Red Hat Streams for Apache Kafka 2.7

Streams for Apache Kafka Proxy の使用

特化機能で Kafka を強化

Last Updated: 2024-09-16

Red Hat Streams for Apache Kafka 2.7 Streams for Apache Kafka Proxy の使用

特化機能で Kafka を強化

法律上の通知

概要

Streams for Apache Kafka Proxy は、フィルターメカニズムを通じて Kafka ベースのシステムを強化するように設計された、Apache Kafka プロトコル対応プロキシーです。現在プレビュー版の Streams for Apache Kafka Proxy には、Record Encryption フィルターが含まれています。このフィルターは、Kafka クラスターに保存されているデータの保存時の暗号化を提供します。

目次

はじめに

RED HAT ドキュメントへのフィードバック (英語のみ)

Red Hat ドキュメントに関するご意見やご感想をお寄せください。

改善を提案するには、Jira 課題を作成し、変更案についてご説明ください。ご要望に迅速に対応できるよう、できるだけ詳細にご記入ください。

前提条件

- Red Hat カスタマーポータルのアカウントがある。このアカウントを使用すると、Red Hat Jira Software インスタンスにログインできます。
  アカウントをお持ちでない場合は、アカウントを作成するように求められます。

手順

1. 以下の Create issue をクリックします。
2. **Summary** テキストボックスに、問題の簡単な説明を入力します。
3. **Description** テキストボックスに、次の情報を入力します。
   - 問題が見つかったページの URL
   - 問題の詳細情報
     他のフィールドの情報はデフォルト値のままにすることができます。
4. レポーター名を追加します。
5. **Create** をクリックして、Jira 課題をドキュメントチームに送信します。

フィードバックをご提供いただきありがとうございました。

テクノロジープレビュー

Streams for Apache Kafka Proxy はテクノロジープレビューです。

テクノロジープレビュー機能は、Red Hat の実稼働環境のサービスレベルアグリーメント (SLA) ではサポートされません。また、機能的に完全ではない可能性があるため、Red Hat はテクノロジープレビュー機能を実稼働環境に実装することは推奨しません。テクノロジープレビューの機能は、最新の技術をいち早く提供して、開発段階で機能のテストやフィードバックの収集を可能にするために提供されます。サポート範囲の詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

第1章 STREAMS FOR APACHE KAFKA PROXY の概要

Streams for Apache Kafka Proxy は、Kafka ベースのシステムを強化するために設計された Apache Kafka プロトコル対応プロキシーです。そのフィルターメカニズムにより、アプリケーションや Kafka クラスター自体を変更することなく、Kafka ベースのシステムに追加の動作を導入できます。

Streams for Apache Kafka Proxy は仲介者として機能し、Kafka クラスターとそのクライアント間の通信を仲介します。このプロキシーは、メッセージの受信、フィルタリング、転送を行います。

1.1. RECORD ENCRYPTION フィルター

Streams for Apache Kafka Proxy の Record Encryption は、Kafka メッセージのセキュリティーを強化します。このフィルターは、業界標準の暗号化技術を使用して Kafka メッセージに暗号化を適用し、Kafka クラスターに保存されているデータの機密性を確保します。Streams for Apache Kafka Proxy はトピックレベルの暗号化を集中管理し、Kafka クラスター全体で暗号化が効率的に行われるようにします。

フィルターはエンベロープ暗号化を使用して、対称暗号鍵でレコードを暗号化します。

エンベロープ暗号化

エンベロープ暗号化は、大量のデータを効率的に暗号化するのに適した業界標準の技術です。データは Data Encryption Key (DEK) で暗号化されます。DEK は、Key Encryption Key (KEK) を使用して暗号化されます。KEK は Key Management System (KMS) に安全に保存されます。

対称暗号鍵

AES (GCM) 256 ビット暗号化対称暗号鍵は、レコードデータの暗号化と復号化に使用されます。

プロセスは以下のようになります。

1. フィルターは、生成アプリケーションからの生成要求をインターセプトし、レコードを暗号化します。
2. 生産要求はブローカーに転送されます。
3. フィルターは、消費アプリケーションからのフェッチ応答をインターセプトし、レコードを復号化します。
4. フェッチ応答は、消費アプリケーションに転送されます。

フィルターはレコード値のみを暗号化します。レコード鍵、ヘッダー、タイムスタンプは暗号化されません。

Kafka クライアントと Kafka ブローカーから見ると、プロセス全体が透過的です。どちらも、レコードが暗号化されていることを認識しておらず、暗号化鍵にアクセスすることも、レコードを保護するための暗号化プロセスに影響を与えることもできません。

フィルターは、鍵マテリアルの安全な保管の最終的な責任を負う Key Management Service (KMS) と統合されます。現在、フィルターは KMS として HashiCorp Vault と統合されていますが、今後 KMS インテグレーションのさらなるサポートが計画されています。

1.1.1. フィルターがレコードを暗号化する方法

フィルターは、次のように生成要求からのレコードを暗号化します。

1. フィルターは適用する KEK を選択します。

2. KEK の DEK を生成するように KMS に要求します。
3. 暗号化された DEK (KEK で暗号化された DEK) を使用してレコードを暗号化します。
4. 元のレコードを暗号レコード (暗号化されたレコード、暗号化された DEK、およびメタデータ) に置き換えます。

フィルターは DEK 再利用ストラテジーを使用します。暗号化されたレコードは、タイムアウトまたは暗号化制限に達するまで、同じ DEK を使用して同じトピックに送信されます。

1.1.2. フィルターを使用してレコードを復号化する方法

フィルターは次のようにフェッチ応答からレコードを復号化します。

1. フィルターは Kafka ブローカーから暗号レコードを受信します。
2. 暗号レコードを構築したプロセスを元に戻します。
3. KMS を使用して DEK を復号化します。
4. 復号化された DEK を使用して暗号化されたレコードを復号化します。
5. 暗号レコードを復号化されたレコードに置き換えます。

フィルターは、復号化されたレコードに対して LRU (最近最も使われていない) ストラテジーを使用します。復号化された DEK は、KMS とのやり取りを減らすためにメモリー内に保持されます。

1.1.3. フィルターで KMS を使用する方法

フィルターをサポートするために、KMS は以下を提供します。

- Key Encryption Key (KEK) を保存するための安全なリポジトリー
- Data Encryption Keys (DEK) を生成および復号化するサービス

KEK は KMS 内に留まります。KMS は、指定された KEK に対して DEK (安全に生成されたランダムデータ) を生成し、DEK と暗号化された DEK を返します。暗号化された DEK には同じデータが含まれますが、KEK で暗号化されています。KMS は DEK を保存しません。DEK はブローカーの暗号レコードの一部として保存されます。

警告

KMS は実行時に使用可能である必要があります。KMS が利用できない場合は、KMS サービスが回復するまで、フィルターを使用した生成と使用は不可能になります。KMS は高可用性 (HA) 設定で使用することを推奨します。

1.1.4. 暗号化されるレコード

Record Encryption フィルターは、レコードの値のみを暗号化し、レコード鍵、ヘッダー、タイムスタンプはそのまま残します。圧縮されたトピック内の削除を表す可能性のある null レコード値は、暗号化されずにブローカーに送信されます。このアプローチにより、圧縮されたトピックが正しく機能するこ

とが保証されます。

1.1.5. 暗号化されていないトピック

一部のトピックを暗号化し、他のトピックを暗号化しないようにシステムを設定できます。これにより、機密情報を含むトピックは暗号化し、機密情報を含まない Kafka トピックは暗号化しないシナリオがサポートされます。

第2章 RECORD ENCRYPTION フィルター用の HASHICORP VAULT の準備

OpenShift クラスターで Record Encryption フィルターを使用して Vault を使用するには、Vault インスタンスに対して次の設定を使用します。

- Record Encryption フィルターは Transit Engine の API に依存しているため、Transit Engine を有効にします。
- エンベロープ暗号化用の Data Encryption Keys (DEK) を生成および復号化する権限を持つ、フィルター専用の Vault ポリシーを作成します。
- フィルターポリシーを含む Vault トークンを取得します。

プロキシーのデプロイメント設定では、Vault Transit Engine サービスの URL が使用されます。

Vault は、既存のインスタンス、クラウドインスタンス、または OpenShift 上にデプロイできます。プロキシーにアクセスできる場合は、Streams for Apache Kafka Proxy と同じ場所に配置することも、リモートでデプロイすることもできます。

OpenShift に Vault をインストールし、アクセスを設定する方法については、HashiCorp Vault 製品のドキュメントを参照してください。

この手順では、Vault を準備するための 2 つのオプションを説明します。

- Streams for Apache Kafka Proxy に付属する一時的なデプロイメント設定の例を使用して、Helm で Vault を OpenShift クラスターにデプロイします。
- 既存の Vault インスタンスを更新しています。

Vault インスタンスを準備したら、Record Encryption フィルター用の Vault ポリシーとトークンを作成する必要があります。

警告

サンプルのデプロイメント設定は、実稼働環境には適していません。

Streams for Apache Kafka には、**examples/proxy/record-encryption/vault** フォルダーにサンプルのインストールアーティファクトが含まれています。このフォルダーには、プロキシーおよび Record Encryption フィルターと互換性のある事前設定済みの Vault デプロイメントファイルが含まれています。

- **amqstreams_proxy_encryption_filter_policy.hcl** は、Record Encryption フィルターの Vault ポリシーを定義します。
- **helm-dev-values.yaml** は、Vault の Helm デプロイメント設定を指定します。

これらのインストールファイルは、プロキシーを試すための簡単なセットアップを提供します。

前提条件

- インストール用に **system:admin** などの **cluster-admin** ロールを持つ OpenShift ユーザーを用意する。
- **oc** コマンドラインツールがインストールされ、管理者権限で OpenShift クラスターに接続するように設定されている。
- **Helm** コマンドラインツールがインストールされ、管理者権限で OpenShift クラスターに接続するように設定されている。
- **proxy** と呼ばれる OpenShift プロジェクトの namespace。これは、プロキシーがデフォルトでインストールされている namespace と同じである。

この手順で使用される **oc** および **Helm** コマンドラインオプションの詳細は、**--help** を参照してください。

Helm デプロイメント設定の例を使用した Vault のデプロイ

1. Streams for Apache Kafka Proxy インストールアーティファクトをダウンロードして展開します。
   プロキシーは、Streams for Apache Kafka ソフトウェアダウンロードページ から入手できます。

   ファイルには、Vault をデプロイするために必要なデプロイメント設定が含まれています。

2. ルートトークンを作成し、メモしておきます。

```
cat /dev/urandom | LC_ALL=C tr -dc 'a-zA-Z0-9' | fold -w 32 | head -n 1 > vault.root.token
export VAULT_TOKEN=$(cat vault.root.token)
```

3. Helm を使用して Vault をインストールします。

```
helm repo add hashicorp https://helm.releases.hashicorp.com
helm install vault hashicorp/vault \
  --create-namespace --namespace=vault \
  --version <helm_version> \
  --values vault/helm-dev-values.yaml \
  --set server.dev.devRootToken=${VAULT_TOKEN} \
  --wait
```

   root トークンは Vault インスタンスに使用されます。

4. デプロイメントのステータスを確認します。

```
oc get pods -n vault
```

   デプロイメント名と準備状態が表示されている出力

```
NAME                    READY  STATUS   RESTARTS
vault-0                 1/1    Running  0
```

   Pod ID は、作成された Pod を識別します。

   デフォルトのデプロイメントでは、単一のプロキシー Pod をインストールします。

READY は、ready/expected 状態のレプリカ数を表示します。STATUS が Running と表示されれば、デプロイメントは成功です。

5. 新しい Vault インスタンスを指す Vault アドレス (**VAULT_ADDR**) 環境変数を作成します。

```
export VAULT_ADDR=$(oc get route -n vault vault --template='https://{{.spec.host}}')
```

6. 管理者として Vault にログインし、Vault Transit シークレットエンジンを有効にします。

```
vault secrets enable transit
```

シークレットエンジンがすでに有効になっている場合は、エラーを無視します。

7. Vault Transit アドレスを指す環境変数を作成します。

```
export VAULT_TRANSIT_URL=${VAULT_ADDR}/v1/transit
```

アドレスはプロキシーデプロイメント設定で使用されます。

8. Vault ポリシーとトークンを作成します。

独自の Vault インスタンスの設定

すでに Kafka インスタンスがインストールされている場合は、それを更新して Streams for Apache Kafka Proxy で使用できます。

1. Vault インスタンスを指す Vault アドレス環境変数 (Enterprise を使用している場合は **VAULT_ADDR** と **VAULT_NAMESPACE**) を作成します。

```
export VAULT_ADDR=https://<vault server>:8200
export VAULT_NAMESPACE=<namespaces>
```

2. 管理者として Vault にログインし、Vault Transit シークレットエンジンを有効にします。

```
vault secrets enable transit
```

シークレットエンジンがすでに有効になっている場合は、エラーを無視します。

3. Vault Transit アドレスを指す環境変数を作成します。

```
export VAULT_TRANSIT_URL=${VAULT_ADDR}/v1/${VAULT_NAMESPACE}/transit
```

アドレスはプロキシーデプロイメント設定で使用されます。

4. Vault インスタンスを参照するようにプロキシーデプロイメント設定を更新します。

```
sed -i "s/\(vaultTransitEngineUrl:\).*$/\1 ${VAULT_TRANSIT_URL}/" */proxy/proxy-config.yaml
```

5. Vault ポリシーとトークンを作成します。

Vault ポリシーとトークンの作成

Vault インスタンスを設定したら、Record Encryption フィルターの Vault ポリシーとトークンを作成します。

1. Vault ポリシーを作成します。

```
vault policy write amqstreams_proxy_encryption_filter_policy
vault/amqstreams_proxy_encryption_filter_policy.hcl
```

Streams for Apache Kafka Proxy に付属する HashiCorp ポリシー定義ファイル (**.hcl**) を使用して、ポリシーを Vault に書き込みます。ポリシーの名前は **amqstreams_proxy_encryption_filter_policy** です。

2. Vault トークンを作成します。

```
vault token create \
  -display-name "amqstreams-proxy encryption filter" \
  -policy=amqstreams_proxy_encryption_filter_policy \
  -no-default-policy \
  -orphan \
  -field=token > vault.encryption.token
```

このコマンドは、ポリシーを指定したトークン、および関連付けられた親トークンまたはデフォルトポリシーのないトークンを作成します。

3. トークンを含むシークレットを作成します。

```
oc create secret generic proxy-encryption-vault-token \
  -n proxy \
  --from-file=encryption-vault-token.txt=vault.encryption.token \
  --dry-run=client \
  -o yaml > base/proxy/proxy-encryption-vault-token-secret.yaml
```

このコマンドは、Vault トークンをシークレットに保存し、**proxy** namespace に YAML ファイルとしてシークレットを作成します。

Record Encryption フィルターを使用して Streams for Apache Kafka をデプロイするときに、**proxy-encryption-vault-token-secret.yaml** シークレットが OpenShift クラスターに適用されます。

ヒント

侵害された鍵の影響を最小限に抑えるために、定期的に鍵をローテーションします。HashiCorp Vault などの Key Management System (KMS) を使用する場合は、KMS に保存されている Key Encryption Key (KEK) をローテーションします。Streams for Apache Kafka Proxy は DEK のローテーションを自動的に管理します。プロキシーが新しい鍵を取得するには、再起動が必要になる場合があります。さらに、暗号化されたメッセージには、鍵のローテーションを示す鍵バージョンメタデータが含まれている必要があります。

第3章 RECORD ENCRYPTION フィルターを使用した STREAMS FOR APACHE KAFKA PROXY のデプロイ

Streams for Apache Kafka Proxy は、Streams for Apache Kafka によって管理される Kafka クラスターとシームレスに統合するように設計されています。さらに、ディストリビューションやプロトコルのバージョンに関係なく、全種の Kafka インスタンスとの互換性も提供します。デプロイメントにパブリッククラウドまたはプライベートクラウドが含まれるかどうか、ローカル開発環境をセットアップしている場合、このガイドの手順はすべての場合に当てはまります。

この手順では、Streams for Apache Kafka Proxy を Record Encryption フィルターとともにデプロイし、OpenShift 上の Streams for Apache Kafka によって管理される Kafka インスタンスでフィルターを使用できるようにします。

Streams for Apache Kafka では、Streams for Apache Kafka Proxy が Kafka クラスターに接続するために必要な設定を含むサンプルのインストールアーティファクトが **examples/proxy/record-encryption** フォルダーに用意されています。

サンプル設定ファイルを使用して、次のタイプのリスナーでプロキシーをデプロイおよび公開します。

- ブローカーごとの **ClusterIP** サービスを使用して OpenShift クラスター内でプロキシーを公開する **cluster-ip** タイプのリスナー
- ブローカーごとの **loadbalancer** サービスを使用してプロキシーを OpenShift クラスターの外部に公開する **loadbalancer** タイプのリスナー

各オプションごとに、次のファイルが提供されます。

- **kustomization.yaml**。プロキシーをデプロイするための Kubernetes カスタマイズを指定します
- **proxy-config.yaml**。プロキシーの **ConfigMap** リソース設定を指定します
- **proxy-service.yaml**。プロキシーサービスの **Service** リソース設定を指定します

ConfigMap リソースは、仮想クラスターとフィルターを指定するための設定を提供します。仮想クラスターは、プロキシーで使用する Kafka クラスターを表します。

前提条件

- サポートされているバージョンを実行している OpenShift クラスター。
- OpenShift クラスター上で実行されている Streams for Apache Kafka によって管理される Kafka クラスター。
- ロードバランサーを介した外部アクセスのプロキシー設定を確認するためにローカルにインストールされた Kafka バイナリー。Kafka バイナリーは、Streams for Apache Kafka ソフトウェアダウンロードページ から入手できる RHEL 上の Streams for Apache Kafka のインストールアーティファクトに含まれています。
- 仮想クラスターとフィルターを作成するための設定が含まれる config map。
- **oc** コマンドラインツールがインストールされ、管理者権限で OpenShift クラスターに接続するように設定されている。
- **Helm** コマンドラインツールがインストールされ、管理者権限で OpenShift クラスターに接続するように設定されている。

- HashiCorp Vault がプロキシー用に設定されており、Streams for Apache Kafka Proxy からアクセスできる。
  Vault インスタンスが Record Encryption フィルター用に設定されていること を確認します。

この手順で使用される **oc** および **Helm** コマンドラインオプションの詳細は、**--help** を参照してください。

Streams for Apache Kafka Proxy では、Streams for Apache Kafka Proxy をインストールするためのファイルに加えて、Kafka クラスターをインストールするための事前設定済みファイルも用意されています。インストールファイルを使用すると、最も簡単にプロキシーをセットアップして試すことができます。ただし、Streams for Apache Kafka および Vault によって管理される Kafka クラスターの独自のデプロイメントを使用することもできます。

この手順では、**Kafka** namespace にデプロイされた **my-cluster** という名前の Kafka クラスターに接続します。Streams for Apache Kafka によって管理されるクラスターと同じ namespace にプロキシーをデプロイするには、**kustomization.yaml** ファイルの **namespace** 設定を変更します。プロキシーは、デフォルトで **proxy** namespace にデプロイされます。この設定を維持する場合は、Streams for Apache Kafka Operator をクラスター全体にインストールする必要があります。

手順

1. Streams for Apache Kafka Proxy インストールアーティファクトをダウンロードして展開します。
   アーティファクトは、Streams for Apache Kafka ソフトウェアダウンロードページ から入手できるインストールおよびサンプルファイルに含まれています。

   ファイルには、**cluster-ip** または **loadbalancer** タイプのリスナーを介して接続するために必要なデプロイメント設定が含まれています。

2. Kafka クラスターにトピックを作成します。

```
oc run -n <my_proxy_namespace> -ti proxy-producer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never \
  -- bin/kafka-topics.sh \
  --bootstrap-server proxy-service:9092 \
  --create -topic my-topic
```

   この例では、インタラクティブな Pod コンテナーを通じて **my-topic** という名前のトピックを作成します。

3. Vault で **my-topic** の鍵を作成します。

```
vault write -f transit/keys/KEK_my-topic
```

4. プロキシーのフィルター設定を提供する **ConfigMap** を編集します。
   Record Encryption フィルター **config** には、HashiCorp Vault KMS の認証情報が必要です。

   Record Encryption フィルターの設定例

```
filters:
  - type: RecordEncryption
    config:
      kms: VaultKmsService (1)
```

```
    kmsConfig:
      vaultTransitEngineUrl: http://vault.vault.svc.cluster.local:8200/v1/transit (2)
      vaultToken:
        passwordFile: /opt/proxy/encryption/token.txt (3)
    selector: TemplateKekSelector (4)
    selectorConfig:
      template: "KEK_${topicName}" (5)
# ...
```

1. 使用する KMS (key management system) の種類。今回は、HashiCorp Vault を使用します。
2. Vault Transit Engine サービスの URL。
3. Vault サービスにアクセスするために必要なトークンを含むファイル。この場所が変更された場合は、プロキシーデプロイメント設定で同様の変更が必要になります。
4. 使用する Key Encryption Key (KEK) セレクター。**${topicName}** は、プロキシーが理解するリテラルです。
5. 特定のトピック名に基づいて KEK を導出するためのテンプレート。

5. Record Encryption フィルターと適切なリスナーを使用して、Streams for Apache Kafka Proxy を OpenShift クラスターにデプロイします。

cluster-ip リスナーを使用したプロキシーのデプロイ

```
cd /examples/proxy/record-encryption/
oc apply -k cluster-ip
```

loadbalancer リスナーを使用したプロキシーのデプロイ

```
cd /examples/proxy/record-encryption/
oc apply -k loadbalancer
```

6. **loadbalancer** リスナーを使用している場合は、作成されたロードバランサーサービスのアドレスを使用するようにプロキシー設定を更新します。
   a. プロキシーサービスの外部アドレスを取得します。

   ```
   LOAD_BALANCER_ADDRESS=$(oc get service -n <my_proxy_namespace> proxy-service --template='{{(index .status.loadBalancer.ingress 0).hostname}}')
   ```

   b. ブローカーアドレスを使用するには、プロキシーサービス設定の **brokerAddressPattern** プロパティーを更新します。

   ```
   sed -i "s/\(brokerAddressPattern:\).*$/\1 ${LOAD_BALANCER_ADDRESS}/" load-balancer/proxy/proxy-config.yaml
   ```

   c. プロキシー設定に変更を適用し、プロキシー Pod を再起動します。

   ```
   oc apply -k load-balancer && oc delete pod -n <my_proxy_namespace> --all
   ```

7. デプロイメントのステータスを確認します。

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

デプロイメント名と準備状態が表示されている出力

```
NAME                  READY  STATUS   RESTARTS
my-cluster-kafka-0       1/1    Running  0
my-cluster-kafka-1       1/1    Running  0
my-cluster-kafka-2       1/1    Running  0
my-cluster-proxy-<pod_id> 1/1    Running  0
```

my-cluster-proxy はプロキシーの名前です。

Pod ID は、作成された Pod を識別します。

デフォルトのデプロイメントでは、単一のプロキシー Pod をインストールします。

READY は、ready/expected 状態のレプリカ数を表示します。STATUS が Running と表示されれば、デプロイメントは成功です。

8. プロキシー経由でメッセージを生成し、Kafka クラスターから直接的または間接的に消費して、指定されたトピックに暗号化が適用されていることを確認します。

3.1. CLUSTER-IP タイプのリスナーを使用する場合のプロキシーの検証

OpenShift クラスター内で Kafka プロデューサーとコンシューマーの対話型 Pod コンテナーを実行して、**cluster-ip** タイプのリスナーを使用するときにプロキシーが機能していることを確認します。

1. プロキシーからメッセージを生成します。

プロキシー経由でメッセージを生成する

```
oc run -n <my_proxy_namespace> -ti proxy-producer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never \
  -- bin/kafka-console-producer.sh \
  --bootstrap-server proxy-service:9092 \
  --topic my-topic
```

2. メッセージが暗号化されていることを示すために、Kafka クラスターから直接メッセージを消費します。

Kafka クラスターから直接メッセージを消費する

```
oc run -n my-cluster -ti cluster-consumer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never
  -- ./bin/kafka-console-consumer.sh  \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 \
```

```
  --topic my-topic \
  --from-beginning \
  --timeout-ms 10000
```

3. プロキシーからメッセージを消費して、自動的に復号化されていることを示します。

Kafka クラスターから直接メッセージを消費する

```
oc run -n <my_proxy_namespace> -ti proxy-consumer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never \
  -- ./bin/kafka-console-consumer.sh \
  --bootstrap-server proxy-service:9092 \
  --topic my-topic --from-beginning --timeout-ms 10000
```

3.2. LOADBALANCER タイプのリスナーを使用する場合のプロキシーの検証

loadbalancer タイプのリスナーを使用するときに、プロキシーを介して Kafka プロデューサーとコンシューマーをローカルで実行して、プロキシーが機能していることを確認します。

1. ロードバランサーアドレスを使用してプロキシーからメッセージを生成します。

プロキシー経由でメッセージを生成する

```
kafka-console-producer \
--bootstrap-server <load_balancer_address>:9092 \
--topic my-topic
```

2. インタラクティブな Pod コンテナーを使用して Kafka クラスターから直接メッセージを消費し、メッセージが暗号化されていることを確認します。

Kafka クラスターから直接メッセージを消費する

```
oc run -n my-cluster -ti cluster-consumer \
  --image=registry.redhat.io/amq-streams/kafka-37-rhel9:2.7.0 \
  --rm=true \
  --restart=Never
  -- ./bin/kafka-console-consumer.sh  \
  --bootstrap-server my-cluster-kafka-bootstrap:9092 \
  --topic my-topic \
  --from-beginning \
  --timeout-ms 10000
```

3. プロキシーからメッセージを消費して、自動的に復号化されていることを示します。

Kafka クラスターから直接メッセージを消費する

```
kafka-console-consumer \
--bootstrap-server <load_balancer_address>:9092 \
--topic my-topic \
--from-beginning \
--timeout-ms 10000
```

第4章 STREAMS FOR APACHE KAFKA PROXY の設定

Streams for Apache Kafka Proxy のリソースを設定して、特定の要件に応じて追加機能を追加し、デプロイメントを微調整します。

4.1. STREAMS FOR APACHE KAFKA PROXY 設定の例

Streams for Apache Kafka Proxy 設定は **ConfigMap** リソースで定義します。**ConfigMap** リソースの **data** プロパティーを使用して、以下を設定します。

- Kafka クラスターを表す仮想クラスター
- Kafka クラスターにあるブローカー通信用のネットワークアドレス
- Kafka デプロイメントに追加機能を導入するためのフィルター

この例では、Record Encryption フィルターの設定を示します。

Streams for Apache Kafka Proxy 設定の例

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: proxy-config
data:
  config.yaml: |
    adminHttp: (1)
      endpoints:
        prometheus: {}
    virtualClusters: (2)
      my-cluster-proxy: (3)
        targetCluster:
          bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093 (4)
          tls: (5)
            trust:
              storeFile: /opt/proxy/trust/ca.p12
              storePassword:
                passwordFile: /opt/proxy/trust/ca.password
        clusterNetworkAddressConfigProvider: (6)
          type: SniRoutingClusterNetworkAddressConfigProvider
          Config:
            bootstrapAddress: mycluster-proxy.kafka:9092
            brokerAddressPattern: broker$(nodeId).mycluster-proxy.kafka
        logNetwork: false (7)
        logFrames: false
        tls: (8)
          key:
            storeFile: /opt/proxy/server/key-material/keystore.p12
            storePassword:
              passwordFile: /opt/proxy/server/keystore-password/storePassword
filters: (9)
  - type: EnvelopeEncryption (10)
    config: (11)
      kms: VaultKmsService
```

```
    kmsConfig:
      vaultTransitEngineUrl: https://vault.vault.svc.cluster.local:8200/v1/transit
      vaultToken:
        passwordFile: /opt/proxy/server/token.txt
      tls: 12
        key:
          storeFile: /opt/cert/server.p12
          storePassword:
            passwordFile: /opt/cert/store.password
          keyPassword:
            passwordFile: /opt/cert/key.password
          storeType: PKCS12
    selector: TemplateKekSelector
    selectorConfig:
      template: "${topicName}"
```

1 プロキシーのメトリクスを有効にします。

2 仮想クラスター設定。

3 仮想クラスターの名前。

4 プロキシーされるターゲットの物理 Kafka クラスターのブートストラップアドレス。

5 ターゲットクラスターへの接続用の TLS 設定。

6 仮想クラスターがネットワークに提示される方法を制御するクラスターネットワークアドレス設定プロバイダーの設定。

7 デフォルトではログ記録は無効になっています。ログプロパティーを **true** に設定して、ネットワークアクティビティー (**logNetwork**) およびメッセージ (**logFrames**) に関連するログを有効にします。

8 クライアントとの接続を保護するための TLS 暗号化。

9 フィルターの設定。

10 フィルターのタイプ。この例では、Record Encryption フィルターです。

11 フィルターのタイプに固有の設定。

12 Record Encryption フィルターには Vault への接続が必要です。必要に応じて、TLS 証明書が保存される鍵の名前を使用して、Vault による TLS 認証の認証情報を指定することもできます。

4.2. 仮想クラスターの設定

Kafka クラスターは、プロキシーによって仮想クラスターとして表されます。クライアントは実際のクラスターではなく仮想クラスターに接続します。Streams for Apache Kafka Proxy をデプロイすると、プロキシーに仮想クラスターを作成するための設定が含まれています。

仮想クラスターにはターゲットクラスターが 1 つだけありますが、複数の仮想クラスターが同じクラスターをターゲットにすることができます。各仮想クラスターはターゲットクラスター上の単一のリスナーをターゲットとするため、Kafka 側の複数のリスナーはプロキシーによって複数の仮想クラスターとして表されます。クライアントは、**bootstrap_servers** アドレスを使用して仮想クラスターに接続します。仮想クラスターには、ターゲットクラスター内の各ブローカーにマップされるブートストラップ

アドレスがあります。クライアントがプロキシーに接続すると、アドレスを書き換えることで通信がターゲットブローカーにプロキシーされます。クライアントへの応答は、仮想クラスターの適切なネットワークアドレスを反映するように書き換えられます。

クライアントからターゲットクラスターへの仮想クラスター接続を保護できます。

Streams for Apache Kafka Proxy では、PEM (Privacy Enhanced Mail)、PKCS #12 (Public-Key Cryptography Standards)、または JKS (Java KeyStore) キーストア形式の鍵と証明書を使用できます。

4.2.1. クライアントからの接続の保護

仮想クラスターへのクライアント接続を保護するには、次の手順を実行して仮想クラスターで TLS を設定します。

認証局から仮想クラスターの CA (証明局) 証明書を取得します。証明書を要求するときは、それが仮想クラスターのブートストラップとブローカーのアドレスの名前と一致していることを確認してください。これには、ワイルドカード証明書とサブジェクト別名 (SAN) が必要になる場合があります。

tls プロパティーを使用して、仮想クラスター設定で TLS 認証情報を指定します。

PKCS #12 設定の例

```
virtualClusters:
  my-cluster-proxy:
    tls:
      key:
        storeFile: <path>/server.p12 (1)
        storePassword:
          passwordFile: <path>/store.password (2)
        keyPassword:
          passwordFile: <path>/key.password (3)
        storeType: PKCS12 (4)
      # ...
```

1. 仮想クラスターのパブリック CA 証明書用の PKCS #12 ストア。
2. PKCS #12 ストアを保護するためのパスワード。
3. (オプション) 鍵のパスワード。パスワードが指定されていない場合は、キーストアのパスワードも鍵の復号化に使用されます。
4. (オプション) キーストアのタイプ。キーストアのタイプが指定されていない場合は、デフォルトの JKS (Java Keystore) タイプが使用されます。

注記

実稼働設定では、Kafka クライアントと仮想クラスターで TLS が推奨されます。

PEM 設定の例

```
virtualClusters:
  my-cluster-proxy:
```

```
    tls:
      key:
        privateKeyFile: <path>/server.key (1)
        certificateFile: <path>/server.crt (2)
        keyPassword:
          passwordFile: <path>/key.password (3)
# …
```

1. 仮想クラスターの秘密鍵。
2. 仮想クラスターのパブリック CA 証明書。
3. (オプション) 鍵のパスワード。

必要に応じて、**insecure** プロパティーを設定して信頼を無効にし、証明書の有効性に関係なく、Kafka クラスターとの安全でない接続を確立します。ただし、このオプションは実稼働環境での使用には推奨されません。

安全でない TLS を有効にする例

```
virtualClusters:
  demo:
    targetCluster:
      bootstrap_servers: myprivatecluster:9092
      tls:
        trust:
          insecure: true (1)
      #...
# …
```

1. 安全でない TLS を有効にします。

4.2.2. ターゲットクラスターへの接続のセキュリティー保護

ターゲットクラスターへの仮想クラスター接続を保護するには、仮想クラスターで TLS を設定します。ターゲットクラスターは、TLS を使用するようにすでに設定されている必要があります。

targetCluster.tls プロパティーを使用して仮想クラスター設定の TLS を指定します。

OpenShift プラットフォームから信頼を継承するには、空のオブジェクト (**{}**) を使用します。このオプションは、ターゲットクラスターがパブリック CA によって署名された TLS 証明書を使用している場合に適しています。

TLS のターゲットクラスター設定の例

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
      bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093
      tls: {}
      #...
```

プライベート CA によって署名された TLS 証明書を使用している場合は、ターゲットクラスターのトラストストア設定を追加する必要があります。

ターゲットクラスターのトラストストア設定の例

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
      bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093
      tls:
        trust:
          storeFile: <path>/trust.p12 (1)
          storePassword:
            passwordFile: <path>/store.password (2)
          storeType: PKCS12 (3)
      #...
```

1. Kafka クラスターのパブリック CA 証明書用の PKCS #12 ストア。
2. パブリック Kafka クラスター CA 証明書にアクセスするためのパスワード。
3. (オプション) キーストアのタイプ。キーストアのタイプが指定されていない場合は、デフォルトの JKS (Java Keystore) タイプが使用されます。

mTLS の場合、仮想クラスターのキーストア設定も追加できます。

mTLS のキーストアとトラストストアの設定例

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
      bootstrap_servers: my-cluster-kafka-bootstrap.kafka.svc.cluster.local:9093:9092
      tls:
        key:
          privateKeyFile: <path>/client.key (1)
          certificateFile: <path>/client.crt (2)
        trust:
          storeFile: <path>/server.crt
          storeType: PEM
# ...
```

1. 仮想クラスターの秘密鍵。
2. 仮想クラスターのパブリック CA 証明書。

実稼働環境外でテストする目的で、**insecure** プロパティーを **true** に設定して TLS をオフにし、Streams for Apache Kafka Proxy が任意の Kafka クラスターに接続できるようにすることができます。

TLS をオフにする設定例

```
virtualClusters:
  my-cluster-proxy:
    targetCluster:
```

```
    bootstrap_servers: myprivatecluster:9092
    tls:
      trust:
        insecure: true
    #...
```

4.3. ネットワークアドレスの設定

仮想クラスターの設定には、ネットワーク通信を管理し、ブローカーアドレス情報をクライアントに提供するネットワークアドレス設定プロバイダーが必要です。

Streams for Apache Kafka Proxy には 2 つの組み込みプロバイダーがあります。

ブローカーアドレスプロバイダー (PortPerBrokerClusterNetworkAddressConfigProvider)

ブローカーごとのネットワークアドレス設定プロバイダーは、仮想クラスターのブートストラップアドレス用に 1 つのポートと、ターゲット Kafka クラスター内のブローカーごとに 1 つのポートを開きます。ポートは動的に維持されます。たとえば、ブローカーがクラスターから削除されると、そのブローカーに割り当てられているポートは閉じられます。

SNI ルーティングアドレスプロバイダー (SniRoutingClusterNetworkAddressConfigProvider)

SNI ルーティングプロバイダーは、すべての仮想クラスターに対して 1 つのポートを開くか、またはそれぞれに 1 つのポートを開きます。Kafka クラスターの場合、クラスター全体または各ブローカーのポートを開くことができます。SNI ルーティングプロバイダーは、SNI 情報を使用してトラフィックをルーティングする場所を決定します。

ブローカーアドレスプロバイダーの設定例

```
clusterNetworkAddressConfigProvider:
  type: PortPerBrokerClusterNetworkAddressConfigProvider
  config:
    bootstrapAddress: mycluster.kafka.com:9192 (1)
    brokerAddressPattern: mybroker-$(nodeId).mycluster.kafka.com (2)
    brokerStartPort: 9193 (3)
    numberOfBrokerPorts: 3 (4)
    bindAddress: 192.168.0.1 (5)
```

(1) Kafka クライアントが使用するブートストラップアドレスのホスト名とポート。

(2) (オプション) ブローカーアドレスを形成するために使用されるブローカーアドレスパターン。定義されていない場合は、デフォルトでブートストラップアドレスのホスト名部分とブローカーに割り当てられたポート番号になります。$(nodeId) トークンは、ブローカーの **node.id** (**node.id** が設定されていない場合は **broker.id**) に置き換えられます。

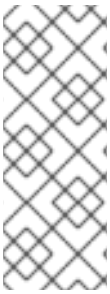
(3) (オプション) ブローカーポート範囲の開始番号。デフォルトはブートストラップアドレスのポートに 1 を加えた値です。

(4) (オプション) 使用可能なブローカーポートの最大数。デフォルトは 3 です。

(5) (オプション) ポートをバインドするときに使用するバインドアドレス。定義されていない場合は、すべてのネットワークインターフェイスがバインドされます。

ブローカーアドレス設定の例では、次のブローカーアドレスが作成されます。

```
mybroker-0.mycluster.kafka.com:9193
mybroker-1.mycluster.kafka.com:9194
mybroker-2.mycluster.kafka.com:9194
```



注記

複数の物理クラスターを含む設定の場合、**numberOfBrokerPorts** が (ブローカーの数 * ブローカーあたりのリスナーの数) + すべてのクラスターのブートストラップリスナーの数に設定されていることを確認します。たとえば、物理クラスターが 2 つあり、それぞれにノードが 3 つ含まれているとします。各ブローカーにリスナーが 1 つある場合、設定の値として 8 が必要です (ブローカーリスナー用の 3 つのポートと各クラスターのブートストラップリスナー用の 1 つのポートで構成)。

SNI ルーティングアドレスプロバイダーの設定例

```
clusterNetworkAddressConfigProvider:
  type: SniRoutingClusterNetworkAddressConfigProvider
  config:
    bootstrapAddress: mycluster.kafka.com:9192 (1)
    brokerAddressPattern: mybroker-$(nodeId).mycluster.kafka.com
    bindAddress: 192.168.0.1
```

1. ブートストラップアドレスとブローカーを含むすべてのトラフィック用の単一のアドレス。

SNI ルーティングアドレス設定では、各ブローカーのルートを生成するために **brokerAddressPattern** の指定が必須です。

第5章 メトリクスの概要

Streams for Apache Kafka Proxy のデプロイメントにメトリクスを導入する場合は、セキュアでない HTTP および Prometheus エンドポイント (/**metrics**) を設定できます。

Streams for Apache Kafka Proxy 設定を定義する **ConfigMap** リソースに以下を追加します。

最小限のメトリクス設定

```
adminHttp:
  endpoints:
    prometheus: {}
```

デフォルトでは、HTTP エンドポイントは **0.0.0.0:9190** をリッスンします。ホスト名とポートは次のように変更できます。

ホスト名とポートを使用したメトリクス設定の例

```
adminHttp:
  host: localhost
  port: 9999
  endpoints:
    prometheus: {}
```

プロキシーに付属するサンプルファイルには、**PodMonitor** リソースが含まれています。OpenShift でユーザー定義プロジェクトの監視を有効にしている場合は、**PodMonitor** リソースを使用してプロキシーメトリクスを取り込むことができます。

PodMonitor リソース設定の例

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: proxy
  labels:
    app: proxy
spec:
  selector:
    matchLabels:
      app: proxy
  namespaceSelector:
    matchNames:
      - proxy
  podMetricsEndpoints:
  - path: /metrics
    port: metrics
```

付録A サブスクリプションの使用

Streams for Apache Kafka は、ソフトウェアサブスクリプションを通じて提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **My Subscriptions** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

Zip および Tar ファイルのダウンロード

zip または tar ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **INTEGRATION AND AUTOMATION** カテゴリーで、**Streams for Apache Kafka** エントリーを見つけます。
3. 必要な Streams for Apache Kafka 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

DNF を使用したパッケージのインストール

パッケージとすべてのパッケージ依存関係をインストールするには、以下を使用します。

```
dnf install <package_name>
```

ローカルディレクトリーからダウンロード済みのパッケージをインストールするには、以下を使用します。

```
dnf install <path_to_download_package>
```

改訂日時: 2024-07-04