Kafka のバージョンを後でアップグレードします。

6. Kafka Pod のイメージを取得して、アップグレードが正常に完了したことを確認します。

```
oc get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

イメージタグには、新しい Operator のバージョンが表示されます。以下に例を示します。

```
registry.redhat.io/amq7/amq-streams-kafka-32-rhel8:2.2.2
```

Cluster Operator はバージョン 2.2 にアップグレードされましたが、管理するクラスターで稼働している Kafka のバージョンは変更されていません。

Cluster Operator のアップグレードの次に、[Kafka のアップグレード](#) を実行する必要があります。

9.5. KAFKA のアップグレード

Cluster Operator を 2.2 にアップグレードした後、次にすべての Kafka ブローカーをサポートされる最新バージョンの Kafka にアップグレードします。

Kafka のアップグレードは、Kafka ブローカーのローリング更新によって Cluster Operator によって実行されます。

Cluster Operator は、Kafka クラスターの設定に基づいてローリング更新を開始します。

Kafka.spec.kafka.config に以下が含まれている場合	Cluster Operator によって開始されるもの
inter.broker.protocol.version と log.message.format.version の両方。	単一のローリング更新。更新後、 inter.broker.protocol.version を手動で更新し、続いて log.message.format.version を更新する必要があります。それぞれを変更すると、ローリング更新がさらにトリガーされます。
inter.broker.protocol.version または log.message.format.version のいずれか。	2つのローリング更新
inter.broker.protocol.version または log.message.format.version の設定なし。	2つのローリング更新



重要

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。ブローカーの **log.message.format.version** プロパティおよびトピックの **message.format.version** プロパティは、非推奨となり、Kafka の今後のリリースで削除されます。

Cluster Operator は、Kafka のアップグレードの一環として、ZooKeeper のローリング更新を開始します。

- ZooKeeper バージョンが変更されなくても、単一のローリング更新が発生します。
- 新しいバージョンの Kafka に新しいバージョンの ZooKeeper が必要な場合、追加のローリング更新が発生します。

9.5.1. Kafka バージョン

Kafka のログメッセージ形式バージョンと inter-broker プロトコルバージョンは、それぞれメッセージに追加されるログ形式バージョンとクラスターで使用される Kafka プロトコルのバージョンを指定します。正しいバージョンが使用されるようにするため、アップグレードプロセスでは、既存の Kafka ブローカーの設定変更と、クライアントアプリケーション (コンシューマーおよびプロデューサー) のコード変更が行われます。

以下の表は、Kafka バージョンの違いを示しています。

表9.1 Kafka バージョンの相違点

Kafka バージョン	Inter-broker プロトコルバージョン	ログメッセージ形式バージョン	ZooKeeper バージョン
3.2.3	3.2	3.2	3.6.3

Kafka バージョン	Inter-broker プロトコルバージョン	ログメッセージ形式バージョン	ZooKeeper バージョン
3.1.0	3.1	3.1	3.6.3

Inter-broker プロトコルバージョン

Kafka では、Inter-broker の通信に使用されるネットワークプロトコルは **Inter-broker プロトコル** と呼ばれます。Kafka の各バージョンには、互換性のあるバージョンの Inter-broker プロトコルがあります。上記の表が示すように、プロトコルのマイナーバージョンは、通常 Kafka のマイナーバージョンと一致するように番号が増加されます。

Inter-broker プロトコルのバージョンは、**Kafka** リソースでクラスター全体に設定されます。これを変更するには、**Kafka.spec.kafka.config** の **inter.broker.protocol.version** プロパティを編集します。

ログメッセージ形式バージョン

プロデューサーが Kafka ブローカーにメッセージを送信すると、特定の形式を使用してメッセージがエンコードされます。この形式は Kafka のリリース間で変更になる可能性があるため、メッセージにはエンコードに使用されたメッセージ形式のバージョンが指定されます。

特定のメッセージ形式のバージョンを設定するために使用されるプロパティは以下のとおりです。

- トピック用の **message.format.version** プロパティ
- Kafka ブローカーの **log.message.format.version** プロパティ

Kafka 3.0.0 以降、メッセージ形式のバージョンの値は **inter.broker.protocol.version** と一致すると見なされ、設定する必要はありません。値は、使用される Kafka バージョンを反映します。

Kafka 3.0.0 以降にアップグレードする場合は、**inter.broker.protocol.version** を更新する際にこの設定を削除できます。それ以外の場合は、アップグレード先の Kafka バージョンに基づいてメッセージ形式のバージョンを設定します。

トピックの **message.format.version** のデフォルト値は、Kafka ブローカーに設定される **log.message.format.version** によって定義されます。トピックの **message.format.version** は、トピック設定を編集すると手動で設定できます。

9.5.2. クライアントをアップグレードするストラテジー

クライアントアプリケーション (Kafka Connect コネクタを含む) のアップグレードに適切な方法は、特定の状況によって異なります。

消費するアプリケーションは、そのアプリケーションが理解するメッセージ形式のメッセージを受信する必要があります。その状態であることを、以下のいずれかの方法で確認できます。

- プロデューサーをアップグレードする **前に**、トピックのすべてのコンシューマーをアップグレードする。
- ブローカーでメッセージをダウンコンバートする。

ブローカーのダウンコンバートを使用すると、ブローカーに余分な負荷が加わるので、すべてのトピックで長期にわたりダウンコンバートに頼るのは最適な方法ではありません。ブローカーの実行を最適化するには、ブローカーがメッセージを一切ダウンコンバートしないようにしてください。

ブローカーのダウンコンバートは 2 通りの方法で設定できます。

- トピックレベルの **message.format.version**。単一のトピックが設定されます。
- ブローカーレベルの **log.message.format.version**。トップレベルの **message.format.version** が設定されていないトピックのデフォルトです。

新バージョンの形式でトピックにパブリッシュされるメッセージは、コンシューマーによって認識されます。これは、メッセージがコンシューマーに送信されるだけでなく、ブローカーがプロデューサーからメッセージを受信するときに、ブローカーがダウンコンバートを実行するからです。

クライアントのアップグレードに使用できる一般的なストラテジーを以下に示します。クライアントアプリケーションをアップグレードするストラテジーは他にもあります。



重要

Kafka 3.0.0 以降にアップグレードすると、各ストラテジーで概説されている手順が変わります。Kafka 3.0.0 以降、メッセージ形式のバージョンの値は **inter.broker.protocol.version** と一致すると見なされ、設定する必要はありません。

ブローカーレベルのコンシューマーの最初のストラテジー

1. コンシューマーとして機能するアプリケーションをすべてアップグレードします。
2. ブローカーレベル **log.message.format.version** を新バージョンに変更します。
3. プロデューサーとして機能するアプリケーションをアップグレードします。

このストラテジーは分かりやすく、ブローカーのダウンコンバートの発生をすべて防ぎます。ただし、所属組織内のすべてのコンシューマーを整然とアップグレードできることが前提になります。また、コンシューマーとプロデューサーの両方に該当するアプリケーションには通用しません。さらにリスクとして、アップグレード済みのクライアントに問題がある場合は、新しい形式のメッセージがメッセージログに追加され、以前のコンシューマーバージョンに戻せなくなる場合があります。

トピックレベルのコンシューマーの最初のストラテジー

トピックごとに以下を実行します。

1. コンシューマーとして機能するアプリケーションをすべてアップグレードします。
2. トピックレベルの **message.format.version** を新バージョンに変更します。
3. プロデューサーとして機能するアプリケーションをアップグレードします。

このストラテジーではブローカーのダウンコンバートがすべて回避され、トピックごとにアップグレードできます。この方法は、同じトピックのコンシューマーとプロデューサーの両方に該当するアプリケーションには通用しません。ここでもリスクとして、アップグレード済みのクライアントに問題がある場合は、新しい形式のメッセージがメッセージログに追加される可能性があります。

ダウンコンバージョンを使用したトピックレベルのコンシューマーの最初のストラテジー

トピックごとに以下を実行します。

1. トピックレベルの **message.format.version** を、旧バージョンに変更します (または、デフォルトがブローカーレベルの **log.message.format.version** のトピックを利用します)。

2. コンシューマーおよびプロデューサーとして機能するアプリケーションをすべてアップグレードします。
3. アップグレードしたアプリケーションが正しく機能することを確認します。
4. トピックレベルの **message.format.version** を新バージョンに変更します。

この戦略にはブローカーのダウンコンバートが必要ですが、ダウンコンバートは一度に1つのトピック (またはトピックの小さなグループ) のみに必要になるので、ブローカーへの負荷は最小限に抑えられます。この方法は、同じトピックのコンシューマーとプロデューサーの両方に該当するアプリケーションにも通用します。この方法により、新しいメッセージ形式バージョンを使用する前に、アップグレードされたプロデューサーとコンシューマーが正しく機能することが保証されます。

この方法の主な欠点は、多くのトピックやアプリケーションが含まれるクラスターでの管理が複雑になる場合があることです。



注記

複数の戦略を適用することもできます。たとえば、最初のアプリケーションおよびトピックには、"per-topic consumers first, with down conversion" 戦略を使用することができます。これが問題なく適用されたら、より効率的な別の戦略の使用を検討できます。

9.5.3. Kafka バージョンおよびイメージマッピング

Kafka のアップグレード時に、**STRIMZI_KAFKA_IMAGES** 環境変数と **Kafka.spec.kafka.version** プロパティの設定について考慮してください。

- それぞれの **Kafka** リソースは **Kafka.spec.kafka.version** で設定できます。
- Cluster Operator の **STRIMZI_KAFKA_IMAGES** 環境変数により、Kafka のバージョンと、指定の **Kafka** リソースでそのバージョンが要求されるときに使用されるイメージをマッピングできます。
 - **Kafka.spec.kafka.image** を設定しないと、そのバージョンのデフォルトのイメージが使用されます。
 - **Kafka.spec.kafka.image** を設定すると、デフォルトのイメージがオーバーライドされません。



警告

Cluster Operator は、Kafka ブローカーの想定されるバージョンが実際にイメージに含まれているかどうかを検証できません。所定のイメージが所定の Kafka バージョンに対応することを必ず確認してください。

9.5.4. Kafka ブローカーおよびクライアントアプリケーションのアップグレード

この手順では、AMQ Streams Kafka クラスターを最新のサポートされる Kafka バージョンにアップグレードする方法を説明します。

新しい Kafka バージョンを現在のバージョンと比較すると、新しいバージョンは **ログメッセージ形式の上位バージョン**、**ブローカー間プロトコルの上位バージョン**、またはその両方をサポートする可能性があります。必要に応じて、これらのバージョンをアップグレードする手順を実行します。詳細は、「[Kafka バージョン](#)」を参照してください。

[クライアントをアップグレードするストラテジー](#) を選択する必要もあります。Kafka クライアントは、この手順の 6 でアップグレードされます。

前提条件

Kafka リソースをアップグレードするには、以下を確認します。

- 両バージョンの Kafka をサポートする Cluster Operator が稼働している。
- **Kafka.spec.kafka.config** には、新しい Kafka バージョンでサポートされないオプションが含まれていない。

手順

1. Kafka クラスタ設定を更新します。

```
oc edit kafka my-cluster
```

2. 設定されている場合、**Kafka.spec.kafka.config** に **log.message.format.version** があり、**inter.broker.protocol.version** が現在の Kafka バージョンのデフォルトに設定されていることを確認します。
たとえば、Kafka のバージョン 3.1.0 から 3.2.3 へのアップグレードは以下のようになります。

```
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.1.0
    config:
      log.message.format.version: "3.1"
      inter.broker.protocol.version: "3.1"
  # ...
```

log.message.format.version および **inter.broker.protocol.version** が設定されていない場合、AMQ Streams では、次のステップの Kafka バージョンの更新後、これらのバージョンを現在のデフォルトに自動的に更新します。



注記

log.message.format.version および **inter.broker.protocol.version** の値は、浮動小数点数として解釈されないように文字列である必要があります。

3. **Kafka.spec.kafka.version** を変更して、新しい Kafka バージョンを指定します。現在の Kafka バージョンのデフォルトで **log.message.format.version** および **inter.broker.protocol.version** のままにします。



注記

kafka.version を変更すると、クラスターのすべてのブローカーがアップグレードされ、新しいブローカーバイナリーの使用が開始されます。このプロセスでは、一部のブローカーは古いバイナリーを使用し、他のブローカーはすでに新しいバイナリーにアップグレードされています。**Inter.broker.protocol.version** を変更しないと、ブローカーはアップグレード中でも相互に通信を継続できます。

たとえば、Kafka 3.1.0 から 3.2.3 へのアップグレードは以下のようになります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.2.3 ❶
    config:
      log.message.format.version: "3.1" ❷
      inter.broker.protocol.version: "3.1" ❸
    # ...
```

- ❶ Kafka のバージョンが新しいバージョンに変更されます。
- ❷ メッセージ形式のバージョンは変更されません。
- ❸ ブローカー間のプロトコルバージョンは変更されません。



警告

新しい Kafka バージョンの **inter.broker.protocol.version** が変更された場合は、Kafka をダウングレードできません。ブローカー間プロトコルのバージョンは、**__consumer_offsets** に書き込まれたメッセージなど、ブローカーによって保存される永続メタデータに使用されるスキーマを判断します。ダウングレードされたクラスターはメッセージを理解しません。

4. Kafka クラスターのイメージが **Kafka.spec.kafka.image** の Kafka カスタムリソースで定義されている場合、**image** を更新して、新しい Kafka バージョンでコンテナイメージを示すようにします。
[Kafka バージョンおよびイメージマッピング](#) を参照してください。
5. 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。
Pod の状態の遷移を監視して、ローリング更新の進捗を確認します。

```
oc get pods my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

ローリング更新により、各 Pod が新バージョンの Kafka のブローカーバイナリーを使用するようになります。

6. クライアントのアップグレードに選択したストラテジーに応じて、新バージョンのクライアントバイナリーを使用するようにすべてのクライアントアプリケーションをアップグレードします。
必要に応じて、Kafka Connect および MirrorMaker の **version** プロパティを新バージョンの Kafka として設定します。
 - a. Kafka Connect では、**KafkaConnect.spec.version** を更新します。
 - b. MirrorMaker では、**KafkaMirrorMaker.spec.version** を更新します。
 - c. MirrorMaker 2.0 の場合は、**KafkaMirrorMaker2.spec.version** を更新します。
7. 設定されている場合、新しい **inter.broker.protocol.version** バージョンを使用するように Kafka リソースを更新します。それ以外の場合は、ステップ9に進みます。
たとえば、Kafka 3.2.3 へのアップグレードは以下のようになります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.2.3
    config:
      log.message.format.version: "3.1"
      inter.broker.protocol.version: "3.2"
      # ...
```

8. Cluster Operator によってクラスターが更新されるまで待ちます。
9. 設定されている場合、新しい **log.message.format.version** バージョンを使用するように Kafka リソースを更新します。それ以外の場合は、ステップ10に進みます。
たとえば、Kafka 3.2.3 へのアップグレードは以下のようになります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.2.3
    config:
      log.message.format.version: "3.2"
      inter.broker.protocol.version: "3.2"
      # ...
```



重要

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。

10. Cluster Operator によってクラスターが更新されるまで待ちます。
 - これで、Kafka クラスターおよびクライアントが新バージョンの Kafka を使用ようになります。

- ブローカーは、ブローカー間プロトコルバージョンと新バージョンの Kafka のメッセージ形式バージョンを使用して、メッセージを送信するように設定されます。

Kafka のアップグレードに従い、必要な場合は以下を行うことができます。

- [Incremental Cooperative Rebalance プロトコル](#)を使用するようにコンシューマーをアップグレード

9.6. コンシューマーの COOPERATIVE REBALANCING へのアップグレード

Kafka コンシューマーおよび Kafka Streams アプリケーションをアップグレードすることで、パーティションの再分散にデフォルトの **Eager Rebalance** プロトコルではなく **Incremental Cooperative Rebalance** プロトコルを使用できます。この新しいプロトコルが Kafka 2.4.0 に追加されました。

コンシューマーは、パーティションの割り当てを Cooperative Rebalance で保持し、クラスタの分散が必要な場合にプロセスの最後でのみ割り当てを取り消します。これにより、コンシューマーグループまたは Kafka Streams アプリケーションが使用不可能になる状態が削減されます。



注記

Incremental Cooperative Rebalance プロトコルへのアップグレードは任意です。Eager Rebalance プロトコルは引き続きサポートされます。

前提条件

- Kafka 3.2.3 に [Kafka ブローカーおよびクライアントアプリケーションをアップグレード済み](#)である。

手順

Incremental Cooperative Rebalance プロトコルを使用するように Kafka コンシューマーをアップグレードする方法:

1. Kafka クライアント **.jar** ファイルを新バージョンに置き換えます。
2. コンシューマー設定で、**partition.assignment.strategy** に **cooperative-sticky** を追加します。たとえば、**range** ストラテジーが設定されている場合は、設定を **range, cooperative-sticky** に変更します。
3. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待ちます。
4. コンシューマー設定から前述の **partition.assignment.strategy** を削除して、グループの各コンシューマーを再設定し、**cooperative-sticky** ストラテジーのみを残します。
5. グループ内の各コンシューマーを順次再起動し、再起動後に各コンシューマーがグループに再度参加するまで待ちます。

Incremental Cooperative Rebalance プロトコルを使用するように Kafka Streams アプリケーションをアップグレードする方法:

1. Kafka Streams の **.jar** ファイルを新バージョンに置き換えます。
2. Kafka Streams の設定で、**upgrade.from** 設定パラメーターをアップグレード前の Kafka バージョンに設定します (例: 2.3)。

3. 各ストリームプロセッサ (ノード) を順次再起動します。
4. **upgrade.from** 設定パラメーターを Kafka Streams 設定から削除します。
5. グループ内の各コンシューマーを順次再起動します。

第10章 AMQ STREAMS のダウングレード

アップグレードしたバージョンの AMQ Streams で問題が発生した場合は、インストールを直前のバージョンに戻すことができます。

YAML インストールファイルを使用して AMQ Streams をインストールした場合は、以前のリリースから YAML インストールファイルを使用して、以下のダウングレード手順を実行できます。

1. 「[Cluster Operator の以前のバージョンへのダウングレード](#)」
2. 「[Kafka のダウングレード](#)」

以前のバージョンの AMQ Streams では使用している Kafka バージョンがサポートされない場合、メッセージに追加されるログメッセージ形式のバージョンが一致すれば Kafka をダウングレードすることができます。



警告

別のインストール方法を使用して AMQ Streams をデプロイした場合は、サポートされるアプローチを使用して AMQ Streams をダウングレードします。ここでは、ダウングレードの手順を使用しないでください。たとえば、Operator Lifecycle Manager (OLM) を使用して AMQ Streams をインストールした場合、デプロイチャネルを以前のバージョンの AMQ Streams に変更することでダウングレードできません。

10.1. CLUSTER OPERATOR の以前のバージョンへのダウングレード

AMQ Streams で問題が発生した場合は、インストールを元に戻すことができます。

この手順では、Cluster Operator デプロイメントを以前のバージョンにダウングレードする方法を説明します。

前提条件

- 既存の Cluster Operator デプロイメントを利用できる。
- [以前のバージョンのインストールファイルがダウンロード済み](#) である必要があります。

手順

1. 既存の Cluster Operator リソース (`/install/cluster-operator` ディレクトリー内) に追加した設定変更を書留めておきます。すべての変更は、以前のバージョンの Cluster Operator によって上書きされます。
2. カスタムリソースを元に戻して、ダウングレードする AMQ Streams バージョンで利用可能なサポート対象の設定オプションを反映します。
3. Cluster Operator を更新します。
 - a. Cluster Operator を実行している namespace に従い、以前のバージョンのインストールファイルを編集します。

Linux の場合は、以下を使用します。

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

MacOS の場合は、以下を使用します。

```
sed -i "s/namespace: */namespace: <my_cluster_operator_namespace>/'
install/cluster-operator/*RoleBinding*.yaml
```

- b. 既存の Cluster Operator **Deployment** で1つ以上の環境変数を編集した場合、**install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** ファイルを編集し、これらの環境変数を使用します。
4. 設定を更新したら、残りのインストーリソースとともにデプロイします。

```
oc replace -f install/cluster-operator
```

ローリング更新が完了するのを待ちます。

5. Kafka Pod のイメージを取得して、ダウングレードが正常に完了したことを確認します。

```
oc get pod my-cluster-kafka-0 -o jsonpath='{.spec.containers[0].image}'
```

イメージタグには、新しい AMQ Streams バージョンと Kafka バージョンが順に示されます。
例: **NEW-STRIMZI-VERSION-kafka-CURRENT-KAFKA-VERSION**

Cluster Operator は以前のバージョンにダウングレードされました。

10.2. KAFKA のダウングレード

Kafka バージョンのダウングレードは、Cluster Operator によって実行されます。

10.2.1. ダウングレードでの Kafka バージョンの互換性

Kafka のダウングレードは、互換性のある現在およびターゲットの [Kafka バージョン](#) と、メッセージがログに記録された状態に依存します。

そのバージョンが、クラスターでこれまで使用された **inter.broker.protocol.version** 設定をサポートしない場合、または新しい **log.message.format.version** を使用するメッセージログにメッセージが追加された場合は、下位バージョンの Kafka に戻すことはできません。

inter.broker.protocol.version は、**__consumer_offsets** に書き込まれたメッセージのスキーマなど、ブローカーによって保存される永続メタデータに使用されるスキーマを判断します。クラスターで以前使用された **inter.broker.protocol.version** が認識されない Kafka バージョンにダウングレードすると、ブローカーが認識できないデータが発生します。

ダウングレードする Kafka のバージョンの関係は次のとおりです。

- ダウングレードする Kafka バージョンの **log.message.format.version** が現行バージョンと **同じ** である場合、Cluster Operator は、ブローカーのローリング再起動を1回実行してダウングレードを行います。
- 別の **log.message.format.version** の場合、ダウングレード後の Kafka バージョンが使用する

バージョンに設定された **log.message.format.version** が常に実行中のクラスターに存在する場合に限り、ダウングレードが可能です。通常は、アップグレードの手順が **log.message.format.version** の変更前に中止された場合にのみ該当します。その場合、ダウングレードには以下が必要です。

- 2つのバージョンで Interbroker プロトコルが異なる場合、ブローカーのローリング再起動が2回必要です。
- 両バージョンで同じ場合は、ローリング再起動が1回必要です。

以前のバージョンでサポートされない **log.message.format.version** が新バージョンで使われていた場合 (**log.message.format.version** のデフォルト値が使われていた場合など)、ダウングレードは実行できません。たとえば以下のリソースの場合、**log.message.format.version** が変更されていないので、Kafka バージョン 3.1.0 にダウングレードできます。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.2.3
    config:
      log.message.format.version: "3.1"
      # ...
```

log.message.format.version が "3.2" に設定されているか、値がない (このためパラメーターに 3.2.3 ブローカーのデフォルト値 3.2 が採用される) 場合は、ダウングレードを実施できません。



重要

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。

10.2.2. Kafka ブローカーおよびクライアントアプリケーションのダウングレード

この手順では、AMQ Streams Kafka クラスターを Kafka の下位 (以前の) バージョンにダウングレードする方法 (3.2.3 から 3.1.0 へのダウングレードなど) を説明します。

前提条件

AMQ Streams Kafka クラスターをダウングレードする前に、**Kafka** リソースについて以下を確認してください。

- 重要: [Kafka バージョンの互換性](#)。
- 両バージョンの Kafka をサポートする Cluster Operator が稼働している。
- **Kafka.spec.kafka.config** に、ダウングレードする Kafka バージョンでサポートされていないオプションが含まれていない。
- **Kafka.spec.kafka.config** に、ダウングレード先の Kafka バージョンでサポートされる **log.message.format.version** と **inter.broker.protocol.version** がある。
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。

手順

1. Kafka クラスター設定を更新します。

```
oc edit kafka KAFKA-CONFIGURATION-FILE
```

2. **Kafka.spec.kafka.version** を変更して、以前のバージョンを指定します。
たとえば、Kafka 3.2.3 から 3.1.0 へのダウングレードは以下のようになります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 3.1.0 1
    config:
      log.message.format.version: "3.1" 2
      inter.broker.protocol.version: "3.1" 3
    # ...
```

- 1** Kafka のバージョンが以前のバージョンに変更されます。
- 2** メッセージ形式のバージョンは変更されません。
- 3** ブローカー間のプロトコルバージョンは変更されません。



注記

log.message.format.version および **inter.broker.protocol.version** の値は、浮動小数点数として解釈されないように文字列である必要があります。

3. Kafka バージョンのイメージが Cluster Operator の **STRIMZI_KAFKA_IMAGES** に定義されているイメージとは異なる場合は、**Kafka.spec.kafka.image** を更新します。
「[Kafka バージョンおよびイメージマッピング](#)」を参照
4. 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。
更新をログで確認するか、Pod 状態の遷移を監視して確認します。

```
oc logs -f CLUSTER-OPERATOR-POD-NAME | grep -E "Kafka version downgrade from [0-9.]+ to [0-9.]+, phase ([0-9]+) of \1 completed"
```

```
oc get pod -w
```

Cluster Operator ログで **INFO** レベルのメッセージを確認します。

```
Reconciliation #NUM(watch) Kafka(NAMESPACE/NAME): Kafka version downgrade from FROM-VERSION to TO-VERSION, phase 1 of 1 completed
```

5. すべてのクライアントアプリケーション (コンシューマー) をダウングレードして、以前のバージョンのクライアントバイナリーを使用します。
これで、Kafka クラスターおよびクライアントは以前の Kafka バージョンを使用ようになります。

- トピックメタデータの保存に ZooKeeper を使用する 1.7 よりも前のバージョンの AMQ Streams に戻す場合は、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-32-rhel8:2.2.2 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

関連情報

- [Topic Operator のトピックストア](#)

第11章 AMQ STREAMS のアンインストール

OpenShift Container Platform Web コンソールまたは CLI を使用して、OperatorHub から OpenShift 4.8 から 4.11 への AMQ Streams をアンインストールできます。

AMQ Streams のインストールに使用したのと同じアプローチを使用します。

AMQ Streams をアンインストールする場合は、デプロイメント専用で作成され、AMQ Streams リソースから参照されるリソースを特定する必要があります。

このようなリソースには以下があります。

- シークレット (カスタム CA および証明書、Kafka Connect Secrets、その他の Kafka シークレット)
- ロギング **ConfigMap** (**external** タイプ)

Kafka、**KafkaConnect**、**KafkaMirrorMaker**、**KafkaBridge** のいずれかの設定で参照されるリソースです。



警告

CustomResourceDefinitions を削除すると、対応するカスタムリソース (**Kafka**、**KafkaConnect**、**KafkaMirrorMaker**、または **KafkaBridge**)、およびそれらに依存するリソース (Deployments、StatefulSets、およびその他の依存リソース) のガベージコレクションが実行されます。

11.1. WEB コンソールを使用した OPERATORHUB からの AMQ STREAMS のアンインストール

この手順では、OperatorHub から AMQ Streams をアンインストールし、デプロイメントに関連するリソースを削除する方法を説明します。

コンソールから手順を実行したり、別の CLI コマンドを使用したりできます。

前提条件

- **cluster-admin** または **strimzi-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform Web コンソールにアクセスできる。
- 削除するリソースを特定している。
AMQ Streams をアンインストールしたら、以下の **oc** CLI コマンドを使用してリソースを検索して、削除されていることを確認できます。

AMQ Streams デプロイメントに関連するリソースを検索するコマンド

```
oc get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

<resource_type> は、**secret** または **configmap** などのチェックするリソースのタイプに置き換えます。

手順

1. OpenShift Web コンソールで、**Operators > Installed Operators**に移動します。
2. **Red Hat Integration - AMQ Streams** Operator がインストールされている場合には、オプションアイコン (3つの縦の点) を選択し、**Uninstall Operator** をクリックします。Operator が **Installed Operators** から削除されます。
3. **Home > Projects** に移動し、AMQ Streams と Kafka コンポーネントがインストールされているプロジェクトを選択します。
4. **Inventory** のオプションをクリックして関連リソースを削除します。リソースには以下が含まれます。
 - Deployments
 - StatefulSets
 - Pod
 - Services
 - ConfigMap
 - Secrets

ヒント

検索を使用して、Kafka クラスターの名前で始まる関連リソースを検索します。また、**Workloads** でもリソースを検索できます。

代替の CLI コマンド

CLI コマンドを使用して、OperatorHub から AMQ Streams をアンインストールできます。

1. AMQ Streams サブスクリプションを削除します。

```
oc delete subscription amq-streams -n openshift-operators
```

2. クラスターサービスバージョン (CSV) を削除します。

```
oc delete csv amqstreams.<version> -n openshift-operators
```

3. 関連する CRD を削除します。

```
oc get crd -l app=strimzi -o name | xargs oc delete
```

11.2. CLI を使用した AMQ STREAMS のアンインストール

この手順では、**oc** コマンドラインツールを使用して AMQ Streams をアンインストールし、デプロイメントに関連するリソースを削除する方法を説明します。

前提条件

- **cluster-admin** または **strimzi-admin** 権限を持つアカウントを使用して、OpenShift クラスターにアクセスできる。
- 削除するリソースを特定している。
AMQ Streams をアンインストールしたら、以下の **oc** CLI コマンドを使用してリソースを検索して、削除されていることを確認できます。

AMQ Streams デプロイメントに関連するリソースを検索するコマンド

```
oc get <resource_type> --all-namespaces | grep <kafka_cluster_name>
```

<resource_type> は、**secret** または **configmap** などのチェックするリソースのタイプに置き換えます。

手順

1. Cluster Operator **Deployment**、関連する **CustomResourceDefinitions**、および **RBAC** リソースを削除します。
Cluster Operator のデプロイに使用するインストールファイルを指定します。

```
oc delete -f install/cluster-operator
```

2. 前提条件で特定したリソースを削除します。

```
oc delete <resource_type> <resource_name> -n <namespace>
```

<resource_type> は削除するリソースのタイプに、<resource_name> はリソースの名前に置き換えます。

シークレットの削除例

```
oc delete secret my-cluster-clients-ca -n my-project
```

第12章 AMQ STREAMS でのメータリングの使用

OCP 4 で利用可能なメータリングツールを使用して、異なるデータソースからメータリングレポートを生成できます。クラスター管理者として、メータリングを使用してクラスターの内容を分析できます。独自のクエリーを作成するか、事前定義 SQL クエリーを使用して、利用可能な異なるデータソースからデータを処理する方法を定義できます。Prometheus をデフォルトのデータソースとして使用すると、Pod、namespace、およびその他ほとんどの OpenShift リソースのレポートを生成できます。

OpenShift のメータリング Operator を使用すると、インストールされた AMQ Streams コンポーネントを分析し、Red Hat サブスクリプションに準拠しているかどうかを判断できます。

AMQ Streams でメータリングを使用するには、まず OpenShift Container Platform に [メータリング Operator](#) をインストールし、設定する必要があります。

12.1. メータリングリソース

メータリングには、メータリングのデプロイメントやインストール、およびメータリングが提供するレポート機能を管理するために使用できるリソースが多数含まれています。メータリングは以下の CRD を使用して管理されます。

表12.1 メータリングリソース

名前	説明
MeteringConfig	デプロイメントのメータリングスタックを設定します。メータリングスタック設定用の各コンポーネントを制御するカスタマイズおよび設定オプションが含まれます。
Report	使用するクエリー、クエリーを実行するタイミングおよび頻度、および結果を保存する場所を制御します。
ReportQuery	ReportDataSources 内に含まれるデータに対して分析を実行するために使用される SQL クエリーが含まれます。
ReportDataSource	ReportQuery および Report で利用可能なデータを制御します。メータリング内で使用できるように複数の異なるデータベースへのアクセスの設定を可能にします。

12.2. AMQ STREAMS のメータリングラベル

以下の表では、AMQ Streams インフラストラクチャーコンポーネントおよびインテグレーションのメータリングラベルがリスト表示されています。

表12.2 メータリングラベル

ラベル	使用できる値
com.company	Red_Hat
rht.prod_name	Red_Hat_Integration

ラベル	使用できる値
rht.prod_ver	2022.Q3
rht.comp	AMQ_Streams
rht.comp_ver	2.2
rht.subcomp	<p>インフラストラクチャー</p> <p>cluster-operator</p> <p>entity-operator</p> <p>zookeeper</p> <hr/> <p>アプリケーション</p> <p>kafka-broker</p> <p>kafka-connect</p> <p>kafka-connect-build</p> <p>kafka-mirror-maker2</p> <p>kafka-mirror-maker</p> <p>cruise-control</p> <p>kafka-bridge</p> <p>kafka-exporter</p> <p>drain-cleaner</p>
rht.subcomp_t	<p>infrastructure</p> <p>application</p>

例

- インフラストラクチャーの例 (インフラストラクチャーコンポーネントが **entity-operator** の場合)

```

com.company=Red_Hat
rht.prod_name=Red_Hat_Integration
rht.prod_ver=2022.Q3
rht.comp=AMQ_Streams
rht.comp_ver=2.2
rht.subcomp=entity-operator
rht.comp_t=infrastructure

```

- アプリケーションの例 (インテグレーションのデプロイメント名が **kafka-bridge** の場合)

```
com.company=Red_Hat  
rht.prod_name=Red_Hat_Integration  
rht.prod_ver=2022.Q3  
rht.comp=AMQ_Streams  
rht.comp_ver=2.2  
rht.subcomp=kafka-bridge  
rht.comp_t=application
```

付録A サブスクリプションの使用

AMQ Streams は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **My Subscriptions** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **INTEGRATION AND AUTOMATION** カテゴリで、**AMQ Streams for Apache Kafka** エントリーを見つけます。
3. 必要な AMQ Streams 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

DNF を使用したパッケージのインストール

パッケージとすべてのパッケージ依存関係をインストールするには、以下を使用します。

```
dnf install <package_name>
```

ローカルディレクトリーからダウンロード済みのパッケージをインストールするには、以下を使用します。

```
dnf install <path_to_download_package>
```

改訂日時: 2023-10-24