Red Hat AMQ 2021.Q3

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

OpenShift Container Platform 上で AMQ Streams 1.8 を使用

Last Updated: 2023-02-23

Red Hat AMQ 2021.Q3 OpenShift での AMQ Streams のデプロイおよびアップグレード

OpenShift Container Platform 上で AMQ Streams 1.8 を使用

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

法律上の通知

概要

本ガイドでは、AMQ Streams のデプロイおよびアップグレード手順を説明します。

目次

多様性を受け入れるオープンソースの強化

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

第1章 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。

本ガイドでは、AMQ Streams をデプロイおよびアップグレードするすべての方法の手順を取り上げ、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。追加のデプロイメントオプションには、メトリクスの導入手順が含まれます。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デプロイメント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設計されています。

1.1. AMQ STREAMS による KAFKA のサポート

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

Operator は以下のプロセスを単純化します。

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

1.2. AMQ STREAMS の OPERATOR

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

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

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

Cluster Operator

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

Entitiy Operator

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

Topic Operator

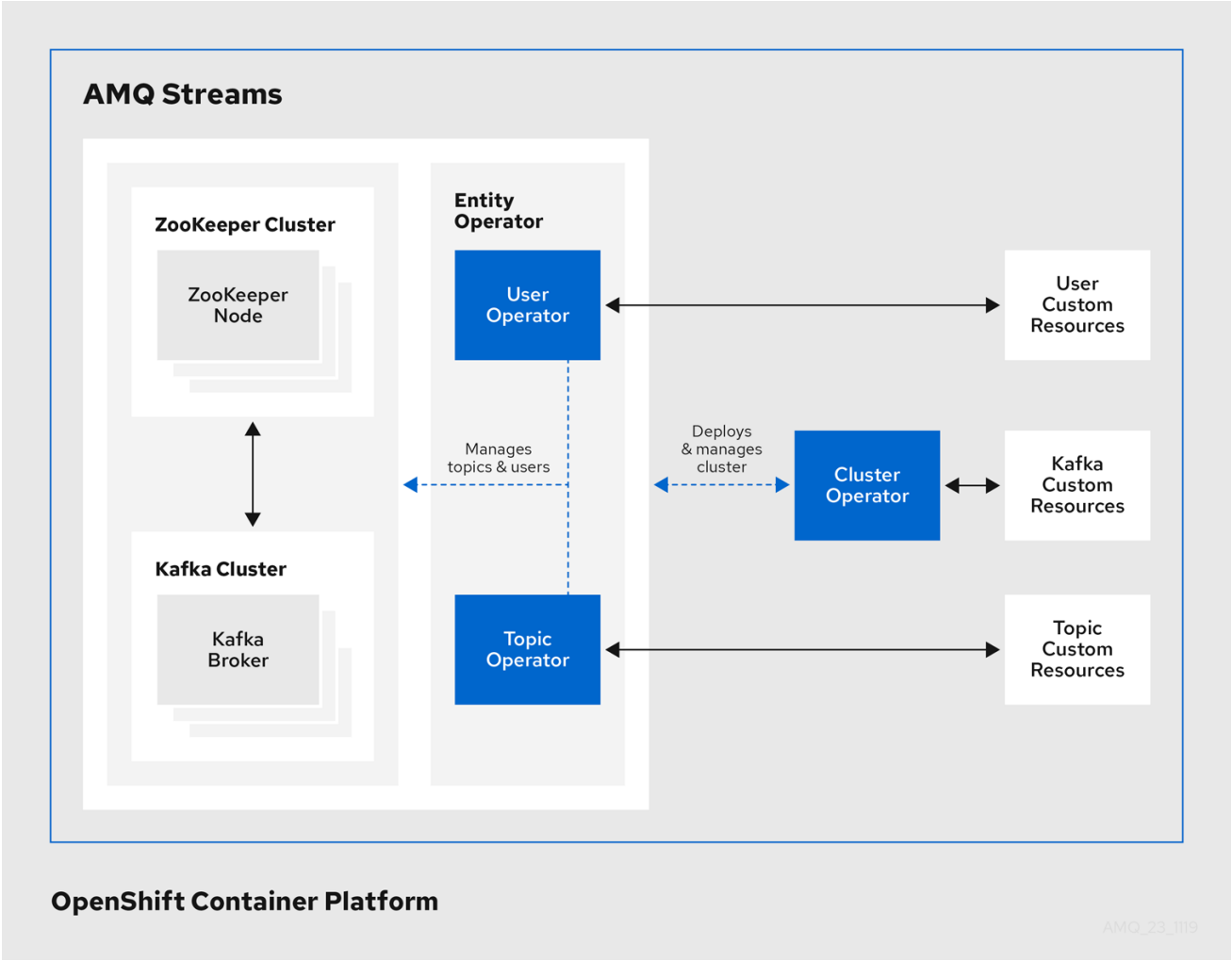
Kafka トピックを管理します。

User Operator

Kafka ユーザーを管理します。

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

AMQ Streams アーキテクチャー内の Operator



1.2.1. Cluster Operator

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

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

- Kafka Bridge

クラスターのデプロイメントにはカスタムリソースが使用されます。

たとえば、以下のように Kafka クラスターをデプロイします。

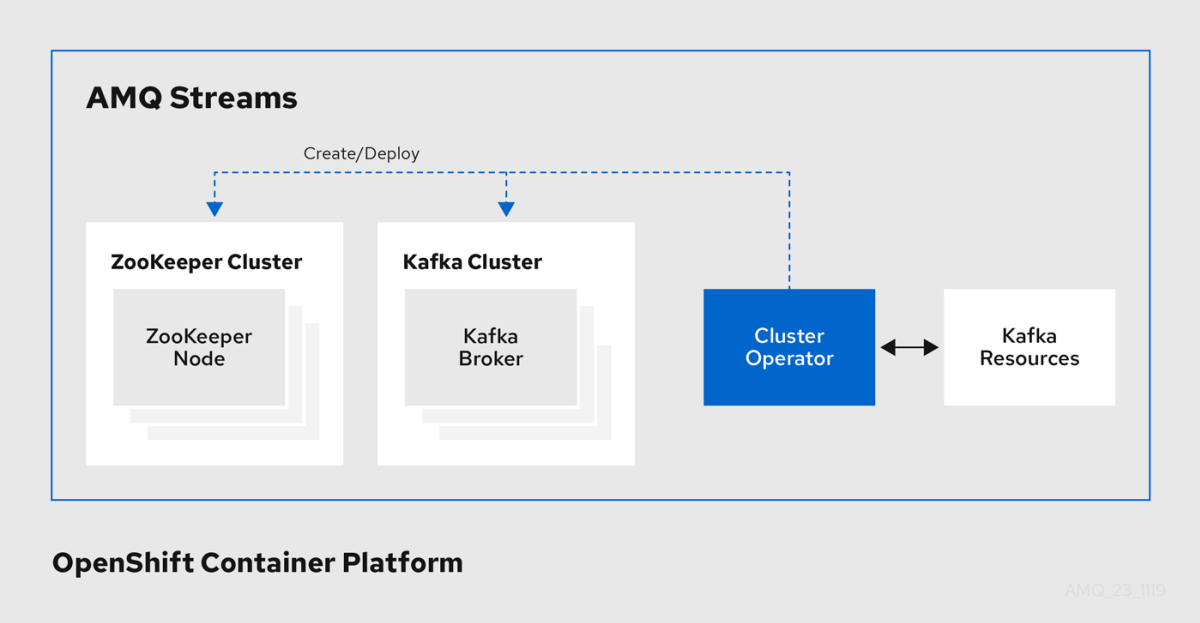
- クラスター設定のある **Kafka** リソースが OpenShift クラスター内で作成されます。
- **Kafka** リソースに宣言された内容を基にして、該当する Kafka クラスターが Cluster Operator によってデプロイされます。

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

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

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

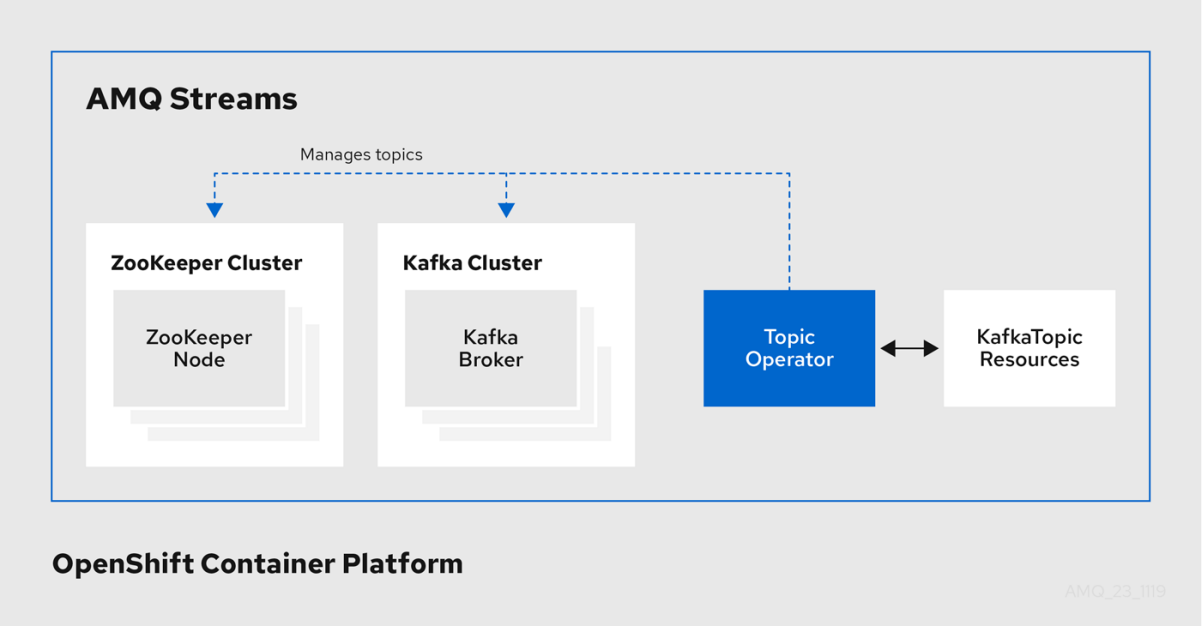
Cluster Operator のアーキテクチャー例



1.2.2. Topic Operator

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

Topic Operator のアーキテクチャー例



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

KafkaTopic とトピックの関係は次のとおりです。

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

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

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

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

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

1.2.3. User Operator

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

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

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

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

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

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

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

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

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

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

詳細は「Feature gates」を参照してください。

1.3. 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。

その他のリソース

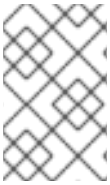
- 「Extend the Kubernetes API with CustomResourceDefinitions」

1.3.1. AMQ Streams カスタムリソースの例

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

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

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



注記

カスタムリソースの管理は、AMQ Streams 管理者のみが行えます。詳細は、『**Deploying and Upgrading AMQ Streams on OpenShift**』の「Designating AMQ Streams administrators」を参照してください。

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

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

警告

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

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

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

Kafka トピックの CRD

```
apiVersion: kafka.strimzi.io/v1beta2
kind: CustomResourceDefinition
metadata: 1
  name: kafkatopics.kafka.strimzi.io
  labels:
    app: strimzi
spec: 2
  group: kafka.strimzi.io
  versions:
    v1beta2
  scope: Namespaced
  names:
    # ...
    singular: kafkatopic
    plural: kafkatopics
    shortNames:
    - kt 3
  additionalPrinterColumns: 4
      # ...
```

```
  subresources:
    status: {} 5
  validation: 6
    openAPIV3Schema:
      properties:
        spec:
          type: object
          properties:
            partitions:
              type: integer
              minimum: 1
            replicas:
              type: integer
              minimum: 1
              maximum: 32767
    # ...
```

1 CRD を識別するためのトピック CRD、その名前および名前のメタデータ。

2 この CRD に指定された項目には、トピックの API にアクセスするため URL に使用されるグループ (ドメイン) 名、複数名、およびサポートされるスキーマバージョンが含まれます。他の名前は、CLI のインスタンスリソースを識別するために使用されます。たとえば、**oc get kafkaShortNametopic my-topic** や **oc get kafkatopics** などです。

3 ShortName は CLI コマンドで使用できます。たとえば、**oc get kafkatopic** の代わりに **oc get kt** を略名として使用できます。

4 カスタムリソースで **get** コマンドを使用する場合に示される情報。

5 リソースの スキーマ参照 に記載されている CRD の現在の状態。

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 が特定されます。
2. トピックまたはユーザーが属する Kafka クラスターの名前 (**Kafka** リソースの名前と同じ) を定義する、**KafkaTopic** および **KafkaUser** リソースのみに適用可能なラベル。
3. 指定内容には、トピックのパーティション数およびレプリカ数や、トピック自体の設定パラメーターが示されています。この例では、メッセージがトピックに保持される期間や、ログのセグメントファイルサイズが指定されています。
4. **KafkaTopic** リソースのステータス条件。**lastTransitionTime** で **type** 条件が **Ready** に変更されています。

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

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

1.4. AMQ STREAMS のインストール方法

AMQ Streams を OpenShift 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	サポートされるプラットフォーム
インストールアーティファクト (YAML ファイル)	AMQ Streams のダウンロードサイト から **amq-streams-x.y.z-ocp-install-examples.zip** ファイルをダウンロードします。次に、**oc**を使用して YAML インストールアーティファクトを OpenShift クラスターにデプロイします。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。	OpenShift 4.6 および 4.8
OperatorHub	OperatorHub で **Red Hat Integration - AMQ Streams Operator** 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。	OpenShift 4.6 および 4.8

柔軟性が重要な場合は、インストールアーティファクトによる方法を選択します。Web コンソールを使用して標準設定で AMQ Streams を OpenShift 4.6 および 4.8 にインストールする場合は、OperatorHub による方法を選択します。OperatorHub を使用すると、自動更新も利用できます。

どちらの方法でも、Cluster Operator が OpenShift クラスターにインストールされます。**同じ** 方法を使用して、Kafka クラスターから順に他のコンポーネントをデプロイします。インストールアーティファクトによる方法を使用している場合は、YAML ファイルのサンプルが提供されます。OperatorHub を使用している場合は、AMQ Streams Operator によって Kafka コンポーネントを OpenShift Web コンソールからインストールできるようになります。

AMQ Streams インストールアーティファクト

AMQ Streams インストールアーティファクトには、OpenShift にデプロイできるさまざまな YAML ファイルが含まれ、**oc**を使用して以下を含むカスタムリソースが作成されます。

- デプロイメント
- Custom Resource Definition (CRD)
- ロールおよびロールバインディング
- サービスアカウント

YAML インストールファイルは、Cluster Operator、Topic Operator、User Operator、および Strimzi Admin ロールに提供されます。

OperatorHub

OpenShift 4 以上では、**Operator Lifecycle Manager** (OLM) を使用することにより、クラスター管理者はクラスター全体で実行されるすべての Operator やそれらの関連サービスをインストール、更新、および管理できます。OLM は、Kubernetes のネイティブアプリケーション (Operator) を効率的に自動化された拡張可能な方法で管理するために設計されたオープンソースツールキットの **Operator Framework** の一部です。

OperatorHub は OpenShift Web コンソールの一部です。クラスター管理者はこれを使用して Operator を検出、インストール、およびアップグレードできます。Operator は OperatorHub からプルでき、単一の namespace またはすべての namespace への OpenShift クラスターにインストールできます。Operator は OLM で管理できます。エンジニアリングチームは OLM を使用して、開発、テスト、および本番環境でソフトウェアを独立管理できます。

Red Hat Integration - AMQ Streams Operator

Red Hat Integration - AMQ Streams Operatorは OperatorHub からインストールできます。AMQ Streams Operator のインストール後、必要な CRD およびロールベースアクセス制御 (RBAC) リソースと共に Cluster Operator が OpenShift クラスターにデプロイされます。Kafka コンポーネントは OpenShift Web コンソールからインストールする必要があります。

その他のリソース

インストールアーティファクトを使用した AMQ Streams のインストール:

- 「単一の namespace を監視対象とする Cluster Operator のデプロイメント」
- 「複数の namespace を監視対象とする Cluster Operator のデプロイメント」
- 「すべての namespace を対象とする Cluster Operator のデプロイメント」

OperatorHub からの AMQ Streams のインストール:

- 「OperatorHub からの AMQ Streams Operator のデプロイ」

- OpenShift ドキュメント『Operator』

第2章 AMQ STREAMS でデプロイされるもの

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 にデプロイするために提供されます。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 メトリクスデータを抽出する **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。

2.1. デプロイメントの順序

OpenShift クラスターへのデプロイメントで必要な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

2.2. その他のデプロイメント設定オプション

本書のデプロイメント手順では、AMQ Streams で提供されるインストール YAML ファイルのサンプルを使用するデプロイメントを説明します。手順では、検討する必要がある重要な設定事項について説明しますが、使用できる設定オプションをすべて取り上げるわけではありません。

カスタムリソースを使用するとデプロイメントを改良できます。

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できます。カスタムリソースによる設定の詳細は、『**Using AMQ Streams on OpenShift**』の「Deployment configuration」を参照してください。

2.2.1. Kafka のセキュリティー

デプロイメントでは、Cluster Operator はクラスター内でのデータの暗号化および認証に対して自動で TLS 証明書を設定します。

AMQ Streams では、『**AMQ Streams on OpenShift の使用**』で説明する **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を管理して、Kafka クラスターとクライアント間のデータ交換をセキュアにします。
- 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。

2.2.2. デプロイメントの監視

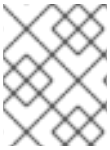
AMQ Streams は、デプロイメントを監視する追加のデプロイメントオプションをサポートします。

- Prometheus および Grafana を Kafka クラスターでデプロイ し、メトリクスを抽出して、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。
- 『**Using AMQ Streams on OpenShift**』で説明するように、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います。

第3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
- Kafka Connect S2I （Source-to-Image) ビルドの Red Hat レジストリーでの認証方法 (必要な場合)
- AMQ Streams コンテナー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。

3.1. デプロイメントの前提条件

AMQ Streams のデプロイする場合、以下を確認してください。

- OpenShift 4.6 および 4.8 クラスターが利用可能である。
  AMQ Streams は AMQ Streams Strimzi 0.24.x をベースとしています。
- **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こと。

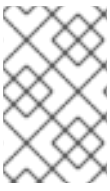
3.2. AMQ STREAMS リリースアーティファクトのダウンロード

AMQ Streams をインストールするには、AMQ Streams のダウンロードページ から **amq-streams-<version>-ocp-install-examples.zip** ファイルをダウンロードし、リリースアーティファクトを展開します。

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。



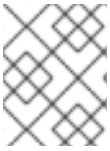
注記

AMQ Streams コンテナーイメージは、Red Hat Ecosystem Catalog から使用することもできます。しかし、提供される YAML ファイルを使用して AMQ Streams をデプロイすることが推奨されます。

3.3. KAFKA CONNECT S2I のコンテナーレジストリーでの認証

OpenShift ビルドおよび S2I(Source-to-Image)を使用してコンテナーイメージを作成する前に、Red Hat コンテナーレジストリー(**registry.redhat.io**) で認証を設定する必要があります。

コンテナーレジストリーは、AMQ Streams コンテナーイメージを Red Hat Ecosystem Catalog に保存するために使用されます。カタログには、S2I がサポートされる Kafka Connect ビルダーイメージが含まれます。OpenShift ビルドは、ソースコードおよびバイナリーと共にこのビルダーイメージをプルし、これを使用して新しいコンテナーイメージをビルドします。



注記

Red Hat コンテナーレジストリーによる認証は、Kafka Connect S2I を使用する場合のみ必要です。他の AMQ Streams コンポーネントには必要ありません。

前提条件

- OpenShift Container Platform クラスターへアクセスできるクラスター管理者権限。
- Red Hat カスタマーポータルのアカウントのログイン詳細。付録A *サブスクリプションの使用* を参照してください。

手順

1. 必要であれば、管理者として OpenShift クラスターにログインします。

```
oc login --user system:admin --token=my-token --server=https://my-cluster.example.com:6443
```

2. Kafka Connect S2I クラスターが含まれるプロジェクトを開きます。

```
oc project CLUSTER-NAME
```

注記

Kafka Connect S2I クラスターがすでにデプロイ されている可能性があります。

3. Red Hat カスタマーポータルアカウントを使用して **docker-registry** シークレットを作成します。**PULL-SECRET-NAME** は、作成するシークレット名に置き換えます。

```
oc create secret docker-registry PULL-SECRET-NAME \
 --docker-server=registry.redhat.io \
 --docker-username=CUSTOMER-PORTAL-USERNAME \
 --docker-password=CUSTOMER-PORTAL-PASSWORD \
 --docker-email=EMAIL-ADDRESS
```

以下の出力が表示されるはずです。

```
secret/PULL-SECRET-NAME created
```

重要

この **docker-registry** シークレットを、registry **.redhat.io** に対して認証されるすべての OpenShift プロジェクトに作成する必要があります。

4. シークレットをサービスアカウントにリンクして、シークレットをイメージをプルするために使用します。サービスアカウント名は、OpenShift Pod が使用する名前と一致する必要があります。

```
oc secrets link SERVICE-ACCOUNT-NAME PULL-SECRET-NAME --for=pull
```

たとえば、**デフォルトのサービスアカウント** と **my-secret** という名前のシークレットを使用します。

```
oc secrets link default my-secret --for=pull
```

5. シークレットを **builder** サービスアカウントにリンクし、ビルドイメージをプッシュおよびプルするためにシークレットを使用します。

```
oc secrets link builder PULL-SECRET-NAME
```

注記

Red Hat のユーザー名とパスワードを使用してプルシークレットを作成したくない場合は、レジストリーサービスアカウントを使用して認証トークンを作成できます。

その他のリソース

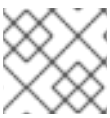
- 「OpenShift ビルドおよび S2I (Source-to-Image) を使用したコンテナーイメージの作成」
- Red Hat コンテナーレジストリーの認証 (Red Hat ナレッジベース)
- Red Hat カスタマーポータルのレジストリーサービスアカウント

3.4. コンテナーイメージを独自のレジストリーにプッシュ

AMQ Streams のコンテナーイメージは 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ー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** コンテナー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。

コンテナーイメージ	namespace／リポジトリー	説明
Kafka	• registry.redhat.io/amq7/amq-streams-kafka-28-rhel7:1.8.0 • registry.redhat.io/amq7/amq-streams-kafka-27-rhel7:1.8.0	次を含む、Kafka を実行するための AMQ Streams イメージ。 • Kafka Broker • Kafka Connect / S2I • Kafka Mirror Maker • ZooKeeper 3.5.7 • TLS Sidecars
Operator	• registry.redhat.io/amq7/amq-streams-rhel7-operator:1.8.0	Operator を実行するための AMQ Streams イメージ。 • Cluster Operator • Topic Operator • User Operator • Kafka Initializer
Kafka Bridge	• registry.redhat.io/amq7/amq-streams-bridge-rhel7:1.8.0	AMQ Streams Kafka Bridge を稼働するための AMQ Streams イメージ

3.5. AMQ STREAMS の管理者の指名

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、これらのリソースを表示、作成、編集、および削除する権限は OpenShift クラスター管理者に制限されます。AMQ Streams には、このような権限を他のユーザーに割り当てするために使用する 2 つのクラスターロールがあります。

- **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** ロールに集約され、**strimzi-admin** は **edit** および **admin** ロールに集約されます。集約により、これらのロールを同様の権限を持つユーザーに割り当てする必要がない可能性があります。

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 の管理者を指名できます。

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) リソースが必要です。

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4章 OPERATORHUB からの AMQ STREAMS のデプロイ

Red Hat Integration - AMQ Streams Operator を使用して、OperatorHub から AMQ Streams をデプロイします。

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 のデプロイ
- AMQ Streams Operator を使用した Kafka コンポーネントのデプロイ

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

Red Hat Integration Operator 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。

その他のリソース

Red Hat Integration Operator のインストールおよび使用に関する詳細は、『Installing the Red Hat Integration Operator on OpenShift』を参照してください。

4.2. OPERATORHUB からの AMQ STREAMS OPERATOR のデプロイ

OperatorHub から AMQ Streams Operator をインストールして、Cluster Operator を OpenShift クラスターにデプロイできます。

警告

適切な更新チャネルを使用するようにしてください。サポートされるバージョンの OpenShift を使用している場合、デフォルトの **stable** チャネルから **安全** に AMQ Streams をインストールできます。ただし、サポートされていない OpenShift のバージョンを使用している場合は、特に自動更新が有効になっている状態で stable チャネルから AMQ Streams をインストールすることは **安全ではありません**。これは、クラスターが OpenShift リリースによってサポートされない新しいコンポーネントを自動更新で受け取るためです。

前提条件

- **Red Hat Operator** の **OperatorSource** が OpenShift クラスターで有効になっている必要があります。適切な **OperatorSource** が有効になっていれば OperatorHub に Red Hat Operator が表示されます。詳細は、『**Operator**』を参照してください。

- インストールには、Operator を OperatorHub からインストールするための権限を持つユーザーが必要です

手順

1. OpenShift Web コンソールで、**Operators** > **OperatorHub** をクリックします。
2. **Streaming & Messaging** カテゴリーの **AMQ Streams** Operator を検索または閲覧します。
3. **Red Hat Integration - AMQ Streams Operator**タイルをクリックし、右側のサイドバーで **Install** をクリックします。
4. **Create Operator Subscription** 画面で、以下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**: AMQ Streams Operator の更新チャネルを選択します。
     - **stable** チャネル (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られます。
     - **amq-streams-X.Y.x** チャネルには、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られます。
   - **Installation Mode**: AMQ Streams Operator をクラスターのすべての namespace にインストール (デフォルト) するか、特定の namespace インストールするかを選択します。namespace を使用して関数を分離することが推奨されます。特定の namespace を Kafka クラスターおよびその他の AMQ Streams コンポーネントの専用とすることが推奨されます。
   - **Approval Strategy**: 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『Operator』を参照してください。
5. **Subscribe** をクリックすると、AMQ Streams Operator が OpenShift クラスターにインストールされます。
   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 またはすべての namespace にデプロイされます。
6. **Installed Operators** 画面で、インストールの進捗を確認します。AMQ Streams Operator は、ステータスが **InstallSucceeded** に変更されると使用できます。

次に、AMQ Streams Operator を使用して、Kafka クラスターから順に Kafka コンポーネントをデプロイできます。

その他のリソース

- 「AMQ Streams Operator を使用した Kafka コンポーネントのデプロイ」
- 「AMQ Streams のインストール方法」

- 「Kafka クラスターのデプロイメント」

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

AMQ Streams Operator を Openshift Container Platform にインストールすると、Kafka コンポーネントをユーザーインターフェースからインストールできます。

インストールできる Kafka コンポーネント:

- Kafka
- Kafka Connect
- Kafka Connect Source to Image (S2I)
- Kafka MirrorMaker
- Kafka MirrorMaker 2
- Kafka Topic
- Kafka User
- Kafka Bridge
- Kafka Connector
- Kafka Rebalance

前提条件

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

手順

1. **Installed Operators** に移動し、**Red Hat Integration - AMQ Streams Operator**をクリックして **Operator details** ページを表示します。
2. **Provided APIs** から、インストールする Kafka コンポーネントの **Create Instance** をクリックします。
   各コンポーネントのデフォルト設定は CRD **spec** プロパティーにカプセル化されます。
3. (任意設定) インストールを実行する前に、**form** または **YAML** ビューからインストールの指定内容を設定します。
4. **Create** をクリックして、選択したコンポーネントのインストールを開始します。
   状態が **Succeeded** に変わるまで待ちます。

その他のリソース

- 「OperatorHub からの AMQ Streams Operator のデプロイ」

第5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

OperatorHub で AMQ Streams Operator を使用して AMQ Streams をデプロイする代わりに、インストールアーティファクトを使用できます。AMQ Streams のデプロイメント環境の準備 が整ったら、以下を実行できます。

- Kafka クラスターの作成方法
- 要件に応じてその他の Kafka コンポーネントをデプロイする任意の手順。
  - Kafka Connect
  - Kafka MirrorMaker
  - Kafka Bridge

これらの手順は、OpenShift クラスターが利用可能で稼働していることを想定しています。

AMQ Streams は AMQ Streams Strimzi 0.24.x をベースとしています。ここでは、OpenShift 4.6 および 4.8 に AMQ Streams をデプロイする方法を説明します。

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。

5.1. 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、これを行うためにデプロイメントファイルのサンプルが提供されます。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 を使用してこれを管理できません。これは、たとえば OpenShift 外部で実行されている Kafka クラスターに適用されます。ただし、Topic Operator および User Operator をスタンドアロンコンポーネントとしてデプロイおよび使用することもできます。

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、Kafka クラスターデプロイメントの単一の namespace で **KafkaTopics** および **KafkaUsers** を監視します。

Kafka クラスターを Topic Operator および User Operator とデプロイ

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

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

5.1.1. 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は以下を説明します。

- 以下を**監視**するよう Cluster Operator をデプロイする方法。
  - 単一の namespace
  - 複数の namespace
  - すべての namespace
- 代替のデプロイメント

5.1.1.1. 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**。
- Source2Image がサポートされる Kafka Connect クラスターの **KafkaConnectS2I**。
- Kafka Connect クラスターでコネクター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 Bridge インスタンスの **KafkaBridge**。

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 によってクラスターの詳細がリソースより取得されます。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構成する OpenShift リソースで該当する更新が Operator によって実行されます。

クラスターの望ましい状態がリソースのクラスターに反映されるようにするため、リソースへのパッチ適用後またはリソースの削除後にリソースが再作成されます。この操作は、サービスの中断を引き起こすローリングアップデート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。

5.1.1.2. 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ように 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 がインストールされる namespace を使用するように、AMQ Streams の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. Cluster Operator が正常にデプロイされたことを確認します。

```
oc get deployments
```

5.1.1.3. 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 全体で AMQ Streams リソースを監視するように Cluster Operator をデプロイする方法を説明します。

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 がインストールされる namespace を使用するように、AMQ Streams の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. **install/cluster-operator/060-Deployment-strimzi-cluster-operator.yaml** ファイルを編集し、Cluster Operator によって監視されるすべての 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-rhel7-operator:1.8.0
           imagePullPolicy: IfNotPresent
           env:
           - name: STRIMZI_NAMESPACE
             value: watched-namespace-1,watched-namespace-2,watched-namespace-3
   ```

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** に対してこれを行います。

```
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
```

4. Cluster Operator をデプロイします。

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

5. Cluster Operator が正常にデプロイされたことを確認します。

```
oc get deployments
```

5.1.1.4. 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 全体で AMQ Streams リソースを監視するように Cluster Operator をデプロイする方法を説明します。

このモードで実行している場合、Cluster Operator によって、新規作成された namespace でクラスターが自動的に管理されます。

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 がインストールされる namespace を使用するように、AMQ Streams の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. **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-rhel7-operator:1.8.0
        imagePullPolicy: IfNotPresent
        env:
        - name: STRIMZI_NAMESPACE
          value: "*"
        # ...
```

3. 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。

4. 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
```

5. Cluster Operator が正常にデプロイされたことを確認します。

```
oc get deployments
```

5.1.2. 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 クラスター
  - 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 クラスター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。

5.1.2.1. Kafka クラスターのデプロイメント

この手順では、Cluster Operator を使用して Kafka クラスターを OpenShift 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って **Kafka** リソースが作成されます。

AMQ Streams では、デプロイメントの YAML ファイルのサンプルは **examples/kafka/** にあります。

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ーをデプロイします。

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、永続クラスターをデプロイします。

kafka-persistent-single.yaml

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

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、その内容は厳密に Pod のライフサイクルと関連し、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、**PersistentVolumes** 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。たとえば、YAML ファイルを変更しなくても Amazon AWS デプロイメントで Amazon EBS ボリュームを使用できます。**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。

サンプル YAML ファイルは、サポートされる最新の Kafka バージョンを指定し、サポートされるログメッセージ形式バージョンの設定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 のアップグレード時に、これらのプロパティーの更新が必要になります。

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** プロパティーを編集します。

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: 2.8.0
    #...
    config:
      #...
      log.message.format.version: 2.8
      inter.broker.protocol.version: 2.8
  # ...
```

Kafka リソースの設定に関する詳細は、『**Using AMQ Streams on OpenShift**』の「Kafka cluster configuration」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. **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. Kafka クラスターが正常にデプロイされたことを確認します。

   ```
   oc get deployments
   ```

5.1.2.2. 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** プロパティーを設定し、**topicOperator** が含まれるようにします。デフォルトでは、Topic Operator は Kafka クラスターデプロイメントの namespace で **KafkaTopics** を監視します。

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** プロパティーの設定に関する詳細は、『**Using AMQ Streams on OpenShift**』の「Configuring the Entity Operator」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** プロパティーを編集し、**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** schema reference」に記載されているプロパティーを使用して、Topic Operator の **spec** を設定します。
すべてのプロパティー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 <your-file>
```

5.1.2.3. 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** プロパティーを設定し、**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 **KafkaUsers** を監視します。

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** プロパティーの設定に関する詳細は、『**Using AMQ Streams on OpenShift**』の「Configuring the Entity Operator」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** プロパティーを編集し、**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. 『Using AMQ Streams on OpenShift』の「EntityUserOperatorSpec schema reference」に記載されているプロパティーを使用して、User Operator の **spec** を設定します。
すべてのプロパティー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。

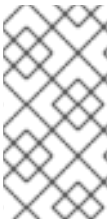
3. リソースを作成または更新します。

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

5.1.3. 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

Cluster Operator を使用してこれらの operator をデプロイする代わりに、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 Topic Operator および User Operator と使用する場合は、スタンドアロンデプロイメントを検討してください。

スタンドアロンデプロイメントでは、Kafka を OpenShift の外部で実行できます。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に一致するようにします。



注記

スタンドアロンデプロイメントでは、Topic Operator および User Operator は AMQ Streams Kafka クラスター (つまり Cluster Operator を使用してデプロイされた Kafka クラスター) に接続**しません**。クラスターとの接続が確立されていても、operator は機能しません。

5.1.3.1. 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 クラスターと操作できます。

スタンドアロンデプロイメントファイルが提供されます。**05-Deployment-strimzi-topic-operator.yaml** デプロイメントファイルを設定し、Topic Operator が Kafka クラスターに接続できるようにする環境変数を追加します。

前提条件

- Topic Operator が接続する Kafka クラスターが稼働している。

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** プロパティーを編集します。

スタンドアロン 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"
```

(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ためのラベルセレクター。

。

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が原因で、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 によって使用される 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。

2. **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 の設定例

```
# ....
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、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ために使用される認証局の公開鍵 (**ca.crt**) の値が含まれています。

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 クライアント認証に対して新しいユーザー証明書を署名するための認証局の秘密鍵 (**ca.key**) 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

3. スタンドアロン Topic Operator をデプロイします。

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

4. スタンドアロン Topic Operator が正常にデプロイされていることを確認します。

```
oc describe deployment strimzi-topic-operator
```

スタンドアロン Topic Operator は、Replicas エントリーに **1 available が表示され** ればデプロイされます。

注記

OpenShift への接続が低速な場合や Topic Operator イメージがこれまでダウンロードされたことがない場合は、デプロイメントに遅延が発生することがあります。

5.1.3.2. スタンドアロン User Operator のデプロイ

この手順では User Operator をユーザー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 User 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 クラスターと操作できます。

スタンドアロンデプロイメントファイルが提供されます。**05-Deployment-strimzi-user-operator.yaml** デプロイメントファイルを編集し、User Operator が Kafka クラスターに接続できるようにする環境変数を追加します。

前提条件

- User Operator が接続する Kafka クラスターが稼働している。

手順

1. 以下の **env** プロパティーを **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_ZOOKEEPER_CONNECT 5
        value: my-cluster-zookeeper-client:2181
      - name: STRIMZI_LABELS 6
        value: "strimzi.io/cluster=my-cluster"
      - name: STRIMZI_FULL_RECONCILIATION_INTERVAL_MS 7
        value: "120000"
      - name: STRIMZI_ZOOKEEPER_CONNECT 8
        value: my-cluster-zookeeper-client:2181
      - name: STRIMZI_ZOOKEEPER_SESSION_TIMEOUT_MS 9
        value: "18000"
      - name: STRIMZI_LOG_LEVEL 10
        value: INFO
      - name: STRIMZI_GC_LOG_ENABLED 11
        value: "true"
      - name: STRIMZI_CA_VALIDITY 12
        value: "365"
      - name: STRIMZI_CA_RENEWAL 13
        value: "30"
      - name: STRIMZI_JAVA_OPTS 14
        value: "-Xmx=512M -Xms=256M"
      - name: STRIMZI_JAVA_SYSTEM_PROPERTIES 15
        value: "-Djavax.net.debug=verbose -DpropertyName=value"
```

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. 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。
6. User Operator によって管理される **KafkaUser** リソースを識別するために使用されるラベルセレクター。
7. 定期的な調整の間隔 (秒単位)。デフォルトは **120000** (2 分) です。
8. 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。

9 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。

10 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。

11 ガベッジコレクション (GC) ロギングを有効にします。デフォルトは **true** です。

12 認証局の有効期限。デフォルトは **365** 日です。

13 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。

14 (任意) User Operator を実行する JVM によって使用される Java オプション。

15 (任意) User Operator に設定されたデバッグ (**-D**) オプション。

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 の設定例

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

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. スタンドアロン User Operator をデプロイします。

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

4. スタンドアロン User Operator が正常にデプロイされていることを確認します。

```
oc describe deployment strimzi-user-operator
```

Replicas エントリーに **1 available** が表示されれば、スタンドアロン User Operator はデプロイされています。

注記

OpenShift への接続が低速な場合や User Operator イメージがこれまでダウンロードされたことがない場合は、デプロイメントに遅延が発生することがあります。

5.2. 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 では、**コネクター** の概念を使用し、スケーラビリティーと信頼性を維持しながら Kafka クラスターで大量のデータを出し入れするためのフレームワークが提供されます。

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。

本セクションの手順では以下の方法を説明します。

- **KafkaConnect** リソースを使用した Kafka Connect のデプロイ
- 複数の Kafka Connect インスタンスの実行
- 接続の確立に必要なコネクターが含まれる Kafka Connect の作成
- KafkaConnector リソースまたは Kafka Connect REST API を使用したコネクターの作成および管理
- KafkaConnector リソースを Kafka Connect にデプロイ
- KafkaConnector リソースにアノテーションを付けて Kafka コネクターを再起動
- KafkaConnector リソースにアノテーションを付けて Kafka コネクタータスクを再起動

注記

コネクター という用語は、Kafka Connect クラスター内で実行されているコネクターインスタンスや、コネクター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 **コネクター** という用語を使用します。

5.2.1. Kafka Connect 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 Connect 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。

Kafka Connect クラスターは **Deployment** として実装されます。その **Deployment** には、コネクターのワークロードを **タスク** として分布するノード (**ワーカー** とも呼ばれる) の設定可能な数が含まれるため、メッセージフローのスケーラビリティーや信頼性が高くなります。

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

この手順では、AMQ Streams にある以下のサンプルファイルを使用します。

- **examples/connect/kafka-connect.yaml**

KafkaConnect リソース（または Source-to-Image(S **2I)サポートのある KafkaConnectS** 2I リソース）の設定に関する詳細は、『**AMQ Streams on OpenShift の使用**』の「Kafka Connect クラスターの設定」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必要があります。
- 稼働中の Kafka クラスターが必要です。

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. Kafka Connect が正常にデプロイされたことを確認します。

```
oc get deployments
```

5.2.2. 複数インスタンスの Kafka Connect 設定

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

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

1. インスタンスが属する Kafka Connect クラスターグループ。
2. コネクターオフセットを保存する Kafka トピック。
3. コネクターおよびタスクステータスの設定を保存する Kafka トピック。
4. コネクターおよびタスクステータスの更新を保存する Kafka トピック。

注記

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

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

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

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

5.2.3. コネクタープラグインでの Kafka Connect の拡張

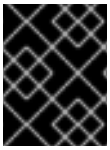
Kafka Connect の AMQ Streams コンテナーイメージには、ファイルベースのデータを Kafka クラスターで出し入れするために 2 つの組み込みコネクターが含まれています。

表5.1 ファイルコネクター

ファイルコネクター	説明
FileStreamSourceConnector	ファイル (ソース) から Kafka クラスターにデータを転送します。
FileStreamSinkConnector	Kafka クラスターからファイル (シンク) にデータを転送します。

ここの手順では、以下を行って、独自のコネクタークラスをコネクターイメージに追加する方法を説明します。

- AMQ Streams を使用した新しいコンテナーイメージの自動作成
- Kafka Connect ベースイメージからコンテナーイメージを作成 (手作業または継続インテグレーションを使用)
- OpenShift ビルドおよび S2I (Source-to-Image) を使用してコンテナーイメージを作成します (OpenShift の場合のみ)



重要

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

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

この手順では、AMQ Streams が追加のコネクターで新しいコンテナー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。コネクタープラグインは、**KafkaConnect** カスタムリソースの **.spec.build.plugins** プロパティーを使用して定義します。AMQ Streams はコネクタープラグインを自動的にダウンロードし、新しいコンテナーイメージに追加します。コンテナーは、**.spec.build.output** に指定されたコンテナーリポジトリーにプッシュされ、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。
- コンテナーレジストリー。

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ー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ーレジストリーだけでなく、Quay や Docker Hub などのパブリックレジストリーもサポートします。

手順

1. **.spec.build.output** でコンテナーレジストリーを指定し、**.spec.build.plugins** で追加のコネクターを指定して、 **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: 962a12151bdf9a5a30627eebac739955a4fd95a08d373b86bdcea2b4d0c27dd6e1edd5cb548045e115e33a9e69b1b2a352bee24df035a0447cb820077af00c03
      - name: camel-telegram
        artifacts:
          - type: tgz
            url: https://repo.maven.apache.org/maven2/org/apache/camel/kafkaconnector/camel-telegram-kafka-connector/0.7.0/camel-telegram-kafka-connector-0.7.0-package.tar.gz
            sha512sum: a9b1ac63e3284bea7836d7d24d84208c49cdf5600070e6bd1535de654f6920b74ad950d51733e8020bf4187870699819f54ef5859c7846ee4081507f48873479
    #...
```

(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ーレジストリーの設定。

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

2. リソースを作成または更新します。

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

3. 新しいコンテナー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。

4. Kafka Connect REST API または KafkaConnector カスタムリソースを使用して、追加したコネクタープラグインを使用します。

その他のリソース

詳細は、『**AMQ Streams on OpenShift の使用**』を参照してください。

- Kafka Connect **Build** schema reference

5.2.3.2. 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 コンテナーイメージを、追加のコネクター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ー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

1. **registry.redhat.io/amq7/amq-streams-kafka-28-rhel7:1.8.0** 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-28-rhel7:1.8.0
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
```

2. コンテナー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ーレジストリーにプッシュします。
4. 新しいコンテナー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** プロパティーを編集します。
     設定された場合、このプロパティーによって 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** (コネクターではない) 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ーイメージを示すようにした後、Cluster Operator を再インストールします。

関連情報

詳細は、『**AMQ Streams on OpenShift の使用**』を参照してください。

- Container image configuration and the **KafkaConnect.spec.image property**

- Cluster Operator configuration and the **STRIMZI_KAFKA_CONNECT_IMAGES** variable

5.2.3.3. OpenShift ビルドおよび S2I (Source-to-Image) を使用したコンテナーイメージの作成

この手順では、OpenShift ビルド と S2I (Source-to-Image) フレームワークを使用して、新しいコンテナーイメージを作成する方法を説明します。

OpenShift ビルドは、S2I がサポートされるビルダーイメージとともに、ユーザー提供のソースコードおよびバイナリーを取得し、これらを使用して新しいコンテナーイメージを構築します。構築後、コンテナーイメージは OpenShfit のローカルコンテナーイメージリポジトリーに格納され、デプロイメントで使用可能になります。

S2I がサポートされる Kafka Connect ビルダーイメージは、**registry.redhat.io/amq7/amq-streams-kafka-28-rhel7:1.8.0** イメージの一部として、Red Hat Ecosystem Catalog で提供されます。このS2I イメージは、バイナリー (プラグインおよびコネクターとともに) を取得し、**/tmp/kafka-plugins/s2i** ディレクトリーに格納されます。このディレクトリーから、Kafka Connect デプロイメントとともに使用できる新しい Kafka Connect イメージを作成します。改良されたイメージの使用を開始すると、Kafka Connect は **/tmp/kafka-plugins/s2i** ディレクトリーからサードパーティープラグインをロードします。

重要

ビルド設定が **KafkaConnect リソースに導入されたため** 、AMQ Streams はデータコネクションに必要なコネクタープラグインでコンテナーイメージを自動的にビルドできるようになりました。そのため、Source-to-Image (S2I) を使用した Kafka Connect のサポートは非推奨となり、AMQ Streams 1.8 以降で削除されます。この変更に備えるため、Kafka Connect S2I インスタンスを Kafka Connect インスタンスに移行 できます。

手順

1. コマンドラインで **oc apply** コマンドを使用し、Kafka Connect の S2I クラスターを作成およびデプロイします。

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

2. Kafka Connect プラグインでディレクトリーを作成します。

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

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

3. **oc start-build** コマンドで、準備したディレクトリーを使用してイメージの新しいビルドを開始します。

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

注記

ビルドの名前は、デプロイされた Kafka Connect クラスターと同じになります。

4. ビルドが完了したら、Kafka Connect のデプロイメントによって新しいイメージが自動的に使用されます。

5.2.4. コネクターの作成および管理

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

コネクターは特定の **コネクタークラス** のインスタンスで、メッセージに関して関連する外部システムとの通信方法を認識しています。コネクターは多くの外部システムで使用でき、独自のコネクターを作成することもできます。

ソース および **シンク** タイプのコネクターを作成できます。

ソースコネクター

ソースコネクターは、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ーです。

シンクコネクター

シンクコネクターは、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ーです。

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

- KafkaConnector リソース (KafkaConnectors と呼ばれます)
- Kafka Connect REST API

API を使用すると、以下を行うことができます。

- コネクターインスタンスのステータスの確認。
- 稼働中のコネクターの再設定。
- コネクターインスタンスのコネクタータスク数の増減。
- コネクターの再起動。
- 失敗したタスクを含むコネクタータスクの再起動。
- コネクターインスタンスの一時停止。
- 一時停止したコネクターインスタンスの再開。
- コネクターインスタンスの削除。

5.2.4.1. KafkaConnector リソース

KafkaConnectors を使用すると、Kafka Connect のコネクターインスタンスを OpenShift ネイティブに作成および管理できるため、cURL などの HTTP クライアントが必要ありません。その他の Kafka リソースと同様に、コネクターの望ましい状態を OpenShift クラスターにデプロイされた KafkaConnector YAML ファイルに宣言し、コネクターイン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インスタンスを管理した後、更新を適用します。アノテーションは、コネクターインスタンスおよびコネクタータスクを手動で再起動するために使用されます。該当する KafkaConnector を削除して、コネクターを削除します。

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 クラスターのために有効にするには、**KafkaConnect** リソースでアノテーションを使用する必要があります。手順は、『**Using AMQ Streams on OpenShift**』の「Configuring Kafka Connect」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インスタンスの設定を更新します。

AMQ Streams には、**examples/connect/source-connector.yaml** という名前のサンプル **KafkaConnector** が含まれます。この例を使用して、「サンプル KafkaConnector リソースのデプロイ」 の説明に従って **FileStreamSourceConnector** および **FileStreamSinkConnector** を作成および管理できます。

5.2.4.2. Kafka Connect REST API の可用性

Kafka Connect REST API は、**<connect-cluster-name>-connect-api** サービスとして 8083 番ポートで使用できます。

KafkaConnectors が有効になっている場合、Kafka Connect REST API に直接手作業で追加された変更は Cluster Operator によって元に戻されます。

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

5.2.5. サンプル KafkaConnector リソースのデプロイ

AMQ Streams には、**examples/connect/source-connector.yaml に KafkaConnector の例** が含まれています。これにより、Kafka ライセンスファイルの各行（サンプルファイルソース）を 1 つの Kafka トピックに送信する基本的な **FileStreamSourceConnector** インスタンスが作成されます。

この手順では、以下を作成する方法を説明します。

- Kafka ライセンスファイル（ソース）からデータを読み取り、データをメッセージとして Kafka トピックに書き込む **FileStreamSourceConnector**。
- 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**。

注記

実稼働環境で、「コネクタープラグインでの Kafka Connect の拡張」 の説明どおりに、必要な Kafka Connect コネクターが含まれるコンテナー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。ここで説明するように、コンテナーでこれらのコネクターを実行することは、実稼働のユースケースには適していません。

前提条件

- Kafka Connect デプロイメント。
- Kafka Connect デプロイメントで KafkaConnectors が有効になっている。
- Cluster Operator が稼働している必要があります。

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 1
  labels:
    strimzi.io/cluster: my-connect-cluster 2
spec:
  class: org.apache.kafka.connect.file.FileStreamSourceConnector 3
  tasksMax: 2 4
  config: 5
    file: "/opt/kafka/LICENSE" 6
    topic: my-topic 7
    # ...
```

1. コネクター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。
2. コネクターインスタンスを作成する Kafka Connect クラスターの名前。コネクターは、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。
3. 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。

4 コネクターが作成できる Kafka Connect **Tasks** の最大数。

5 キーと値のペアとしてのコネクター設定。

6 このサンプルソースコネクター設定では、**/opt/kafka/LICENSE** ファイルからデータが読み取られます。

7 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 (1)
  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ー設定。

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. コネクター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. コンテナーで、**kafka -console-consumer.sh** を実行して、ソースコネクター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ーの設定オプション

コネクター設定は KafkaConnector リソースの **spec.config** プロパティーで定義されます。

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。他のコネクターは異なる設定オプションをサポートします。

表5.2 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、標準入力が使用されます。
topic	List	Null	データのパブリッシュ先となる Kafka トピック。

表5.3 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。

その他のリソース

- 「コネクターの作成および管理」

5.2.6. Kafka コネクターの再起動の実行

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

前提条件

- Cluster Operator が稼働している必要があります。

手順

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

```
oc get KafkaConnector
```

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

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

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

関連情報

- 『Deploying and Upgrading』の「Creating and managing connectors」。

5.2.7. Kafka コネクタータスクの再起動の実行

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

前提条件

- Cluster Operator が稼働している必要があります。

手順

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

```
oc get KafkaConnector
```

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

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

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

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

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

関連情報

- 『**Deploying and Upgrading**』の「Creating and managing connectors」。

5.3. 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。このプロセスはミラーリングと言われ、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します。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。

5.3.1. 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って、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。

この手順では、AMQ Streams にある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**

KafkaMirrorMaker または**KafkaMirrorMaker 2** リソースの設定に関する詳細は、『**AMQ Streams on OpenShift の使用**』の「Kafka MirrorMaker クラスターの設定」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 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. MirrorMaker が正常にデプロイされたことを確認します。

```
oc get deployments
```

5.4. 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。

5.4.1. 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って **KafkaBridge** リソースが作成されます。

この手順では、AMQ Streams にある以下のサンプルファイルを使用します。

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

KafkaBridge リソースの設定に関する詳細は、『**AMQ Streams on OpenShift の使用**』の「Kafka Bridge クラスターの設定」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 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. Kafka Bridge が正常にデプロイされたことを確認します。

```
oc get deployments
```

第6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ー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す。『**Using AMQ Streams on OpenShift**』で説明する 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。

6.1. 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ー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必要があります。

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-28-rhel7:1.8.0
--rm=true --restart=Never -- bin/kafka-console-producer.sh --broker-list 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-28-rhel7:1.8.0
--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。

6.2. OPENSHIFT 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、ベアメタル等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 ブローカーの simple、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、必ず同等の Kafka 設定と一致するようにしてください。

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

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) および PEM 形式 (.crt) の証明書を利用できます。この手順では、PKCS #12 証明書について説明します。

前提条件

前提条件

- クライアントが Kafka クラスターを使用できる必要があります。
- 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必要があります。

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、「Generic Kafka listener schema reference」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 によって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 ブートストラップおよびブローカーサービスのアドレス を提供しない場合、その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 の名前で構成されます。たとえば、**my-cluster-kafka-listener1-bootstrap-myproject** (**CLUSTER-NAME**-kafka-**LISTENER-NAME**-bootstrap-**NAMESPACE**) となります。**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-CONFIG-FILE
```

Kafka クラスターは、TLS 認証を使用する Kafka ブローカーリスナーと共に設定されます。

Kafka ブローカー Pod ごとにサービスが作成されます。

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

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 ID を検証するためのクラスター CA 証明書も、**Kafka** リソースと同じ名前で作成されます。

3. **Kafka** リソースのステータスから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[?
(@.type=="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 (1)
spec:
  authentication:
    type: tls (2)
  authorization:
    type: simple
    acls: (3)
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Read
      - resource:
          type: topic
          name: my-topic
          patternType: literal
        operation: Describe
      - resource:
          type: group
          name: my-group
          patternType: literal
        operation: Read
```

(1) 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。

(2) 認証は **tls** として指定されます。

(3) 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。

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。

以下は例になります。

```
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 証明書を目的の証明書形式に抽出します。

```
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詳細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. ユーザーシークレットから目的の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 証明書の認証詳細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。このプロパティーは、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

- Listener authentication options
- Kafka authorization options
- 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7章 AMQ STREAMS のメトリクスおよびダッシュボードの設定

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。メトリクスは、Kafka、ZooKeeper、および AMQ Streams の他の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 からキーメトリクスを使用します。次に、Grafana はこれらのメトリクス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。

OpenShift Container Platform 4.x では、AMQ Streams は **ユーザー定義プロジェクトのモニタリング** (OpenShift の機能) を使用し、Prometheus の設定プロセスを容易にします。

OpenShift Container Platform 3.11 では、Prometheus および Alertmanager コンポーネントを別々にクラスターにデプロイする必要があります。

OpenShift Container Platform のバージョンに関係なく、AMQ Streams に Prometheus メトリクス設定をデプロイ して開始する必要があります。

次に、OpenShift Container Platform のバージョンに適した手順に従います。

- 「OpenShift 4 での Kafka メトリクスおよびダッシュボードの表示」
- 「OpenShift 3.11 での Kafka メトリクスおよびダッシュボードの表示」

Prometheus および Grafana が設定されると、Grafana ダッシュボードおよびアラートルールのサンプルを使用して Kafka クラスターを監視できます。

追加の監視オプション

Kafka Exporter は、コンシューマーラグに関連する追加の監視を提供する任意のコンポーネントです。AMQ Streams で Kafka Exporter を使用する場合は、「Configure the **Kafka** resource to deploy Kafka Exporter with your Kafka cluster」を参照してください。

さらに、分散トレーシングを設定してメッセージをエンドツーエンドで追跡するように、デプロイメントを設定することもできます。詳細は、『**Using AMQ Streams on OpenShift**』の「Distributed tracing」を参照してください。

その他のリソース

- Prometheus ドキュメント
- Grafana ドキュメント
- Kafka ドキュメントの 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 ドキュメントの 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。

7.1. 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のサンプルファイルは、**examples/metrics** ディレクトリー にあります。以下のリストが示すように、一部のファイルは OpenShift Container Platform 3.11 のみで使用され、OpenShift Container Platform 4.x では使用されません。

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 - OPENSHIFT 3.11 ONLY 3
├── prometheus-alertmanager-config
│   └── alert-manager-config.yaml 4
├── prometheus-install
│   ├── alert-manager.yaml - OPENSHIFT 3.11 ONLY 5
│   ├── prometheus-rules.yaml 6
│   ├── prometheus.yaml - OPENSHIFT 3.11 ONLY 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. Grafana ダッシュボードのサンプル
2. Grafana イメージのインストールファイル。
3. **OPENSHIFT 3.11 のみ該当**: 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クスをスクレープする追加の Prometheus 設定。これらのメトリクスは、ノード上の OpenShift cAdvisor エージェントおよび kubelet から直接提供されます。
4. Alertmanager による通知送信のためのフック定義。
5. **OPENSHIFT 3.11 のみ該当**: 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。
7. **OPENSHIFT 3.11 のみ該当**: Prometheus イメージのインストールリソースファイル。
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 から直接メトリクスデータをスクレープできます。
9. メトリクス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クス設定。

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クス設定。

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クス設定。

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。

7.1.1. Grafana ダッシュボードのサンプル

Grafana ダッシュボードのサンプルは、以下のリソースを監視するために提供されます。

AMQ Streams Kafka

以下のメトリクスを表示します。

- オンラインのブローカーの数
- クラスター内のアクティブなコントローラーの数
- 非同期レプリカがリーダーに選択される割合
- オンラインのレプリカ
- 複製の数が最低数未満であるパーティションの数
- 最小の In-Sync レプリカ数にあるパーティション
- 最小の In-Sync レプリカ数未満のパーティション
- アクティブなリーダーを持たないため、書き込みや読み取りができないパーティション
- Kafka ブローカー Pod のメモリー使用量
- 集約された Kafka ブローカー Pod の CPU 使用率
- Kafka ブローカー Pod のディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数
- 受信バイトレートの合計
- 送信バイトレートの合計
- 受信メッセージレート
- 生成要求レートの合計
- バイトレート

- 生成要求レート
- 取得要求レート
- ネットワークプロセッサーの平均時間アイドル率
- リクエストハンドラーの平均時間アイドル率
- ログサイズ

AMQ Streams ZooKeeper

以下のメトリクスを表示します。

- ZooKeeper アンサンブルのクォーラムサイズ
- **アクティブな** 接続の数
- サーバーのキューに置かれたリクエストの数
- ウォッチャーの数
- ZooKeeper Pod のメモリー使用量
- 集約された ZooKeeper Pod の CPU 使用率
- ZooKeeper Pod のディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数
- サーバーがクライアントリクエストに応答するまでの時間 (最大、最小、および平均)

AMQ Streams Kafka Connect

以下のメトリクスを表示します。

- 受信バイトレートの合計
- 送信バイトレートの合計
- ディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間

AMQ Streams Kafka MirrorMaker 2

以下のメトリクスを表示します。

- コネクターの数
- タスクの数
- 受信バイトレートの合計

- 送信バイトレートの合計
- ディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間

AMQ Streams の Operator

以下のメトリクスを表示します。

- カスタムリソース
- 1時間あたりの成功したカスタムリソース調整の数
- 1時間あたりの失敗したカスタムリソース調整の数
- 1時間あたりのロックなしの調整の数
- 1時間あたりの開始された調整の数
- 1時間あたりの定期的な調整の数
- 最大の調整時間
- 平均の調整時間
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数

ダッシュボードは、AMQ Streams の Kafka Bridge および Cruise Control コンポーネントにも提供されます。

すべてのダッシュボードは、JVM メトリクスの他に、各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、Operator ダッシュボードは、処理中の調整またはカスタムリソースの数に関する情報を提供します。

7.1.2. Prometheus メトリクス設定の例

AMQ Streams は、Prometheus JMX Exporter を使用して、Prometheus によってスクレープされる HTTP エンドポイントを使用して JMX 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ることがあります。

AMQ Streams では、再ラベル付けルールがすでに定義されたカスタムリソース設定 YAML ファイルのサンプルが提供されます。Prometheus メトリクス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、メトリクス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7.1 メトリクス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** および **KafkaConnectS2I**	**kafka-connect-metrics.yaml**
Kafka MirrorMaker 2.0	**KafkaMirrorMaker2**	**kafka-mirror-maker-2-metrics.yaml**
Kafka Bridge	**KafkaBridge**	**kafka-bridge-metrics.yaml**
Cruise Control	**Kafka**	**kafka-cruise-control-metrics.yaml**

関連情報

- 「Prometheus メトリクス設定のデプロイ」
- 再ラベル付けの使用方法の詳細は、Prometheus ドキュメントの「Configuration」を参照してください。

7.2. PROMETHEUS メトリクス設定のデプロイ

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。

再ラベル付けルールの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。

- 使用するカスタムリソース定義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。

7.2.1. Prometheus メトリクス設定のカスタムリソースへのコピー

Grafana ダッシュボードを監視に使用するには、メトリクス設定サンプルをカスタムリソースにコピーします。

この手順では、**Kafka** リソースが更新されますが、監視をサポートするすべてのコンポーネントについて手順は同じです。

手順

デプロイメントの **Kafka** リソースごとに以下の手順を実行します。

1. エディターで **Kafka** リソースを更新します。

```
oc edit kafka KAFKA-CONFIG-FILE
```

2. **kafka-metrics.yaml** の設定例を、ユーザーの **Kafka** リソース定義にコピーします。

3. ファイルを保存し、更新したリソースが調整されるのを待ちます。

7.2.2. Prometheus メトリクス設定での Kafka クラスターのデプロイメント

Grafana ダッシュボードを監視に使用するには、メトリクス設定でサンプル Kafka クラスターをデプロイできます。

この手順では、**kafka-metrics.yaml** ファイルが **Kafka** リソースに使用されます。

手順

- メトリクス設定サンプルで Kafka クラスターをデプロイします。

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

7.3. OPENSHIFT 4 での KAFKA メトリクスおよびダッシュボードの表示

AMQ Streams が OpenShift Container Platform 4.x 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である場合、**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クスを消費します。

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独自のプロジェクト内の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 メトリクスを Grafana ユーザーインターフェー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のモニタリングを提供します。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**。

Grafana バージョン 6.3 は、サポートされる最小バージョンです。

前提条件

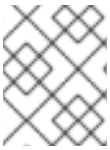
- YAML ファイルのサンプルを使用して、Prometheus メトリクス設定がデプロイされている 必要があります。
- **ユーザー定義プロジェクトの監視**が有効になっている必要があります。OpenShift Container Platform クラスターに、クラスター管理者が作成した **cluster-monitoring-config** ConfigMap が存在する必要があります。詳しい情報は、以下の資料を参照してください。
  - OpenShift Container Platform 4.6 の「ユーザー定義プロジェクトのモニタリングの有効化」。
  - OpenShift Container Platform 4.5 の「独自のサービスのモニタリングの有効化」。
- ユーザー定義のプロジェクトを監視するには、クラスター管理者がユーザーに **monitoring-rules-edit** または **monitoring-edit** ロールを割り当て済みである必要があります。以下を参照してください。
  - OpenShift Container Platform 4.6 の「ユーザーに対するユーザー定義のプロジェクトをモニターするパーミッションの付与」。
  - OpenShift Container Platform 4.5 の「WEB コンソールを使用したユーザーパーミッションの付与」

手順の概要

OpenShift Container Platform 4.x 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

7.3.1. Prometheus リソースのデプロイ



注記

OpenShift Container Platform 4.x で AMQ Streams を実行している場合は、この手順を使用します。

Kafka メトリクスを使用するよう Prometheus を有効にするには、サンプルメトリクスファイルで **PodMonitor** リソースを設定およびデプロイします。**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プします。

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

- 稼働中の Kafka クラスターが必要です。

- AMQ Streams で 提供されるアラートルールのサンプル を確認します。

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 4 での Kafka メトリクス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** プロパティーを編集します。

```
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) メトリクスをスクレープする 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
```

その他のリソース

- OpenShift Container Platform 4.6 の『モニタリング』ガイド。
- 「アラートルールの例」

7.3.2. Grafana のサービスアカウントの作成

注記

OpenShift Container Platform 4.x で AMQ Streams を実行している場合は、この手順を使用します。

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。

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 1
```

```
roleRef:
  kind: ClusterRole
  name: cluster-monitoring-view
  apiGroup: rbac.authorization.k8s.io
```

1 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
```

関連情報

- 「OpenShift 4 での Kafka メトリクスおよびダッシュボードの表示」

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

注記

OpenShift Container Platform 4.x で AMQ Streams を実行している場合は、この手順を使用します。

この手順では、OpenShift Container Platform 4.x モニタリングスタックに対して設定された Grafana アプリケーションをデプロイする方法を説明します。

OpenShift Container Platform 4.x 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す。

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** プロパティーに貼り付けます。

```
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}" (1)
  editable: true
```

(1) **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** で構成される 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:6.3.0
        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
```

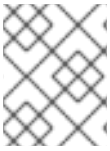
5. Grafana アプリケーションを、Kafka クラスターが含まれるプロジェクトにデプロイします。

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

関連情報

- 「OpenShift 4 での Kafka メトリクスおよびダッシュボードの表示」
- OpenShift Container Platform 4.6 の『モニタリング』ガイド。

7.3.4. Grafana サービスへのルートの作成



注記

OpenShift Container Platform 4.x で AMQ Streams を実行している場合は、この手順を使用します。

Grafana サービスを公開するルートを介して、Grafana ユーザーインターフェースにアクセスできます。

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

手順

- **grafana** サービスへのルートの作成:

```
oc create route edge MY-GRAFANA-ROUTE --service=grafana --namespace=KAFKA-NAMESPACE
```

関連情報

- 「OpenShift 4 での Kafka メトリクスおよびダッシュボードの表示」

7.3.5. Grafana ダッシュボードサンプルのインポート

注記

OpenShift Container Platform 4.x で AMQ Streams を実行している場合は、この手順を使用します。

Grafana ユーザーインターフェースを使用して Grafana ダッシュボードのサンプルをインポートします。

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
- 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. **Dashboards** > **Manage**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に、ステップ 1-7 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。

その他のリソース

- 「Grafana サービスへのルートの作成」
- 「OpenShift 4 での Kafka メトリクスおよびダッシュボードの表示」

7.4. OPENSHIFT 3.11 での KAFKA メトリクスおよびダッシュボードの表示

AMQ Streams が OpenShift Container Platform 3.11 にデプロイされた場合、Prometheus を使用して AMQ Streams で提供される Grafana ダッシュボードのサンプルのモニタリングデータを提供できます。Prometheus コンポーネントをクラスターに手動でデプロイする必要があります。

Grafana ダッシュボードのサンプルを実行するには、以下を行う必要があります。

1. メトリクス設定を Kafka クラスターリソースに追加します。
2. Prometheus および Prometheus Alertmanager をデプロイします。
3. Grafana のデプロイメント

注記

このセクションで参照されるリソースは、まず監視を設定することを目的としており、これらはサンプルとしてのみ提供されます。実稼働環境で Prometheus または Grafana を設定、実行するためにサポートがさらに必要な場合は、それぞれのコミュニティーに連絡してください。

7.4.1. Prometheus のサポート

AMQ Streams が OpenShift Container Platform 3.11 にデプロイされた場合は、Prometheus サーバーはサポート**されません**。しかし、メトリクスを公開するために使用される Prometheus エンドポイントと Prometheus JMX Exporter はサポートされます。

Prometheus を使用して監視を行う場合に備え、詳細な手順とメトリクス設定ファイルのサンプルが提供されます。

7.4.2. Prometheus の設定

注記

OpenShift Container Platform 3.11 で AMQ Streams を実行している場合は、以下の手順を使用します。

Prometheus では、システム監視とアラート通知のオープンソースのコンポーネントセットが提供されます。

ここでは、AMQ Streams が OpenShift Container Platform 3.11 にデプロイされている場合に、提供された Prometheus イメージと設定ファイルを使用して、Prometheus サーバーを実行および管理する方法を説明します。

前提条件

- 互換性のあるバージョンの Prometheus および Grafana を OpenShift Container Platform 3.11 クラスターにデプロイしている。
- Prometheus サーバー Pod の実行に使用されるサービスアカウントが OpenShift API サーバーにアクセスできる。これにより、サービスアカウントはメトリクスの取得元となるクラスターにある Pod の一覧を取得できます。
  詳細は、「Discovering services」 を参照してください。

7.4.2.1. Prometheus の設定

AMQ Streams では、Prometheus サーバーの設定ファイルのサンプル が提供されます。

デプロイメント用に Prometheus イメージが提供されます。

- **prometheus.yaml**

Prometheus 関連の追加設定も、以下のファイルに含まれています。

- **prometheus-additional.yaml**
- **prometheus-rules.yaml**
- **strimzi-pod-monitor.yaml**

Prometheus が監視データを取得するには、互換性のあるバージョンの Prometheus を OpenShift Container Platform 3.11 クラスターにデプロイしている必要があります。

次に、設定ファイルを使用して Prometheus をデプロイ します。

7.4.2.2. Prometheus リソース

Prometheus 設定を適用すると、以下のリソースが OpenShift クラスターに作成され、Prometheus Operator によって管理されます。

- **ClusterRole**。コンテナーメトリクスのために Kafka と ZooKeeper の Pod、cAdvisor および kubelet によって公開される health エンドポイントを読み取る権限を Prometheus に付与します。
- **ServiceAccount**。これで Prometheus Pod が実行されます。
- **ClusterRoleBinding**。**ClusterRole** を **ServiceAccount** にバインドします。
- **Deployment**。Prometheus Operator Pod を管理します。
- **PodMonitor**。Prometheus Pod の設定を管理します。
- **Prometheus**。Prometheus Pod の設定を管理します。
- **PrometheusRule**。Prometheus Pod のアラートルールを管理します。
- **Secret**。Prometheus の追加設定を管理します。
- **Service**。クラスターで稼働するアプリケーションが Prometheus に接続できるようにします (例: Prometheus をデータソースとして使用する Grafana)。

7.4.2.3. Prometheus のデプロイメント

Kafka クラスターの監視データを取得するには、独自の Prometheus デプロイメントを使用するか、Prometheus Docker イメージのインストールリソースサンプルファイルと Prometheus 関連リソースの YAML ファイル を適用して Prometheus をデプロイすることができます。

デプロイメントプロセスでは、**ClusterRoleBinding** が作成され、デプロイメントのために指定された namespace で Alertmanager インスタンスが検出されます。

前提条件

- 提供されるアラートルールのサンプルを確認します。

手順

1. Prometheus のインストール先となる namespace に従い、Prometheus インストールファイル (**prometheus.yaml**) を変更します。
   Linux の場合は、以下を使用します。

   ```
   sed -i 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
   ```

   MacOS の場合は、以下を使用します。

   ```
   sed -i '' 's/namespace: .*/namespace: my-namespace/' prometheus.yaml
   ```

2. **PodMonitor** リソースを **strimzi-service-monitor.yaml** で編集し、Pod からメトリクスデータをスクレープする Prometheus ジョブを定義します。
   namespaceSelector.matchNames プロパティーを、メトリクスのスクレープ元の Pod が実行されている namespace で更新します。

PodMonitor は、Apache Kafka、ZooKeeper、Operator、Kafka Bridge、および Cruise Control の Pod から直接データをスクレープするために使用されます。

3. **prometheus.yaml** インストールファイルを編集し、ノードから直接メトリクスをスクレープするための追加設定を含めます。
提供される Grafana ダッシュボードが表示する 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クスは、ノード上の OpenShift cAdvisor エージェントおよび kubelet から直接提供されます。

   a. 設定ファイル (**prometheus-additional.yaml** in the **examples/metrics/prometheus-additional-properties** ディレクトリー) から **Secret** リソースを作成します。

   ```
   oc apply -f prometheus-additional.yaml
   ```

   b. **prometheus.yaml** ファイルで **additionalScrapeConfigs** プロパティーを編集して、**Secret** の名前と **prometheus-additional.yaml** ファイルを含めます。

4. Prometheus リソースをデプロイします。

```
oc apply -f strimzi-pod-monitor.yaml
oc apply -f prometheus-rules.yaml
oc apply -f prometheus.yaml
```

7.4.3. Prometheus Alertmanager の設定

Prometheus Alertmanager は、アラートを処理して通知サービスにルーティングするためのプラグインです。Alertmanager は、アラートルールを基にして潜在的な問題と見られる状態を通知し、監視で必要な条件に対応します。

7.4.3.1. Alertmanager の設定

AMQ Streams には、Prometheus Alertmanager の設定ファイルのサンプルが含まれます。

設定ファイルは、Alertmanager をデプロイするためのリソースを定義します。

- **alert-manager.yaml**

追加の設定ファイルには、Kafka クラスターから通知を送信するためのフック定義が含まれます。

- **alert-manager-config.yaml**

Alertmanger で Prometheus アラートの処理を可能にするには、設定ファイルを使用して以下を行います。

- Alertmanager のデプロイ。

7.4.3.2. アラートルール

アラートルールによって、メトリクス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。

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。

その他のリソース

アラートルールの設定についての詳細は、Prometheus ドキュメントの「Configuration」を参照してください。

7.4.3.3. アラートルールの例

Kafka および ZooKeeper メトリクスのアラートルールのサンプルは AMQ Streams に含まれており、Prometheus デプロイメントで使用できます。

アラートルールの定義に関する一般的な留意点:

- **for** プロパティーはルールと併用され、アラートがトリガーされる前に条件が維持されなければならない期間を決定します。
- 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** メトリクス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。

Kafka アラートルール

UnderReplicatedPartitions

現在のブローカーがリードレプリカでありながら、パーティションのトピックに設定された **min.insync.replicas** よりも複製数が少ないパーティションの数が示されます。このメトリクスにより、フォロワーレプリカをホストするブローカーの詳細が提供されます。リーダーからこれらのフォロワーへの複製が追い付いていません。その理由として、現在または過去にオフライン状態になっていたり、過剰なスロットリングが適用されたブローカー間の複製であることが考えられます。この値がゼロより大きい場合にアラートが発生し、複製の数が最低数未満であるパーティションの情報がブローカー別に通知されます。

AbnormalControllerState

現在のブローカーがクラスターのコントローラーであるかどうかを示します。メトリクスは 0 または 1 です。クラスターのライフサイクルでは、1 つのブローカーのみかコントローラーとなるはずで、クラスターには常にアクティブなコントローラーが存在する必要があります。複数のブローカーがコントローラーであることが示される場合は問題になります。そのような状態が続くと、すべてのブローカーのこのメトリクスの合計値が 1 でない場合にアラートが発生します。合計値が 0 であればアクティブなコントローラーがなく、合計値が 1 を超えればコントローラーが複数あることを意味します。

UnderMinIsrPartitionCount

書き込み操作の完了を通知しなければならないリード Kafka ブローカーの ISR (In-Sync レプリカ) が最小数 (**min.insync.replicas** を使用して指定) に達していないことを示します。このメトリクスでは、ブローカーがリードし、In-Sync レプリカの数が最小数に達していない、パーティションの数が定義されます。この値がゼロより大きい場合にアラートが発生し、完了通知 (ack) が最少数未満であった各ブローカーのパーティション数に関する情報が提供されます。

OfflineLogDirectoryCount

ハードウェア障害などの理由によりオフライン状態であるログディレクトリーの数を示します。そのため、ブローカーは受信メッセージを保存できません。この値がゼロより大きい場合にアラートが発生し、各ブローカーのオフライン状態であるログディレクトリーの数に関する情報が提供されます。

KafkaRunningOutOfSpace

データの書き込みに使用できる残りのディスク容量を示します。この値が 5GiB 未満になるとアラートが発生し、永続ボリューム要求 (Persistent Volume Claim、PVC) ごとに容量不足のディスクに関する情報が提供されます。しきい値は **prometheus-rules.yaml** で変更できます。

ZooKeeper アラートルール

AvgRequestLatency

サーバーがクライアントリクエストに応答するまでの時間を示します。この値が 10 (tick) を超えるとアラートが発生し、各サーバーの平均リクエストレイテンシーの実際の値が通知されます。

OutstandingRequests

サーバーでキューに置かれたリクエストの数を示します。この値は、サーバーが処理能力を超えるリクエストを受信すると上昇します。この値が 10 よりも大きい場合にアラートが発生し、各サーバーの未処理のリクエスト数が通知されます。

ZookeeperRunningOutOfSpace

このメトリクスは、ZooKeeper へのデータ書き込みに使用できる残りのディスク容量を示します。この値が 5GiB 未満になるとアラートが発生し、永続ボリューム要求 (Persistent Volume Claim、PVC) ごとに容量不足のディスクに関する情報が提供されます。

7.4.3.4. Alertmanager のデプロイメント

Alertmanager をデプロイするには、設定ファイルのサンプルを適用します。

AMQ Streams に含まれる設定サンプルでは、Slack チャネルに通知を送信するように Alertmanager を設定します。

デプロイメントで以下のリソースが定義されます。

- **Alertmanager**。Alertmanager Pod を管理します。
- **Secret**。Alertmanager の設定を管理します。
- **Service**。参照しやすいホスト名を提供し、他のサービスが Alertmanager に接続できるようにします (Prometheus など)。

前提条件

- メトリクスが Kafka クラスターリソースに設定されている必要があります。
- Prometheus がデプロイされている必要があります。

手順

1. Alertmanager 設定ファイル (**examples/metrics/prometheus-alertmanager-config** ディレクトリーの **alert-manager-config.yaml**) から **Secret** リソースを作成します。

```
oc apply -f alert-manager-config.yaml
```

2. 以下を行って、**alert-manager-config.yaml** ファイルを更新します。
   - **slack_api_url** プロパティーを、Slack ワークスペースのアプリケーションに関連する Slack API URL の実際の値に置き換えます。
   - **channel** プロパティーを、通知が送信される実際の Slack チャネルに置き換えます。
3. Alertmanager をデプロイします。

```
oc apply -f alert-manager.yaml
```

7.4.4. Grafana の設定

Grafana では、Prometheus メトリクスを視覚化できます。

AMQ Streams で提供される Grafana ダッシュボードサンプルをデプロイして有効化できます。

7.4.4.1. Grafana のデプロイメント

Prometheus メトリクスを視覚化するには、独自の Grafana インストールを使用するか、examples/metrics ディレクトリーにある **grafana.yaml** ファイルを適用して Grafana をデプロイできます。

前提条件

- メトリクスが Kafka クラスターリソースに設定されている必要があります。
- Prometheus および Prometheus Alertmanager がデプロイされている必要があります。

手順

1. Grafana をデプロイします。

```
oc apply -f grafana.yaml
```

2. Grafana ダッシュボードを有効にします。

7.4.4.2. Grafana ダッシュボードサンプルの有効化

AMQ Streams には、Grafana のダッシュボード設定ファイルのサンプル が含まれています。ダッシュボードのサンプルは、**examples/metrics/grafana-dashboards** ディレクトリーの以下の JSON ファイルで提供されます。

- **strimzi-kafka.json**
- **strimzi-zookeeper.json**
- **strimzi-operators.json**
- **strimzi-kafka-connect.json**
- **strimzi-kafka-mirror-maker-2.json**
- **strimzi-kafka-bridge.json**

- **strimzi-cruise-control.json**
- **strimzi-kafka-exporter.json**

ダッシュボードのサンプルは、主なメトリクスの監視を開始するための雛形として使用できますが、使用できるすべてのメトリックスを対象としていません。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。

Prometheus および Grafana の設定後に、Grafana ダッシュボードで AMQ Streams データを可視化できます。

注記

アラート通知ルールは定義されていません。

ダッシュボードにアクセスする場合、**port-forward** コマンドを使用して Grafana Pod からホストにトラフィックを転送できます。

注記

Grafana Pod の名前はユーザーごとに異なります。

手順

1. Grafana サービスの詳細を取得します。

```
oc get service grafana
```

以下に例を示します。

NAME	TYPE	CLUSTER-IP	PORT(S)
grafana	ClusterIP	172.30.123.40	3000/TCP

ポート転送用のポート番号を書き留めておきます。

2. **port-forward** を使用して、Grafana ユーザーインターフェースを **localhost:3000** にリダイレクトします。

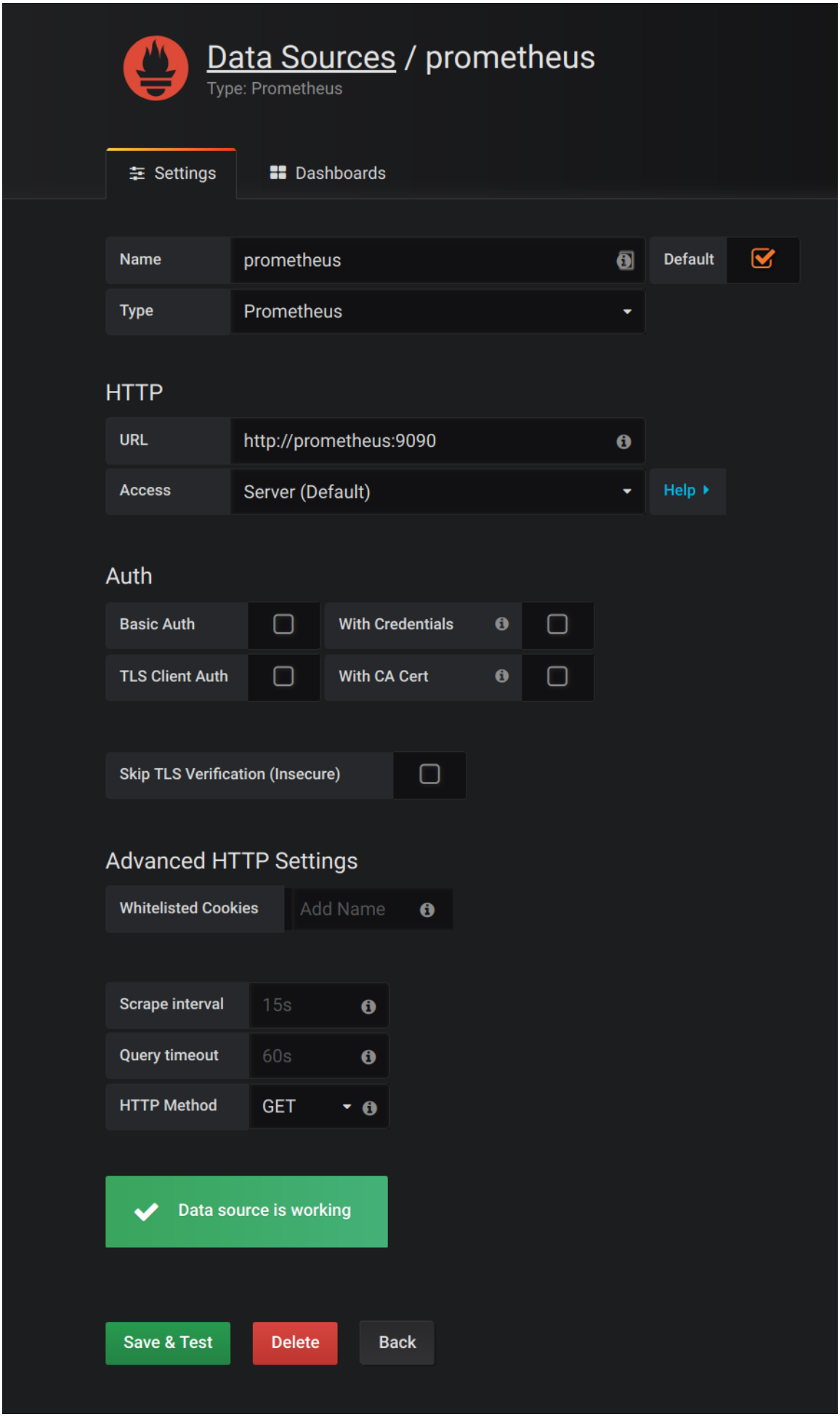
```
oc port-forward svc/grafana 3000:3000
```

3. Web ブラウザーで **http://localhost:3000** を指定します。
Grafana のログインページが表示されます。

4. 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。

5. Prometheus を **データソース** として追加します。
   - 名前を指定します。
   - **Prometheus** をタイプとして追加します。

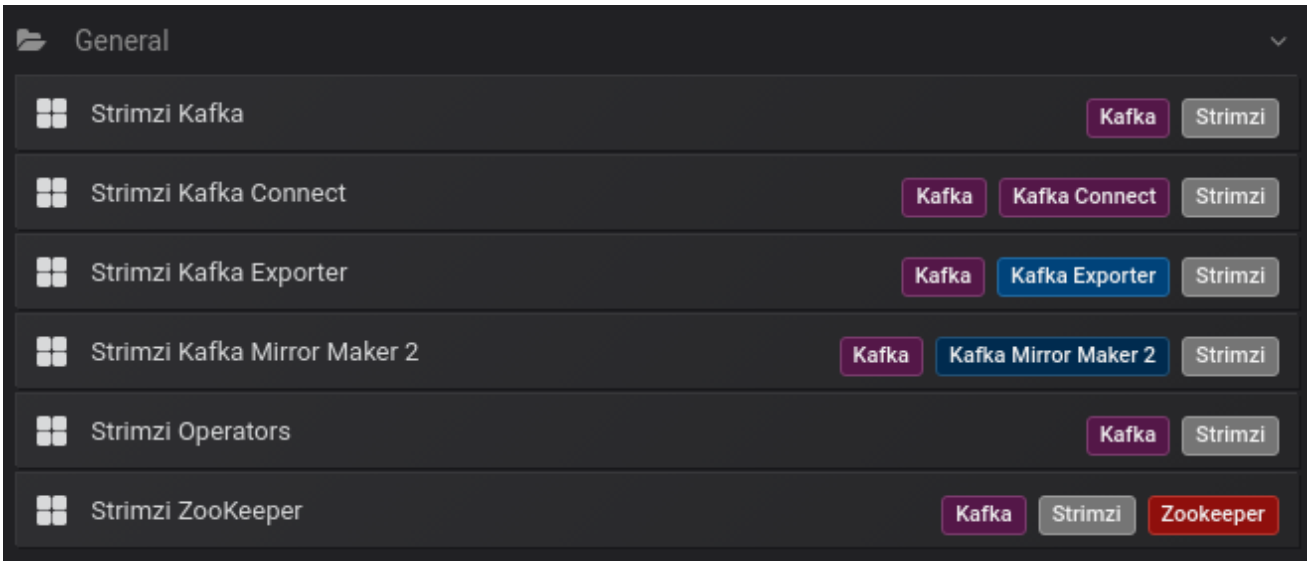
- Prometheus サーバーの URL (http://prometheus-operated:9090) を指定します。
詳細を追加したら、保存して接続をテストします。


Data Sources / prometheus
Type: Prometheus
Settings
Dashboards
Name
prometheus
Default
Type
Prometheus
HTTP
URL
http://prometheus:9090
Access
Server (Default)
Help
Auth
Basic Auth
With Credentials
TLS Client Auth
With CA Cert
Skip TLS Verification (Insecure)
Advanced HTTP Settings
Whitelisted Cookies
Add Name
Scrape interval
15s
Query timeout
60s
HTTP Method
GET
Data source is working
Save & Test
Delete
Back

6. **Dashboards → Import** から、ダッシュボードのサンプルをアップロードするか、JSON を直接貼り付けます。

7. 上部のヘッダーでダッシュボードのドロップダウンメニューをクリックし、表示するダッシュボードを選択します。
   Prometheus サーバーが AMQ Streams クラスターのメトリクスを収集すると、それがダッシュボードに反映されます。

図7.1 ダッシュボードの選択オプション



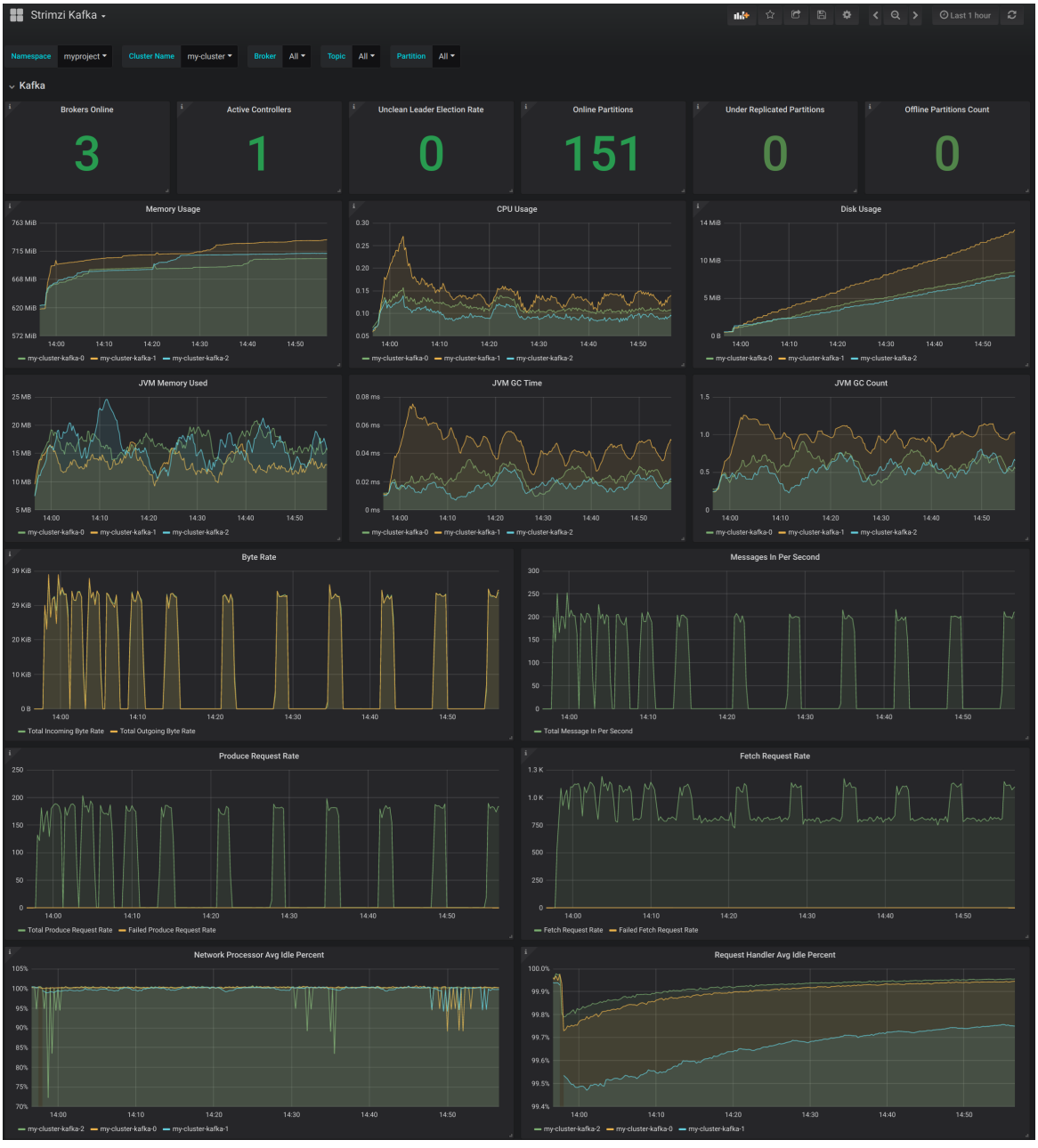
AMQ Streams Kafka

以下のメトリクスを表示します。

- オンラインのブローカーの数
- クラスター内のアクティブなコントローラーの数
- 非同期レプリカがリーダーに選択される割合
- オンラインのレプリカ
- 複製の数が最低数未満であるパーティションの数
- 最小の In-Sync レプリカ数にあるパーティション
- 最小の In-Sync レプリカ数未満のパーティション
- アクティブなリーダーを持たないため、書き込みや読み取りができないパーティション
- Kafka ブローカー Pod のメモリー使用量
- 集約された Kafka ブローカー Pod の CPU 使用率
- Kafka ブローカー Pod のディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数

- 受信バイトレートの合計
- 送信バイトレートの合計
- 受信メッセージレート
- 生成要求レートの合計
- バイトレート
- 生成要求レート
- 取得要求レート
- ネットワークプロセッサーの平均時間アイドル率
- リクエストハンドラーの平均時間アイドル率
- ログサイズ

図7.2 AMQ Streams Kafka ダッシュボード



AMQ Streams ZooKeeper

以下のメトリクスを表示します。

- ZooKeeper アンサンブルのクォーラムサイズ
- **アクティブな** 接続の数
- サーバーのキューに置かれたリクエストの数
- ウォッチャーの数
- ZooKeeper Pod のメモリー使用量
- 集約された ZooKeeper Pod の CPU 使用率
- ZooKeeper Pod のディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数
- サーバーがクライアントリクエストに応答するまでの時間 (最大、最小、および平均)

AMQ Streams の Operator

以下のメトリクスを表示します。

- カスタムリソース
- 1時間あたりの成功したカスタムリソース調整の数
- 1時間あたりの失敗したカスタムリソース調整の数
- 1時間あたりのロックなしの調整の数
- 1時間あたりの開始された調整の数
- 1時間あたりの定期的な調整の数
- 最大の調整時間
- 平均の調整時間
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数

AMQ Streams Kafka Connect

以下のメトリクスを表示します。

- 受信バイトレートの合計
- 送信バイトレートの合計

- ディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間

AMQ Streams Kafka MirrorMaker 2

以下のメトリクスを表示します。

- コネクターの数
- タスクの数
- 受信バイトレートの合計
- 送信バイトレートの合計
- ディスク使用量
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間

AMQ Streams Kafka Bridge

「Kafka Bridge の監視」 を参照してください。

AMQ Streams Cruise Control

「Cruise Control の監視」 を参照してください。

AMQ Streams Kafka Exporter

「Kafka Exporter Grafana ダッシュボードの有効化」 を参照してください。

7.5. KAFKA EXPORTER の追加

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 Exporter は、Kafka クラスターとのデプロイメントを実現するために AMQ Streams で提供され、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クス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。

ビルトイン Kafka メトリクスの監視のために Prometheus および Grafana をすでに使用している場合、Kafka Exporter Prometheus エンドポイントをスクレープするように Prometheus を設定することもできます。

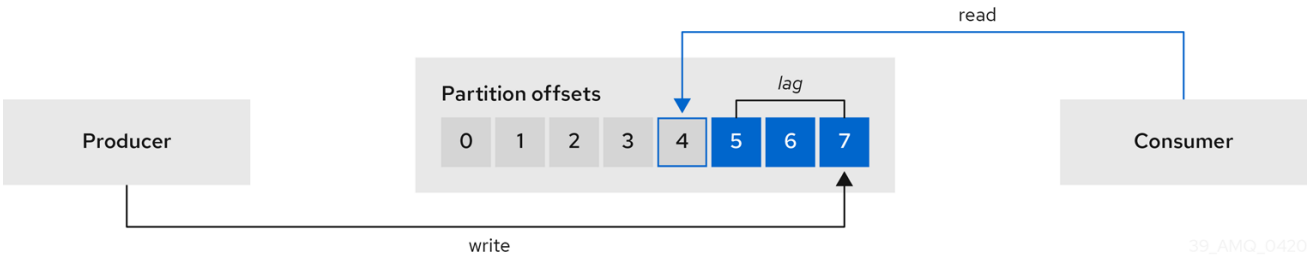
AMQ Streams には、**examples/metrics/grafana-dashboards/strimzi-kafka-exporter.json** に Kafka Exporter ダッシュボードのサンプルが含まれています。

7.5.1. コンシューマーラグの監視

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



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、リアルタイム処理の達成から遠ざかります。

たとえば、パージされていない古いデータの大量消費や、予定外のシャットダウンが、コンシューマーラグの原因となることがあります。

コンシューマーラグの削減
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ーを増強します。

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の場合、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。

7.5.2. Kafka Exporter アラートルールの例

メトリクスをデプロイメントに導入するステップが実行済みである場合、Kafka Exporter をサポートするアラート通知ルールを使用するよう Kafka クラスターがすでに設定された状態になっています。

Kafka Exporter のルールは **prometheus-rules.yaml** に定義されており、Prometheus でデプロイされます。詳細は、「Prometheus」を参照してください。

Kafka Exporter に固有のサンプルのアラート通知ルールには以下があります。

UnderReplicatedPartition

トピックで複製の数が最低数未満であり、ブローカーがパーティションで十分な複製を作成してい

ないことを警告するアラートです。デフォルトの設定では、トピックに複製の数が最低数未満のパーティションが 1 つ以上ある場合のアラートになります。このアラートは、Kafka インスタンスがダウンしているか Kafka クラスターがオーバーロードの状態であることを示す場合があります。レプリケーションプロセスを再起動するには、Kafka ブローカーの計画的な再起動が必要な場合があります。

TooLargeConsumerGroupLag

特定のトピックパーティションでコンシューマーグループのラグが大きすぎることを警告するアラートです。デフォルト設定は 1000 レコードです。ラグが大きい場合、コンシューマーが遅すぎてプロデューサーの処理に追い付いてない可能性があります。

NoMessageForTooLong

トピックが一定期間にわたりメッセージを受信していないことを警告するアラートです。この期間のデフォルト設定は 10 分です。この遅れは、設定の問題により、プロデューサーがトピックにメッセージを公開できないことが原因である可能性があります。

これらのルールのデフォルト設定は、特定のニーズに合わせて調整してください。

その他のリソース

- *7章AMQ Streams のメトリクスおよびダッシュボードの設定*
- 「メトリクスファイルの例」
- 「アラートルール」

7.5.3. Kafka Exporter メトリクスの公開

ラグ情報は、Grafana で示す Prometheus メトリクスとして Kafka Exporter によって公開されます。

Kafka Exporter は、ブローカー、トピック、およびコンシューマーグループのメトリクスデータを公開します。これらのメトリクスは、**strimzi-kafka-exporter** ダッシュボードのサンプルに表示されます。

抽出されるデータを以下に示します。

表7.2 ブローカーメトリクスの出力

名前	詳細
kafka_brokers	Kafka クラスターに含まれるブローカーの数

表7.3 トピックメトリクスの出力

名前	詳細
kafka_topic_partitions	トピックのパーティション数
kafka_topic_partition_current_offset	ブローカーの現在のトピックパーティションオフセット
kafka_topic_partition_oldest_offset	ブローカーの最も古いトピックパーティションオフセット

名前	詳細
kafka_topic_partition_in_sync_replica	トピックパーティションの In-Sync レプリカ数
kafka_topic_partition_leader	トピックパーティションのリーダーブローカー ID
kafka_topic_partition_leader_is_preferred	トピックパーティションが優先ブローカーを使用している場合は、**1** が示されます。
kafka_topic_partition_replicas	このトピックパーティションのレプリカ数
kafka_topic_partition_under_replicated_partition	トピックパーティションの複製の数が最低数未満である場合に **1** が示されます。

表7.4 コンシューマーグループメトリクスの出力

名前	詳細
kafka_consumergroup_current_offset	コンシューマーグループの現在のトピックパーティションオフセット
kafka_consumergroup_lag	トピックパーティションのコンシューマーグループの現在のラグ (概算値)

1 つ以上のコンシューマーグループにゼロよりも大きいラグがある場合、コンシューマーグループメトリクスは Kafka Exporter ダッシュボードのみに表示されます。

7.5.4. Kafka Exporter の設定

この手順では、**KafkaExporter** プロパティーから **Kafka** リソースの Kafka Exporter を設定する方法を説明します。

Kafka リソースの設定に関する詳細は、『**Using AMQ Streams on OpenShift**』の「Kafka cluster configuration」を参照してください。

この手順では、Kafka Exporter 設定に関連するプロパティーを取り上げます。

これらのプロパティーは、Kafka クラスターのデプロイメントまたは再デプロイメントの一部として設定できます。

前提条件

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

手順

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

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafkaExporter:
    image: my-registry.io/my-org/my-exporter-cluster:latest (1)
    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ーイメージの設定。

(2) メトリクス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) メトリクス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 ロギン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) Healthcheck の readiness プローブ。

9 ヘルスチェックの liveness プローブ。

2. リソースを作成または更新します。

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

次のステップ

Kafka Exporter の設定およびデプロイ後に、Grafana を有効にして Kafka Exporter ダッシュボードを表示できます。

関連情報

KafkaExporterTemplate schema reference.

7.5.5. Kafka Exporter Grafana ダッシュボードの有効化

AMQ Streams には、Grafana のダッシュボード設定ファイルのサンプル が含まれています。Kafka Exporter ダッシュボードは、JSON ファイルとして提供され、examples/metrics ディレクトリーに含まれています。

- **strimzi-kafka-exporter.json**

Kafka Exporter を Kafka クラスターでデプロイした場合、公開されるメトリクスデータを Grafana ダッシュボードで可視化できます。

前提条件

- Kafka が Kafka Exporter メトリクスの設定によってデプロイされている必要があります。
- Prometheus および Prometheus Alertmanager が Kafka クラスターにデプロイされている必要があります。
- Grafana が Kafka クラスターにデプロイされている必要があります。

この手順では、Grafana ユーザーインターフェースにアクセスでき、Prometheus がデータソースとして追加されていることを前提とします。ユーザーインターフェースに初めてアクセスする場合は、「Grafana」を参照してください。

手順

1. Grafana ユーザーインターフェースにアクセスします。
2. **Strimzi Kafka Exporter** ダッシュボードを選択します。
   メトリクスデータが収集されると、Kafka Exporter のチャートにデータが反映されます。

   AMQ Streams Kafka Exporter

   以下のメトリクスを表示します。

   - トピックの数
   - パーティションの数
   - レプリカの数

- In-Sync レプリカの数
- 複製の数が最低数未満であるパーティションの数
- 最小の In-Sync レプリカ数にあるパーティション
- 最小の In-Sync レプリカ数未満のパーティション
- 優先ノードにないパーティション
- 毎秒のトピックからのメッセージ
- 毎秒消費されるトピックからのメッセージ
- コンシューマーグループごとに毎分消費されるメッセージ
- コンシューマーグループごとのラグ
- パーティションの数
- 最新のオフセット
- 最も古いオフセット

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ごとのラグ** 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。

7.6. KAFKA BRIDGE の監視

ビルトイン Kafka メトリクスの監視のために Prometheus および Grafana をすでに使用している場合、Kafka Bridge Prometheus エンドポイントをスクレープするように Prometheus を設定することもできます。

Kafka Bridge の Grafana ダッシュボードのサンプルは以下を提供します。

- さまざまなエンドポイントへの HTTP 接続および関連リクエストに関する情報
- ブリッジによって使用される Kafka コンシューマーおよびプロデューサーに関する情報
- ブリッジ自体からの JVM メトリクス

7.6.1. Kafka Bridge の設定

KafkaBridge リソースの **enableMetrics** プロパティーを使用して、Kafka Bridge メトリクスを有効にできます。

このプロパティーは、Kafka Bridge のデプロイメントまたは再デプロイメントの一部として設定できます。

以下に例を示します。

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

```
metadata:
  name: my-bridge
spec:
  # ...
  bootstrapServers: my-cluster-kafka:9092
  http:
    # ...
  enableMetrics: true
  # ...
```

7.6.2. Kafka Bridge Grafana ダッシュボードの有効化

Kafka Bridge を Kafka クラスターでデプロイした場合、Grafana により公開されるメトリクスデータを表示するように Grafana を有効化できます。

Kafka Bridge ダッシュボードは、JSON ファイルとして提供され、examples/metrics ディレクトリーに含まれています。

- **strimzi-kafka-bridge.json**

メトリクスデータが収集されると、Kafka Bridge のチャートにデータが反映されます。

Kafka Bridge

以下のメトリクスを表示します。

- Kafka Bridge への HTTP 接続の数
- 処理中の HTTP リクエストの数
- HTTP メソッドごとに 1 秒あたり処理されるリクエスト
- レスポンスコード (2XX、4XX、5XX) ごとの要求レートの合計
- 1 秒あたりの受信および送信バイト数
- Kafka Bridge エンドポイントごとのリクエスト
- Kafka Bridge 自体によって使用される Kafka コンシューマー、プロデューサー、および関連するオープン接続の数
- Kafka プロデューサー:
  - 1 秒あたり送信される平均のレコード数 (トピックごとにグループ化)
  - 1 秒あたりすべてのブローカーに送信される発信バイト数 (トピックごとにグループ化)
  - エラーとなったレコードの 1 秒あたりの平均数 (トピックごとにグループ化)
- Kafka コンシューマー:
  - 1 秒あたり消費される平均のレコード数 (clientId-topic ごとにグループ化)
  - 1 秒あたり消費される平均のバイト数 (clientId-topic ごとにグループ化)
  - 割り当てられるパーティション (clientId ごとにグループ化)
- 使用されている JVM メモリー

- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数

7.7. CRUISE CONTROL の監視

ビルトイン Kafka メトリクスの監視のために Prometheus および Grafana をすでに使用している場合、Cruise Control Prometheus エンドポイントをスクレープするように Prometheus を設定することもできます。

Cruise Control の Grafana ダッシュボードのサンプルは以下を提供します。

- 最適化プロポーザルの計算、ゴールの逸脱、クラスターのバランス状況などに関する情報
- リバランスプロポーザルおよび実際のリバランス操作の REST API コールに関する情報
- Cruise Control 自体からの JVM メトリクス

7.7.1. Cruise Control の設定

Kafka リソースの **cruiseControl.metricsConfig** プロパティーを使用して Cruise Control メトリクスを有効にし、公開するメトリクスの JMX エクスポーター設定が含まれる ConfigMap への参照を提供します。

以下は例になります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  # ...
  kafka:
    # ...
  zookeeper:
    # ...
  cruiseControl:
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: my-key
```

7.7.2. Cruise Control Grafana ダッシュボードの有効化

メトリクスを有効にして Cruise Control を Kafka クラスターでデプロイした場合、Grafana により公開されるメトリクスデータを表示するように Grafana を有効化できます。

Cruise Control ダッシュボードは、JSON ファイルとして提供され、examples/metrics ディレクトリーに含まれています。

- **strimzi-cruise-control.json**

メトリクスデータが収集されると、Cruise Control のチャートにデータが反映されます。

Cruise Control

以下のメトリクスを表示します。

- Cruise Control によって監視されるスナップショットウィンドウの数
- 最適化プロポーザルを計算するのに十分なサンプルが含まれるため、有効とみなされる時間枠の数
- プロポーザルまたはリバランスのために実施中の実行の数
- Cruise Control の異常検出コンポーネントによって計算された (デフォルトでは 5 分ごと) Kafka クラスターの現在のバランス状態スコア
- 監視されるパーティションの割合
- 異常検出によって報告された (デフォルトでは 5 分ごと) ゴール逸脱の数
- ブローカーでディスクの読み取り障害が発生する頻度
- メトリクスサンプルの取得に失敗する割合
- 最適化プロポーザルの計算に必要な時間
- クラスターモデルの作成に必要な時間
- Cruise Control の REST API 経由でプロポーザルリクエストまたは実際のリバランスリクエストが実行される頻度
- Cruise Control の REST API 経由でクラスター全体の状態およびユーザータスクの状態が要求される頻度
- 使用されている JVM メモリー
- JVM ガベージコレクションの時間
- JVM ガベージコレクションの数

第8章 AMQ STREAMS のアップグレード

AMQ Streams をバージョン 1.8 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。

アップグレードの一部として、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す。

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 バージョンの一覧は、Red Hat カスタマーポータルの「製品ダウンロード」で確認できます。

アップグレードパス

2 つのアップグレードパスが可能です。

インクリメント

AMQ Streams を以前のマイナーバージョンからバージョン 1.8 にアップグレードします。

マルチバージョン

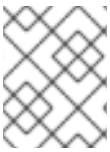
AMQ Streams を 1 回で古いバージョンからバージョン 1.8 にアップグレードします (1 つ以上の中間バージョンを飛ばします)。
たとえば、AMQ Streams 1.5 から直接 AMQ Streams 1.7 にアップグレードします。

1.7 よりも古いバージョンからのアップグレード

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 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 にアップグレードします。
2. カスタムリソースを **v1beta2** に変換します。
3. AMQ Streams の 1.8 以降へのアップグレード



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ができます。

Kafka バージョンのサポート

Kafka バージョン の表には、AMQ Streams 1.8 でサポートされる Kafka バージョンが記載されています。この表では以下に注意してください。

- **最新**の Kafka バージョンは実稼働でサポートされます。
- **最新バージョンより前の** Kafka バージョンは、AMQ Streams 1.8 へのアップグレードの目的でのみサポートされます。

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。

注記

ご使用のバージョンの AMQ Streams によってサポートされれば、上位バージョンの Kafka にアップグレードできます。サポートされる下位バージョンの Kafka にダウングレードできる場合もあります。

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アップデートがトリガーされ、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。ローリングアップデート中に、すべてのブローカーがオンライン状態ではないため、**クラスター全体の可用性** は一時的に低下します。クラスターの可用性が低下すると、ブローカーの障害によってメッセージが失われる可能性が高くなります。

8.1. 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 Strimzi アップグレード手順を **必ず** 完了してください。

1. **v1beta2** API バージョンをサポートするように既存のカスタムリソースを更新します。
   - 「AMQ Streams カスタムリソースのアップグレード」

   AMQ Streams 1.7 にアップグレードした後、AMQ Streams 1.8 以降にアップグレードする前にこれを行います。バージョン 1.7 よりも前のバージョンからのマルチバージョンアップグレードの場合:

   a. この手順を省略し、以下の手順にしたがってバージョン 1.7 にアップグレードします。

   b. この手順に戻り、このアップグレードシーケンスのすべての手順を実行して 1.8 以降にアップグレードします。

2. Cluster Operator を新しい AMQ Streams バージョンに更新します。
   - 「Cluster Operator のアップグレード」

   実施する手法は、Cluster Operator のデプロイ 方法によって異なります。

   - 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、「Upgrading the Cluster Operator」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 から Cluster Operator をデプロイした場合は、Operator Lifecycle Manager (OLM) を使用して AMQ Streams Operator の更新チャネルを新しい AMQ Streams バージョンに変更します。
     選択したアップグレードストラテジーに応じて、チャネルの更新後に以下のいずれかを実行します。
     - 自動アップグレードが開始されます。

- 手動アップグレードでは、インストールを開始する前に承認が必要です。
  OperatorHub を使用した Operator のアップグレードについての詳細は、OpenShift ドキュメントの「Upgrading installed Operators」を参照してください。

3. 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、サポートされる最新の Kafka バージョンにアップグレードします。

   - 「Kafka のアップグレード」
   - 「クライアントをアップグレードするストラテジー」

任意手順: Incremental Cooperative Rebalance のアップグレード

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 アプリケーションのアップグレードを検討します。

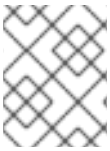
- 「コンシューマーの Cooperative Rebalancing へのアップグレード」

8.2. AMQ STREAMS カスタムリソースのアップグレード

AMQ Streams を 1.8 にアップグレードする前に、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**。これは、AMQ Streams 1.7 にアップグレードした後にいつでも実行できますが、AMQ Streams 1.8 以降にアップグレードする前にアップグレードを完了する必要があります。

重要

カスタムリソースを **v1beta2** にアップグレードすると、Cluster Operator をアップグレードする前に、Cluster Operator がリソースを認識できるように **する必要があります**。



注記

カスタムリソースを **v1beta2** にアップグレードすると、OpenShift v1.22 に必要な OpenShift CRD **v1** に移行するために AMQ Streams が準備されます。

カスタムリソースへの CLI のアップグレード

AMQ Streams では、**API 変換ツール** とそのリリースアーティファクトが提供されます。

AMQ Streams のダウンロードサイト から ZIP または TAR.GZ をダウンロードできます。ツールを使用するには、そのツールを展開して、bin **ディレクトリーで** スクリプトを使用します。

その CLI からツールを使用して、カスタムリソースの形式を 2 つの方法のいずれかで **v1beta2** に変換できます。

- 「API 変換ツールを使用したカスタムリソース設定ファイルの変換」
- 「API 変換ツールを使用したカスタムリソースの直接変換」

カスタムリソースの変換後に、v **1beta2** を CRD **のストレージ** API バージョンとして設定する必要があります。

- 「API 変換ツールを使用した CRD の v1beta2 へのアップグレード」

カスタムリソースへの手動アップグレード

API 変換ツールを使用してカスタムリソースを **v1beta2 に更新する代わりに、v1beta2** を使用するように各カスタムリソースを手動で更新できます。

他のコンポーネントの設定を含む Kafka カスタムリソースを更新します。

- 「v1beta2 をサポートするように Kafka リソースをアップグレード」
- 「v1beta2 をサポートするように ZooKeeper をアップグレード」
- 「v1beta2 をサポートするように Topic Operator をアップグレード」
- 「v1beta2 をサポートするように Entity Operator をアップグレード」
- 「v1beta2 をサポートするように Cruise Control をアップグレード」 **(Cruise Control がデプロイされている場合)**
- 「Kafka リソースの API バージョンを v1beta2 にアップグレード」

デプロイメントに適用される他のカスタムリソースを更新します。

- 「Kafka Connect リソースの v1beta2 へのアップグレード」
- 「Kafka Connect S2I リソースの v1beta2 へのアップグレード」
- 「Kafka MirrorMaker リソースの v1beta2 へのアップグレード」
- 「Kafka MirrorMaker 2.0 リソースの v1beta2 へのアップグレード」
- 「Kafka Bridge リソースの v1beta2 へのアップグレード」

- 「Kafka User リソースの v1beta2 へのアップグレード」
- 「Kafka Topic リソースの v1beta2 へのアップグレード」
- 「Kafka Connector リソースの v1beta2 へのアップグレード」
- 「Kafka Rebalance リソースの v1beta2 へのアップグレード」

手動の手順では、各カスタムリソースに加えた変更が示されます。これらの変更後に、API 変換ツールを使用して CRD をアップグレードする必要があります。

8.2.1. API のバージョン管理

カスタムリソースは、CRD によって OpenShift に追加された API を使用して編集および制御されます。言い換えると、CRD は Kubernetes API を拡張して、カスタムリソースを作成できるようにします。CRD 自体は OpenShift 内のリソースです。これらの CRD は、OpenShift クラスターにインストールされ、カスタムリソースの API のバージョンを定義します。カスタムリソース API の各バージョンで、そのバージョンの独自のスキーマを定義できます。AMQ Streams Operator を含む OpenShift クライアントは、API バージョンが含まれる URL パス (API パス) を使用して Kubernetes API サーバーによって提供されるカスタムリソースにアクセスします。

v1beta2 の導入により、カスタムリソースのスキーマが更新されています。v1alpha1 および v1beta1 のバージョンが削除されました。

v1alpha1 API バージョンは、以下の AMQ Streams カスタムリソースには使用されなくなりました。

- Kafka
- KafkaConnect
- KafkaConnectS2I

- KafkaConnector
- KafkaMirrorMaker
- KafkaMirrorMaker2
- KafkaTopic
- KafkaUser
- KafkaBridge
- KafkaRebalance

v1beta1 API バージョンは、以下の AMQ Streams カスタムリソースには使用されなくなりました。

- Kafka
- KafkaConnect
- KafkaConnectS2I
- KafkaMirrorMaker
- KafkaTopic
- KafkaUser

関連情報

- 「Extend the Kubernetes API with CustomResourceDefinitions」

8.2.2. API 変換ツールを使用したカスタムリソース設定ファイルの変換

この手順では、API 変換ツールを使用して、AMQ Streams カスタムリソースの設定を記述する YAML ファイルを v1beta2 に適用可能な形式に変換する方法を説明します。これを行うには、convert-file(cf)コマンドを使用します。

convert-file コマンドは、複数のドキュメントが含まれる YAML ファイルを変換できます。マルチドキュメントの YAML ファイルでは、含まれる AMQ Streams カスタムリソースがすべて変換されます。AMQ Streams 以外の OpenShift リソースは、変換された出力ファイルに未変更の状態でレプリケートされます。

YAML ファイルの変換後、設定を適用してクラスターのカスタムリソースを更新する必要があります。または、GitOps 同期メカニズムがクラスターの更新に使用される場合は、これを使用して変更を適用できます。変換は、カスタムリソースが OpenShift クラスターで更新される場合にのみ完了します。

または、convert-resource の手順を使用して、カスタムリソースを直接変換することもできます。

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- リリースアーティファクトで提供される API 変換ツールが必要です。
- このツールには Java 11 が必要です。

API 変換ツールや convert-file コマンドで利用可能なフラグの詳細は、CLI help を使用します。

```
bin/api-conversion.sh help
bin/api-conversion.sh help convert-file
```

Windows を使用している場合は、この手順に bin/api-conversion.cmd を使用します。

表8.1 YAML ファイル変換のフラグ

フラグ	説明
-f, --file=NAME-OF-YAML-FILE	変換される AMQ Streams カスタムリソースの YAML ファイルを指定します。
-o, --output=NAME-OF-CONVERTED-YAML-FILE	変換されたカスタムリソースの出力 YAML ファイルを作成します。
--in-place	変換された YAML で元のソースファイルを更新します。

手順

1. convert-file コマンドおよび適切なフラグを指定して、API 変換ツールを実行します。

   例 1: YAML ファイルを変換し、出力を表示しますが、ファイルは変更されません。

   ```
   bin/api-conversion.sh convert-file --file input.yaml
   ```

   例 2: YAML ファイルを変換し、変更を元のソースファイルに書き込みます。

   ```
   bin/api-conversion.sh convert-file --file input.yaml --in-place
   ```

   例 3: YAML ファイルを変換し、変更を新しい出力ファイルに書き込みます。

   ```
   bin/api-conversion.sh convert-file --file input.yaml --output output.yaml
   ```

2. 変換した設定ファイルを使用して、カスタムリソースを更新します。

   ```
   oc apply -f CONVERTED-CONFIG-FILE
   ```

3. カスタムリソースが変換されたことを確認します。

   ```
   oc get KIND CUSTOM-RESOURCE-NAME -o yaml
   ```

8.2.3. API 変換ツールを使用したカスタムリソースの直接変換

この手順では、API 変換ツールを使用して、OpenShift クラスターの AMQ Streams カスタムリソースを直接 v1beta2 に適用可能な形式に変換する方法を説明します。これを行うには、convert-resource(cr)コマンドを使用します。このコマンドは Kubernetes API を使用して変換を行います。

kind プロパティーを基にして、AMQ Streams カスタムリソースのタイプを 1 つ以上指定するか、すべてのタイプを変換できます。特定の namespace またはすべての namespace を変換の対象にすることもできます。1 つの namespace を対象にする場合、その namespace のすべてのカスタムリソースを変換でき、その名前と種類を指定すると単一のカスタムリソースを変換できます。

または、convert-file の手順を使用して、カスタムリソースを記述する YAML ファイルを変換および適用できます。

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- リリースアーティファクトで提供される API 変換ツールが必要です。
- このツールには Java 11 (OpenJDK) が必要です。
- 手順では、以下を行うために RBAC の権限を持つユーザー管理者アカウントが必要です。
  - --name オプションを使用して、変換される AMQ Streams カスタムリソースを取得します。
  - --name オプションを使用せずに変換される AMQ Streams カスタムリソースを一覧表示します。
  - 変換される AMQ Streams カスタムリソースの置き換え。

API 変換ツールや convert-resource コマンドで利用可能なフラグの詳細は、CLI help を使用します。

```
bin/api-conversion.sh help
bin/api-conversion.sh help convert-resource
```

Windows を使用している場合は、この手順に bin/api-conversion.cmd を使用します。

表8.2 カスタムリソースを変換するためのフラグ

フラグ	説明
-k, **--kind**	変換するカスタムリソースの種類を指定します。指定されていない場合はすべてのリソースを変換します。
-a, **--all-namespaces**	すべての namespace でカスタムリソースを変換します。
-n, **--namespace**	OpenShift namespace または OpenShift プロジェクトを指定します。指定されていない場合は現在の namespace を使用します。
--name	**--namespace** と単一のカスタムリソース **--kind** が使用されている場合は、変換されるカスタムリソースの名前を指定します。

手順

1. convert-resource コマンドおよび適切なフラグを指定して、API 変換ツールを実行します。

   例 1: 現在の namespace のすべての AMQ Streams リソースを変換します。

   ```
   bin/api-conversion.sh convert-resource
   ```

   例 2: すべての namespace のすべての AMQ Streams リソースを変換します。

   ```
   bin/api-conversion.sh convert-resource --all-namespaces
   ```

   例 3: my-kafka namespace のすべての AMQ Streams リソースを変換します。

   ```
   bin/api-conversion.sh convert-resource --namespace my-kafka
   ```

例 4: すべての namespace の Kafka リソースのみを変換します。

```
bin/api-conversion.sh convert-resource --all-namespaces --kind Kafka
```

例 5: すべての namespace の Kafka および Kafka Connect リソースを変換します。

```
bin/api-conversion.sh convert-resource --all-namespaces --kind Kafka --kind
KafkaConnect
```

例 6: my- kafka namespace の my-cluster という名前の Kafka カスタムリソースを変換します。

```
bin/api-conversion.sh convert-resource --kind Kafka --namespace my-kafka --name
my-cluster
```

2. カスタムリソースが変換されたことを確認します。

```
oc get KIND CUSTOM-RESOURCE-NAME -o yaml
```

8.2.4. API 変換ツールを使用した CRD の v1beta2 へのアップグレード

この手順では、API 変換ツールを使用して、v 1beta2 に適用可能な形式で AMQ Streams 固有のリソースをインスタンス化および管理するために使用されるスキーマを定義する CRD を変換する方法を説明します。これを実行するには、crd -upgrade コマンドを使用します。

OpenShift クラスター全体のすべての AMQ Streams カスタムリソースを v1beta2 に変換した後に、この手順を実行します。CRD を最初にアップグレードしてからカスタムリソースを変換する場合は、このコマンドを再度実行する必要があります。

このコマンドは CRD の spec.versions を更新し、v 1beta2 を ストレージ API バージョンとして宣言します。このコマンドは、カスタムリソースも更新し、それらのカスタムリソースは v1beta2 に保存されるようにします。新しいカスタムリソースインスタンスはストレージ API バージョンの仕様から作成されるため、1 つの API バージョンのみがストレージバージョンとしてマークされます。

v1beta2 をストレージバージョンとして使用するように CRD をアップグレードした場合は、カスタムリソースで v1beta2 プロパティーのみを使用する必要があります。

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- リリースアーティファクトで提供される API 変換ツールが必要です。
- このツールには Java 11 (OpenJDK) が必要です。
- カスタムリソースが v1beta2 に変換されています。
- 手順では、以下を行うために RBAC の権限を持つユーザー管理者アカウントが必要です。
  - すべての namespace の AMQ Streams カスタムリソースのリスト。
  - 変換される AMQ Streams カスタムリソースの置き換え。
  - CRD の更新。
  - CRD の状態の置き換え。

API 変換ツールの詳細は、CLI で help を使用します。

```
bin/api-conversion.sh help
```

Windows を使用している場合は、この手順に bin/api-conversion.cmd を使用します。

手順

1. これを実行していない場合は、カスタムリソースを v1beta2 を使用するように変換します。

   これは、API 変換ツールを使用して、以下のいずれかの方法で行います。

- 「API 変換ツールを使用したカスタムリソース設定ファイルの変換」
- 「API 変換ツールを使用したカスタムリソースの直接変換」

また、手動で変更することもできます。

2. crd-upgrade コマンドを使用して、API 変換ツールを実行します。

```
bin/api-conversion.sh crd-upgrade
```

3. CRD がアップグレードされ、v1beta2 がストレージバージョンであることを確認します。

たとえば、Kafka トピック CRD の場合は次のとおりです。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: CustomResourceDefinition
metadata:
  name: kafkatopics.kafka.strimzi.io
  #...
spec:
  group: kafka.strimzi.io
  #...
  versions:
  - name: v1beta2
    served: true
    storage: true
    #...
status:
  #...
  storedVersions:
  - v1beta2
```

8.2.5. v1beta2 をサポートするように Kafka リソースをアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。

手順

デプロイメントの Kafka カスタムリソースごとに以下の手順を実行します。

1. エディターで Kafka カスタムリソースを更新します。

```
oc edit kafka KAFKA-CLUSTER
```

2. 「リスナーの汎用リスナー設定への更新」 で説明されているように、新しい汎用リスナー形式に update .spec.kafka.listener をまだ更新していない場合は、に記載のように、新しい汎用リスナー形式に update.spec.kafka.listener を更新します。

警告

古いリスナー形式は API バージョン v1beta2 ではサポートされません。

3. 存在する場合は、アフィニティー を .spec.kafka.affinity から .spec.kafka.template.pod.affinity に移動します。

4. 容認 がある場合は、容認を .spec.kafka.tolerations から .spec.kafka.template.pod.tolerations に移動します。

5. If present, remove .spec.kafka.template.tlsSidecarContainer.

6. If present, remove .spec.kafka.tlsSidecarContainer.

7. 以下のポリシー設定のいずれかが存在する場合:

- .spec.kafka.template.externalBootstrapService.externalTrafficPolicy

- .spec.kafka.template.perPodService.externalTrafficPolicy

a. type: loadbalancer および type: nodeport リスナーの両方で、設定を .spec.kafka.listeners[].configuration.externalTrafficPolicy に移動します。

b. remove .spec.kafka.template.externalBootstrapService.externalTrafficPolicy または .spec.kafka.template.perPodService.externalTrafficPolicy.

8. 以下のロードバランサーリスナー 設定のいずれか が存在する場合：

- .spec.kafka.template.externalBootstrapService.loadBalancerSourceRanges
- .spec.kafka.template.perPodService.loadBalancerSourceRanges

a. type: loadbalancer リスナーの設定を .spec.kafka.listeners[].configuration.loadBalancerSourceRanges に移動します。

b. remove .spec.kafka.template.externalBootstrapService.loadBalancerSourceRanges または .spec.kafka.template.perPodService.loadBalancerSourceRanges

9. type: external ロギングが .spec.kafka.logging に設定されている場合:

ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名と キー の両方を指定します。

```
logging:
  type: external
  valueFrom:
```

```
    configMapKeyRef:
      name: my-config-map
      key: log4j.properties
```

10. .spec.kafka.metrics フィールドを使用してメトリクスを有効にする場合：

    a. JMX Prometheus エクスポーターの YAML 設定をキーの下に保存する新しい ConfigMap を作成します。YAML は、現在 .spec.kafka.metrics フィールドの内容と一致している必要があります。

    ```
    kind: ConfigMap
    apiVersion: v1
    metadata:
      name: kafka-metrics
      labels:
        app: strimzi
    data:
      kafka-metrics-config.yaml: |
        <YAML>
    ```

    b. ConfigMap およびキーを参照する .spec.kafka.metricsConfig プロパティーを追加します。

    ```
    metricsConfig:
      type: jmxPrometheusExporter
      valueFrom:
        configMapKeyRef:
          name: kafka-metrics
          key: kafka-metrics-config.yaml
    ```

    c. old .spec.kafka.metrics フィールドを削除します。

11. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

次のステップ

各 Kafka カスタムリソースについて、ZooKeeper、Topic Operator、Entity Operator、および Cruise Control（デプロイされている場合）の設定をアップグレードして、バージョン v1beta2 をサポートします。これは以下の手順で説明します。

すべての Kafka 設定が更新され v1beta2 をサポートする場合は、Kafka カスタムリソースを v1beta2にアップグレードできます。

8.2.6. リスナーの汎用リスナー設定への更新

AMQ Streams では、Kafka リソースの Kafka リスナーを設定するための Generic KafkaListener スキーマが提供されます。

GenericKafkaListener は、AMQ Streams から削除された KafkaListeners スキーマに置き換わりま した。

GenericKafkaListener スキーマ を使用すると、名前とポートが一意であれば、必要なリスナーをいくつでも設定できます。リスナー設定 は配列として定義されますが、非推奨の形式もサポートされます。

OpenShift クラスター内のクライアントの場合は、プレーン （暗号化なし）または tls 内部 リスナーを作成できます。

OpenShift クラスター外のクライアントの場合は、外部リスナー を作成し、nodeport、loadbalancer、ingress、または route などの接続メカニズムを指定します。

KafkaListeners スキーマは、plain、tlsおよび external リスナーのサブプロパティーを使用し、それぞれに固定ポートを使用していました。アップグレードプロセスの段階で、KafkaListeners スキーマを使用して設定されたリスナーを GenericKafkaListener スキーマの形式に変換する必要があります。

たとえば、現在 Kafka 設定で以下の設定を使用しているとします。

これまでのリスナー設定

```
listeners:
  plain:
    # ...
  tls:
    # ...
  external:
    type: loadbalancer
    # ...
```

以下を使用して、リスナーを新しい形式に変換します。

新しいリスナー設定

```
listeners:
  #...
  - name: plain
    port: 9092
    type: internal
    tls: false (1)
  - name: tls
    port: 9093
    type: internal
    tls: true
  - name: external
    port: 9094
    type: EXTERNAL-LISTENER-TYPE (2)
    tls: true
```

(1) すべてのリスナーに TLS プロパティーが必要になります。

(2) オプション： ingress、loadbalancer、nodeport、route。

必ず 正確な 名前とポート番号を使用してください。

古い形式で 使用される追加の 設定または オーバーライド プロパティーについては、新しい形式に更新する必要があります。

リスナー設定 に導入された変更：

- overrides は configuration セクションとマージされます。
- dnsAnnotations の名前が アノテーションになりました。
- preferredAddressType の名前が preferredNodePortAddressType
- address の名前が alternativeNames
- loadBalancerSourceRanges および externalTrafficPolicy が、現在の非推奨の テンプレートからリスナー設定に移動します。

例として、以下の設定を見てみましょう。

従来の追加リスナー設定

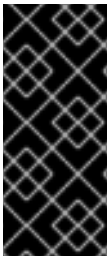
```
listeners:
  external:
    type: loadbalancer
    authentication:
      type: tls
    overrides:
      bootstrap:
        dnsAnnotations:
          #...
```

これを以下に変更します。

新しい追加リスナー設定

```
listeners:
    #...
  - name: external
    port: 9094
```

```
type:loadbalancer
tls: true
authentication:
  type: tls
configuration:
  bootstrap:
    annotations:
      #...
```



重要

後方互換性を維持するため、新しいリスナー設定にある名前およびポート番号を使用する 必要 があります。他の値を使用すると、Kafka リスナーおよび OpenShift サービスの名前が変更されます。

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

8.2.7. v1beta2 をサポートするように ZooKeeper をアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。

手順

デプロイメントの Kafka カスタムリソースごとに以下の手順を実行します。

1. エディターで Kafka カスタムリソースを更新します。

```
oc edit kafka KAFKA-CLUSTER
```

2. 存在する場合は、アフィニティー を .spec.zookeeper.affinity から .spec.zookeeper.template.pod.affinity に移動します。

3. 容認 がある場合は、容認を .spec.zookeeper.tolerations から

.spec.zookeeper.template.pod.tolerations に移動します。

4. If present, remove .spec.zookeeper.template.tlsSidecarContainer.

5. If present, remove .spec.zookeeper.tlsSidecarContainer.

6. type: external ロギングが .spec.kafka.logging に設定されている場合:

   ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

   valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名とキー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j.properties
```

7. .spec.zookeeper.metrics フィールドを使用してメトリクスを有効にする場合：

   a. JMX Prometheus エクスポーターの YAML 設定をキーの下に保存する新しい ConfigMap を作成します。YAML は、現在 .spec.zookeeper.metrics フィールドの内容と一致する必要があります。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kafka-metrics
  labels:
    app: strimzi
data:
  zookeeper-metrics-config.yaml: |
    <YAML>
```

   b.

ConfigMap およびキーを参照する .spec.zookeeper.metricsConfig プロパティーを追加します。

```
metricsConfig:
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: kafka-metrics
      key: zookeeper-metrics-config.yaml
```

c. old .spec.zookeeper.metrics フィールドを削除します。

8. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.8. v1beta2 をサポートするように Topic Operator をアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。

手順

デプロイメントの Kafka カスタムリソースごとに以下の手順を実行します。

1. エディターで Kafka カスタムリソースを更新します。

```
oc edit kafka KAFKA-CLUSTER
```

2. Kafka.spec.topicOperator が使用されている場合は、以下を行います。

a. アフィニティーを .spec.topicOperator.affinity から .spec.entityOperator.template.pod.affinity に移動します。

b. 容認を .spec.topicOperator.tolerations から .spec.entityOperator.template.pod.tolerations に移動します。

c.
Move .spec.topicOperator.tlsSidecar to .spec.entityOperator.tlsSidecar.

d.
アフィニティー、Toleration、および tlsSidecar の移動後に、残りの設定を .spec.topicOperator の .spec.entityOperator.topicOperator に移動します。

3.
type: external ロギングが .spec.topicOperator.logging に設定されている場合:

ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名と キー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j2.properties
```

注記

このステップは、Entity Operator のアップグレード の一部として完了することもできます。

4.
ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.9. v1beta2 をサポートするように Entity Operator をアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。

-

「v1beta2 をサポートするように Topic Operator をアップグレード」 の説明に従って、Kafka.spec.entityOperator が設定されている。

手順

デプロイメントの Kafka カスタムリソースごとに以下の手順を実行します。

1. エディターで Kafka カスタムリソースを更新します。

```
oc edit kafka KAFKA-CLUSTER
```

2. アフィニティーを .spec.entityOperator.affinity から .spec.entityOperator.template.pod.affinity に移動します。

3. 容認を .spec.entityOperator.tolerations から .spec.entityOperator.template.pod.tolerations に移動します。

4. type: external ロギングが .spec.entityOperator.userOperator.logging または. spec.entityOperator.topicOperator.logging で設定されている場合：

ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名とキー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j2.properties
```

5. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.10. v1beta2 をサポートするように Cruise Control をアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- Cruise Control が設定およびデプロイされている必要があります。『Using AMQ Streams on OpenShift』の「Deploying Cruise Control」を参照してください。

手順

Kafka クラスターの Kafka.spec.cruiseControl 設定ごとに以下の手順を実行します。

1. エディターで Kafka カスタムリソースを更新します。

```
oc edit kafka KAFKA-CLUSTER
```

2. type: external ロギングが .spec.cruiseControl.logging で設定されている場合：

   ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

   valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名と キー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j2.properties
```

3. .spec.cruiseControl.metrics フィールドを使用してメトリクスを有効にする場合：

   a. JMX Prometheus エクスポーターの YAML 設定をキーの下に保存する新しい

ConfigMap を作成します。YAML は、現在 .spec.cruiseControl.metrics フィールドの内容と一致する必要があります。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kafka-metrics
  labels:
    app: strimzi
data:
  cruise-control-metrics-config.yaml: |
    <YAML>
```

b. ConfigMap およびキーを示す .spec.cruiseControl.metricsConfig プロパティーを追加します。

```
metricsConfig:
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: kafka-metrics
      key: cruise-control-metrics-config.yaml
```

c. old .spec.cruiseControl.metrics フィールドを削除します。

4. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.11. Kafka リソースの API バージョンを v1beta2 にアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- Kafka カスタムリソース内で以下の設定が更新済みである必要があります。
  - ZooKeeper
  - Topic Operator

- Entitiy Operator
- Cruise Control (Cruise Control がデプロイされている場合)

手順

デプロイメントの Kafka カスタムリソースごとに以下の手順を実行します。

1. エディターで Kafka カスタムリソースを更新します。

```
oc edit kafka KAFKA-CLUSTER
```

2. Kafka カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

3. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.12. Kafka Connect リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。

手順

デプロイメントの KafkaConnect カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaConnect カスタムリソースを更新します。

```
oc edit kafkaconnect KAFKA-CONNECT-CLUSTER
```

2. 以下があるか確認します。

```
KafkaConnect.spec.affinity
```

```
KafkaConnect.spec.tolerations
```

これを以下に変更します。

```
KafkaConnect.spec.template.pod.affinity
```

```
KafkaConnect.spec.template.pod.tolerations
```

たとえば、以下の場合を考えてみましょう。

```
spec:
 # ...
 affinity:
  # ...
 tolerations:
  # ...
```

以下のように変更します。

```
spec:
 # ...
 template:
  pod:
   affinity:
    # ...
   tolerations:
    # ...
```

3. type: external ロギングが .spec.logging に設定されている場合:

ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名とキー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j.properties
```

4. .spec.metrics フィールドを使用してメトリクスを有効にする場合：

   a. JMX Prometheus エクスポーターの YAML 設定をキーの下に保存する新しい ConfigMap を作成します。YAML は、現在 .spec.metrics フィールドの内容と一致している必要があります。

   ```
   kind: ConfigMap
   apiVersion: v1
   metadata:
     name: kafka-connect-metrics
     labels:
       app: strimzi
   data:
     connect-metrics-config.yaml: |
       <YAML>
   ```

   b. ConfigMap およびキーを参照する .spec.metricsConfig プロパティーを追加します。

   ```
   metricsConfig:
     type: jmxPrometheusExporter
     valueFrom:
       configMapKeyRef:
         name: kafka-connect-metrics
         key: connect-metrics-config.yaml
   ```

   c. old .spec.metrics フィールドを削除します。

5. KafkaConnect カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

6. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.13. Kafka Connect S2I リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。

手順

デプロイメントの KafkaConnectS2I カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaConnectS2I カスタムリソースを更新します。

```
oc edit kafkaconnects2i S2I-CLUSTER
```

2. 以下があるか確認します。

```
KafkaConnectS2I.spec.affinity
```

```
KafkaConnectS2I.spec.tolerations
```

これを以下に変更します。

```
KafkaConnectS2I.spec.template.pod.affinity
```

```
KafkaConnectS2I.spec.template.pod.tolerations
```

- たとえば、以下の場合を考えてみましょう。

```
spec:
  # ...
  affinity:
    # ...
  tolerations:
    # ...
```

以下のように変更します。

```
spec:
  # ...
  template:
    pod:
      affinity:
        # ...
      tolerations:
        # ...
```

3. type: external ロギングが .spec.logging に設定されている場合:

ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名とキー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j.properties
```

4. .spec.metrics フィールドを使用してメトリクスを有効にする場合：

a. JMX Prometheus エクスポーターの YAML 設定をキーの下に保存する新しい

ConfigMap を作成します。YAML は、現在 .spec.metrics フィールドの内容と一致している必要があります。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kafka-connect-s2i-metrics
  labels:
    app: strimzi
data:
  connect-s2i-metrics-config.yaml: |
    <YAML>
```

b. ConfigMap およびキーを参照する .spec.metricsConfig プロパティーを追加します。

```
metricsConfig:
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: kafka-connect-s2i-metrics
      key: connect-s2i-metrics-config.yaml
```

c. old .spec.metrics フィールドを削除します。

5. KafkaConnectS2I カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

6. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.14. Kafka MirrorMaker リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。

- MirrorMaker が設定され、デプロイされます。「Kafka MirrorMaker の OpenShift クラスターへのデプロイ」 を参照してください。

手順

デプロイメントの KafkaMirrorMaker カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaMirrorMaker カスタムリソースを更新します。

```
oc edit kafkamirrormaker MIRROR-MAKER
```

2. 以下があるか確認します。

```
KafkaMirrorMaker.spec.affinity
```

```
KafkaMirrorMaker.spec.tolerations
```

これを以下に変更します。

```
KafkaMirrorMaker.spec.template.pod.affinity
```

```
KafkaMirrorMaker.spec.template.pod.tolerations
```

たとえば、以下の場合を考えてみましょう。

```
spec:
  # ...
  affinity:
    # ...
  tolerations:
    # ...
```

以下のように変更します。

```
spec:
```

```
# ...
template:
  pod:
    affinity:
      # ...
    tolerations:
      # ...
```

3. type: external ロギングが .spec.logging に設定されている場合:

   ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

   valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名とキー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j.properties
```

4. .spec.metrics フィールドを使用してメトリクスを有効にする場合：

   a. JMX Prometheus エクスポーターの YAML 設定をキーの下に保存する新しい ConfigMap を作成します。YAML は、現在 .spec.metrics フィールドの内容と一致している必要があります。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kafka-mm-metrics
  labels:
    app: strimzi
data:
  mm-metrics-config.yaml: |
    <YAML>
```

   b. ConfigMap およびキーを参照する .spec.metricsConfig プロパティーを追加します。

```
metricsConfig:
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: kafka-mm-metrics
      key: mm-metrics-config.yaml
```

c. old .spec.metrics フィールドを削除します。

5. KafkaMirrorMaker カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

6. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.15. Kafka MirrorMaker 2.0 リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- MirrorMaker 2.0 が設定され、デプロイされます。「Kafka MirrorMaker の OpenShift クラスターへのデプロイ」 を参照してください。

手順

デプロイメントの KafkaMirrorMaker2 カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaMirrorMaker2 カスタムリソースを更新します。

```
oc edit kafkamirrormaker2 MIRROR-MAKER-2
```

2. 存在する場合は、アフィニティー を .spec.affinity から .spec.template.pod.affinity に移動します。

3. 容認 がある場合は、容認を .spec.tolerations から .spec.template.pod.tolerations に移動します。

4. type: external ロギングが .spec.logging に設定されている場合:

   ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

   valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名と キー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j.properties
```

5. .spec.metrics フィールドを使用してメトリクスを有効にする場合：

   a. JMX Prometheus エクスポーターの YAML 設定をキーの下に保存する新しい ConfigMap を作成します。YAML は、現在 .spec.metrics フィールドの内容と一致している必要があります。

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: kafka-mm2-metrics
  labels:
    app: strimzi
data:
  mm2-metrics-config.yaml: |
    <YAML>
```

b. ConfigMap およびキーを参照する .spec.metricsConfig プロパティーを追加します。

```
metricsConfig:
  type: jmxPrometheusExporter
  valueFrom:
    configMapKeyRef:
      name: kafka-mm2-metrics
      key: mm2-metrics-config.yaml
```

c. old .spec.metrics フィールドを削除します。

6. KafkaMirrorMaker2 カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

7. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.16. Kafka Bridge リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- Kafka Bridge が設定され、デプロイされます。「Kafka Bridge を OpenShift クラスターへデプロイ」 を参照してください。

手順

デプロイメントの KafkaBridge リソースごとに以下の手順を実行します。

1. エディターで KafkaBridge カスタムリソースを更新します。

```
oc edit kafkabridge KAFKA-BRIDGE
```

2. type: external ロギングが KafkaBridge.spec.logging で設定される場合：

ロギング設定が含まれる ConfigMap の名前 を置き換えます。

```
logging:
  type: external
  name: my-config-map
```

valueFrom.configMapKeyRef フィールドで、ロギングが保存される ConfigMap 名とキー の両方を指定します。

```
logging:
  type: external
  valueFrom:
    configMapKeyRef:
      name: my-config-map
      key: log4j2.properties
```

3. KafkaBridge カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

4. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.17. Kafka User リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする User Operator が稼働している必要があります。

手順

デプロイメントの KafkaUser カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaUser カスタムリソースを更新します。

```
oc edit kafkauser KAFKA-USER
```

2. KafkaUser カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

3. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.18. Kafka Topic リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Topic Operator が稼働している必要があります。

手順

デプロイメントの KafkaTopic カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaTopic カスタムリソースを更新します。

```
oc edit kafkatopic KAFKA-TOPIC
```

2. KafkaTopic カスタムリソースの apiVersion を v1beta2 に更新します。

以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

3. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.19. Kafka Connector リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- コネクターインスタンスを管理するために KafkaConnector カスタムリソースがデプロイされている必要があります。「コネクターの作成および管理」 を参照してください。

手順

デプロイメントの KafkaConnector カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaConnector カスタムリソースを更新します。

```
oc edit kafkaconnector KAFKA-CONNECTOR
```

2. KafkaConnector カスタムリソースの apiVersion を v1beta2 に更新します。

   以下を置き換えます。

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

   上のコマンドを、下のコマンドに置き換えます。

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

3. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.2.20. Kafka Rebalance リソースの v1beta2 へのアップグレード

前提条件

- v1beta2 API バージョンをサポートする Cluster Operator が稼働している必要があります。
- Cruise Control が設定およびデプロイされている必要があります。『Using AMQ Streams on OpenShift』の「Deploying Cruise Control」を参照してください。

手順

デプロイメントの KafkaRebalance カスタムリソースごとに以下の手順を実行します。

1. エディターで KafkaRebalance カスタムリソースを更新します。

   ```
   oc edit kafkarebalance KAFKA-REBALANCE
   ```

2. KafkaRebalance カスタムリソースの apiVersion を v1beta2 に更新します。

   以下を置き換えます。

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

上のコマンドを、下のコマンドに置き換えます。

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

3. ファイルを保存し、エディターを終了して更新したカスタムリソースが調整されるのを待ちます。

8.3. CLUSTER OPERATOR のアップグレード

この手順では、AMQ Streams 1.8 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。

OperatorHub ではなくインストール用の YAML ファイルを使用して Cluster Operator をデプロイした場合は、次の手順に従ってください。

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については、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必要があります。
- AMQ Streams 1.8 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覚えておきます。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 1.8 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 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
   ```

   ローリングアップデート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前の Kafka バージョンをサポートしなくなった場合、「Version not found」というエラーメッセージが Cluster Operator によって返されます。そうでない場合は、エラーメッセージは返されません。**

   以下は例になります。

   ```
   "Version 2.4.0 is not supported. Supported versions are: 2.6.0, 2.6.1, 2.7.0."
   ```

   -

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 プロパティーをサポートされる 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。以下は例になります。

```
registry.redhat.io/amq7/amq-streams-kafka-28-rhel7:{ContainerVersion}
```

Cluster Operator はバージョン 1.8 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。

8.4. KAFKA のアップグレード

Cluster Operator を 1.8 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アップデートによって 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アップデートを開始します。

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。	単一のローリングアップデート更新後、**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。それぞれの変更により、追加のローリングアップデートがトリガーされます。
inter.broker.protocol.version または **log.message.format.version** のいずれか。	2 つのローリングアップデート
inter.broker.protocol.version または **log.message.format.version** の設定なし。	2 つのローリングアップデート

Cluster Operator は、Kafka のアップグレードの一環として、ZooKeeper のローリングアップデートを開始します。

- **ZooKeeper バージョンが変更されなくても、単一のローリングアップデートが発生します。**
- **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、追加のローリングアップデートが発生します。**

その他のリソース

- **「Cluster Operator のアップグレード」**
- **「Kafka バージョン」**

8.4.1. Kafka バージョン

Kafka 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。

Kafka のバージョン	Interbroker プロトコルのバージョン	ログメッセージ形式のバージョン	ZooKeeper のバージョン
2.7.0	2.7	2.7	3.5.8
2.7.1	2.7	2.7	3.5.9
2.8.0	2.8	2.8	3.5.9

ブローカー間のプロトコルバージョン

Kafka では、ブローカー間の通信に使用されるネットワークプロトコルはブローカー間プロトコル (Inter-broker protocol) 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ブローカー間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。

ブローカー間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 プロパティーを編集します。

ログメッセージ形式のバージョン

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され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形式のバージョンが指定されます。ブローカーがメッセージをログに追加する前に、メッセージを新しい形式バージョンから特定の旧形式バージョンに変換するように、Kafka ブローカーを設定できます。

Kafka には、メッセージ形式のバージョンを設定する 2 通りの方法があります。

- message.format.version プロパティーをトピックに設定します。
- log.message.format.version プロパティーを Kafka ブローカーに設定します。

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。

本セクションのアップグレード作業では、メッセージ形式のバージョンが log.message.format.version によって定義されることを前提としています。

8.4.2. 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 コネクターを含む) 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 では単一のとピックが設定されます。
- ブローカーレベルの log.message.format.version は、トピックレベルの message.format.version が設定されて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ときでなく、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ストラテジーは複数あります。

コンシューマーを最初にアップグレード

1. 消費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。

トピック単位でコンシューマーを最初にアップグレード

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。

トピック単位でコンシューマーを最初にアップグレード、ダウンコンバートあり

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。

このストラテジー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。

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。

注記

複数のストラテジーを適用することもできます。たとえば、最初のいくつかのアプリケーションとトピックに、「トピック単位でコンシューマーを最初にアップグレード、ダウンコンバートあり」のストラテジーを適用します。これが問題なく適用されたら、より効率的な別のストラテジーの使用を検討できます。

8.4.3. 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 プロパティー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す。

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。

8.4.4. 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 クラスター設定を更新します。

```
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 2.7.0 から 2.8.0 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: 2.7.0
    config:
      log.message.format.version: "2.7"
      inter.broker.protocol.version: "2.7"
      # ...
```

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も相互に通信を継続できます。

たとえば、Kafka 2.7.0 から 2.8.0 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: 2.8.0 (1)
  config:
   log.message.format.version: "2.7" (2)
   inter.broker.protocol.version: "2.7" (3)
   # ...
```

(1) Kafka のバージョンが新しい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。

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ー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、ローリングアップデートの完了を待ちます。

Pod の状態の遷移を監視して、ローリングアップデートの進捗を確認します。

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

ローリングアップデートにより、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ことが保証されます。

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 プロパティーを新バージョンの 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 2.8.0 へのアップグレードでは以下のようになります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.8.0
    config:
      log.message.format.version: "2.7"
      inter.broker.protocol.version: "2.8"
      # ...
```

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 2.8.0 へのアップグレードでは以下のようになります。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.8.0
    config:
      log.message.format.version: "2.8"
      inter.broker.protocol.version: "2.8"
      # ...
```

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 を使用するようになります。
- ブローカーは、ブローカー間プロトコルのバージョンと、新しいバージョンの Kafka のメッセージ形式の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。

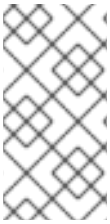
Kafka のアップグレードに従い、必要な場合は以下を行うことができます。

- リスナーを GenericKafkaListener スキーマ に更新
- 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
- 既存のカスタムリソースの更新

8.5. 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、クラスター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 2.8.0 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ことが必要です。

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には以下を行います。

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には以下を行います。

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ー (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。

その他のリソース

- Apache Kafka ドキュメントの「Notable changes in 2.4.0」。

第9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。

以下のダウングレードを実行できます。

1. Cluster Operator を以前の AMQ Streams バージョンに戻します。
   - 「Cluster Operator の以前のバージョンへのダウングレード」
2. 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、以前の Kafka バージョンにダウングレードします。
   - 「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。

9.1. 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必要があります。
- 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覚えておきます。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
   ```

   ローリングアップデートが完了するのを待ちます。

5. **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 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。

9.2. 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。

9.2.1. 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メッセージ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 2.7.0 にダウングレードできます。

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
spec:
  # ...
  kafka:
    version: 2.8.0
    config:
      log.message.format.version: "2.7"
      # ...
```

log.message.format.version が "2.8" に設定されているか、値がない（このためパラメーターに 2.8.0 ブローカーのデフォルト値 2.8.0）場合は、ダウングレードは実施できません。

9.2.2. 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 (2.8.0 から 2.7.0 へのダウングレードなど) を説明します。

前提条件

Kafka リソースをダウングレードするには、以下を確認します。

- 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。

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 2.8.0 から 2.7.0 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: 2.7.0 (1)
  config:
   log.message.format.version: "2.7" (2)
   inter.broker.protocol.version: "2.7" (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 の値と 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、文字列にする必要があります。

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」 を参照してください。

4. エディターを保存して終了し、ローリングアップデートの完了を待ちます。

更新をログで確認するか、または Pod 状態の遷移を監視して確認します。

```
oc logs -f 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 バージョンを使用するようになります。

6. トピックメタデータの保存に ZooKeeper を使用する 0.22 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-28-rhel7:1.8.0 --rm=true --restart=Never -- ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi-topic-operator-kstreams-topic-store-changelog --delete && ./bin/kafka-topics.sh --bootstrap-server localhost:9092 --topic __strimzi_store_topic --delete
```

関連情報

- Topic Operator のトピックストア

付録A サブスクリプションの使用

AMQ Streams は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. サブスクリプション に移動します。
3. Activate a subscription に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合は、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの Product Downloads ページにログインします。
2. INTEGRATION AND AUTOMATION カテゴリーで Red Hat AMQ Streams エントリーを見つけます。
3. 必要な AMQ Streams 製品を選択します。Software Downloads ページが開きます。

4. コンポーネントの **Download** リンクをクリックします。

Revised on 2021-12-18 13:38:49 +1000