Red Hat build of Debezium 2.3.4

Installing Debezium on OpenShift

For use with Red Hat build of Debezium 2.3.4 on OpenShift Container Platform
Red Hat build of Debezium 2.3.4 Installing Debezium on OpenShift

For use with Red Hat build of Debezium 2.3.4 on OpenShift Container Platform
Abstract

This guide describes how to install Red Hat build of Debezium on OpenShift Container Platform with AMQ Streams.
# Table of Contents

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

CHAPTER 1. DEBEZIUM OVERVIEW ........................................ 4

CHAPTER 2. INSTALLING DEBEZIUM CONNECTORS .......................... 5
  2.1. KAFKA TOPIC CREATION RECOMMENDATIONS 5
  2.2. DEBEZIUM DEPLOYMENT ON AMQ STREAMS 5
    2.2.1. Deploying Debezium with AMQ Streams 6
    2.2.2. Verifying that the Debezium connector is running 12

APPENDIX A. USING YOUR SUBSCRIPTION .................................. 17
  Accessing your account 17
  Activating a subscription 17
  Downloading zip and tar files 17
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 Apache Kafka Connect connectors for the following common databases:

- Db2
- MySQL
- MongoDB
- Oracle
- 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 schema history topics for the Debezium Db2, MySQL, Oracle, and SQL Server connectors

For each of the preceding connectors a database schema history topic is required. Whether you manually create the database schema history topic, use the Kafka broker to create the topic automatically, or use Kafka Connect to create the topic, ensure that the topic is configured with the following settings:

- 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 Apicurio 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.
- Building a new container image automatically in Deploying and Managing 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 cluster. After AMQ Streams builds the Kafka Connect image, you create **KafkaConnector** custom resources to start the connectors that are 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 Managing AMQ Streams on OpenShift.
- Kafka Connect is deployed on AMQ Streams
- You have a Red Hat build of Debezium 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 for storing new container images. You must explicitly create an ImageStream for the cluster. ImageStreams are not available by default. For more information about ImageStreams, see Managing image streams in the OpenShift Container Platform documentation.

Procedure

1. Log in to the OpenShift cluster.

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

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

In the example that follows, the custom resource is configured to download the following artifacts:

- The Debezium connector archive.
- The Red Hat build of Apicurio Registry archive. The Apicurio Registry is an optional component. Add the Apicurio Registry component only if you intend to use Avro serialization with the connector.
- The Debezium scripting SMT archive and the associated scripting engine that you want to use with the Debezium connector. The SMT archive and scripting language dependencies are optional components. Add these components only if you intend to use the Debezium content-based routing SMT or filter SMT.

```yaml
apiVersion: kafka.strimzi.io/v1beta2
```
Table 2.1. Descriptions of Kafka Connect configuration settings

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sets the <code>strimzi.io/use-connector-resources</code> annotation to &quot;true&quot; to enable the Cluster Operator to use KafkaConnect resources to configure connectors in this Kafka Connect cluster.</td>
</tr>
<tr>
<td>2</td>
<td>The <code>spec.build</code> configuration specifies where to store the build image and lists the plug-ins to include in the image, along with the location of the plug-in artifacts.</td>
</tr>
<tr>
<td>3</td>
<td>The <code>build.output</code> specifies the registry in which the newly built image is stored.</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>4</td>
<td>Specifies the name and image name for the image output. Valid values for <code>output.type</code> are <code>docker</code> to push into a container registry such as Docker Hub or Quay, or <code>imagestream</code> to push the image to an internal OpenShift ImageStream. To use an ImageStream, an ImageStream resource must be deployed to the cluster. For more information about specifying the <code>build.output</code> in the KafkaConnect configuration, see the AMQ Streams Build schema reference in Configuring AMQ Streams on OpenShift.</td>
</tr>
<tr>
<td>5</td>
<td>The <code>plugins</code> 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 <code>name</code>, and information for about the artifacts that are required to build the connector. Optionally, for each connector plug-in, you can include other components that you want to be available for use with the connector. For example, you can add Service Registry artifacts, or the Debezium scripting component.</td>
</tr>
<tr>
<td>6</td>
<td>The value of <code>artifacts.type</code> specifies the file type of the artifact specified in the <code>artifacts.url</code>. Valid types are <code>zip</code>, <code>tgz</code>, or <code>jar</code>. Debezium connector archives are provided in <code>.zip</code> file format. The <code>type</code> value must match the type of the file that is referenced in the <code>url</code> field.</td>
</tr>
<tr>
<td>7</td>
<td>The value of <code>artifacts.url</code> specifies the address of an HTTP server, such as a Maven repository, that stores the file for the connector artifact. Debezium connector artifacts are available in the Red Hat Maven repository. The OpenShift cluster must have access to the specified server.</td>
</tr>
<tr>
<td>8</td>
<td>(Optional) Specifies the artifact <code>type</code> and <code>url</code> for downloading the Apicurio Registry component. Include the Apicurio Registry artifact, only if you want the connector to use Apache Avro to serialize event keys and values with the Red Hat build of Apicurio Registry, instead of using the default JSON converter.</td>
</tr>
<tr>
<td>9</td>
<td>(Optional) Specifies the artifact <code>type</code> and <code>url</code> for the Debezium scripting SMT archive to use with the Debezium connector. Include the scripting SMT only if you intend to use the Debezium content-based routing SMT or filter SMT. To use the scripting SMT, you must also deploy a JSR 223-compliant scripting implementation, such as groovy.</td>
</tr>
<tr>
<td>Item</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>10</td>
<td>(Optional) Specifies the artifact type and url for the JAR files of a JSR 223-compliant scripting implementation, which is required by the Debezium scripting SMT.</td>
</tr>
</tbody>
</table>

**IMPORTANT**

If you use AMQ Streams to incorporate the connector plugin into your Kafka Connect image, for each of the required scripting language components artifacts.url must specify the location of a JAR file, and the value of artifacts.type must also be set to jar. Invalid values cause the connector fails at runtime.

To enable use of the Apache Groovy language with the scripting SMT, the custom resource in the example retrieves JAR files for the following libraries:

- groovy
- groovy-jsr223 (scripting agent)
- groovy-json (module for parsing JSON strings)

As an alternative, the Debezium scripting SMT also supports the use of the JSR 223 implementation of GraalVM JavaScript.

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

apiVersion: kafka.strimzi.io/v1beta2
customresourcetype: KafkaConnector
description: Debezium connector
metadata:
  labels:
    strimzi.io/cluster: debezium-kafka-connect-cluster
ame: inventory-connector-mysql
spec:
class: io.debezium.connector.mysql.MySqlConnector
tasksMax: 1
```
Table 2.2. Descriptions of connector configuration settings

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The name of the connector to register with the Kafka Connect cluster.</td>
</tr>
<tr>
<td>2</td>
<td>The name of the connector class.</td>
</tr>
<tr>
<td>3</td>
<td>The number of tasks that can operate concurrently.</td>
</tr>
<tr>
<td>4</td>
<td>The connector’s configuration.</td>
</tr>
<tr>
<td>5</td>
<td>The address of the host database instance.</td>
</tr>
<tr>
<td>6</td>
<td>The port number of the database instance.</td>
</tr>
<tr>
<td>7</td>
<td>The name of the account that Debezium uses to connect to the database.</td>
</tr>
<tr>
<td>8</td>
<td>The password that Debezium uses to connect to the database user account.</td>
</tr>
<tr>
<td>9</td>
<td>Unique numeric ID of the connector.</td>
</tr>
<tr>
<td>10</td>
<td>The topic prefix for the database instance or cluster. The specified name must be formed only from alphanumeric characters or underscores. Because the topic prefix 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. This 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.</td>
</tr>
<tr>
<td>11</td>
<td>The list of tables from which the connector captures change events.</td>
</tr>
</tbody>
</table>

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

```bash
cfg: config:
schema.history.internal.kafka.bootstrap.servers: debezium-kafka-cluster-kafka-bootstrap.debezium.svc.cluster.local:9092
schema.history.internal.kafka.topic: schema-changes.inventory
database.hostname: mysql.debezium-mysql.svc.cluster.local
database.port: 3306
database.user: debezium
database.password: dbz
database.server.id: 184054
topic.prefix: inventory-connector-mysql
table.include.list: inventory.*
```
For example,

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

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

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

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

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
    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**

<table>
<thead>
<tr>
<th>NAME</th>
<th>CLUSTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTITIONS</td>
<td>REPLICA FACTOR</td>
</tr>
<tr>
<td>connect-cluster-configs</td>
<td>1</td>
</tr>
<tr>
<td>connect-cluster-offsets</td>
<td>25</td>
</tr>
<tr>
<td>connect-cluster-status</td>
<td>5</td>
</tr>
<tr>
<td>consumer-offsets---84e7a678d08f4bd226872e5cdd4eb527fad1c6a</td>
<td>1</td>
</tr>
<tr>
<td>inventory-connector-mysql.inventory.addresses---1b6beaf7b2eb57d177d922be90ca2b210c9a56480</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>inventory-connector-mysql.inventory.customers---9931e04ec92ecc0924f4406af3fdace7545c483b</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>inventory-connector-mysql.inventory.geom---9f7e136091f071bf49ca59bf99e86c713ee58dd5</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>inventory-connector-mysql.inventory.orders---ac5e98ac6a5d91e04d8ec0dc9078a1ece439081d</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>inventory-connector-mysql.inventory.products---df0746db116844ece22977ab611c21b56f82dce</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>inventory-connector-mysql.inventory.products_on_hand---8649e0f17ffcc9212e266e31a7aeaa4585e5c6b5</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>schema-changes.inventory</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>strimzi-store-topic---effb8e3e057afce1ecf67c3f5d8e4e3f177fc55</td>
<td>debezium-kafka-cluster</td>
</tr>
<tr>
<td>strimzi-topic-operator-kstreams-topic-store-changelog---b75e702040b99be8a9263134de3507fc0cc4017b</td>
<td>debezium-kafka-cluster</td>
</tr>
</tbody>
</table>
3. Check topic content.

- From a terminal window, enter the following command:

```bash
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,

```bash
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**

```json
{"schema":{"type":"struct","fields":
["type":"int32","optional":false,"field":"product_id"],"optional":false,"name":"inventory-connector-mysql.inventory.products_on_hand.Key","payload":{"product_id":101}}
{"schema":{"type":"struct","fields":
["type":"int32","optional":false,"field":"product_id"],"optional":false,"name":"inventory-connector-mysql.inventory.products_on_hand.Value","field":"before"},
{"type":"struct","fields":
["type":"int32","optional":false,"field":"product_id"],"optional":false,"field":"quantity"],"optional":false,"name":"inventory-connector-mysql.inventory.products_on_hand.Value","field":"after"},
{"type":"string","optional":false,"field":"version"],
{"type":"string","optional":false,"field":"connector"],
{"type":"string","optional":false,"field":"name"],
{"type":"int64","optional":false,"field":"ts_ms"},
{"type":"string","optional":true,"name":"io.debezium.data.Enum","version":1,"parameters":
["allowed":"true,last,false"],"default":"false","field":"snapshot"],
{"type":"string","optional":false,"field":"db"],
{"type":"string","optional":true,"field":"sequence"],
{"type":"string","optional":true,"field":"table"],
{"type":"int64","optional":false,"field":"server_id"],
{"type":"string","optional":true,"field":"gtid"],
{"type":"string","optional":false,"field":"file"],
{"type":"int64","optional":false,"field":"pos"],
{"type":"int32","optional":false,"field":"row"],
{"type":"int64","optional":true,"field":"thread"],
{"type":"string","optional":true,"field":"query"}],"optional":false,"name":"io.debezium.connector.mysql.Source","field":"source"],
{"type":"string","optional":false,"field":"op"],
{"type":"int64","optional":true,"field":"ts_ms"},
{"type":"string","optional":false,"field":"id"],
{"type":"int64","optional":false,"field":"total_order"],
```
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
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](http://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](http://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](http://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 2023-10-11 18:33:25 UTC*