Red Hat Integration 2020.Q1

Getting Started with Debezium

For use with Debezium 1.0
Abstract

This guide describes how to get started using Debezium.
# Table of Contents

PREFACE .................................................................................................................. 3

CHAPTER 1. INTRODUCTION TO DEBEZIUM .......................................................... 4

CHAPTER 2. STARTING THE SERVICES ................................................................... 5
  2.1. SETTING UP A KAFKA CLUSTER .................................................................. 5
  2.2. DEPLOYING KAFKA CONNECT .................................................................. 6
  2.3. DEPLOYING A MYSQL DATABASE .............................................................. 6

CHAPTER 3. CREATING A CONNECTOR TO MONITOR THE INVENTORY DATABASE .... 9

CHAPTER 4. VIEWING CHANGE EVENTS ................................................................. 13
  4.1. VIEWING A CREATE EVENT ....................................................................... 13
  4.2. UPDATING THE DATABASE AND VIEWING THE UPDATE EVENT .............. 19
  4.3. DELETING A RECORD IN THE DATABASE AND VIEWING THE DELETE EVENT 21
  4.4. RESTARTING THE KAFKA CONNECT SERVICE .......................................... 23

CHAPTER 5. NEXT STEPS ......................................................................................... 26
This tutorial demonstrates how to use Debezium to monitor a MySQL database. As the data in the database changes, you will see the resulting event streams.

In this tutorial you will start the Debezium services in OpenShift, run a MySQL database server with a simple example database, and use Debezium to monitor the database for changes.

**Prerequisites**

Before you can use Debezium to monitor a MySQL database, you must have:

- Access to an OpenShift Container Platform 4.x cluster with `cluster-admin` privileges
- The AMQ Streams 1.4 OpenShift installation and example files
  You can download these files from the [AMQ Streams download site](#).
- The Debezium 1.0.0 MySQL Connector
  You can download these files from the [Red Hat Integration download site](#).

**NOTE**

These prerequisites apply to the MySQL connector. Other Debezium connectors may have different prerequisites.
CHAPTER 1. INTRODUCTION TO DEBEZIUM

Debezium is a distributed platform that turns your existing databases into event streams, so applications can see and respond immediately to each row-level change in the databases.

Debezium is built on top of Apache Kafka and provides Kafka Connect compatible connectors that monitor specific database management systems. Debezium records the history of data changes in Kafka logs, from where your application consumes them. This makes it possible for your application to easily consume all of the events correctly and completely. Even if your application stops unexpectedly, it will not miss anything: when the application restarts, it will resume consuming the events where it left off.

Debezium includes multiple connectors. In this tutorial, you will use the MySQL connector.
CHAPTER 2. STARTING THE SERVICES

Using Debezium requires AMQ Streams and the Debezium connector service. To start the services needed for this tutorial, you must:

1. Use AMQ Streams to set up a single-node Kafka cluster in OpenShift
2. Deploy Kafka Connect with the Debezium MySQL Connector plugin
3. Deploy a MySQL database

2.1. SETTING UP A KAFKA CLUSTER

You use AMQ Streams to set up a Kafka cluster. This procedure deploys a single-node Kafka cluster.

Procedure

1. In your OpenShift 4.x cluster, create a new project:

   $ oc new-project cdc-tutorial

2. Change to the directory where you downloaded the AMQ Streams 1.4 OpenShift installation and example files.

3. Deploy the AMQ Streams Cluster Operator.
   The Cluster Operator is responsible for deploying and managing Kafka clusters within an OpenShift cluster. This command deploys the Cluster Operator to watch just the project that you created:

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

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

4. Verify that the Cluster Operator is running.
   This command shows that the Cluster Operator is running, and that all of the Pods are ready:

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

5. Deploy the Kafka cluster.
   This command uses the kafka-ephemeral-single.yaml Custom Resource to create an ephemeral Kafka cluster with three ZooKeeper nodes and one Kafka node:

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

6. Verify that the Kafka cluster is running.
   This command shows that the Kafka cluster is running, and that all of the Pods are ready:

   $ 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
2.2. DEPLOYING KAFKA CONNECT

After setting up a Kafka cluster, you deploy the Kafka Connect Source-to-Image (S2I) service. This service provides a framework for managing the Debezium MySQL connector.

**Procedure**

1. Deploy the Kafka Connect Source-to-Image (S2I) service:
   This command deploys the Kafka Connect S2I service using the example YAML file for a single-node Kafka cluster:

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

2. Verify that the Kafka Connect service is running.
   This command shows that the Kafka Connect service is running, and that the Pod is ready:

   ```sh
   $ 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. Start a new build of the Kafka Connect image using the Debezium MySQL Connector plugin.
   This command uses the Debezium MySQL Connector plugin that you previously downloaded:

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

4. Verify that the build has completed.
   This command shows that the new build is complete (*my-connect-cluster-connect-2*). The Debezium MySQL Connector is installed:

   ```sh
   $ 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. DEPLOYING A MYSQL DATABASE

At this point, you have deployed a Kafka cluster and the Kafka Connect service with the Debezium MySQL Database Connector. However, you still need a database server from which Debezium can capture changes. In this procedure, you will start a MySQL server with an example database.

**Procedure**

1. Start a MySQL database.
   This command starts a MySQL database server preconfigured with an example *inventory* database:
2. Configure credentials for the MySQL database.
   This command updates the deployment configuration for the MySQL database to add the user
   name and password:

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

3. Verify that the MySQL database is running.
   This command shows that the MySQL database is running, and that the Pod is ready:

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

4. Open a new terminal and log into the sample inventory database.
   This command opens a MySQL command line client in the Pod that is running the MySQL
   database. It uses the user name and password that you previously configured:

   $ 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. List the tables in the inventory database.

   mysql> show tables;
   +---------------------+
   | Tables_in_inventory |
   +---------------------+
   | addresses           |
   | customers           |
   | geom                |
   | orders              |
   | products            |
   | products_on_hand    |
   +---------------------+
   6 rows in set (0.00 sec)
6. Explore the database and view the pre-loaded data.
   This example shows the customers table:

```
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)
```
After starting the Kafka, Debezium, and MySQL services, you are ready to create a connector instance to monitor the inventory database.

In this procedure, you will create the connector instance by creating a KafkaConnector Custom Resource (CR) that defines the connector instance, and then applying it. After applying the CR, the connector instance will start monitoring the inventory database’s binlog. The binlog records all of the database’s transactions (such as changes to individual rows and changes to the schemas). When a row in the database changes, Debezium generates a change event.

**NOTE**

Typically, you would likely use the Kafka tools to manually create the necessary topics, including specifying the number of replicas. However, for this tutorial, Kafka is configured to automatically create the topics with just one replica.

**Procedure**

1. Open the `examples/kafka-connect/kafka-connect-s2i-single-node-kafka.yaml` file that you used to deploy Kafka Connect. Before you can create the MySQL connector instance, you must first enable connector resources in the KafkaConnectS2I Custom Resource (CR).

2. In the `metadata.annotations` section, enable Kafka Connect to use connector resources. This example adds an annotation to the `examples/kafka-connect/kafka-connect-s2i-single-node-kafka.yaml` example file:

   ```yaml
   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. Apply the updated `kafka-connect-s2i-single-node-kafka.yaml` file to update the KafkaConnectS2I CR.

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

4. Create a MySQL connector instance to monitor the inventory database. This example creates a KafkaConnector CR that defines the MySQL connector instance:

   ```yaml
   inventory-connector.yaml
   
   apiVersion: kafka.strimzi.io/v1alpha1
   kind: KafkaConnector
   metadata:
   ```
The name of the connector.

Only one task should operate at any one time. Because the MySQL connector reads the MySQL server's binlog, using a single connector task ensures proper order and event handling. The Kafka Connect service uses connectors to start one or more tasks that do the work, and it automatically distributes the running tasks across the cluster of Kafka Connect services. If any of the services stop or crash, those tasks will be redistributed to running services.

The connector's configuration.

The database host, which is the name of the container running the MySQL server (mysql).

A unique server ID and name. The server name is the logical identifier for the MySQL server or cluster of servers. This name will be used as the prefix for all Kafka topics.

Only changes in the inventory database will be detected.

The connector will store the history of the database schemas in Kafka using this broker (the same broker to which you are sending events) and topic name. Upon restart, the connector will recover the schemas of the database that existed at the point in time in the binlog when the connector should begin reading.

5. Apply the connector instance.

$ oc apply -f inventory-connector.yaml

The inventory-connector connector is registered and starts to run against the inventory database.

6. Verify that inventory-connector was created and has started to monitor the inventory database.

You can verify the connector instance by watching the Kafka Connect log output as inventory-connector starts.

   a. Display the Kafka Connect log output:
Review the log output and verify that the initial snapshot has been executed. These lines show that the initial snapshot has started:

```
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&zeroDateTimeBehavior=CONVERT_TO_NULL&connectTimeout=30000 with user 'debezium' with locking mode 'minimal' (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-snapshot]
```

The snapshot involves a number of steps:

```
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 17:57:30,843 INFO   using binlog 'mysql-bin.000003' at position '154' and gtid '' (io.debezium.connector.mysql.SnapshotReader) [debezium-mysqlconnector-dbserver1-snapshot]
```

```
```

After completing the snapshot, Debezium begins monitoring the **inventory** database’s **binlog** for change events:
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]
...

CHAPTER 4. VIEWING CHANGE EVENTS

After deploying the Debezium MySQL connector, it starts monitoring the inventory database for data change events.

When you watched the connector start up, you saw that events were written to the following topics with the dbserver1 prefix (the name of the connector):

- **dbserver1**: The schema change topic to which all of the DDL statements are written.
- **dbserver1.inventory.products**: Captures change events for the products table in the inventory database.
- **dbserver1.inventory.products_on_hand**: Captures change events for the products_on_hand table in the inventory database.
- **dbserver1.inventory.customers**: Captures change events for the customers table in the inventory database.
- **dbserver1.inventory.orders**: Captures change events for the orders table in the inventory database.

For this tutorial, you will explore the dbserver1.inventory.customers topic. In this topic, you will see different types of change events to see how the connector captured them:

- Viewing a create event
- Updating the database and viewing the update event
- Deleting a record in the database and viewing the delete event
- Restarting Kafka Connect and changing the database

### 4.1. VIEWING A CREATE EVENT

By viewing the dbserver1.inventory.customers topic, you can see how the MySQL connector captured create events in the inventory database. In this case, the create events capture new customers being added to the database.

**Procedure**

1. Open a new terminal and use **kafka-console-consumer** to consume the dbserver1.inventory.customers topic from the beginning of the topic.
   
   This command runs a simple consumer (**kafka-console-consumer.sh**) in the Pod that is running Kafka (**my-cluster-kafka-0**):

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

   The consumer returns four messages (in JSON format), one for each row in the customers table. Each message contains the event records for the corresponding table row.
There are two JSON documents for each event: a key and a value. The key corresponds to the row’s primary key, and the value shows the details of the row (the fields that the row contains, the value of each field, and the type of operation that was performed on the row).

2. For the last event, review the details of the key.
   Here are the details of the key of the last event (formatted for readability):

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

   The event has two parts: a schema and a payload. The schema contains a Kafka Connect schema describing what is in the payload. In this case, the payload is a struct named `dbserver1.inventory.customers.Key` that is not optional and has one required field (id of type int32).

   The payload has a single id field, with a value of 1004.

   By reviewing the key of the event, you can see that this event applies to the row in the `inventory.customers` table whose id primary key column had a value of 1004.

3. Review the details of the same event’s value.
   The event’s value shows that the row was created, and describes what it contains (in this case, the id, first_name, last_name, and email of the inserted row).

   Here are the details of the value of the last event (formatted for readability):

   ```json
   {
     "schema": {
       "type": "struct",
       "fields": [
         {
           "type": "struct",
           "fields": [
             {
               "type": "int32",
               "optional": false,
               "field": "id"
             },
             {
               "type": "string",
               "optional": false,
               "field": "first_name"
             }
           ],
           "optional": true,
           "name": "dbserver1.inventory.customers.Customer"
         },
         "optional": false,
         "name": "dbserver1.inventory.customers.Customer".
       },
       "payload": {
         "id": 1004,
         "first_name": "John",
         "last_name": "Doe",
         "email": "john.doe@example.com"
       }
     }
   }
   ```
"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"
This portion of the event is much longer, but like the event’s key, it also has a schema and a payload. The schema contains a Kafka Connect schema named \texttt{dbserver1.inventory.customers.Envelope} (version 1) that can contain five fields:

- **op**
  A required field that contains a string value describing the type of operation. Values for the MySQL connector are \texttt{c} for create (or insert), \texttt{u} for update, \texttt{d} for delete, and \texttt{r} for read (in the case of a non-initial snapshot).

- **before**
  An optional field that, if present, contains the state of the row before the event occurred. The structure will be described by the \texttt{dbserver1.inventory.customers.Value} Kafka Connect schema, which the \texttt{dbserver1} connector uses for all rows in the
inventory.customers table.

**after**

An optional field that, if present, contains the state of the row after the event occurred. The structure is described by the same `dbserver1.inventory.customers.Value` Kafka Connect schema used in **before**.

**source**

A required field that contains a structure describing the source metadata for the event, which in the case of MySQL, contains several fields: the connector name, the name of the binlog file where the event was recorded, the position in that binlog file where the event appeared, the row within the event (if there is more than one), the names of the affected database and table, the MySQL thread ID that made the change, whether this event was part of a snapshot, and, if available, the MySQL server ID, and the timestamp in seconds.

**ts_ms**

An optional field that, if present, contains the time (using the system clock in the JVM running the Kafka Connect task) at which the connector processed the event.

**NOTE**

The JSON representations of the events are much longer than the rows they describe. This is because, with every event key and value, Kafka Connect ships the schema that describes the payload. Over time, this structure may change. However, having the schemas for the key and the value in the event itself makes it much easier for consuming applications to understand the messages, especially as they evolve over time.

The Debezium MySQL connector constructs these schemas based upon the structure of the database tables. If you use DDL statements to alter the table definitions in the MySQL databases, the connector reads these DDL statements and updates its Kafka Connect schemas. This is the only way that each event is structured exactly like the table from where it originated at the time the event occurred. However, the Kafka topic containing all of the events for a single table might have events that correspond to each state of the table definition.

The JSON converter includes the key and value schemas in every message, so it does produce very verbose events.

4. Compare the event’s key and value schemas to the state of the inventory database. In the terminal that is running the MySQL command line client, run the following statement:

```sql
mysql> SELECT * FROM customers;
+----------------+---------+------------+-----------------------+
<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
<th>email</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Sally</td>
<td>Thomas</td>
<td><a href="mailto:sally.thomas@acme.com">sally.thomas@acme.com</a></td>
</tr>
<tr>
<td>1002</td>
<td>George</td>
<td>Bailey</td>
<td><a href="mailto:gbailey@foobar.com">gbailey@foobar.com</a></td>
</tr>
<tr>
<td>1003</td>
<td>Edward</td>
<td>Walker</td>
<td><a href="mailto:ed@walker.com">ed@walker.com</a></td>
</tr>
<tr>
<td>1004</td>
<td>Anne</td>
<td>Kretchmar</td>
<td><a href="mailto:annek@noanswer.org">annek@noanswer.org</a></td>
</tr>
</tbody>
</table>
+----------------+---------+------------+-----------------------+
4 rows in set (0.00 sec)
```

This shows that the event records you reviewed match the records in the database.
4.2. UPDATING THE DATABASE AND VIEWING THE UPDATE EVENT

Now that you have seen how the Debezium MySQL connector captured the create events in the inventory database, you will now change one of the records and see how the connector captures it.

By completing this procedure, you will learn how to find details about what changed in a database commit, and how you can compare change events to determine when the change occurred in relation to other changes.

Procedure

1. In the terminal that is running the MySQL command line client, run the following statement:

   ```
   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. View the updated customers table:

   ```
   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. Switch to the terminal running kafka-console-consumer to see a new fifth event. By changing a record in the customers table, the Debezium MySQL connector generated a new event. You should see two new JSON documents: one for the event’s key, and one for the new event’s value.

   Here are the details of the key for the update event (formatted for readability):

   ```
   {
   "schema": {
   "type": "struct",
   "name": "dbserver1.inventory.customers.Key",
   "optional": false,
   "fields": [
   {
   "field": "id",
   "type": "int32",
   "optional": false
   }
   ]
   },
   "payload": {
   "id": 1004
   }
   }
   ```
This key is the same as the key for the previous events.

Here is that new event’s value. There are no changes in the schema section, so only the payload section is shown (formatted for readability):

```json
{
    "schema": {...},
    "payload": {
        "before": {
            "id": 1004,
            "first_name": "Anne",
            "last_name": "Kretchmar",
            "email": "annek@noanswer.org"
        },
        "after": {
            "id": 1004,
            "first_name": "Anne Marie",
            "last_name": "Kretchmar",
            "email": "annek@noanswer.org"
        },
        "source": {
            "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",
        "ts_ms": 1486501486308
    }
}
```

1. The before field now has the state of the row with the values before the database commit.
2. The after field now has the updated state of the row, and the first_name value is now Anne Marie.
3. The source field structure has many of the same values as before, except that the ts_sec and pos fields have changed (the file might have changed in other circumstances).
4. The op field value is now u, signifying that this row changed because of an update.
5. The ts_ms field shows the time stamp for when Debezium processed this event.

By viewing the payload section, you can learn several important things about the update event:

- By comparing the before and after structures, you can determine what actually changed in the affected row because of the commit.
By reviewing the source structure, you can find information about MySQL’s record of the change (providing traceability).

By comparing the payload section of an event to other events in the same topic (or a different topic), you can determine whether the event occurred before, after, or as part of the same MySQL commit as another event.

4.3. DELETING A RECORD IN THE DATABASE AND VIEWING THE DELETE EVENT

Now that you have seen how the Debezium MySQL connector captured the create and update events in the inventory database, you will now delete one of the records and see how the connector captures it.

By completing this procedure, you will learn how to find details about delete events, and how Kafka uses log compaction to reduce the number of delete events while still enabling consumers to get all of the events.

Procedure

1. In the terminal that is running the MySQL command line client, run the following statement:

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

   **NOTE**
   
   If the above command fails with a foreign key constraint violation, then you must remove the reference of the customer address from the addresses table using the following statement:

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

2. Switch to the terminal running kafka-console-consumer to see two new events.

   By deleting a row in the customers table, the Debezium MySQL connector generated two new events.

3. Review the key and value for the first new event.

   Here are the details of the key for the first new event (formatted for readability):

   ```
   {
   "schema": {
   "type": "struct",
   "name": "dbserver1.inventory.customers.Key",
   "optional": false,
   "fields": [
   {
   "field": "id",
   "type": "int32",
   "optional": false
   }
   ],
   "payload": {
   ```
This key is the same as the key in the previous two events you looked at.

Here is the value of the first new event (formatted for readability):

```
{
  "schema": {...},
  "payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,
    "source": {
      "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",
    "ts_ms": 1486501558315
  }
}
```

1. The **before** field now has the state of the row that was deleted with the database commit.
2. The **after** field is **null** because the row no longer exists.
3. The **source** field structure has many of the same values as before, except the **ts_sec** and **pos** fields have changed (the **file** might have changed in other circumstances).
4. The **op** field value is now **d**, signifying that this row was deleted.
5. The **ts_ms** field shows the time stamp for when Debezium processes this event.

Thus, this event provides a consumer with the information that it needs to process the removal of the row. The old values are also provided, because some consumers might require them to properly handle the removal.

4. Review the **key** and **value** for the second new event.
   Here is the **key** for the second new event (formatted for readability):

   ```
   \"id\": 1004
   ```
Once again, this key is exactly the same key as in the previous three events you looked at.

Here is the value of that same event (formatted for readability):

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

If Kafka is set up to be log compacted, it will remove older messages from the topic if there is at least one message later in the topic with same key. This last event is called a tombstone event, because it has a key and an empty value. This means that Kafka will remove all prior messages with the same key. Even though the prior messages will be removed, the tombstone event means that consumers can still read the topic from the beginning and not miss any events.

### 4.4. RESTARTING THE KAFKA CONNECT SERVICE

Now that you have seen how the Debezium MySQL connector captures create, update, and delete events, you will now see how it can capture change events even when it is not running.

The Kafka Connect service automatically manages tasks for its registered connectors. Therefore, if it goes offline, when it restarts, it will start any non-running tasks. This means that even if Debezium is not running, it can still report changes in a database.

In this procedure, you will stop Kafka Connect, change some data in the database, and then restart Kafka Connect to see the change events.

**Procedure**

1. Stop the Kafka Connect service.
   a. Open the deployment configuration for the Kafka Connect service.

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

   The deployment configuration opens:
b. Change the `spec.replicas` value to **0**.

c. Save the deployment configuration.

d. Verify that the Kafka Connect service has stopped.
   This command shows that the Kafka Connect service is completed, and that no Pods are running:

   ```
   $ 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. While the Kafka Connect service is down, switch to the terminal running the MySQL client, and add a new record to the database.

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

3. Restart the Kafka Connect service.

   a. Open the deployment configuration for the Kafka Connect service.

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

   The deployment configuration opens:

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

   b. Change the `spec.replicas` value to **1**.

   c. Save the deployment configuration.

   d. Verify that the Kafka Connect service has restarted.
      This command shows that the Kafka Connect service is running, and that the Pod is ready:

   ```
   $ 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. Switch to the terminal that is running **kafka-console-consumer** and review the messages.
You should see the record that you created when Kafka Connect was offline (formatted for readability):

```json
{
    ...
    "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
    }
}
```
CHAPTER 5. NEXT STEPS

After completing the tutorial, consider the following next steps:

- Explore the tutorial further.
  Use the MySQL command line client to add, modify, and remove rows in the database tables, and see the effect on the topics. Keep in mind that you cannot remove a row that is referenced by a foreign key.

- Plan a Debezium deployment.
  You can install Debezium in OpenShift or on Red Hat Enterprise Linux. For more information, see the following:
  - Installing Debezium on OpenShift
  - Installing Debezium on RHEL