Red Hat build of Debezium 2.5.4

Getting Started with Debezium

For use with Red Hat build of Debezium 2.5.4

Last Updated: 2024-04-08

Red Hat build of Debezium 2.5.4 Getting Started with Debezium

For use with Red Hat build of Debezium 2.5.4

Legal Notice

Abstract

This guide describes how to get started using Red Hat build of Debezium.

Table of Contents

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.

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.

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation.

To propose improvements, open a Jira issue and describe your suggested changes. Provide as much detail as possible to enable us to address your request quickly.

Prerequisite

- You have a Red Hat Customer Portal account. This account enables you to log in to the Red Hat Jira Software instance.
  If you do not have an account, you will be prompted to create one.

Procedure

1. Click the following link: Create issue.
2. In the **Summary** text box, enter a brief description of the issue.
3. In the **Description** text box, provide the following information:
   - The URL of the page where you found the issue.
   - A detailed description of the issue.
     You can leave the information in any other fields at their default values.
4. Click **Create** to submit the Jira issue to the documentation team.

Thank you for taking the time to provide feedback.

CHAPTER 1. ABOUT THIS TUTORIAL

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 Managing AMQ Streams on OpenShift.
- You have a Red Hat build of Debezium 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 for storing new container images. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Additional resources:

- Managing image streams on OpenShift Container Platform.

CHAPTER 2. INTRODUCTION TO DEBEZIUM

Debezium is a distributed platform that converts information from your existing databases into event streams, enabling applications to detect, and immediately respond to row-level changes in the databases.

Debezium is built on top of Apache Kafka and provides a set of Kafka Connect compatible connectors. Each of the connectors works with a specific database management system (DBMS). Connectors record the history of data changes in the DBMS by detecting changes as they occur, and streaming a record of each change event to a Kafka topic. Consuming applications can then read the resulting event records from the Kafka topic.

By taking advantage of Kafka's reliable streaming platform, Debezium makes it possible for applications to consume changes that occur in a database correctly and completely. Even if your application stops unexpectedly, or loses its connection, it does not miss events that occur during the outage. After the application restarts, it resumes reading from the topic from the point where it left off.

The tutorial that follows shows you how to deploy and use the Debezium MySQL connector with a simple configuration. For more information about deploying and using Debezium connectors, see the connector documentation.

Additional resources

- Debezium connector for Db2
- Debezium connector for MongoDB
- Debezium connector for MySQL
- Debezium connector for Oracle Database
- Debezium connector for PostgreSQL
- Debezium connector for SQL Server

CHAPTER 3. 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. ImageStream example
4. Verify the connector deployment

3.1. 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:

```
$ oc new-app -l app=mysql --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:

```
$ oc set env deployment/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:

```
$ 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:

```
$ oc exec mysql-1-2gzx5 -it -- mysql -u mysqluser -pmysqlpw 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 data that it contains, for example, view 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)
```

3.2. 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.

Container images can be stored in an external container registry, such as **quay.io**, or in an OpenShift ImageStream. Because ImageStreams are not created automatically, to store container images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect.

After AMQ Streams builds and stores the Kafka Connect image, use the **KafkaConnector** custom resource to start the connector.

Prerequisites

- AMQ Streams is running on an OpenShift cluster.
- The AMQ Streams Cluster Operator is installed to the OpenShift cluster.
- If you prefer to store the KafkaConnect container image in an OpenShift ImageStream, an ImageStream is available.
- Apache Kafka and Kafka Connect are running on AMQ Streams.

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.
   The following example shows an excerpt from a **dbz-connect.yaml** file that describes a **KafkaConnect** custom resource.
   The **metadata.annotations** and **spec.build** properties are required.

Example 3.1. A dbz-connect.yaml file that defines aKafkaConnect custom resource that includes a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnect
metadata:
  name: my-connect-cluster
  annotations:
    strimzi.io/use-connector-resources: "true" 1
spec:
  replicas: 1
  version: 3.6.0
  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/2.5.4.Final-redhat-00001/debezium-connector-mysql-2.5.4.Final-redhat-00001-plugin.zip 7
```

```
    bootstrapServers: my-cluster-kafka-bootstrap:9093

  ...
```

Table 3.1. Descriptions of Kafka Connect configuration settings

Item	Description
1	Sets the **strimzi.io/use-connector-resources** annotation to **"true"** to enable the Cluster Operator to use **KafkaConnector** resources to configure connectors in this Kafka Connect cluster.
2	The **spec.build** 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.
3	The **build.output** specifies the registry in which the newly built image is stored.
4	Specifies the name and image name for the image output. Valid values for **output.type** are **docker** to push into a container registry like Docker Hub or Quay, or **imagestream** 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 **build.output** in the KafkaConnect configuration, see the AMQ Streams Build schema reference documentation
5	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.
6	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.
7	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.

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 AMQ 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 3.2. A mysql-inventory-connector.yaml file that defines the KafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: my-connect-cluster
  name: inventory-connector 1
spec:
  class: io.debezium.connector.mysql.MySqlConnector 2
  tasksMax: 1 3
  config: 4
    database.hostname: mysql 5
    database.port: 3306 6
    database.user: debezium 7
    database.password: dbz 8
    database.server.id: 184054
    topic.prefix: dbserver1 9
    table.include.list: inventory.* 10
    schema.history.internal.kafka.bootstrap.servers: 'my-cluster-kafka-bootstrap:9092' 11
    schema.history.internal.kafka.topic: schema-changes.inventory 12
```

Table 3.2. Descriptions of connector configuration settings

Item	Description
1	The name of the connector to register with the Kafka Connect cluster.
2	The name of the connector class.
3	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.
4	The connector's configuration.
5	The hostname or address of the MySQL database instance.
6	The port number of the database instance.
7	The name of the user account through which Debezium connects to the database.

Item	Description
8	The password that Debezium uses to connect to the database user account.
9	Topic prefix for the MySQL server or cluster. This string prefixes the names of every Kafka topic that the connector sends event records to.
10	The list of tables from which the connector captures change events. The connector detects changes only if they occur in the **inventory** table.
11	List of Kafka brokers that the connector uses to write and recover DDL statements to the database schema history topic. This is the same broker that the connector sends change event records to. After a restart, the connector recovers the database schemas that existed at the point in the binlog when the connector resumes reading.
12	Name of the database schema history topic. This topic is for internal use only and should not be used by consumers.

5. Create the connector resource by running the following command:

```
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.database.dbname** 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.

3.3. EXAMPLE: A SIMPLE OPENSHIFT IMAGESTREAM OBJECT DEFINITION

The following example shows a simple **ImageStream** object definition

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: kafka-connect-dbz-mysql
spec:
  lookupPolicy:
    local: true
```

Additional resources:

- Creating and managing images and imagestreams in OpenShift Container Platform .

3.4. 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 3.3. 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:
    dbserver1
    dbserver1.inventory.addresses
    dbserver1.inventory.customers
    dbserver1.inventory.geom
    dbserver1.inventory.orders
    dbserver1.inventory.products
    dbserver1.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, **dbserver1.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d**.

     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 get kafkatopics
```

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

Example 3.4. KafkaTopic resource status

```
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---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a my-cluster
50 1 True
dbserver1---a96f69b23d6118ff415f772679da623fbbb99421 my-cluster 1 1 True
dbserver1.inventory.addresses---
1b6beaf7b2eb57d177d92be90ca2b210c9a56480  my-cluster 1 1 True
dbserver1.inventory.customers---9931e04ec92ecc0924f4406af3fdace7545c483b
my-cluster 1 1    True
dbserver1.inventory.geom---9f7e136091f071bf49ca59bf99e86c713ee58dd5  my-
cluster 1 1    True
dbserver1.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d
my-cluster 1 1    True
dbserver1.inventory.products---df0746db116844cee2297fab611c21b56f82dcef
my-cluster 1 1    True
dbserver1.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5  my-cluster 1   1 True
schema-changes.inventory my-cluster    1           1       True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55                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 \
>     --property print.key=true \
>     --topic=<topic-name>
```

For example,

```
 oc exec -n debezium  -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.products_on_hand
```

The format for specifying the topic name is the same as the **oc describe** command returns in Step 1, for example, **dbserver1.inventory.addresses**.

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

Example 3.5. Content of a Debezium change event


```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"dbserver1.inventory.products_on_hand.Key"},"payload":{"product_id":101}} {"schema":
{"type":"struct","fields":[{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"dbserver1.inventory.products_on_hand.Value","field":"before"},{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"},
{"type":"int32","optional":false,"field":"quantity"}],"optional":true,"name":"dbserver1.inventory.products_on_hand.Value","field":"after"},{"type":"struct","fields":
[{"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":true,"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":true,"field":"gtid"},{"type":"string","optional":false,"field":"file"},
{"type":"int64","optional":false,"field":"pos"},{"type":"int32","optional":false,"field":"row"},
{"type":"int64","optional":true,"field":"thread"},
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connector.mysql.Source","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":"transaction"}],"optional":false,"name":"dbserver1.inventory.products_on_hand.Envelope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":{"version":"2.5.4.Final-redhat-
00001","connector":"mysql","name":"dbserver1","ts_ms":1638985247805,"snapshot":"true",
"db":"inventory","sequence":null,"table":"products_on_hand","server_id":0,"gtid":null,"file":"mysql-
bin.000003","pos":156,"row":0,"thread":null,"query":null},"op":"r","ts_ms":1638985247805,"transaction":null}}
```


In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **dbserver1.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 4. VIEWING CHANGE EVENTS

After deploying the Debezium MySQL connector, it starts capturing changes to the **inventory** database.

When the connector starts, it writes events to a set of Apache Kafka topics, each of which represents one of the tables in the MySQL database. The name of each topic begins with the name of the database server, **dbserver1**.

The connector writes to the following Kafka topics:

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.

The remainder of this tutorial examines the **dbserver1.inventory.customers** Kafka topic. As you look more closely at the topic, you'll see how it represents different types of change events, and find information about the connector captured each event.

The tutorial contains the following sections:

- 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**):

```
$ 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):

```
{
  "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):

```
{
  "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": "2.5.4.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": "r",
    "ts_ms": 1486500577691
  }
}
```


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 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

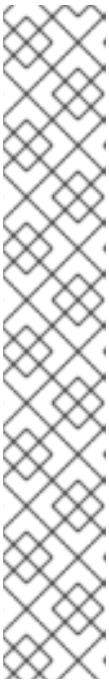
An optional field that, if present, contains the state of the row *after* the event occurred. The structure is described by the same **dbserver1.inventory.customers.Value** Kafka Connect schema used in **before**.

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:

```
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)
```

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):

```
{
  "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": "2.5.4.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
  }
}
```

Table 4.1. Descriptions of fields in the payload of an update event value

Item	Description
1	The **before** field shows the values present in the row before the database commit. The original **first_name** value is **Anne**.

Item	Description
2	The **after** field shows the state of the row after the change event. 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 a timestamp that indicates 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": {
    "id": 1004
  }
}
```

   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": {  1
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,  2
    "source": {  3
      "name": "2.5.4.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 1 1 1 The **before** field now has the state of the row that was deleted with the database commit.

2 2 2 2 The **after** field is **null** because the row no longer exists.

3 3 3 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 4 4 4 The **op** field value is now **d**, signifying that this row was deleted.

5 5 5 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):

```
{
  "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):

```
{
  "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 configuration for the Kafka Connect deployment:

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

   The deployment configuration opens:

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

   b. Change the **spec.replicas** value to **0**.

   c. Save the 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 deployment/my-connect-cluster-connect
```

The deployment configuration opens:

```
apiVersion: apps.openshift.io/v1
kind: Deployment
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):

```
{
  ...
  "payload":{
    "id":1005
  }
}
{
  ...
  "payload":{
    "before":null,
    "after":{
      "id":1005,
      "first_name":"Sarah",
      "last_name":"Thompson",
      "email":"kitt@acme.com"
    },
    "source":{
      "version":"2.5.4.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 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

Revised on 2024-04-08 22:09:24 UTC