You are viewing documentation for the current development version of Debezium.
If you want to view the latest stable version of this page, please go here.

Tutorial

Table of Contents

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

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

Prerequisites

- Docker is installed and running.

  This tutorial uses Docker and the Debezium Docker images to run the required services. You should use the latest version of Docker. For more information, see the Docker Engine installation documentation.

  NOTE

NOTE

This example can also be run using Podman. For more information see Podman

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.

Starting the services

Using Debezium requires three separate services: ZooKeeper, Kafka, and the Debezium connector service. In this tutorial, you will set up a single instance of each service using Docker and the Debezium Docker images.

To start the services needed for this tutorial, you must:

- Start Zookeeper
- Start Kafka
- Start a MySQL database
- Start a MySQL command line client
- Start Kafka Connect

Considerations for running Debezium with Docker

This tutorial uses Docker and the Debezium Docker images to run the ZooKeeper, Kafka, Debezium, and MySQL services. Running each service in a separate container simplifies the setup so that you can see Debezium in action.

NOTE

reliability, replication, and fault tolerance. Typically, you would either deploy these services on a platform like OpenShift or Kubernetes that manages multiple Docker containers running on multiple hosts and machines, or you would install on dedicated hardware.

You should be aware of the following considerations for running Debezium with Docker:

- The containers for ZooKeeper and Kafka are ephemeral.

  ZooKeeper and Kafka would typically store their data locally inside the containers, which would require you to mount directories on the host machine as volumes. That way, when the containers are stopped, the persisted data remains. However, this tutorial skips this setup - when a container is stopped, all persisted data is lost. This way, cleanup is simple when you complete the tutorial.

  NOTE

  For more information about storing persistent data, see the documentation for the Docker images.

- This tutorial requires you to run each service in a different container.

  To avoid confusion, you will run each container in the foreground in a separate terminal. This way, all of the output of a container will be displayed in the terminal used to run it.

  NOTE

  Docker also allows you to run a container in *detached* mode (with the `-d` option), where the container is started and the `docker` command returns immediately. However, detached mode containers do not display their output in the terminal. To see the output, you would need to use the `docker logs --follow --name <container-name>` command. For more information, see the Docker documentation.

Starting Zookeeper

ZooKeeper is the first service you must start.

Procedure

1. Open a terminal and use it to start ZooKeeper in a container.

   This command runs a new container using version 1.6 of the `debezium/zookeeper` image:

   SHELL

   ```
   $ docker run -it --rm --name zookeeper -p 2181:2181 -p 2888:2888 -p 3888:3888
   debezium/zookeeper:1.6
   ```

   -it
   The container is interactive, which means the terminal's standard input and output are attached to the container.

   --rm

--name zookeeper

The name of the container.

-p 2181:2181 -p 2888:2888 -p 3888:3888

Maps three of the container's ports to the same ports on the Docker host. This enables other containers (and applications outside of the container) to communicate with ZooKeeper.

NOTE

If you use Podman, run the following command:s

SHELL

```
$ sudo podman pod create --name=dbz --publish "9092,3306,8083"
$ sudo podman run -it --rm --name zookeeper --pod dbz debezium/zookeeper:1.6
```

1. Verify that ZooKeeper started and is listening on port `2181`.

   You should see output similar to the following:

SHELL

```
Starting up in standalone mode
ZooKeeper JMX enabled by default
Using config: /zookeeper/conf/zoo.cfg
2017-09-21 07:15:55,417 - INFO  [main:QuorumPeerConfig@134] - Reading configuration from:
/zookeeper/conf/zoo.cfg
2017-09-21 07:15:55,419 - INFO  [main:DatadirCleanupManager@78] - autopurge.snapRetainCount
set to 3
2017-09-21 07:15:55,419 - INFO  [main:DatadirCleanupManager@79] - autopurge.purgeInterval
set to 1
...
port 0.0.0.0/0.0.0.0:2181  ①
```

① This line indicates that ZooKeeper is ready and listening on port 2181. The terminal will continue to show additional output as ZooKeeper generates it.

Starting Kafka

After starting ZooKeeper, you can start Kafka in a new container.

NOTE

Debezium 1.6.0.Final has been tested against multiple versions of Kafka Connect. Please refer to the Debezium Test Matrix to determine compatibility between Debezium and Kafka Connect.

Procedure

1. Open a new terminal and use it to start Kafka in a container.

   This command runs a new container using version 1.6 of the `debezium/kafka` image:

SHELL

```
$ docker run -it --rm --name kafka -p 9092:9092 --link zookeeper:zookeeper
debezium/kafka:1.6
```

The container is interactive, which means the terminal's standard input and output are attached to the container.

`--rm`
: The container will be removed when it is stopped.

`--name kafka`
: The name of the container.

`-p 9092:9092`
: Maps port `9092` in the container to the same port on the Docker host so that applications outside of the container can communicate with Kafka.

`--link zookeeper:zookeeper`
: Tells the container that it can find ZooKeeper in the `zookeeper` container, which is running on the same Docker host.

NOTE

If you use Podman, run the following command:

```
$ sudo podman run -it --rm --name kafka --pod dbz debezium/kafka:1.6
```

SHELL

NOTE

In this tutorial, you will always connect to Kafka from within a Docker container. Any of these containers can communicate with the `kafka` container by linking to it. If you needed to connect to Kafka from *outside* of a Docker container, you would have to set the `-e` option to advertise the Kafka address through the Docker host (`-e ADVERTISED_HOST_NAME=` followed by either the IP address or resolvable host name of the Docker host).

2. Verify that Kafka started.

   You should see output similar to the following:

```
...
2017-09-21 07:16:59,085 - INFO  [main-EventThread:ZkClient@713] - zookeeper state changed
(SyncConnected)
2017-09-21 07:16:59,218 - INFO  [main:Logging$class@70] - Cluster ID =
LPtcBFxzRvOzDSXhc6AamA
...
2017-09-21 07:16:59,649 - INFO  [main:Logging$class@70] - [Kafka Server 1], started  ①
```

SHELL

① The Kafka broker has successfully started and is ready for client connections. The terminal will continue to show additional output as Kafka generates it.

Starting a MySQL database

At this point, you have started ZooKeeper and Kafka, but you still need a database server from which Debezium can capture changes. In this procedure, you will start a MySQL server with an

Procedure

1. Open a new terminal, and use it to start a new container that runs a MySQL database server preconfigured with an `inventory` database.

   This command runs a new container using version 1.6 of the `debezium/example-mysql` image, which is based on the mysql:5.7 image. It also defines and populates a sample `inventory` database:

   ```
   $ docker run -it --rm --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=debezium -e
   MYSQL_USER=mysqluser -e MYSQL_PASSWORD=mysqlpw debezium/example-mysql:1.6
   ```
   SHELL

   -it
   : The container is interactive, which means the terminal's standard input and output are attached to the container.

   --rm
   : The container will be removed when it is stopped.

   --name mysql
   : The name of the container.

   -p 3306:3306
   : Maps port `3306` (the default MySQL port) in the container to the same port on the Docker host so that applications outside of the container can connect to the database server.

   -e MYSQL_ROOT_PASSWORD=debezium -e MYSQL_USER=mysqluser -e MYSQL_PASSWORD=mysqlpw
   : Creates a user and password that has the minimum privileges required by the Debezium MySQL connector.

NOTE

If you use Podman, run the following command:

```
$ sudo podman run -it --rm --name mysql --pod dbz -e MYSQL_ROOT_PASSWORD=debezium -e
MYSQL_USER=mysqluser -e MYSQL_PASSWORD=mysqlpw debezium/example-mysql:1.6
```
SHELL

1. Verify that the MySQL server starts.

   The MySQL server starts and stops a few times as the configuration is modified. You should see output similar to the following:

   ```
   ...
   017-09-21T07:18:50.824629Z 0 [Note] mysqld: ready for connections.
   Version: '5.7.19-log'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306  MySQL Community
   Server (GPL)
   ```
   SHELL

After starting MySQL, you start a MySQL command line client so that you access the sample `inventory` database.

Procedure

1. Open a new terminal, and use it to start the MySQL command line client in a container.

   This command runs a new container using the mysql:5.7 image, and defines a shell command to run the MySQL command line client with the correct options:

   ```
   SHELL
   $ docker run -it --rm --name mysqlterm --link mysql --rm mysql:5.7 sh -c 'exec mysql -
   h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT" -uroot -
   p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
   ```

   `-it`
   : The container is interactive, which means the terminal's standard input and output are attached to the container.

   `--rm`
   : The container will be removed when it is stopped.

   `--name mysqlterm`
   : The name of the container.

   `--link mysql`
   : Links the container to the `mysql` container.

NOTE

If you use Podman, run the following command:

```
SHELL
$ sudo podman run -it --rm --name mysqlterm --pod dbz --rm mysql:5.7 sh -c 'exec mysql -h 0.0.0.0 -uroot
-pdebezium'
```

1. Verify that the MySQL command line client started.

   You should see output similar to the following:

   ```
   MYSQL
   mysql: [Warning] Using a password on the command line interface can be insecure.
   Welcome to the MySQL monitor.  Commands end with ; or \g.
   Your MySQL connection id is 3
   Server version: 5.7.17-log MySQL Community Server (GPL)

   Copyright (c) 2000, 2016, 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.
   ```

2. At the `mysql>` command prompt, switch to the inventory database:

```SQL
mysql> use inventory;
```

3. List the tables in the database:

```SQL
mysql> show tables;
+---------------------+
| Tables_in_inventory |
+---------------------+
| addresses           |
| customers           |
| geom                |
| orders              |
| products            |
| products_on_hand    |
+---------------------+
6 rows in set (0.00 sec)
```

4. Use the MySQL command line client to explore the database and view the pre-loaded data in the database.

   For example:

```SQL
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)
```

Starting Kafka Connect

After starting MySQL and connecting to the `inventory` database with the MySQL command line client, you start the Kafka Connect service. This service exposes a REST API to manage the Debezium MySQL connector.

Procedure

1. Open a new terminal, and use it to start the Kafka Connect service in a container.

   This command runs a new container using the 1.6 version of the `debezium/connect` image:

```SHELL
$ docker run -it --rm --name connect -p 8083:8083 -e GROUP_ID=1 -e
CONFIG_STORAGE_TOPIC=my_connect_configs -e OFFSET_STORAGE_TOPIC=my_connect_offsets -e
```

-it
: The container is interactive, which means the terminal's standard input and output are attached to the container.

--rm
: The container will be removed when it is stopped.

--name connect
: The name of the container.

-p 8083:8083
: Maps port `8083` in the container to the same port on the Docker host. This enables applications outside of the container to use Kafka Connect's REST API to set up and manage new container instances.

-e CONFIG_STORAGE_TOPIC=my_connect_configs -e OFFSET_STORAGE_TOPIC=my_connect_offsets -e STATUS_STORAGE_TOPIC=my_connect_statuses
: Sets environment variables required by the Debezium image.

--link zookeeper:zookeeper --link kafka:kafka --link mysql:mysql
: Links the container to the containers that are running ZooKeeper, Kafka, and the MySQL server.

NOTE

If you use Podman, run the following command:

```
$ sudo podman run -it --rm --name connect --pod dbz -e GROUP_ID=1 -e
CONFIG_STORAGE_TOPIC=my_connect_configs -e OFFSET_STORAGE_TOPIC=my_connect_offsets -e
STATUS_STORAGE_TOPIC=my_connect_statuses debezium/connect:1.6
```
SHELL

NOTE

If you provide a `--hostname` command option then Kafka Connect REST API will **not** listen on the `localhost` interface. This can cause issues when the REST port is being exposed.

If this is a problem then set environment variable `REST_HOST_NAME=0.0.0.0` which will ensure that REST API will be accessible from all interfaces.

1. Verify that Kafka Connect started and is ready to accept connections.

   You should see output similar to the following:

```
...
2020-02-06 15:48:33,939 INFO   ||  Kafka version: 2.4.0
[org.apache.kafka.common.utils.AppInfoParser]
...
```
SHELL

```
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2020-02-06 15:48:34,485 INFO   ||  [Worker clientId=connect-1, groupId=1] Finished starting
connectors and tasks   [org.apache.kafka.connect.runtime.distributed.DistributedHerder]
```

2. Use the Kafka Connect REST API to check the status of the Kafka Connect service.

   Kafka Connect exposes a REST API to manage Debezium connectors. To communicate with the Kafka Connect service, you can use the `curl` command to send API requests to port 8083 of the Docker host (which you mapped to port 8083 in the `connect` container when you started Kafka Connect).

   NOTE

   These commands use `localhost` . If you are using a non-native Docker platform (such as Docker Toolbox), replace `localhost` with the IP address of your Docker host.

   a. Open a new terminal and check the status of the Kafka Connect service:

   ```
   $ curl -H "Accept:application/json" localhost:8083/
   {"version":"2.7.0","commit":"cb8625948210849f"}  ①
   ```

   ① The response shows that Kafka Connect version 2.7.0 is running.

   b. Check the list of connectors registered with Kafka Connect:

   ```
   $ curl -H "Accept:application/json" localhost:8083/connectors/
   []  ①
   ```

   ① No connectors are currently registered with Kafka Connect.

Deploying the MySQL connector

After starting the Debezium and MySQL services, you are ready to deploy the Debezium MySQL connector so that it can start monitoring the sample MySQL database (`inventory`).

At this point, you are running the Debezium services, a MySQL database server with a sample `inventory` database, and the MySQL command line client that is connected to the database. To deploy the MySQL connector, you must:

- Register the MySQL connector to monitor the `inventory` database

  After the connector is registered, it will start monitoring the database server's `binlog` and it will generate change events for each row that changes.

- Watch the MySQL connector start

each task it must complete before it can start monitoring the `binlog`.

Registering a connector to monitor the `inventory` database

By registering the Debezium MySQL connector, the connector will start monitoring the MySQL database server's `binlog`. The `binlog` records all of the database's transactions (such as changes to individual rows and changes to the schemas). When a row in the database changes, Debezium generates a change event.

NOTE

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

Procedure

1. Review the configuration of the Debezium MySQL connector that you will register.

   Before registering the connector, you should be familiar with its configuration. In the next step, you will register the following connector:

```JSON
{
  "name": "inventory-connector",  ①
  "config": {  ②
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "tasks.max": "1",  ③
    "database.hostname": "mysql",  ④
    "database.port": "3306",
    "database.user": "debezium",
    "database.password": "dbz",
    "database.server.id": "184054",  ⑤
    "database.server.name": "dbserver1",  ⑤
    "database.include.list": "inventory",  ⑥
    "database.history.kafka.bootstrap.servers": "kafka:9092",  ⑦
    "database.history.kafka.topic": "schema-changes.inventory"  ⑦
  }
}
```

① The name of the connector.

② The connector's configuration.

③ Only one task should operate at any one time. Because the MySQL connector reads the MySQL server's `binlog`, using a single connector task ensures proper order and event handling. The Kafka Connect service uses connectors to start one or more tasks that do the work, and it automatically distributes the running tasks across the cluster of Kafka Connect services. If any of the services stop or crash, those tasks will be redistributed to running services.

④ The database host, which is the name of the Docker container running the MySQL server (`mysql`). Docker manipulates the network stack within the containers so that each linked container can be resolved with `/etc/hosts` using the container name for the host name. If MySQL were running on a normal network, you would specify the IP address or resolvable host name for this value.

⑥ Only changes in the `inventory` database will be detected.

⑦ The connector will store the history of the database schemas in Kafka using this broker (the same broker to which you are sending events) and topic name. Upon restart, the connector will recover the schemas of the database that existed at the point in time in the `binlog` when the connector should begin reading.

For more information, see MySQL connector configuration properties.

2. Open a new terminal, and use the `curl` command to register the Debezium MySQL connector.

   This command uses the Kafka Connect service's API to submit a `POST` request against the `/connectors` resource with a JSON document that describes the new connector (called `inventory-connector`).

   This command uses `localhost` to connect to the Docker host. If you are using a non-native Docker platform, replace `localhost` with the IP address of of your Docker host.

```shell
$ curl -i -X POST -H "Accept:application/json" -H "Content-Type:application/json"
localhost:8083/connectors/ -d '{ "name": "inventory-connector", "config": {
"connector.class": "io.debezium.connector.mysql.MySqlConnector", "tasks.max": "1",
"database.hostname": "mysql", "database.port": "3306", "database.user": "debezium",
"database.password": "dbz", "database.server.id": "184054", "database.server.name":
"dbserver1", "database.include.list": "inventory",
"database.history.kafka.bootstrap.servers": "kafka:9092", "database.history.kafka.topic":
"dbhistory.inventory" } }'
```

NOTE

Windows users may need to escape the double-quotes. For example:

```shell
$ curl -i -X POST -H "Accept:application/json" -H "Content-Type:application/json"
localhost:8083/connectors/ -d '{ \"name\": \"inventory-connector\", \"config\": {
\"connector.class\": \"io.debezium.connector.mysql.MySqlConnector\", \"tasks.max\": \"1\",
\"database.hostname\": \"mysql\", \"database.port\": \"3306\", \"database.user\": \"debezium\",
\"database.password\": \"dbz\", \"database.server.id\": \"184054\", \"database.server.name\":
\"dbserver1\", \"database.include.list\": \"inventory\",
\"database.history.kafka.bootstrap.servers\": \"kafka:9092\", \"database.history.kafka.topic\":
\"dbhistory.inventory\" } }'
```

Otherwise, you might see an error like the following:

```json
{"error_code":500,"message":"Unexpected character ('n' (code 110)): was expecting double-quote to
start field name\n at [Source:
(org.glassfish.jersey.message.internal.ReaderInterceptorExecutor$UnCloseableInputStream); line: 1,
column: 4]"}
```

NOTE

```
$ curl -i -X POST -H "Accept:application/json" -H "Content-Type:application/json"
localhost:8083/connectors/ -d '{ "name": "inventory-connector", "config": { "connector.class":
"io.debezium.connector.mysql.MySqlConnector", "tasks.max": "1", "database.hostname": "0.0.0.0",
"database.port": "3306", "database.user": "debezium", "database.password": "dbz", "database.server.id":
"184054", "database.server.name": "dbserver1", "database.include.list": "inventory",
"database.history.kafka.bootstrap.servers": "0.0.0.0:9092", "database.history.kafka.topic":
"dbhistory.inventory" } }'
```
SHELL

1. Verify that `inventory-connector` is included in the list of connectors:

```
$ curl -H "Accept:application/json" localhost:8083/connectors/
["inventory-connector"]
```
SHELL

2. Review the connector's tasks:

```
$ curl -i -X GET -H "Accept:application/json" localhost:8083/connectors/inventory-connector
```
SHELL

You should see a response similar to the following (formatted for readability):

```
HTTP/1.1 200 OK
Date: Thu, 06 Feb 2020 22:12:03 GMT
Content-Type: application/json
Content-Length: 531
Server: Jetty(9.4.20.v20190813)

{
  "name": "inventory-connector",
  ...
  "tasks": [
    {
      "connector": "inventory-connector",  ①
      "task": 0
    }
  ]
}
```
JSON

① The connector is running a single task (task `0`) to do its work. The connector only supports a single task, because MySQL records all of its activities in one sequential `binlog`. This means the connector only needs one reader to get a consistent, ordered view of all of the events.

Watching the connector start

When you register a connector, it generates a large amount of log output in the Kafka Connect container. By reviewing this output, you can better understand the process that the connector goes through from the time it is created until it begins reading the MySQL server's `binlog`.

After registering the `inventory-connector` connector, you can review the log output in the Kafka Connect container (`connect`) to track the connector's status.

```
...
2017-09-21 07:23:59,051 INFO   ||  Connector inventory-connector config updated
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,550 INFO   ||  Rebalance started
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,550 INFO   ||  Finished stopping tasks in preparation for rebalance
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,550 INFO   ||  (Re-)joining group 1
[org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2017-09-21 07:23:59,556 INFO   ||  Successfully joined group 1 with generation 2
[org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2017-09-21 07:23:59,556 INFO   ||  Joined group and got assignment: Assignment{error=0,
leader='connect-1-4d60cb71-cb93-4388-8908-6f0d299a9d94', leaderUrl='http://172.17.0.7:9092/',
offset=1, connectorIds=[inventory-connector], taskIds=[]}
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,557 INFO   ||  Starting connectors and tasks using config offset 1
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,557 INFO   ||  Starting connector inventory-connector
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
...
```

SHELL

Further down, you should see output like the following from the connector:

```
...
2017-09-21 07:24:01,151 INFO   MySQL|dbserver1|task  Kafka version : 0.11.0.0
[org.apache.kafka.common.utils.AppInfoParser]
2017-09-21 07:24:01,151 INFO   MySQL|dbserver1|task  Kafka commitId : cb8625948210849f
[org.apache.kafka.common.utils.AppInfoParser]
2017-09-21 07:24:01,584 INFO   MySQL|dbserver1|task  Found no existing offset, so preparing to
perform a snapshot   [io.debezium.connector.mysql.MySqlConnectorTask]
2017-09-21 07:24:01,614 INFO   ||  Source task WorkerSourceTask{id=inventory-connector-0}
finished initialization and start   [org.apache.kafka.connect.runtime.WorkerSourceTask]
2017-09-21 07:24:01,615 INFO   MySQL|dbserver1|snapshot  Starting snapshot for
jdbc:mysql://mysql:3306/?
useInformationSchema=true&nullCatalogMeansCurrent=false&useSSL=false&useUnicode=true&characterEncod
8&characterSetResults=UTF-8&zeroDateTimeBehavior=convertToNull with user 'debezium'
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,617 INFO   MySQL|dbserver1|snapshot  Snapshot is using user 'debezium' with
these MySQL grants:   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,618 INFO   MySQL|dbserver1|snapshot        GRANT SELECT, RELOAD, SHOW
DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'debezium'@'%'
[io.debezium.connector.mysql.SnapshotReader]
...
```

SHELL

The Debezium log output uses *mapped diagnostic contexts* (MDC) to provide thread-specific information in the log output, and make it easier to understand what is happening in the multi-threaded Kafka Connect service. This includes the connector type (`MySQL` in the above log messages), the logical name of the connector (`dbserver1` above), and the connector's activity (`task` , `snapshot` and `binlog`).

In the log output above, the first few lines involve the `task` activity of the connector, and report some bookkeeping information (in this case, that the connector was started with no prior offset). The

started using the "debezium" MySQL user along with the MySQL grants associated with that user.

TIP

If the connector is not able to connect, or if it does not see any tables or the `binlog`, check these grants to ensure that all of those listed above are included.

Next, the connector reports the relevant MySQL server settings that it has found. One of the most important is `binlog_format`, which is set to `ROW`:

SHELL

```
2017-09-21 07:24:01,618 INFO   MySQL|dbserver1|snapshot  MySQL server variables related to change
data capture:   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,625 INFO   MySQL|dbserver1|snapshot         binlog_cache_size
= 32768                                          [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,625 INFO   MySQL|dbserver1|snapshot         binlog_checksum
= CRC32                                          [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,625 INFO   MySQL|dbserver1|snapshot
binlog_direct_non_transactional_updates       = OFF
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,625 INFO   MySQL|dbserver1|snapshot         binlog_error_action
= ABORT_SERVER                                   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_format
= ROW                                            [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_group_commit_sync_delay
= 0                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot
binlog_group_commit_sync_no_delay_count       = 0
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_gtid_simple_recovery
= ON                                             [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_max_flush_queue_time
= 0                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_order_commits
= ON                                             [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_row_image
= FULL                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_rows_query_log_events
= OFF                                            [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         binlog_stmt_cache_size
= 32768                                          [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_set_client
= utf8                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_set_connection
= utf8                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_set_database
= latin1                                         [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_set_filesystem
= binary                                         [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_set_results
= utf8                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_set_server
= latin1                                         [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_set_system
= utf8                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         character_sets_dir
= /usr/share/mysql/charsets/                     [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         collation_connection
= utf8_general_ci                                [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         collation_database
```

```
= latin1_swedish_ci                                [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         enforce_gtid_consistency
= OFF                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,626 INFO   MySQL|dbserver1|snapshot         gtid_executed_compression_period
= 1000                                             [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         gtid_mode
= OFF                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         gtid_next
= AUTOMATIC                                        [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         gtid_owned
=                                                  [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         gtid_purged
=                                                  [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         innodb_api_enable_binlog
= OFF                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         innodb_locks_unsafe_for_binlog
= OFF                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         innodb_version
= 5.7.19                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         log_statements_unsafe_for_binlog
= ON                                               [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         max_binlog_cache_size
= 18446744073709547520                             [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         max_binlog_size
= 1073741824                                       [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         max_binlog_stmt_cache_size
= 18446744073709547520                             [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         protocol_version
= 10                                               [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         session_track_gtids
= OFF                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         slave_type_conversions
=                                                  [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         sync_binlog
= 1                                                [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         system_time_zone
= UTC                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         time_zone
= SYSTEM                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         tls_version
= TLSv1,TLSv1.1                                    [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         tx_isolation
= REPEATABLE-READ                                  [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         tx_read_only
= OFF                                              [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         version
= 5.7.19-log                                       [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         version_comment
= MySQL Community Server (GPL)                     [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,627 INFO   MySQL|dbserver1|snapshot         version_compile_machine
= x86_64                                           [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,628 INFO   MySQL|dbserver1|snapshot         version_compile_os
= Linux                                            [io.debezium.connector.mysql.SnapshotReader]
...
```

Next, the connector reports the nine steps that make up the snapshot operation:

SHELL

```
...
2017-09-21 07:24:01,628 INFO   MySQL|dbserver1|snapshot  Step 0: disabling autocommit and
```

```
consistent snapshot   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,634 INFO   MySQL|dbserver1|snapshot  Step 2: flush and obtain global read
lock to prevent writes to database   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,636 INFO   MySQL|dbserver1|snapshot  Step 3: read binlog position of MySQL
primary server   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,638 INFO   MySQL|dbserver1|snapshot          using binlog 'mysql-bin.000003'
at position '154' and gtid ''   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,638 INFO   MySQL|dbserver1|snapshot  Step 4: read list of available databases
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,638 INFO   MySQL|dbserver1|snapshot          list of available databases is:
[information_schema, inventory, mysql, performance_schema, sys]
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,639 INFO   MySQL|dbserver1|snapshot  Step 5: read list of available tables in
each database   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,641 INFO   MySQL|dbserver1|snapshot          including 'inventory.customers'
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,641 INFO   MySQL|dbserver1|snapshot          including 'inventory.orders'
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,641 INFO   MySQL|dbserver1|snapshot          including 'inventory.products'
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,641 INFO   MySQL|dbserver1|snapshot          including
'inventory.products_on_hand'   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,642 INFO   MySQL|dbserver1|snapshot          'mysql.columns_priv' is filtered
out, discarding   [io.debezium.connector.mysql.SnapshotReader]
...
2017-09-21 07:24:01,670 INFO   MySQL|dbserver1|snapshot         snapshot continuing with
database(s): [inventory]   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,670 INFO   MySQL|dbserver1|snapshot  Step 6: generating DROP and CREATE
statements to reflect current database schemas:   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,679 INFO   MySQL|dbserver1|snapshot         SET character_set_server=latin1,
collation_server=latin1_swedish_ci;   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,724 WARN   MySQL|dbserver1|task  Error while fetching metadata with
correlation id 1 : {dbhistory.inventory=LEADER_NOT_AVAILABLE}
[org.apache.kafka.clients.NetworkClient]
2017-09-21 07:24:01,853 INFO   MySQL|dbserver1|snapshot         DROP TABLE IF EXISTS
`inventory`.`products_on_hand`   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,861 INFO   MySQL|dbserver1|snapshot         DROP TABLE IF EXISTS
`inventory`.`customers`   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,864 INFO   MySQL|dbserver1|snapshot         DROP TABLE IF EXISTS
`inventory`.`orders`   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,866 INFO   MySQL|dbserver1|snapshot         DROP TABLE IF EXISTS
`inventory`.`products`   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,881 INFO   MySQL|dbserver1|snapshot         DROP DATABASE IF EXISTS
`inventory`   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,889 INFO   MySQL|dbserver1|snapshot         CREATE DATABASE `inventory`
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,893 INFO   MySQL|dbserver1|snapshot         USE `inventory`
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,914 INFO   MySQL|dbserver1|snapshot         CREATE TABLE `customers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `email` (`email`)
) ENGINE=InnoDB AUTO_INCREMENT=1005 DEFAULT CHARSET=latin1
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,932 INFO   MySQL|dbserver1|snapshot         CREATE TABLE `orders` (
  `order_number` int(11) NOT NULL AUTO_INCREMENT,
  `order_date` date NOT NULL,
  `purchaser` int(11) NOT NULL,
  `quantity` int(11) NOT NULL,
```

```
  KEY `order_customer` (`purchaser`),
  KEY `ordered_product` (`product_id`),
  CONSTRAINT `orders_ibfk_1` FOREIGN KEY (`purchaser`) REFERENCES `customers` (`id`),
  CONSTRAINT `orders_ibfk_2` FOREIGN KEY (`product_id`) REFERENCES `products` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=10005 DEFAULT CHARSET=latin1
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,937 INFO   MySQL|dbserver1|snapshot         CREATE TABLE `products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `description` varchar(512) DEFAULT NULL,
  `weight` float DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=110 DEFAULT CHARSET=latin1
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,941 INFO   MySQL|dbserver1|snapshot         CREATE TABLE `products_on_hand` (
  `product_id` int(11) NOT NULL,
  `quantity` int(11) NOT NULL,
  PRIMARY KEY (`product_id`),
  CONSTRAINT `products_on_hand_ibfk_1` FOREIGN KEY (`product_id`) REFERENCES `products` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,947 INFO   MySQL|dbserver1|snapshot  Step 7: releasing global read lock to
enable MySQL writes   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,949 INFO   MySQL|dbserver1|snapshot  Step 7: blocked writes to MySQL for a
total of 00:00:00.312   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,950 INFO   MySQL|dbserver1|snapshot  Step 8: scanning contents of 4 tables
while still in transaction   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,953 INFO   MySQL|dbserver1|snapshot  Step 8: - scanning table
'inventory.customers' (1 of 4 tables)   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,958 INFO   MySQL|dbserver1|snapshot  Step 8: - Completed scanning a total of
4 rows from table 'inventory.customers' after 00:00:00.005
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,959 INFO   MySQL|dbserver1|snapshot  Step 8: - scanning table
'inventory.orders' (2 of 4 tables)   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,014 INFO   MySQL|dbserver1|snapshot  Step 8: - Completed scanning a total of
4 rows from table 'inventory.orders' after 00:00:00.055
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,016 INFO   MySQL|dbserver1|snapshot  Step 8: - scanning table
'inventory.products' (3 of 4 tables)   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,017 INFO   MySQL|dbserver1|snapshot  Step 8: - Completed scanning a total of
9 rows from table 'inventory.products' after 00:00:00.001
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,018 INFO   MySQL|dbserver1|snapshot  Step 8: - scanning table
'inventory.products_on_hand' (4 of 4 tables)   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,019 INFO   MySQL|dbserver1|snapshot  Step 8: - Completed scanning a total of
9 rows from table 'inventory.products_on_hand' after 00:00:00.001
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,020 INFO   MySQL|dbserver1|snapshot  Step 8: scanned 26 rows in 4 tables in
00:00:00.069   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,020 INFO   MySQL|dbserver1|snapshot  Step 9: committing transaction
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:02,021 INFO   MySQL|dbserver1|snapshot  Completed snapshot in 00:00:00.405
[io.debezium.connector.mysql.SnapshotReader]
...
```

Each of these steps reports what the connector is doing to perform the consistent snapshot. For example, Step 6 involves reverse engineering the DDL `create` statements for the tables that are being captured. Step 7 releases the global write lock just 0.3 seconds after acquiring it, and Step 8

this case, the connector completed its consistent snapshot in just over one second.

NOTE

The snapshot process will take longer with your databases, but the connector outputs enough log messages that you can track what it is working on, even when the tables have a large number of rows. And although an exclusive write lock is used at the beginning of the snapshot process, it should not last very long even for large databases. This is because the lock is released before any data is copied. For more information, see the MySQL connector documentation.

Next, Kafka Connect reports some "errors". However, you can safely ignore these warnings: these messages just mean that *new* Kafka topics were created and that Kafka had to assign a new leader for each one:

```
...
2017-09-21 07:24:02,632 WARN   ||  Error while fetching metadata with correlation id 1 :
{dbserver1=LEADER_NOT_AVAILABLE}   [org.apache.kafka.clients.NetworkClient]
2017-09-21 07:24:02,775 WARN   ||  Error while fetching metadata with correlation id 5 :
{dbserver1.inventory.customers=LEADER_NOT_AVAILABLE}   [org.apache.kafka.clients.NetworkClient]
2017-09-21 07:24:02,910 WARN   ||  Error while fetching metadata with correlation id 9 :
{dbserver1.inventory.orders=LEADER_NOT_AVAILABLE}   [org.apache.kafka.clients.NetworkClient]
2017-09-21 07:24:03,045 WARN   ||  Error while fetching metadata with correlation id 13 :
{dbserver1.inventory.products=LEADER_NOT_AVAILABLE}   [org.apache.kafka.clients.NetworkClient]
2017-09-21 07:24:03,179 WARN   ||  Error while fetching metadata with correlation id 17 :
{dbserver1.inventory.products_on_hand=LEADER_NOT_AVAILABLE}
[org.apache.kafka.clients.NetworkClient]
...
```

Finally, the log output shows that the connector has transitioned from its snapshot mode into continuously reading the MySQL server's `binlog`:

```
...
Sep 21, 2017 7:24:03 AM com.github.shyiko.mysql.binlog.BinaryLogClient connect
INFO: Connected to mysql:3306 at mysql-bin.000003/154 (sid:184054, cid:7)
2017-09-21 07:24:03,373 INFO   MySQL|dbserver1|binlog  Connected to MySQL binlog at mysql:3306,
starting at binlog file 'mysql-bin.000003', pos=154, skipping 0 events plus 0 rows
[io.debezium.connector.mysql.BinlogReader]
2017-09-21 07:25:01,096 INFO   ||  Finished WorkerSourceTask{id=inventory-connector-0}
commitOffsets successfully in 18 ms   [org.apache.kafka.connect.runtime.WorkerSourceTask]
...
```

Viewing change events

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

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

The schema change topic to which all of the DDL statements are written.

dbserver1.inventory.products
: Captures change events for the `products` table in the `inventory` database.

dbserver1.inventory.products_on_hand
: Captures change events for the `products_on_hand` table in the `inventory` database.

dbserver1.inventory.customers
: Captures change events for the `customers` table in the `inventory` database.

dbserver1.inventory.orders
: Captures change events for the `orders` table in the `inventory` database.

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

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

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 it to start the `watch-topic` utility to watch the `dbserver1.inventory.customers` topic from the beginning of the topic.

   The `watch-topic` utility is very simple and limited in functionality. It is not intended to be used by an application to consume events. In that scenario, you would instead use Kafka consumers and the applicable consumer libraries that offer full functionality and flexibility.

   This command runs the `watch-topic` utility in a new container using the 1.6 version of the `debezium/kafka` image:

   ```
   $ docker run -it --rm --name watcher --link zookeeper:zookeeper --link kafka:kafka
   debezium/kafka:1.6 watch-topic -a -k dbserver1.inventory.customers
   ```
   SHELL

   -a
   : Watches all events since the topic was created. Without this option, `watch-topic` would only show the events recorded after you start watching.

   -k

NOTE

If you use Podman, run the following command:

```shell
$ sudo podman run -it --rm --name watcher --pod dbz debezium/kafka:1.6 watch-topic -a -k
dbserver1.inventory.customers
```

The `watch-topic` utility returns the event records from the `customers` table. There are four events, one for each row in the table. Each event is formatted in JSON, because that is how you configured the Kafka Connect service. There are two JSON documents for each event: one for the key, and one for the value.

You should see output similar to the following:

```shell
Using ZOOKEEPER_CONNECT=172.17.0.2:2181
Using KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://172.17.0.7:9092
Using KAFKA_BROKER=172.17.0.3:9092
Contents of topic dbserver1.inventory.customers:
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"id"}],"optional":false,"name":"dbserver1.inventory.c
{"id":1001}}
...
```

NOTE

This utility keeps watching the topic, so any new events will automatically appear as long as the utility is running.

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
  }
}
```

schema describing what is in the payload. In this case, the payload is a `Struct` named `dbserver1.inventory.customers.Key` that is not optional and has one required field (`id` of type `int32`).

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

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

3. Review the details of the same event's *value*.

   The event's *value* shows that the row was created, and describes what it contains (in this case, the `id` , `first_name` , `last_name` , and `email` of the inserted row).

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

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

```
        "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",
```

```
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "db"
          },
          {
            "type": "string",
            "optional": true,
            "field": "table"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mysql.Source",
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "dbserver1.inventory.customers.Envelope",
    "version": 1
  },
  "payload": {
    "before": null,
    "after": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.6.0.Final",
      "name": "dbserver1",
      "server_id": 0,
      "ts_sec": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,
      "snapshot": true,
      "thread": null,
      "db": "inventory",
      "table": "customers"
    },
    "op": "c",
    "ts_ms": 1486500577691
  }
}
```

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

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. Alternatively, you can use Apache Avro as a serialization format, which results in far smaller event messages. This is because it transforms each Kafka Connect schema into an Avro schema and stores the Avro schemas in a separate Schema Registry service. Thus, when the Avro converter serializes an event message, it places only a unique identifier for the schema along with an Avro-encoded binary representation of the value. As a result, the serialized messages that are transferred over the wire and stored in Kafka are far smaller than what you have seen here. In fact, the Avro Converter is able to use Avro schema evolution techniques to maintain the history of each schema in the Schema Registry.

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;
+------+------------+-----------+-----------------------+
| 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.

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.

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

```SQL
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
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 `watch-topic` 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": {  ①
    "id": 1004,
    "first_name": "Anne",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "after": {  ②
    "id": 1004,
    "first_name": "Anne Marie",
    "last_name": "Kretchmar",
    "email": "annek@noanswer.org"
  },
  "source": {  ③
    "name": "1.6.0.Final",
    "name": "dbserver1",
    "server_id": 223344,
    "ts_sec": 1486501486,
    "gtid": null,
    "file": "mysql-bin.000003",
    "pos": 364,
    "row": 0,
    "snapshot": null,
    "thread": 3,
    "db": "inventory",
    "table": "customers"
  },
  "op": "u",  ④
  "ts_ms": 1486501486308  ⑤
}
}
```

① 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.

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 `watch-topic` 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
      }
    ]
  },
  "payload": {
    "id": 1004
  }
}
```

   This *key* is the same as the *key* in the previous two events you looked at.

JSON


```
{
  "schema": {...},
  "payload": {
    "before": {  ①
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,  ②
    "source": {  ③
      "name": "1.6.0.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 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",
```

```
    },
    "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.

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. Open a new terminal and use it to stop the `connect` container that is running the Kafka Connect service:

SHELL

```
$ docker stop connect
```

The `connect` container is stopped, and the Kafka Connect service gracefully shuts down.

Because you ran the container with the `--rm` option, Docker removes the container once it stops.

few records.

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

SQL

The records are added to the database. However, because Kafka Connect is not running, `watch-topic` does not record any updates.

NOTE

In a production system, you would have enough brokers to handle the producers and consumers, and to maintain a minimum number of in-sync replicas for each topic. Therefore, if enough brokers failed such that there were no longer the minimum number of ISRs, Kafka would become unavailable. In this scenario, producers (like the Debezium connectors) and consumers will wait for the Kafka cluster or network to recover. This means that, temporarily, your consumers might not see any change events as data is changed in the databases. This is because no change events are being produced. As soon as the Kafka cluster is restarted or the network recovers, Debezium will resume producing change events, and your consumers will resume consuming events where they left off.

3. Open a new terminal, and use it to restart the Kafka Connect service in a container.

   This command starts Kafka Connect using the same options you used when you initially started it:

```
$ docker run -it --rm --name connect -p 8083:8083 -e GROUP_ID=1 -e
CONFIG_STORAGE_TOPIC=my_connect_configs -e OFFSET_STORAGE_TOPIC=my_connect_offsets -e
STATUS_STORAGE_TOPIC=my_connect_statuses --link zookeeper:zookeeper --link kafka:kafka --
link mysql:mysql debezium/connect:1.6
```

SHELL

   The Kafka Connect service starts, connects to Kafka, reads the previous service's configuration, and starts the registered connectors that will resume where they last left off.

   Here are the last few lines from this restarted service:

```
...
2017-09-21 07:38:48,385 INFO   MySQL|dbserver1|task  Kafka version : 0.11.0.0
[org.apache.kafka.common.utils.AppInfoParser]
2017-09-21 07:38:48,386 INFO   MySQL|dbserver1|task  Kafka commitId : cb8625948210849f
[org.apache.kafka.common.utils.AppInfoParser]
2017-09-21 07:38:48,390 INFO   MySQL|dbserver1|task  Discovered coordinator 172.17.0.4:9092
(id: 2147483646 rack: null) for group inventory-connector-dbhistory.
[org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2017-09-21 07:38:48,390 INFO   MySQL|dbserver1|task  Revoking previously assigned partitions
[] for group inventory-connector-dbhistory
[org.apache.kafka.clients.consumer.internals.ConsumerCoordinator]
2017-09-21 07:38:48,391 INFO   MySQL|dbserver1|task  (Re-)joining group inventory-connector-
dbhistory   [org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2017-09-21 07:38:48,402 INFO   MySQL|dbserver1|task  Successfully joined group inventory-
connector-dbhistory with generation 1
[org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2017-09-21 07:38:48,403 INFO   MySQL|dbserver1|task  Setting newly assigned partitions
```

SHELL

```
2017-09-21 07:38:48,888 INFO   MySQL|dbserver1|task  Step 0: Get all known binlogs from
MySQL   [io.debezium.connector.mysql.MySqlConnectorTask]
2017-09-21 07:38:48,903 INFO   MySQL|dbserver1|task  MySQL has the binlog file 'mysql-
bin.000003' required by the connector   [io.debezium.connector.mysql.MySqlConnectorTask]
Sep 21, 2017 7:38:49 AM com.github.shyiko.mysql.binlog.BinaryLogClient connect
INFO: Connected to mysql:3306 at mysql-bin.000003/154 (sid:184054, cid:10)
2017-09-21 07:38:49,045 INFO   MySQL|dbserver1|binlog  Connected to MySQL binlog at
mysql:3306, starting at binlog file 'mysql-bin.000003', pos=154, skipping 0 events plus 0
rows   [io.debezium.connector.mysql.BinlogReader]
2017-09-21 07:38:49,046 INFO   ||  Source task WorkerSourceTask{id=inventory-connector-0}
finished initialization and start   [org.apache.kafka.connect.runtime.WorkerSourceTask]
```

These lines show that the service found the offsets previously recorded by the last task before it was shut down, connected to the MySQL database, started reading the `binlog` from that position, and generated events from any changes in the MySQL database since that point in time.

4. Switch to the terminal running `watch-topic` to see events for the two new records you created when Kafka Connect was offline:

JSON


```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"id"}],"optional":false,"name":"dbserver1.inventory.c
{"id":1005}}    {"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
{"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
{"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.connect
{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"}],"optional":false,"name":"dbserver1.inventory.
{"before":null,"after":
{"id":1005,"first_name":"Sarah","last_name":"Thompson","email":"kitt@acme.com"},"source":
{"version":"1.6.0.Final","name":"dbserver1","server_id":223344,"ts_sec":1490635153,"gtid":null,
bin.000003","pos":1046,"row":0,"snapshot":null,"thread":3,"db":"inventory","table":"customers"}

{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"id"}],"optional":false,"name":"dbserver1.inventory.c
{"id":1006}}    {"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
```

```
{"type":"string","optional":false,"field":"email"}],"optional":true,"name":"dbserver1.inventory
{"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.connect
{"type":"string","optional":false,"field":"op"},
{"type":"int64","optional":true,"field":"ts_ms"}],"optional":false,"name":"dbserver1.inventory.
{"before":null,"after":
{"id":1006,"first_name":"Kenneth","last_name":"Anderson","email":"kander@acme.com"},"source":
{"version":"1.6.0.Final","name":"dbserver1","server_id":223344,"ts_sec":1490635160,"gtid":null,
bin.000003","pos":1356,"row":0,"snapshot":null,"thread":3,"db":"inventory","table":"customers"}
```


These events are *create* events that are similar to what you saw previously. As you see, Debezium still reports all of the changes in a database even when it is not running (as long as it is restarted before the MySQL database purges from its `binlog` the commits that were missed).

Cleaning up

After you are finished with the tutorial, you can use Docker to stop all of the running containers.

Procedure

1. Stop each of the containers:

```
SHELL
$ docker stop mysqlterm watcher connect mysql kafka zookeeper
```

Docker stops each container. Because you used the `--rm` option when you started them, Docker also removes them.

NOTE

If you use Podman, run the following command:

```
SHELL
$ sudo podman pod kill dbz
$ sudo podman pod rm dbz
```

1. Verify that all of the processes have stopped and have been removed:

```
SHELL
$ docker ps -a
```

If any of the processes are still running, stop them using `docker stop <process-name>` or `docker stop <containerId>`.

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. You may need to run a separate `watch-topic` command for each topic. Keep in mind that you cannot remove a row that is referenced by a foreign key.
- Try running the tutorial with Debezium connectors for Postgres, MongoDB, SQL Server, and Oracle.

  You can use the Docker Compose version of this tutorial located in the Debezium examples repository. Docker Compose files are provided for running the tutorial with MySQL, Postgres, MongoDB, SQL Server, and Oracle.