Red Hat Integration 2022.Q1

Getting Started with Debezium

For use with Debezium 1.7
Getting Started with Debezium

For use with Debezium 1.7
Abstract

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

**PREFACE**  ................................................................................................................................. 3  
MAKING OPEN SOURCE MORE INCLUSIVE ................................................................. 3

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

**CHAPTER 2. STARTING THE SERVICES** ............................................................................. 5

2.1. DEPLOYING A MYSQL DATABASE .................................................................................. 5
2.2. DEPLOYING KAFKA CONNECT ....................................................................................... 6
2.3. VERIFYING THE CONNECTOR DEPLOYMENT .............................................................. 10

**CHAPTER 3. VIEWING CHANGE EVENTS** ............................................................................. 15

3.1. VIEWING A CREATE EVENT .......................................................................................... 15
3.2. UPDATING THE DATABASE AND VIEWING THE UPDATE EVENT ................................. 21
3.3. DELETING A RECORD IN THE DATABASE AND VIEWING THE DELETE EVENT ........... 23
3.4. RESTARTING THE KAFKA CONNECT SERVICE ........................................................... 25

**CHAPTER 4. NEXT STEPS** ..................................................................................................... 28
PREFACE

This tutorial demonstrates how to use Debezium to capture updates in a MySQL database. As the data in the database changes, you can see the resulting event streams.

The tutorial includes the following steps:

- Deploy a MySQL database server with a simple example database to OpenShift.
- Apply a custom resource in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium MySQL connector plug-in.
- Create the Debezium MySQL connector resource to capture changes in the database.
- Verify the connector deployment.
- View the change events that the connector emits to a Kafka topic from the database.

Prerequisites

- You are familiar with OpenShift and AMQ Streams.
- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.
- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- You have a Red Hat Integration license.
- You know how to use OpenShift administration tools. The OpenShift oc CLI client is installed or you have access to the OpenShift Container Platform web console.
- Depending on how you intend to store the Kafka Connect build image, you must either have permission to access a container registry, or you must create an ImageStream resource on OpenShift:
  - To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub
    - An account and permissions to create and manage images in the registry.
  - To store the build image as a native OpenShift ImageStream
    - An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
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 with Kafka and Kafka Connect, a database, and the Debezium connector service. To run the services for this tutorial, you must:

1. Deploy a MySQL database
2. Deploy Kafka Connect with the Debezium MySQL Connector plug-in
3. Verify the connector deployment

2.1. DEPLOYING A MYSQL DATABASE

Deploy a MySQL database server that includes an example inventory database that includes several tables that are pre-populated with data. The Debezium MySQL connector will capture changes that occur in the sample tables and transmit the change event records to an Apache Kafka topic.

Procedure

1. Start a MySQL database by running the following command, which starts a MySQL database server configured with an example inventory database:

   ```bash
   $ oc new-app --name=mysql quay.io/debezium/example-mysql:latest
   ```

2. Configure credentials for the MySQL database by running the following command, which updates the deployment configuration for the MySQL database to add the user name and password:

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

3. Verify that the MySQL database is running by invoking the following command, which is followed by the output that shows that the MySQL database is running, and that the pod is ready:

   ```bash
   $ 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. The client uses the user name and password that you previously configured:

   ```bash
   $ oc exec mysql-1-2gzx5 -it -- mysql -u mysqluser -pmysqlpw inventory
   ```

   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.
5. List the tables in the inventory database:

```sql
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 data that it contains, for example, view the customers table:

```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)
```

### 2.2. DEPLOYING KAFKA CONNECT

After you deploy the MySQL database, use AMQ Streams to build a Kafka Connect container image that includes the Debezium MySQL connector plug-in. During the deployment process, you create and use the following custom resources (CRs):

- **A KafkaConnect CR** that defines your Kafka Connect instance and includes information about the MySQL connector artifacts to include in the image.

- **A KafkaConnector CR** that provides details that include information that the MySQL connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector CR**.

During the build process, the AMQ Streams Operator transforms input parameters in the **KafkaConnect** custom resource, including Debezium connector definitions, into a Kafka Connect container image. The build downloads the necessary artifacts from the Red Hat Maven repository, and incorporates them into the image. The newly created container is pushed to the container registry that is specified in .spec.build.output, and is used to deploy a Kafka Connect pod. After AMQ Streams builds the Kafka Connect image, use the **KafkaConnector** custom resource to start the connector.
Procedure

1. Log in to the OpenShift cluster and create or open a project, for example `debezium`.

2. Create a Debezium KafkaConnect custom resource (CR) for the connector, or modify an existing one. For example, create a KafkaConnect CR that specifies the `metadata.annotations` and `spec.build` properties, as shown in the following example. Save the file with a name such as `dbz-connect.yaml`.

   **Example 2.1. A `dbz-connect.yaml` file that defines a KafkaConnect custom resource that includes a Debezium connector**

   ```yaml
   apiVersion: kafka.strimzi.io/v1beta2
   kind: KafkaConnect
   metadata:
     name: my-connect-cluster
   annotations:
     strimzi.io/use-connector-resources: "true" ①
   spec:
     version: 3.00
     build: ②
       output: ③
         type: imagestream ④
         image: debezium-streams-connect:latest
     plugins: ⑤
       - name: debezium-connector-mysql
         artifacts:
           - type: zip ⑥
             url: https://maven.repository.redhat.com/ga/io/debezium/debezium-connector-mysql/1.7.2.Final-redhat-<build_number>/debezium-connector-mysql-1.7.2.Final.zip ⑦
   bootstrapServers: my-cluster-kafka-bootstrap:9093
   ```

   **Table 2.1. Descriptions of Kafka Connect configuration settings**

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sets the <code>strimzi.io/use-connector-resources</code> annotation to &quot;true&quot; to enable the Cluster Operator to use KafkaConnect resources to configure connectors in this Kafka Connect cluster.</td>
</tr>
<tr>
<td>2</td>
<td>The <code>spec.build</code> configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.</td>
</tr>
<tr>
<td>3</td>
<td>The <code>build.output</code> specifies the registry in which the newly built image is stored.</td>
</tr>
<tr>
<td>4</td>
<td>Specifies the name and image name for the image output. Valid values for <code>output.type</code> are <code>docker</code> to push into a container registry like Docker Hub or Quay, or <code>imagestream</code> to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the <code>build.output</code> in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation.</td>
</tr>
</tbody>
</table>
The `plugins` configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in `name`, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.

The value of `artifacts.type` specifies the file type of the artifact specified in the `artifacts.url`. Valid types are `zip`, `tgz`, or `jar`. Debezium connector archives are provided in .zip file format. JDBC driver files are in .jar format. The `type` value must match the type of the file that is referenced in the `url` field.

The value of `artifacts.url` specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. The OpenShift cluster must have access to the specified server.

---

### Table: Item Description

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>The <strong>plugins</strong> configuration lists all of the connectors that you want to include in the Kafka Connect image. For each entry in the list, specify a plug-in <strong>name</strong>, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.</td>
</tr>
<tr>
<td>6</td>
<td>The value of <code>artifacts.type</code> specifies the file type of the artifact specified in the <code>artifacts.url</code>. Valid types are <code>zip</code>, <code>tgz</code>, or <code>jar</code>. Debezium connector archives are provided in .zip file format. JDBC driver files are in .jar format. The <strong>type</strong> value must match the type of the file that is referenced in the <strong>url</strong> field.</td>
</tr>
<tr>
<td>7</td>
<td>The value of <code>artifacts.url</code> specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. The OpenShift cluster must have access to the specified server.</td>
</tr>
</tbody>
</table>

---

3. Apply the **KafkaConnect** build specification to the OpenShift cluster by entering the following command:

```
oc create -f dbz-connect.yaml
```

Based on the configuration specified in the custom resource, the Streams Operator prepares a Kafka Connect image to deploy. After the build completes, the Operator pushes the image to the specified registry or ImageStream, and starts the Kafka Connect cluster. The connector artifacts that you listed in the configuration are available in the cluster.

4. Create a **KafkaConnector** resource to define an instance of the MySQL connector. For example, create the following **KafkaConnector** CR, and save it as `debezium-inventory-connector.yaml`.

**Example 2.2.** A `mysql-inventory-connector.yaml` file that defines the **KafkaConnector** custom resource for a Debezium connector

```yaml
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: my-connect-cluster
name: inventory-connector
spec:
class: io.debezium.connector.mysql.MySqlConnector
tasksMax: 1
config:
database.hostname: mysql
database.port: 3306
database.user: debezium
database.password: dbz
```
Table 2.2. Descriptions of connector configuration settings

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The name of the connector to register with the Kafka Connect cluster.</td>
</tr>
<tr>
<td>2</td>
<td>The name of the connector class.</td>
</tr>
<tr>
<td>3</td>
<td>Only one task should operate at any one time. Use a single connector task to ensure proper order and event handling as the MySQL connector reads the MySQL server’s binlog. The Kafka Connect service uses connectors to start one or more tasks to complete the work. It automatically distributes the running tasks across the cluster of Kafka Connect services. If services stop or crash, tasks are redistributed to running services.</td>
</tr>
<tr>
<td>4</td>
<td>The connector’s configuration.</td>
</tr>
<tr>
<td>5</td>
<td>The database host, which is the name of the container that runs the MySQL server (mysql).</td>
</tr>
<tr>
<td>6</td>
<td>The port number of the database instance.</td>
</tr>
<tr>
<td>7</td>
<td>The name of the user account through which Debezium connects to the database.</td>
</tr>
<tr>
<td>8</td>
<td>The password for the database user account.</td>
</tr>
<tr>
<td>9</td>
<td>The name of the database to capture changes from.</td>
</tr>
<tr>
<td>10</td>
<td>The logical name of the database instance or cluster. The server name is the logical identifier for the MySQL server or cluster of servers. This name is used as the prefix for all Kafka topics.</td>
</tr>
<tr>
<td>11</td>
<td>The list of tables from which the connector captures change events. The connector detects changes in the inventory database only.</td>
</tr>
<tr>
<td>12</td>
<td>Specifies the Kafka broker and topic that the connector uses to store the history of the database schemas (the same broker to which you are sending events). After a restart, the connector recovers the database schemas that existed at the point in the binlog when the connector resumes reading.</td>
</tr>
</tbody>
</table>

5. Create the connector resource by running the following command:
For example,

```
oc create -n <namespace> -f <kafkaConnector>.yaml
```

For example,

```
oc create -n debezium -f mysql-inventory-connector.yaml
```

The connector is registered to the Kafka Connect cluster and starts to run against the database that is specified by `spec.config.databasedbname` in the `KafkaConnector` CR. After the connector pod is ready, Debezium is running.

You are now ready to verify that the connector was created and has started to capture changes in the `inventory` database.

### 2.3. VERIFYING THE CONNECTOR DEPLOYMENT

If the connector starts correctly without errors, it creates a topic for each table that the connector is configured to capture. Downstream applications can subscribe to these topics to retrieve information events that occur in the source database.

To verify that the connector is running, you perform the following operations from the OpenShift Container Platform web console, or through the OpenShift CLI tool (oc):

- Verify the connector status.
- Verify that the connector generates topics.
- Verify that topics are populated with events for read operations ("op":"r") that the connector generates during the initial snapshot of each table.

**Prerequisites**

- A Debezium connector is deployed to AMQ Streams on OpenShift.
- The OpenShift `oc` CLI client is installed.
- You have access to the OpenShift Container Platform web console.

**Procedure**

1. Check the status of the `KafkaConnector` resource by using one of the following methods:
   - From the OpenShift Container Platform web console:
     a. Navigate to **Home** → **Search**.
     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type `KafkaConnector`.
     c. From the **KafkaConnectors** list, click the name of the connector that you want to check, for example `inventory-connector`.
     d. In the **Conditions** section, verify that the values in the **Type** and **Status** columns are set to **Ready** and **True**.
   - From a terminal window:
a. Enter the following command:

```
oc describe KafkaConnector <connector-name> -n <project>
```

For example,

```
oc describe KafkaConnector inventory-connector -n debezium
```

The command returns status information that is similar to the following output:

```
Example 2.3. **KafkaConnector** resource status

Name:        inventory-connector
Namespace:   debezium
Labels:       strimzi.io/cluster=my-connect-cluster
Annotations: <none>
API Version: kafka.strimzi.io/v1beta2
Kind:        KafkaConnector

... Status:
Conditions:
  Last Transition Time:  2021-12-08T17:41:34.897153Z
Status:                  True
Type:                  Ready
Connector Status:
  Connector:
    State:      RUNNING
       worker_id:  10.131.1.124:8083
    Name:        inventory-connector
  Tasks:
    Id:               0
    State:            RUNNING
       worker_id:        10.131.1.124:8083
    Type:               source
  Observed Generation:  1
  Tasks Max:            1
  Topics:
    inventory_connector
    inventory_connector.inventory.addresses
    inventory_connector.inventory.customers
    inventory_connector.inventory.geom
    inventory_connector.inventory.orders
    inventory_connector.inventory.products
    inventory_connector.inventory.products_on_hand
  Events:  <none>
```

2. Verify that the connector created Kafka topics:

- From the OpenShift Container Platform web console.
  a. Navigate to **Home → Search**.
b. On the Search page, click Resources to open the Select Resource box, and then type KafkaTopic.

c. From the KafkaTopics list, click the name of the topic that you want to check, for example, inventory-connector.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d.

d. In the Conditions section, verify that the values in the Type and Status columns are set to Ready and True.

3. Check topic content.

   • From a terminal window, enter the following command:

     ```
     oc get kafkatopics
     ```

     The command returns status information that is similar to the following output:

     ```
     NAME                   CLUSTER  PARTITIONS  REPLICATION FACTOR  READY
     connect-cluster-configs my-cluster   1        1            True
     connect-cluster-offsets my-cluster  25       1            True
     connect-cluster-status my-cluster   5        1            True
     consumer-offsets---84e7a678d08f4bd226872e5cddd4eb527fadc1c6a my-cluster 50 1 True
     inventory-connector---a96f69b23d6118ff415f772679da623fbbb99421 my-cluster 1 1 True
     inventory-connector.inventory.addresses---1b69eaf7b2ebe5d7172d92e90ca2b10c9a56480 my-cluster 1 1 True
     inventory-connector.inventory.customers---9931e04ec92ecc0924f406af3f4dce7545c483b my-cluster 1 1 True
     inventory-connector.inventory.geom---9f7e136091f07bfa49ca59bf99e86c713ee58dd5 my-cluster 1 1 True
     inventory-connector.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d my-cluster 1 1 True
     inventory-connector.inventory.products---df0746db116844ce2297fbb61c215f682dce5 my-cluster 1 1 True
     inventory-connector.inventory.products-on-hand---8649e0f77fccc9212e266e31a7aee85e56c5 my-cluster 1 1 True
     schema-changes.inventory my-cluster    1           1       True
     strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3f177fc55 my-cluster 1 1 True
     strimzi-topic-operator-kstreams-topic-store-changelog---b75e702040b99be8a9263134de3507fc0cc4017b my-cluster 1 1 True
     ```

3. Check topic content.

   • From a terminal window, enter the following command:

     ```
     oc exec -n <project> -it <kafka-cluster> -- /opt/kafka/bin/kafka-console-consumer.sh \
     > --bootstrap-server localhost:9092 \
     > --from-beginning \
     ```
For example,

```bash
oc exec -n debezium -ti my-cluster-kafka-0 /opt/kafka/bin/kafka-console-consumer.sh
> --bootstrap-server localhost:9092
> --property print.key=true
> --topic=inventory_connector.inventory.products_on_hand
```

The format for specifying the topic name is the same as the `oc describe` command returns in Step 1, for example, `inventory_connector.inventory.addresses`.

For each event in the topic, the command returns information that is similar to the following output:

**Example 2.5. Content of aDebezium change event**

```json
{"schema":{"type":"struct","fields":
  [["type":"int32","optional":false,"field":"product_id"],"optional":false,"name":"inventory_connector.inventory.products_on_hand.Key"],"payload":{"product_id":101}}
{"schema":
  [{"type":"struct","fields":
    [{"type":"int32","optional":false,"field":"product_id"},
    {"type":"int32","optional":false,"field":"quantity"},"optional":true,"name":"inventory_connector.inventory.products_on_hand.Value","field":"before"},
    {"type":"int32","optional":false,"field":"product_id"},
    {"type":"int32","optional":false,"field":"after"},"optional":true,"name":"inventory_connector.inventory.products_on_hand.Value","field":"after"],"optional":false,"name":"io.debezium.connector.mysql.Source","field":
  [{"type":"string","optional":false,"field":"version"},
  {"type":"string","optional":false,"field":"connector"},
  {"type":"string","optional":false,"field":"name"},
  {"type":"int64","optional":false,"field":"ts_ms"},
  {"type":"string","optional":false,"name":"io.debezium.data.Enum","version":1,"parameters":
    {"allowed":true,"last":false},"default":false,"field":"snapshot"},
  {"type":"string","optional":false,"field":"db"},
  {"type":"string","optional":true,"field":"sequence"},
  {"type":"string","optional":true,"field":"table"},
  {"type":"int64","optional":false,"field":"server_id"},
  {"type":"string","optional":false,"field":"file","gtid"},
  {"type":"int64","optional":false,"field":"pos"},
  {"type":"int32","optional":true,"field":"thread"},
  {"type":"string","optional":false,"field":"query"},"optional":false,"name":"io.debezium.connector.mysql.Source","field":
    [{"type":"string","optional":false,"field":"source"}],
  {"type":"string","optional":false,"field":"op"},
  {"type":"int64","optional":true,"field":"ts_ms"},
  {"type":"struct","fields":
    [{"type":"string","optional":false,"field":"id"},
    {"type":"int64","optional":false,"field":"total_order"},
    {"type":"int64","optional":false,"field":"data_collection_order"},
    {"optional":true,"field":"snapshot"},
    {"optional":false,"name":"inventory_connector.inventory.products_on_hand.Envelope"},
    {"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
      {"version":"1.7.2.Final-redhat-00001","connector":"mysql","name":"inventory_connector","ts_ms":1638985247805,"snapshot":true,"db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":null}}}
```
In the preceding example, the payload value shows that the connector snapshot generated a read ("op" = "r") event from the table inventory.products_on_hand. The "before" state of the product_id record is null, indicating that no previous value exists for the record. The "after" state shows a quantity of 3 for the item with product_id 101.

You are now ready to view change events that the Debezium connector captures from the inventory database.
CHAPTER 3. VIEWING CHANGE EVENTS

Three events were written to the following topics, whose names all start with `dbserver1`, which is the name of the connector:

- `dbserver1`: The schema change topic to which DDL statements that apply to the tables for which changes are being captured are written.
- `dbserver1.inventory.products`: Receives change event records for the `products` table in the `inventory` database.
- `dbserver1.inventory.products_on_hand`: Receives change event records for the `products_on_hand` table in the `inventory` database.
- `dbserver1.inventory.customers`: Receives change event records for the `customers` table in the `inventory` database.
- `dbserver1.inventory.orders`: Receives change event records for the `orders` table in the `inventory` database.

For this tutorial, you will explore the `dbserver1.inventory.customers` topic. In this topic, you will view 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

### 3.1. 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 \
   --from-beginning \
   --property print.key=true \
   --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"
                    }
                ],
                "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": "version"
    }
  ]
}
"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 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 c for create (or insert), u for update, d for delete, and 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 dbserver1.inventory.customers.Value Kafka Connect schema, which the 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:anne@noanswer.org">anne@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.
3.2. 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:

```sql
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 Marie</td>
<td>Kretchmar</td>
<td>annekJnoanswer.org</td>
</tr>
</tbody>
</table>
+----+------------+-----------+-----------------------+
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):

```json
{
"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):

```
{  
  "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": "1.7.2.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.

3.3. 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:

   ```sql
   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:

   ```sql
   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):

   ```json
   {
   "schema": {
   "type": "struct",
   "name": "dbserver1.inventory.customers.Key",
   "optional": false,
   "fields": [
   {
   "field": "id",
   "type": "int32",
   "optional": false
   }
   ```
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": "1.7.2.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):
   
   ```json
   {
   "schema": {
   "type": "struct",
   "name": "dbserver1.inventory.customers.Key"
   "optional": false,
   "fields": [
   { "field": "id",
   "type": "int32",
   "optional": false
   }
   ],
   "payload": {
   "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.

   3.4. 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:

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

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:

   ```yaml
   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.sh`. New events pop up as they arrive.

5. Examine 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":"1.7.2.Final",
    "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 4. 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

Revised on 2022-04-13 09:50:32 UTC