Red Hat Integration 2020.q1

Debezium スタートガイド

Debezium 1.0 向け

Last Updated: 2023-02-28

Red Hat Integration 2020.q1 Debezium スタートガイド

Debezium 1.0 向け

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、Debezium を使用する方法を説明します。

目次

前書き

このチュートリアルでは、Debezium を使用して MySQL データベースを監視する方法を説明します。データベースのデータが変更されると、生成されるイベントストリームが表示されます。

このチュートリアルでは、OpenShift で Debezium サービスを開始し、データベース例を使用して MySQL データベースサーバーを実行し、Debezium を使用して変更のためにデータベースを監視します。

前提条件

Debezium を使用して MySQL データベースを監視する前に、以下が必要です。

- **cluster-admin** 権限での OpenShift Container Platform 4.x クラスターへのアクセス
- AMQ Streams 1.4 OpenShift インストールおよびサンプルファイル。
  AMQ Streams のダウンロードサイト からファイルをダウンロードできます。
- Debezium 1.0.0 MySQL コネクター
  Red Hat Integration のダウンロードサイト からファイルをダウンロードできます。

注記

上記は MySQL コネクターを対象とする前提条件です。その他の Debezium コネクターの前提条件は異なる場合があります。

第1章 DEBEZIUM の紹介

Debezium は、既存のデータベースをイベントストリームに変える分散型プラットフォームで、アプリケーションはデータベースの行レベルの変更を即座に確認し、対応することが可能になります。

Debezium は Apache Kafka 上に構築され、特定のデータベース管理システムを監視する Kafka Connect 対応のコネクターが提供されます。Debezium では、アプリケーションがデータを使用する場所から、データ変更の履歴が Kafka のログに記録されます。これにより、アプリケーションはすべてのイベントを簡単、正確、かつ完全に使用することができます。アプリケーションが予期せず停止しても、見逃すものはありません。アプリケーションが再起動すると、停止した場所からイベントの使用を再開します。

Debezium には、複数のコネクターが含まれています。このチュートリアルでは、MySQL コネクターを使用します。

第2章 サービスの起動

Debezium を使用するには、AMQ Streams と Debezium コネクターサービスが必要です。このチュートリアルに必要なサービスを起動するには、以下を行う必要があります。

1. AMQ Streams を使用した OpenShift での単一ノード Kafka クラスターの設定
2. Debezium MySQL Connector プラグインでの Kafka Connect のデプロイ
3. MySQL データベースのデプロイ

2.1. KAFKA クラスターの設定

AMQ Streams を使用して Kafka クラスターを設定します。この手順では、単一ノードの Kafka クラスターをデプロイします。

手順

1. OpenShift 4.x クラスターで、新しいプロジェクトを作成します。

```
$ oc new-project cdc-tutorial
```

2. AMQ Streams 1.4 OpenShift インストールとサンプルファイルをダウンロードしたディレクトリーに移動します。

3. AMQ Streams Cluster Operator をデプロイします。
Cluster Operator は、OpenShift クラスター内で Kafka クラスターのデプロイおよび管理を行います。このコマンドは、Cluster Operator をデプロイし、作成したプロジェクトのみを監視します。

```
$ sed -i 's/namespace: .*/namespace: cdc-tutorial/' install/cluster-operator/*RoleBinding*.yaml

$ oc apply -f install/cluster-operator -n cdc-tutorial
```

4. Cluster Operator が稼働していることを確認します。
このコマンドを実行すると、Cluster Operator は稼働中で、すべての Pod の準備ができていることを確認できます。

```
$ oc get pods
NAME                                        READY   STATUS    RESTARTS   AGE
strimzi-cluster-operator-5c6d68c54-l4gdz    1/1     Running   0          46s
```

5. Kafka クラスターをデプロイします。
このコマンドは、**kafka-ephemeral-single.yaml** カスタムリソースを使用して、3 つの ZooKeeper ノードと 1 つの Kafka ノードを持つ Kafka の一時クラスターを作成します。

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

6. Kafka クラスターが稼働していることを確認します。
このコマンドを実行すると、Kafka クラスターは稼働中で、すべての Pod の準備ができていることを確認できます。

```
$ oc get pods
```

```
NAME                                          READY   STATUS    RESTARTS   AGE
my-cluster-entity-operator-5b5d4f7c58-8gnq5   3/3     Running   0          41s
my-cluster-kafka-0                            2/2     Running   0          70s
my-cluster-zookeeper-0                        2/2     Running   0          107s
my-cluster-zookeeper-1                        2/2     Running   0          107s
my-cluster-zookeeper-2                        2/2     Running   0          107s
strimzi-cluster-operator-5c6d68c54-l4gdz      1/1     Running   0          8m53s
```

2.2. KAFKA CONNECT のデプロイ

Kafka クラスターの設定後、Kafka Connect Source-to-Image(S2I)サービスをデプロイします。このサービスは、Debezium MySQL コネクターを管理するためのフレームワークを提供します。

手順

1. Kafka Connect Source-to-Image(S2I)サービスをデプロイします。
   このコマンドは、単一ノードの Kafka クラスターの YAML ファイルのサンプルを使用して Kafka Connect S2I サービスをデプロイします。

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

2. Kafka Connect サービスが実行していることを確認します。
   このコマンドを実行すると、Kafka Connect サービスが稼働中で、Pod の準備ができていることを確認できます。

```
$ oc get pods -l strimzi.io/name=my-connect-cluster-connect
NAME                                     READY   STATUS    RESTARTS   AGE
my-connect-cluster-connect-1-dxcs9       1/1     Running   0          7m
```

3. Debezium MySQL Connector プラグインを使用して、Kafka Connect イメージの新しいビルドを開始します。
   このコマンドは、以前にダウンロードした Debezium MySQL Connector プラグインを使用します。

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

4. ビルドが完了したことを確認します。
   このコマンドは、新しいビルドが完了したことを示しています(**my-connect-cluster-connect-2**)。Debezium MySQL Connector がインストールされている。

```
$ oc get build
NAME                         TYPE     FROM     STATUS     STARTED          DURATION
my-connect-cluster-connect-1 Source            Complete   9 minutes ago    2m10s
my-connect-cluster-connect-2 Source   Binary   Complete   4 minutes ago    2m2s
```

2.3. MYSQL データベースのデプロイ

この時点で、Kafka クラスターおよび Kafka Connect サービスが Debezium MySQL Database Connector にデプロイされました。ただし、Debezium による変更のキャプチャーを可能にするデータベースサーバーが必要です。この手順では、サンプルデータベースを使用して MySQL サーバーを起動します。

手順

1. MySQL データベースを起動します。
   このコマンドにより、サンプルの **インベントリー** データベースで事前に設定された MySQL データベースサーバーを起動します。

```
$ oc new-app --name=mysql debezium/example-mysql:1.0
```

2. MySQL データベースのクレデンシャルを設定します。
   このコマンドにより、MySQL データベースのデプロイメント設定が更新され、ユーザー名とパスワードが追加されます。

```
$ oc set env dc/mysql MYSQL_ROOT_PASSWORD=debezium  MYSQL_USER=mysqluser
MYSQL_PASSWORD=mysqlpw
```

3. MySQL データベースが実行中であることを確認します。
   このコマンドにより、MySQL データベースが実行され、Pod の準備ができていることを確認できます。

```
$ oc get pods -l app=mysql
NAME           READY   STATUS    RESTARTS   AGE
mysql-1-2gzx5   1/1     Running   1          23s
```

4. 新しいターミナルを開き、**inventory** データベースのサンプルにログインします。
   このコマンドは、MySQL データベースを実行している Pod で MySQL コマンドラインクライアントを開きます。以前に設定したユーザー名とパスワードを使用します。

```
$ oc exec mysql-1-2gzx5 -it -- mysql -u mysqluser -p mysqlpw inventory
mysql: [Warning] Using a password on the command line interface can be insecure.
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.7.29-log MySQL Community Server (GPL)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

5. **inventory** データベースのテーブルを一覧表示します。

```
mysql> show tables;
+----------------------+
| Tables_in_inventory |
+----------------------+
| addresses           |
| customers           |
```

```
| geom                |
| orders              |
| products            |
| products_on_hand    |
+---------------------+
6 rows in set (0.00 sec)
```

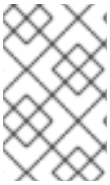
6. データベースを調べ、事前に読み込んだデータを表示します。
以下の例は、customer テーブルを示しています。

```
mysql> select * from customers;
+------+------------+-----------+-----------------------+
| id   | first_name | last_name | email                 |
+------+------------+-----------+-----------------------+
| 1001 | Sally      | Thomas    | sally.thomas@acme.com |
| 1002 | George     | Bailey    | gbailey@foobar.com    |
| 1003 | Edward     | Walker    | ed@walker.com         |
| 1004 | Anne       | Kretchmar | annek@noanswer.org    |
+------+------------+-----------+-----------------------+
4 rows in set (0.00 sec)
```

第3章 インベントリー データベースを監視するコネクターの作成

Kafka、Debezium、および MySQL サービスを起動すると、コネクターインスタンスを作成して **インベントリー** データベースを監視する準備が整います。

この手順では、コネクターインスタンスを定義する **KafkaConnector** カスタムリソース(CR)を作成して適用することで、コネクターインスタンスを作成します。CR の適用後、コネクターインスタンスは **インベントリー** データベースの **binlog** の監視を開始します。**binlog** は、データベースのトランザクション（個別の行の変更やスキーマの変更など）をすべて記録します。データベースの行が変更されると、Debezium は変更イベントを生成します。



注記

通常、Kafka ツールを使用して、レプリカ数の指定など、必要なトピックを手動で作成します。ただし、このチュートリアルでは、1 つのレプリカのみでトピックを自動作成するように Kafka が設定されています。

手順

1. Kafka Connect のデプロイに使用した **examples/kafka-connect/kafka-connect-s2i-single-node-kafka.yaml** ファイルを開きます。
   MySQL コネクターインスタンスを作成する前に、**KafkaConnectS2I** カスタムリソース(CR)でコネクターリソースを有効にする必要があります。

2. **metadata.annotations** セクションで、Kafka Connect がコネクターリソースを使用できるようにします。
   この例では、アノテーションを **examples/kafka-connect/kafka-connect-s2i-single-node-kafka.yaml** ファイルに追加します。

   kafka-connect-s2i-single-node-kafka.yaml

   ```
   apiVersion: kafka.strimzi.io/v1beta1
   kind: KafkaConnectS2I
   metadata:
     name: my-connect-cluster
     annotations:
       strimzi.io/use-connector-resources: "true"
   spec:
     ...
   ```

3. 更新された **kafka-connect-s2i-single-node-kafka.yaml** ファイルを適用して **KafkaConnectS2I** CR を更新します。

   ```
   $ oc apply -f kafka-connect-s2i-single-node-kafka.yaml
   ```

4. MySQL コネクターインスタンスを作成し、**インベントリー** データベースを監視します。
   この例では、MySQL コネクターインスタンスを定義する **KafkaConnector** CR を作成します。

   inventory-connector.yaml

   ```
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnector
   metadata:
     name: inventory-connector (1)
   ```

```
    labels:
      strimzi.io/cluster: my-connect-cluster
  spec:
    class: io.debezium.connector.mysql.MySqlConnector
    tasksMax: 1 (2)
    config: (3)
      database.hostname: mysql (4)
      database.port: 3306
      database.user: debezium
      database.password: dbz
      database.server.id: 184054 (5)
      database.server.name: dbserver1 (6)
      database.whitelist: inventory (7)
      database.history.kafka.bootstrap.servers: my-cluster-kafka-bootstrap:9092 (8)
      database.history.kafka.topic: schema-changes.inventory (9)
```

(1) コネクターの名前。

(2) 1 度に 1 つのタスクのみが動作する必要があります。MySQL コネクターは MySQL サーバーの **binlog** を読み取るため、単一のコネクタータスクを使用することで、順序とイベントの処理が適切に行われるようになります。Kafka Connect サービスはコネクターを使用して作業を行う 1 つ以上のタスクを開始し、実行中のタスクを自動的に Kafka Connect サービスのクラスター全体に分散します。いずれかのサービスが停止またはクラッシュすると、これらのタスクは稼働中のサービスに再分散されます。

(3) コネクターの設定。

(4) データベースホスト。これは、MySQL サーバーを実行しているコンテナーの名前です (**mysql**)。

(5) (6) 一意なサーバー ID および名前。サーバー名は、MySQL サーバーまたはサーバーのクラスターの論理識別子です。この名前は、すべての Kafka トピックのプレフィックスとして使用されます。

(7) **inventory** データベースの変更のみが検出されます。

(8) (9) コネクターは、このブローカー (イベントの送信先となるブローカーと同じ) とトピック名を使用して、データベーススキーマの履歴を Kafka に保存します。再起動時に、コネクターは、コネクターが読み取りを開始すべき時点で **binlog** に存在したデータベースのスキーマを復元します。

5. コネクターインスタンスを適用します。

```
$ oc apply -f inventory-connector.yaml
```

inventory-connector コネクターは登録され、**inventory** データベースに対して実行が開始されます。

6. **inventory-connector** が作成され、**インベントリー** データベースの監視を開始したことを確認します。
Kafka Connect のログ出力を **inventory-connector** の起動時に監視して、コネクターインスタンスを確認することができます。

   a. Kafka Connect のログ出力を表示します。

```
$ oc logs $(oc get pods -o name -l strimzi.io/name=my-connect-cluster-connect)
```

b. ログの出力を確認し、初回のスナップショットが実行されたことを確認します。
以下の行は、初回のスナップショットが開始されたことを表しています。

```
...
2020-02-21 17:57:30,801 INFO Starting snapshot for jdbc:mysql://mysql:3306/?
useInformationSchema=true&nullCatalogMeansCurrent=false&useSSL=false&useUnicode=
true&characterEncoding=UTF-8&characterSetResults=UTF-
8&zeroDateTimeBehavior=CONVERT_TO_NULL&connectTimeout=30000 with user
'debezium' with locking mode 'minimal' (io.debezium.connector.mysql.SnapshotReader)
[debezium-mysqlconnector-dbserver1-snapshot]
2020-02-21 17:57:30,805 INFO Snapshot is using user 'debezium' with these MySQL
grants: (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-
dbserver1-snapshot]
...
```

スナップショットには、複数のステップが関係します。

```
...
2020-02-21 17:57:30,822 INFO Step 0: disabling autocommit, enabling repeatable read
transactions, and setting lock wait timeout to 10
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,836 INFO Step 1: flush and obtain global read lock to prevent
writes to database (io.debezium.connector.mysql.SnapshotReader) [debezium-
mysqlconnector-dbserver1-snapshot]
2020-02-21 17:57:30,839 INFO Step 2: start transaction with consistent snapshot
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,840 INFO Step 3: read binlog position of MySQL master
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:30,843 INFO   using binlog 'mysql-bin.000003' at position '154' and gtid
'' (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
...
2020-02-21 17:57:34,423 INFO Step 9: committing transaction
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
2020-02-21 17:57:34,424 INFO Completed snapshot in 00:00:03.632
(io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-
snapshot]
...
```

スナップショットの完了後、Debezium は変更イベントについて **インベントリー** データベースの **binlog** の監視を開始します。

```
...
2020-02-21 17:57:35,584 INFO Transitioning from the snapshot reader to the binlog
reader (io.debezium.connector.mysql.ChainedReader) [task-thread-inventory-connector-
0]
2020-02-21 17:57:35,613 INFO Creating thread debezium-mysqlconnector-dbserver1-
binlog-client (io.debezium.util.Threads) [task-thread-inventory-connector-0]
```

```
2020-02-21 17:57:35,630 INFO Creating thread debezium-mysqlconnector-dbserver1-
binlog-client (io.debezium.util.Threads) [blc-mysql:3306]
Feb 21, 2020 5:57:35 PM com.github.shyiko.mysql.binlog.BinaryLogClient connect
INFO: Connected to mysql:3306 at mysql-bin.000003/154 (sid:184054, cid:5)
2020-02-21 17:57:35,775 INFO Connected to MySQL binlog at mysql:3306, starting at
binlog file 'mysql-bin.000003', pos=154, skipping 0 events plus 0 rows
(io.debezium.connector.mysql.BinlogReader) [blc-mysql:3306]
...
```

第4章 変更イベントの表示

Debezium MySQL コネクターのデプロイ後、データ変更イベントの **インベントリー** データベースの監視が開始されます。

コネクターが起動すると、**dbserver1** 接頭辞（コネクターの名前）でイベントが以下のトピックに書き込まれたことを確認できます。

dbserver1

すべての DDL ステートメントが書き込まれるスキーマ変更トピック。

dbserver1.inventory.products

インベントリー データベースの **製品** テーブルの変更イベントを取得します。

dbserver1.inventory.products_on_hand

インベントリー データベースの **products_on_hand** テーブルの変更イベントを取得します。

dbserver1.inventory.customers

インベントリー データベースの **顧客** テーブルの変更イベントをキャプチャーします。

dbserver1.inventory.orders

インベントリー データベース内の **orders** テーブルの変更イベントをキャプチャーします。

このチュートリアルでは、**dbserver1.inventory.customers** トピックについて詳しく見ていきます。このトピックでは、異なるタイプの変更イベントを表示し、コネクターがそれらのイベントをキャプチャーする方法を確認します。

- **作成** イベントの表示
- データベースの更新および **更新** イベントの表示
- データベースのレコードの削除および **削除** イベントの表示
- Kafka Connect の再起動およびデータベースの変更

4.1. 作成 イベントの表示

dbserver1.inventory.customers トピックを表示すると、MySQL コネクターが **inventory** データベースの **作成** イベントをどのようにキャプチャーしたが分かります。この場合、**作成** イベントは、データベースに追加された新規顧客をキャプチャーします。

手順

1. 新しいターミナルを開き、**kafka-console-consumer** を使用してトピックの最初から **dbserver1.inventory.customers** トピックを使用します。
このコマンドは、Kafka (**my-cluster-kafka-0**) を実行している Pod で簡単なコンシューマー (**kafka-console-consumer.sh**) を実行します。

```
$ oc exec -it my-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-consumer.sh \
  --bootstrap-server localhost:9092 \
  --from-beginning \
  --property print.key=true \
  --topic dbserver1.inventory.customers
```

コンシューマーは、 **customers** テーブルの各行に 1 つずつ、4 つのメッセージ (JSON 形式) を返します。各メッセージには、対応するテーブル行のイベントレコードが含まれます。

各イベントには、**キー** と **値** という 2 つの JSON ドキュメントがあります。キーは行のプライマリーキーに対応し、値は行の詳細 (行に含まれるフィールド、各フィールドの値、および行で実行された操作のタイプ) を表します。

2. 最後のイベントでは、**キー** の詳細を確認します。
最後のイベントの **キー** の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema":{
    "type":"struct",
      "fields":[
        {
          "type":"int32",
          "optional":false,
          "field":"id"
        }
      ],
    "optional":false,
    "name":"dbserver1.inventory.customers.Key"
  },
  "payload":{
    "id":1004
  }
}
```

このイベントには、**schema** と **payload** の 2 つの部分があります。**schema** には、ペイロードの内容を記述する Kafka Connect スキーマが含まれています。この場合、ペイロードは **dbserver1.inventory.customers.Key** という名前の**構造**で、これはオプションではなく、必須フィールドが 1 つあります（タイプ **int32**の**ID** ）。

payload には、値が **1004** の **id** フィールドが 1 つあります。

イベントの **key** を確認すると、このイベントは **id** の主キー列の値が **1004** である **inventory.customers** テーブルの行に提供されることが分かります。

3. 同じイベントの **値** の詳細を確認します。
イベントの **値** は、行が作成されたことを示し、その行に含まれる内容が記載されています (この場合は挿入された行の **id**、**first_name**、**last_name**、および **email**)。

最後のイベントの **値**の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
```

```
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "dbserver1.inventory.customers.Value",
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "dbserver1.inventory.customers.Value",
        "field": "after"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "string",
            "optional": true,
            "field": "version"
          },
          {
            "type": "string",
            "optional": false,
```

```
            "field": "name"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "server_id"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "ts_sec"
          },
          {
            "type": "string",
            "optional": true,
            "field": "gtid"
          },
          {
            "type": "string",
            "optional": false,
            "field": "file"
          },
          {
            "type": "int64",
            "optional": false,
            "field": "pos"
          },
          {
            "type": "int32",
            "optional": false,
            "field": "row"
          },
          {
            "type": "boolean",
            "optional": true,
            "field": "snapshot"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "db"
          },
          {
            "type": "string",
            "optional": true,
            "field": "table"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mysql.Source",
        "field": "source"
```

```
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "dbserver1.inventory.customers.Envelope",
    "version": 1
  },
  "payload": {
    "before": null,
    "after": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.0.3.Final",
      "name": "dbserver1",
      "server_id": 0,
      "ts_sec": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "snapshot": true,
      "thread": null,
      "db": "inventory",
      "table": "customers"
    },
    "op": "c",
    "ts_ms": 1486500577691
  }
}
```


イベントのこの部分ははるかに長くなりますが、イベントの **キー** と同様に **schema** と **payload** もあります。**schema** には、**dbserver1.inventory.customers.Envelope** (バージョン 1)という名前の Kafka Connect スキーマが含まれており、5 つのフィールドを追加できます。

op

操作のタイプを記述する文字列値が含まれる必須フィールド。MySQL コネクターの値は、**c** (作成または挿入)、**u** (更新)、**d** (削除)、および **r** (読み取り、初回のスナップショットでない場合) です。

before

任意のフィールド。存在する場合は、イベント発生**前**の行の状態が含まれます。この構造は、**dbserver1.inventory.customers.Value** Kafka Connect スキーマによって記述され、**dbserver1** コネクターは **inventory.customers** テーブルのすべての行に使用します。

after

任意のフィールド。存在する場合は、イベント発生**後**の行の状態が含まれます。この構造は、**before** で使用されるのと同じ **dbserver1.inventory.customers.Value** Kafka Connect スキーマで記述されます。

source

イベントのソースメタデータを記述する構造が含まれる必須のフィールド。MySQL の場合は複数のフィールドが含まれます: コネクター名、イベントが記録された **binlog** ファイルの名前、**binlog** ファイルでのイベント発生位置、イベント内の行 (複数ある場合)、影響を受けるデータベースおよびテーブルの名前、変更を行った MySQL スレッド ID、このイベントはスナップショットの一部であったかどうか、MySQL サーバー ID (ある場合)、および秒単位のタイムスタンプ。

ts_ms

任意のフィールド。存在する場合は、コネクターがイベントを処理した時間 (Kafka Connect タスクを実行する JVM のシステムクロックを使用) が含まれます。

注記

イベントの JSON 表現は、記述される行よりもはるかに長くなります。これは、すべてのイベントキーと値で Kafka Connect は **ペイロード** を記述する **スキーマ** を提供するためです。今後、この構造が変更される可能性があります。ただし、特に使用する側のアプリケーションが時間とともに進化する場合は、キーと値のスキーマがイベント自体にあると、メッセージを理解するのが非常に容易になります。

Debezium MySQL コネクターは、データベーステーブルの構造に基づいてこれらのスキーマを構築します。DDL ステートメントを使用して MySQL データベースのテーブル定義を変更する場合、コネクターはこれらの DDL ステートメントを読み取り、Kafka Connect スキーマを更新します。これは、イベント発生時にイベントの発生元となったテーブルと全く同じように、各イベントが構造化される唯一の方法です。ただし、単一テーブルのイベントがすべて含まれる Kafka トピックには、テーブル定義の各状態に対応するイベントが含まれる場合があります。

JSON コンバーターにはすべてのメッセージのキーおよび値スキーマが含まれるため、非常に詳細なイベントを生成します。

4. イベントの **キー** および **値** スキーマを、**inventory** データベースの状態と比較します。MySQL コマンドラインクライアントを実行しているターミナルで、以下のステートメントを実行します。

```
mysql> SELECT * FROM customers;
+------+------------+-----------+-----------------------+
| id   | first_name | last_name | email                 |
+------+------------+-----------+-----------------------+
| 1001 | Sally      | Thomas    | sally.thomas@acme.com |
| 1002 | George     | Bailey    | gbailey@foobar.com    |
| 1003 | Edward     | Walker    | ed@walker.com         |
| 1004 | Anne       | Kretchmar | annek@noanswer.org    |
+------+------------+-----------+-----------------------+
4 rows in set (0.00 sec)
```

これは、確認したイベントレコードがデータベースのレコードと一致することを示しています。

4.2. データベースの更新および 更新 イベントの表示

Debezium MySQL コネクターが **inventory** データベースで **作成** イベントをキャプチャーする方法を確認しました。次に、レコードの1つを変更し、コネクターがこれをどのようにキャプチャーするかを見てみましょう。

この手順を完了すると、データベースのコミットで変更した内容の詳細を確認する方法と、変更イベントを比較して、他の変更と関連していつ変更が発生したかを判断する方法について学ぶことができます。

手順

1. MySQL コマンドラインクライアントを実行しているターミナルで、以下のステートメントを実行します。

```
mysql> UPDATE customers SET first_name='Anne Marie' WHERE id=1004;
Query OK, 1 row affected (0.05 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

2. 更新された **customers** テーブルを表示します。

```
mysql> SELECT * FROM customers;
+------+------------+-----------+-----------------------+
| id   | first_name | last_name | email                 |
+------+------------+-----------+-----------------------+
| 1001 | Sally      | Thomas    | sally.thomas@acme.com |
| 1002 | George     | Bailey    | gbailey@foobar.com    |
| 1003 | Edward     | Walker    | ed@walker.com         |
| 1004 | Anne Marie | Kretchmar | annek@noanswer.org    |
+------+------------+-----------+-----------------------+
4 rows in set (0.00 sec)
```

3. **kafka-console-consumer** を実行しているターミナルに切り替え、**新しい** 5 番目のイベントを確認します。
customers テーブルのレコードを変更することで、Debezium MySQL コネクターは新しいイベントを生成しました。新しい JSON ドキュメントが 2 つあるはずです。1つはイベントの **キー** のドキュメントで、もう1つは新しいイベントの **値** のドキュメントです。

   更新 イベントの **キー** の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": {
    "type": "struct",
    "name": "dbserver1.inventory.customers.Key"
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
  "payload": {
```

```
      "id": 1004
    }
  }
```

この **キー** は、以前のイベントの **キー** と同じです。

新しいイベントの **値** は次のとおりです。**schema** セクションには変更がないため、**payload** セクションのみを表しています (書式を調整して読みやすくしてあります)。

```
{
 "schema": {...},
 "payload": {
  "before": { 1
    "id": 1004,
    "first_name": "Anne",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "after": { 2
    "id": 1004,
    "first_name": "Anne Marie",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "source": { 3
    "name": "1.0.3.Final",
    "name": "dbserver1",
    "server_id": 223344,
    "ts_sec": 1486501486,
    "gtid": null,
    "file": "mysql-bin.000003",
    "pos": 364,
    "row": 0,
    "snapshot": null,
    "thread": 3,
    "db": "inventory",
    "table": "customers"
  },
  "op": "u", 4
  "ts_ms": 1486501486308 5
 }
}
```

1. **before** フィールドは、データベースのコミット前の行と値の状態を表しています。
2. **after** フィールドは、更新された行の状態を表し、**first_name** の値は **Anne Marie** になっています。
3. **source** フィールド構造には以前と同じ値が多数ありますが、**ts_sec** および **pos** フィールドは更新されています (他の状況では **file** が変更されることがあります)。
4. **op** フィールドの値は **u** になっており、更新によってこの行が変更されたことを示しています。
5. The **ts_ms** フィールドは、Debezium がこのイベントを処理したときのタイムスタンプを示します。

payload セクションを表示すると、**更新** イベントに関する重要な情報を確認できます。

- **before** と **after** 構造を比較することで、コミットが原因で影響を受けた行で実際に何が変更されたかを判断できます。
- **ソース** 構造を確認して、MySQL の変更の記録に関する情報を確認できます（トレーサビリティーを提供）。
- イベントの **payload** セクションと、同じトピック（または別のトピック）の他のイベントを比較することで、別のイベントと同じ MySQL コミットの前、後、または一部としてイベントが発生したかどうかを判断できます。

4.3. データベースのレコードの削除および削除 イベントの表示

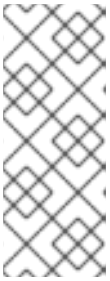
Debezium MySQL コネクターが **inventory** データベースで **作成** および **更新** イベントをキャプチャーする方法を確認しました。次に、レコードの 1 つを削除し、コネクターがこれをどのようにキャプチャーするかを見てみましょう。

この手順を完了すると、**削除** イベントの詳細を見つける方法と、Kafka が **ログコンパクション** を使用して、コンシューマーがすべてのイベントを取得できる状態で **削除** イベントの数を減らす方法を説明します。

手順

1. MySQL コマンドラインクライアントを実行しているターミナルで、以下のステートメントを実行します。

```
mysql> DELETE FROM customers WHERE id=1004;
Query OK, 1 row affected (0.00 sec)
```



注記

上記のコマンドが外部キー制約違反で失敗する場合は、以下のステートメントを使用して、**addresses** テーブルから顧客アドレスの参照を削除する必要があります。

```
mysql> DELETE FROM addresses WHERE customer_id=1004;
```

2. **kafka-console-consumer** を実行しているターミナルに切り替え、**2 つ** の新しいイベントを表示します。
customers テーブルの行を削除することで、Debezium MySQL コネクターは 2 つの新しいイベントを生成しました。

3. 最初の新規イベントの **キー** および **値** を確認します。
最初の新規イベントの **キー** の詳細は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": {
    "type": "struct",
    "name": "dbserver1.inventory.customers.Key"
```

```
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
  "payload": {
    "id": 1004
  }
}
```

この **キー** は、これまで確認した 2 つのイベントの **キー** と同じです。

最初の新規イベントの **値** は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": {...},
  "payload": {
    "before": { (1)
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null, (2)
    "source": { (3)
      "name": "1.0.3.Final",
      "name": "dbserver1",
      "server_id": 223344,
      "ts_sec": 1486501558,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 725,
      "row": 0,
      "snapshot": null,
      "thread": 3,
      "db": "inventory",
      "table": "customers"
    },
    "op": "d", (4)
    "ts_ms": 1486501558315 (5)
  }
}
```

(1) **before** フィールドは、データベースのコミットで削除した行の状態を表しています。

(2) **after** フィールドは **null** で、行が存在しなくなったことが分かります。

(3) **source** フィールド構造には以前と同じ値が多数ありますが、**ts_sec** および **pos** フィールドは更新されています (他の状況では **file** が変更されることがあります)。

(4) **op** フィールドの値は **d** になっており、この行が削除されたことを示しています。

5 The **ts_ms** フィールドは、Debezium がこのイベントを処理したときのタイムスタンプを示します。

よって、このイベントは、行の削除を処理に必要な情報をコンシューマーに提供します。古い値も提供されます。これは、コンシューマーによっては削除を適切に処理するのに古い値が必要になることがあるからです。

4. 2 つ目の新規イベントの **キー** および **値** を確認します。
2 つ目の新規イベントの **値** は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": {
    "type": "struct",
    "name": "dbserver1.inventory.customers.Key"
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
  "payload": {
    "id": 1004
  }
}
```

繰り返しになりますが、この **キー** は、これまで確認した 3 つのイベントのキーと同じです。

同じイベントの **値** は以下のとおりです (書式を調整して読みやすくしてあります)。

```
{
  "schema": null,
  "payload": null
}
```

Kafka が **ログコンパクション** に設定されている場合、トピックの後半に同じキーを持つメッセージが 1 つ以上あると、トピックから古いメッセージが削除されます。この最後のイベントには、キーと空の値があるため、**tombstone** (トゥームストーン) イベントと呼ばれます。これは、Kafka が同じキーを持つこれまでのメッセージをすべて削除することを意味します。これまでのメッセージが削除されても、tombstone イベントであるため、コンシューマーは最初からトピックを読み取ることができ、イベントを見逃しません。

4.4. KAFKA CONNECT サービスの再起動

Debezium MySQL コネクターが作成、更新、および削除イベントをキャプチャーする方法を確認しました。次に、稼働していない場合でもどのように変更イベントをキャプチャーするかを見てみましょう。

Kafka Connect サービスは、登録されたコネクターのタスクを自動的に管理します。したがって、オフラインになると、再起動時に稼働していないタスクがすべて開始されます。つまり、Debezium が稼働していない場合でも、変更をデータベースに報告できます。

この手順では、Kafka Connect を停止し、データベースのデータを一部変更した後、Kafka Connect を再起動して変更イベントを確認します。

手順

1. Kafka Connect サービスを停止します。
   a. Kafka Connect サービスのデプロイメント設定を開きます。

   ```
   $ oc edit dc/my-connect-cluster-connect
   ```

   デプロイメント設定が表示されます。

   ```
   apiVersion: apps.openshift.io/v1
   kind: DeploymentConfig
   metadata:
     ...
   spec:
     replicas: 1
   ...
   ```

   b. **spec.replicas** の値を **0** に変更します。

   c. デプロイメント設定を保存します。

   d. Kafka Connect サービスが停止したことを確認します。
   このコマンドを実行すると、Kafka Connect サービスが完了し、Pod が実行されていないことを確認できます。

   ```
   $ oc get pods -l strimzi.io/name=my-connect-cluster-connect
   NAME                                READY   STATUS      RESTARTS   AGE
   my-connect-cluster-connect-1-dxcs9          0/1     Completed   0          7h
   ```

2. Kafka Connect サービスが停止している間に、MySQL クライアントを実行しているターミナルに切り替え、新しいレコードをデータベースに追加します。

   ```
   mysql> INSERT INTO customers VALUES (default, "Sarah", "Thompson", "kitt@acme.com");
   ```

3. Kafka Connect サービスを再起動します。
   a. Kafka Connect サービスのデプロイメント設定を開きます。

   ```
   $ oc edit dc/my-connect-cluster-connect
   ```

   デプロイメント設定が表示されます。

   ```
   apiVersion: apps.openshift.io/v1
   kind: DeploymentConfig
   metadata:
     ...
   spec:
     replicas: 0
   ...
   ```

b. **spec.replicas** の値を **1** に変更します。

c. デプロイメント設定を保存します。

d. Kafka Connect サービスが再起動したことを確認します。
このコマンドを実行すると、Kafka Connect サービスが稼働中で、Pod の準備ができていることを確認できます。

```
$ oc get pods -l strimzi.io/name=my-connect-cluster-connect
NAME                                     READY   STATUS      RESTARTS   AGE
my-connect-cluster-connect-2-q9kkl       1/1     Running     0          74s
```

4. **kafka-console-consumer** を実行しているターミナルに切り替え、メッセージを確認します。
Kafka Connect がオフライン時に作成したレコードが表示されるはずです（書式を調整して読みやすくしてあります）。

```
{
  ...
  "payload":{
    "id":1005
  }
}
{
  ...
  "payload":{
    "before":null,
    "after":{
      "id":1005,
      "first_name":"Sarah",
      "last_name":"Thompson",
      "email":"kitt@acme.com"
    },
    "source":{
      "version":"{debezium-version}",
      "connector":"mysql",
      "name":"dbserver1",
      "ts_ms":1582581502000,
      "snapshot":"false",
      "db":"inventory",
      "table":"customers",
      "server_id":223344,
      "gtid":null,
      "file":"mysql-bin.000004",
      "pos":364,
      "row":0,
      "thread":5,
      "query":null
    },
    "op":"c",
    "ts_ms":1582581502317
  }
}
```

第5章 次のステップ

チュートリアルが完了したら、以下のステップを検討します。

- チュートリアルをさらに試してみる。
  MySQL コマンドラインクライアントを使用して、データベーステーブルの行を追加、変更、および削除し、トピックへの影響を確認します。外部キーによって参照される行は削除できないことに注意してください。

- Debezium のデプロイメントを計画する。
  Debezium を OpenShift または Red Hat Enterprise Linux にインストールできます。詳細は、以下を参照してください。

  - Debezium の OpenShift へのインストール

  - Debezium の RHEL へのインストール