Cloudera Runtime 7.3.1

Apache Spark integration with Schema Registry

Date published: 2020-07-28
Date modified: 2024-12-10

https://docs.cloudera.com/

Legal Notice

Contents

Apache Spark 3 integration with Schema Registry

Apache Spark 3 integrated with Schema Registry provides a library to leverage Schema Registry for managing Spark schemas and to serialize and/or de-serialize messages in Spark data sources and sinks.

Note: The Apache Spark 3 integration with Schema Registry is available in CDS 3.3 Powered by Apache Spark from version 3.3.2 for CDP Private Cloud Base 7.1.9 SP1.

Running the example programs

The examples illustrate the API usage and how to integrate with Schema Registry. The examples can be run from IDE (for example, Intellij) by specifying a master URL or by using spark3-submit.

```
spark3-submit --jars /opt/cloudera/parcels/CDH/lib/spark-schema-registry-for
-spark3/spark-schema-registry-for-spark3_2.12-jar-with-dependencies.jar \
--class com.hortonworks.spark.registry.examples.classname \
/opt/cloudera/parcels/CDH/lib/spark-schema-registry-for-spark3/examples/sp
ark-schema-registry-for-spark3-examples_2.12.jar [***SCHEMA-REGISTRY-URL***]
 \
bootstrap-servers input-topic output-topic checkpoint-location
```

Using the APIs

Typically in a Spark application you define the Spark schema for the data you are going to process:

```
// the schema for truck events
val schema = StructType(Seq(
  StructField("driverId", IntegerType, nullable = false),
  StructField("truckId", IntegerType, nullable = false),
  StructField("miles", LongType, nullable = false),
  StructField("eventType", StringType, nullable = false),
  ...
  ...
)

// read Json string messages from the data source
val messages = spark
      .readStream
      .format(...)
      .option(...)
      .load()

// parse the messages using the above schema and do further operations
val df = messages
      .select(from_json($"value".cast("string"), schema).alias("value"))
      ...
// project (driverId, truckId, miles) for the events where miles > 300
val filtered = df.select($"value.driverId", $"value.truckId", $"value.miles"
)
      .where("value.miles > 300")
```

However, this approach is not practical because the schema information is tightly coupled with the code. The code needs to be changed when the schema changes, and there is no ability to share or reuse the schema between the message producers and the applications that consume the messages.

Using Schema Registry is a better solution because it enables you to manage different versions of the schema and define compatibility policies.

Configuration

The Schema Registry integration comes as a utility method which can be imported into the scope.

```
import com.hortonworks.spark.registry.util._
```

Before invoking the APIs, you need to define an implicit SchemaRegistryConfig which will be passed to the APIs. The main configuration parameter is the schema registry URL.

```
// the schema registry client config
val config = Map[String, Object]("[***SCHEMA.REGISTRY.URL***]" -> schemaR
egistryUrl)
// the schema registry config that will be implicitly passed
implicit val srConfig:SchemaRegistryConfig = SchemaRegistryConfig(config)
```

SSL configuration

SchemaRegistryConfig expects the following SSL configuration properties:

```
"schema.registry.client.ssl.protocol" -> "SSL",
"schema.registry.client.ssl.trustStoreType" -> "JKS",
"schema.registry.client.ssl.trustStorePath" -> "/var/lib/cloudera-scm-agent/
agent-cert/cm-auto-global_truststore.jks",
"schema.registry.client.ssl.trustStorePassword" ->
 "[***CHANGEMECLIENTPWD***]"
```

Fetching Spark schema by name

The API supports fetching the Schema Registry schema as a Spark schema.

- sparkSchema(schemaName: String)

  Returns the spark schema corresponding to the latest version of schema defined in the Schema Registry.
- sparkSchema(schemaName: String, version: Int)

  Returns the spark schema corresponding to the given version of schema defined in the Schema Registry.

Using the Schema Registry integration, the example previously shown can be simplified, as there is no need to explicitly specify the Spark schema in the code:

```
// retrieve the translated "Spark schema" by specifying the schema registry
schema name
val schema = sparkSchema(name)

// parse the messages using the above schema and do further operations
val df = messages
         .select(from_json($"value".cast("string"), schema).alias("value"))
         ...

// project (driverId, truckId, miles) for the events where miles > 300
val filtered = df.select($"value.driverId", $"value.truckId", $"value.miles
")
```

```
                    .where("value.miles > 300")
```

Serializing messages using Schema Registry

The following method can be used to serialize the messages from Spark to Schema Registry binary format using schema registry serializers.

- to_sr(data: Column, schemaName: String, topLevelRecordName: String, namespace: String)

  Converts a Spark column data to binary format of Schema Registry. This looks up a Schema Registry schema for the schemaName that matches the input and automatically registers a new schema, if not found. The topoLevelRecordName and namespace are optional and will be mapped to Avro top level record name and record namespace.

De-serializing messages using Schema Registry

The following methods can be used to de-serialize Schema Registry serialized messages into Spark columns.

- from_sr(data: Column, schemaName: String)

  Converts Schema Registry binary format to Spark column, using the latest version of the schema.
- from_sr(data: Column, schemaName: String, version: Int)

  Converts Schema Registry binary format to Spark column using the given Schema Registry schema name and version.

Serialization - deserialization example

The following is an example that uses the from_sr to de-serialize Schema Registry formatted messages into Spark, transforms and serializes it back to Schema Registry format using to_sr, and writes to a data sink.

This example assumes Spark Structured Streaming use cases, but it should work well for the non-streaming use cases as well (read and write).

```
// Read schema registry formatted messages and deserialize to spark columns.
val df = messages
      .select(from_sr($"value", topic).alias("message"))
// project (driverId, truckId, miles) for the events where miles > 300
val filtered = df.select($"message.driverId", $"message.truckId", $"message.
miles")
      .where("message.miles > 300")
// write the output as schema registry serialized bytes to a sink
// should produce events like {"driverId":14,"truckId":25,"miles":373}
val query = filtered
      .select(to_sr(struct($"*"), outSchemaName).alias("value"))
      .writeStream
      .format(..)
      .start()
```

The output schema outSchemaName is automatically published to the Schema Registry if it does not exist.

Building and deploying your app

Add a Maven dependency in your project to make use of the library and build your application JAR file:

```
<dependency>
    <groupId>com.hortonworks</groupId>
    <artifactId>spark-schema-registry-for-spark3_2.12</artifactId>
    <version>version</version>
```

```
 </dependency>
```

Once the application JAR file is built, deploy it by adding the dependency in spark3-submit using --packages:

```
spark3-submit --packages com.hortonworks:spark-schema-registry-for-spark3_2.
12:version \
--conf spark.jars.repositories=[***HTTPS://REPOSITORY.EXAMPLE.COM***] \
--class YourApp \
your-application-jar \
args ...
```

Make sure the package is published in a local or online available repository.

If the package is not published to an available repository, or your Spark application cannot access external networks, you can use an uber JAR file instead:

```
spark3-submit --master [***MASTER URL****] \
--jars /opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-registry-for-s
park3/spark-schema-registry-for-spark3_2.12-jar-with-dependencies.jar \
--class YourApp \
your-application-jar \
args ...
```

Running in a Kerberos-enabled cluster

The library works in a Kerberos setup, where Spark and Schema Registry has been deployed on a Kerberos-enabled cluster.

To configure, set up the appropriate JAAS config for RegistryClient (and KafkaClient, if the Spark data source or sink is Kafka).

As an example, to run the SchemaRegistryAvroExample in a Kerberos setup, follow these steps:

1. Create a keytab (for example, app.keytab) with the login user and principal you want to run the application.
2. Create an app_jaas.conf file and specify the keytab and principal created in Step 1.

   If deploying to YARN, the keytab and conf files will be distributed as YARN local resources. They will be placed in the current directory of the Spark YARN container, and the location needs to be specified as ./app.keytab.

   ```
   RegistryClient {
       com.sun.security.auth.module.Krb5LoginModule required
       useKeyTab=true
       keyTab="./app.keytab"
       storeKey=true
       useTicketCache=false
       principal="[***PRINCIPAL***]";
   };

   KafkaClient {
      com.sun.security.auth.module.Krb5LoginModule required
      useKeyTab=true
      keyTab="./app.keytab"
      storeKey=true
      useTicketCache=false
      serviceName="kafka"
      principal="[***PRINCIPAL***]";
   };
   ```

3. Provide the required ACLs for the kafka topics (in-topic, out-topic) for the principal.
4. Use spark3-submit to pass the JAAS configuration file with extraJavaOptions. (And also as local resource files in YARN cluster mode.)

```
spark3-submit --master yarn --deploy-mode cluster \
    --keytab app.keytab --principal [***PRINCIPAL***] \
    --files app_jaas.conf#app_jaas.conf,app.keytab#app.keytab \
    --jars /opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-registry-f
or-spark3/spark-schema-registry-for-spark3_2.12-jar-with-dependencies.jar
 \
    --conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.c
onfig=./app_jaas.conf" \
    --conf "spark.driver.extraJavaOptions=-Djava.security.auth.login.co
nfig=./app_jaas.conf" \
    --class com.hortonworks.spark.registry.examples.SchemaRegistryAvroEx
ample \
/opt/cloudera/parcels/SPARK3/lib/spark3/spark-schema-registry-for-spark3/
examples/spark-schema-registry-for-spark3-examples_2.12.jar \
    [***SCHEMA-REGISTRY-URL***] bootstrap-server in-topic out-topic che
ckpoint-dir SASL_PLAINTEXT
```

Unsupported features

Apache Spark 3 integration with Schema Registry is not supported in pyspark.