Red Hat Integration 2022.Q2

Installing Debezium on OpenShift

For use with Debezium 1.7 on OpenShift Container Platform

Last Updated: 2022-05-23

Red Hat Integration 2022.Q2 Installing Debezium on OpenShift

For use with Debezium 1.7 on OpenShift Container Platform

Legal Notice

Abstract

This guide describes how to install Red Hat Debezium on OpenShift Container Platform with AMQ Streams.

Table of Contents

PREFACE

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. DEBEZIUM OVERVIEW

Debezium for Red Hat Integration is a distributed platform that captures database operations, creates data change event records for row-level operations, and streams change event records to Apache Kafka topics. Debezium is built on Apache Kafka and is deployed and integrated with AMQ Streams.

Debezium captures row-level changes to a database table and passes corresponding change events to AMQ Streams. Applications can read these *change event streams* and access the change events in the order in which they occurred.

Debezium is the upstream community project for Debezium for Red Hat Integration.

Debezium has multiple uses, including:

- Data replication
- Updating caches and search indexes
- Simplifying monolithic applications
- Data integration
- Enabling streaming queries

Debezium provides connectors (based on Kafka Connect) for the following common databases:

- Db2
- MySQL
- MongoDB
- Oracle (Technology Preview)
- PostgreSQL
- SQL Server

CHAPTER 2. INSTALLING DEBEZIUM CONNECTORS

Install Debezium connectors through AMQ Streams by extending Kafka Connect with connector plug-ins. Following a deployment of AMQ Streams, you can deploy Debezium as a connector configuration through Kafka Connect.

2.1. KAFKA TOPIC CREATION RECOMMENDATIONS

Debezium stores data in multiple Apache Kafka topics. The topics must either be created in advance by an administrator, or you can configure Kafka Connect to configure topics automatically.

The following list describes limitations and recommendations to consider when creating topics:

Database history topics for MySQL, SQL Server, Db2, and Oracle connectors

- Infinite or very long retention.
- Replication factor of at least three in production environments.
- Single partition.

Other topics

- When you enable Kafka log compaction so that only the *last* change event for a given record is saved, set the following topic properties in Apache Kafka:
  - **min.compaction.lag.ms**
  - **delete.retention.ms**
    To ensure that topic consumers have enough time to receive all events and delete markers, specify values for the preceding properties that are larger than the maximum downtime that you expect for your sink connectors. For example, consider the downtime that might occur when you apply updates to sink connectors.
- Replicated in production.
- Single partition.
  You can relax the single partition rule, but your application must handle out-of-order events for different rows in the database. Events for a single row are still totally ordered. If you use multiple partitions, the default behavior is that Kafka determines the partition by hashing the key. Other partition strategies require the use of single message transformations (SMTs) to set the partition number for each record.

2.2. DEBEZIUM DEPLOYMENT ON AMQ STREAMS

To set up connectors for Debezium on Red Hat OpenShift Container Platform, you use AMQ Streams to build a Kafka Connect container image that includes the connector plug-in for each connector that you want to use. After the connector starts, it connects to the configured database and generates change event records for each inserted, updated, and deleted row or document.

Beginning with Debezium 1.7, the preferred method for deploying a Debezium connector is to use AMQ Streams to build a Kafka Connect container image that includes the 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 connector artifacts needs to include in the image.
- A **KafkaConnector** CR that provides details that include information the connector uses to access the source database. After AMQ Streams starts the Kafka Connect pod, you start the connector by applying the **KafkaConnector** CR.

In the build specification for the Kafka Connect image, you can specify the connectors that are available to deploy. For each connector plug-in, you can also specify other components that you want to make available for deployment. For example, you can add Service Registry artifacts, or the Debezium scripting component. When AMQ Streams builds the Kafka Connect image, it downloads the specified artifacts, and incorporates them into the image.

The **spec.build.output** parameter in the **KafkaConnect** CR specifies where to store the resulting Kafka Connect container image. Container images can be stored in a Docker registry, or in an OpenShift ImageStream. To store images in an ImageStream, you must create the ImageStream before you deploy Kafka Connect. ImageStreams are not created automatically.

NOTE

If you use a **KafkaConnect** resource to create a cluster, afterwards you cannot use the Kafka Connect REST API to create or update connectors. You can still use the REST API to retrieve information.

Additional resources

- Configuring Kafka Connect in Using AMQ Streams on OpenShift.
- Creating a new container image automatically using AMQ Streams in Deploying and Upgrading AMQ Streams on OpenShift.

2.2.1. Deploying Debezium with AMQ Streams

You follow the same steps to deploy each type of Debezium connector. The following section describes how to deploy a Debezium MySQL connector.

With earlier versions of AMQ Streams, to deploy Debezium connectors on OpenShift, you were required to first build a Kafka Connect image for the connector. The current preferred method for deploying connectors on OpenShift is to use a build configuration in AMQ Streams to automatically build a Kafka Connect container image that includes the Debezium connector plug-ins that you want to use.

During the build process, the AMQ Streams Operator transforms input parameters in a **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 or another configured HTTP server. The newly created container is pushed to the container registry that is specified in **.spec.build.output**, and is used to deploy a Kafka Connect pod. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that included in the build.

Prerequisites

- You have access to an OpenShift cluster on which the cluster Operator is installed.
- The AMQ Streams Operator is running.

- An Apache Kafka cluster is deployed as documented in Deploying and Upgrading AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat Integration license.
- 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 need registry permissions or you must create an ImageStream resource:

  To store the build image in an image registry, such as Red Hat Quay.io or Docker Hub

  - An account and permissions to create and manage images in the registry.

  To store the build image as a native OpenShift ImageStream

  - An ImageStream resource is deployed to the cluster. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default.

Procedure

1. Log in to the OpenShift cluster.
2. Create a new Debezium **KafkaConnect** custom resource (CR) for the connector. For example, create a **KafkaConnect** CR that specifies the **metadata.annotations** and **spec.build** properties, as shown in the following example. Save the file with a name such as **dbz-connect.yaml**.

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

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

```
connect-converter/2.0-redhat-<build-number>/apicurio-registry-distro-connect-converter-
2.0-redhat-<build-number>.zip
        - type: zip
          url: https://maven.repository.redhat.com/ga/io/debezium/debezium-
scripting/1.7.2.Final/debezium-scripting-1.7.2.Final.zip

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

Table 2.1. 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 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 each connector that you want to deploy.
For example, create the following **KafkaConnector** CR, and save it as **mysql-inventory-connector.yaml**

Example 2.2. A mysql-inventory-connector.yaml file that defines theKafkaConnector custom resource for a Debezium connector

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaConnector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
  name: inventory-connector-mysql (1)
spec:
  class: io.debezium.connector.mysql.MySqlConnector (2)
  tasksMax: 1 (3)
  config: (4)
    database.history.kafka.bootstrap.servers: 'debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092'
    database.history.kafka.topic: schema-changes.inventory
    database.hostname: mysql.debezium-mysql.svc.cluster.local (5)
    database.port: 3306 (6)
    database.user: debezium (7)
    database.password: dbz (8)
    database.dbname: mydatabase (9)
    database.server.name: inventory_connector_mysql (10)
    database.include.list: public.inventory (11)
```

Table 2.2. 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	The number of tasks that can operate concurrently.
4	The connector's configuration.
5	The address of the host 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 for the database user account.
9	The name of the database to capture changes from.
10	The logical name of the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the logical name is used as the prefix for any Kafka topics that receive change events from this connector, the name must be unique among the connectors in the cluster. The namespace is also used in the names of related Kafka Connect schemas, and the namespaces of a corresponding Avro schema if you integrate the connector with the Avro connector.
11	The list of tables from which the connector captures change events.

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

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

For example,

```
oc create -n debezium -f {context}-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 the Debezium deployment.

2.2.2. Verifying that the Debezium connector is running

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-mysql**.
     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-mysql -n debezium
```

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

Example 2.3. KafkaConnector resource status

```
Name:         inventory-connector-mysql
Namespace:    debezium
Labels:       strimzi.io/cluster=debezium-kafka-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-mysql
    Tasks:
      Id:               0
```

```
    State:           RUNNING
    worker_id:       10.131.1.124:8083
  Type:              source
 Observed Generation:  1
 Tasks Max:           1
 Topics:
  inventory_connector_mysql
  inventory_connector_mysql.inventory.addresses
  inventory_connector_mysql.inventory.customers
  inventory_connector_mysql.inventory.geom
  inventory_connector_mysql.inventory.orders
  inventory_connector_mysql.inventory.products
  inventory_connector_mysql.inventory.products_on_hand
Events:  <none>
```

2. Verify that the connector created Kafka topics:

   - From the OpenShift Container Platform web console.

     a. Navigate to **Home → Search**.

     b. On the **Search** page, click **Resources** to open the **Select Resource** box, and then type **KafkaTopic**.

     c. From the **KafkaTopics** list, click the name of the topic that you want to check, for example, **inventory-connector-mysql.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 2.4. KafkaTopic resource status

     ```
     NAME                                                                                                CLUSTER
     PARTITIONS   REPLICATION FACTOR   READY
     connect-cluster-configs                                                                              debezium-
     kafka-cluster   1            1                 True
     connect-cluster-offsets                                                                              debezium-
     kafka-cluster   25           1                 True
     connect-cluster-status                                                                               debezium-
     kafka-cluster   5            1                 True
     consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fadc1c6a
     debezium-kafka-cluster   50           1                 True
     inventory-connector-mysql---a96f69b23d6118ff415f772679da623fbbb99421
     debezium-kafka-cluster   1            1                 True
     inventory-connector-mysql.inventory.addresses---
     1b6beaf7b2eb57d177d92be90ca2b210c9a56480          debezium-kafka-cluster
     ```

```
1          1          True
inventory-connector-mysql.inventory.customers---
9931e04ec92ecc0924f4406af3fdace7545c483b          debezium-kafka-cluster   1
1          True
inventory-connector-mysql.inventory.geom---
9f7e136091f071bf49ca59bf99e86c713ee58dd5          debezium-kafka-cluster
1          1          True
inventory-connector-mysql.inventory.orders---
ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d          debezium-kafka-cluster
1          1          True
inventory-connector-mysql.inventory.products---
df0746db116844cee2297fab611c21b56f82dcef          debezium-kafka-cluster   1
1          True
inventory-connector-mysql.inventory.products-on-hand---
8649e0f17ffcc9212e266e31a7aeea4585e5c6b5   debezium-kafka-cluster   1
1          True
schema-changes.inventory
debezium-kafka-cluster   1          1          True
strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3ff177fc55
debezium-kafka-cluster   1          1          True
strimzi-topic-operator-kstreams-topic-store-changelog---
b75e702040b99be8a9263134de3507fc0cc4017b  debezium-kafka-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 debezium-kafka-cluster-kafka-0 -- /opt/kafka/bin/kafka-console-
consumer.sh \
>     --bootstrap-server localhost:9092 \
>     --from-beginning \
>     --property print.key=true \
>     --topic=inventory_connector_mysql.inventory.products_on_hand
```

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

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

   Example 2.5. Content of a Debezium change event

```
{"schema":{"type":"struct","fields":
[{"type":"int32","optional":false,"field":"product_id"}],"optional":false,"name":"inventory_conne
ctor_mysql.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":"inventory_connector
_mysql.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":"inventory_connector
_mysql.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.connecto
r.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":"transacti
on"}],"optional":false,"name":"inventory_connector_mysql.inventory.products_on_hand.Env
elope"},"payload":{"before":null,"after":{"product_id":101,"quantity":3},"source":
{"version":"1.7.2.Final-redhat-
00001","connector":"mysql","name":"inventory_connector_mysql","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,"t
ransaction":null}}

In the preceding example, the **payload** value shows that the connector snapshot generated a read (**"op" ="r"**) event from the table **inventory.products_on_hand**. The **"before"** state of the **product_id** record is **null**, indicating that no previous value exists for the record. The **"after"** state shows a **quantity** of **3** for the item with **product_id 101**.

You can run Debezium with multiple Kafka Connect service clusters and multiple Kafka clusters. The number of connectors that you can deploy to a Kafka Connect cluster depends on the volume and rate of database events.

Next steps

For more information about deploying specific connectors, see the following topics in the Debezium User Guide:

- Deploying the Db2 connector
- Deploying the MongoDB connector
- Deploying the MySQL connector

- Deploying the Oracle connector
- Deploying the PostgreSQL connector
- Deploying the SQL Server connector

APPENDIX A. USING YOUR SUBSCRIPTION

Debezium is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

Accessing your account

1. Go to access.redhat.com.
2. If you do not already have an account, create one.
3. Log in to your account.

Activating a subscription

1. Go to access.redhat.com.
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

Downloading zip and tar files

To access zip or tar files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at access.redhat.com/downloads.
2. Scroll down to **INTEGRATION AND AUTOMATION**.
3. Click **Red Hat Integration** to display the Red Hat Integration downloads page.
4. Click the **Download** link for your component.

Revised on 2022-05-23 09:57:50 UTC