SQL Component

SQL Component

The **sql:** component allows you to work with databases using JDBC queries. The difference between this component and JDBC component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

This component uses **`spring-jdbc`** behind the scenes for the actual SQL handling.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sql</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

The SQL component also supports:

- a JDBC based repository for the Idempotent Consumer EIP pattern. See further below.
- a JDBC based repository for the Aggregator EIP pattern. See further below.

URI format

From Camel 2.11 onwards this component can create both consumer (e.g. `from()`) and producer endpoints (e.g. `to()`).

In previous versions, it could only act as a producer.

This component can be used as a Transactional Client.

The SQL component uses the following endpoint URI notation:

```
sql:select * from table where id=# order by name[?options]
```

From Camel 2.11 onwards you can use named parameters by using :`#name_of_the_parameter` style as shown:

```
 sql:select * from table where id=:#myId order by name[?options]
```

When using named parameters, Camel will lookup the names from, in the given precedence:
1. from message body if its a `java.util.Map`
2. from message headers

If a named parameter cannot be resolved, then an exception is thrown.

From **Camel 2.14** onward you can use Simple expressions as parameters as shown:

```
sql:select * from table where id=:#${property.myId} order by name[?options]
```

Notice that the standard ? symbol that denotes the parameters to an SQL query is substituted with the # symbol, because the ? symbol is used to specify options for the endpoint. The ? symbol replacement can be configured on endpoint basis.

From **Camel 2.17** onwards you can externalize your SQL queries to files in the classpath or file system as shown:

```
sql:classpath:sql/myquery.sql[?options]
```

And the myquery.sql file is in the classpath and is just a plain text

```
select * from table where id = :#${property.myId} order by name
```

In the file you can use multilines and format the SQL as you wish. And also use comments such as the – dash line.

You can append query options to the URI in the following format, `?option=value&option=value&...`

Options

Option	Type	Default	Description
`batch`	`boolean`	`false`	**Camel 2.7.5, 2.8.4 and 2.9:** Execute SQL batch update statements. See notes below on how the treatment of the inbound message body changes if this is set to `true`.
`dataSourceRef`	`String`	`null`	**Deprecated and will be removed in Camel 3.0:** Reference to a `DataSource` to look up in the registry. Use `dataSource=#theName` instead.
`dataSource`	`String`	`null`	**Camel 2.11:** Reference to a `DataSource` to look up in the registry.
`placeholder`	`String`	`#`	**Camel 2.4:** Specifies a character that will be replaced to ? in SQL query. Notice, that it is simple `String.replaceAll()` operation and no SQL parsing is involved (quoted strings will also change). This replacement is **only** happening if the endpoint is created using the `SqlComponent`. If you manually create the endpoint, then use the expected ? sign instead.
`usePlaceholder`	`boolean`	`true`	**Camel 2.17:** Sets whether to use placeholder and replace all placeholder characters with ? sign in the SQL queries.
`template.<xxx>`		`null`	Sets additional options on the Spring `JdbcTemplate` that is used behind the scenes to execute the queries. For instance, `template.maxRows=10`. For detailed documentation, see the JdbcTemplate javadoc documentation.
`allowNamedParameters`	`boolean`	`true`	**Camel 2.11:** Whether to allow using named parameters in the queries.
`processingStrategy`			**Camel 2.11: SQL consumer only:** Allows to plugin to use a custom `org.apache.camel.component.sql.SqlProcessingStrategy` to execute queries when the consumer has processed the rows/batch.
`prepareStatementStrategy`			**Camel 2.11:** Allows to plugin to use a custom `org.apache.camel.component.sql.SqlPrepareStatementStrategy` to control preparation of the query and prepared statement.

consumer.delay	long	500	**Camel 2.11: SQL consumer only:** Delay in milliseconds between each poll.
consumer.initialDelay	long	1000	**Camel 2.11: SQL consumer only:** Milliseconds before polling starts.
consumer.useFixedDelay	boolean	false	**Camel 2.11: SQL consumer only:** Set to true to use fixed delay between polls, otherwise fixed rate is used. See ScheduledExecutorService in JDK for details.
maxMessagesPerPoll	int	0	**Camel 2.11: SQL consumer only:** An integer value to define the maximum number of messages to gather per poll. By default, no maximum is set.
useIterator	boolean	true	**Camel 2.11: SQL consumer only:** If true each row returned when polling will be processed individually. If false the entire java.util.List of data is set as the IN body. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
routeEmptyResultSet	boolean	false	**Camel 2.11: SQL consumer only:** Whether to route a single empty Exchange if there was no data to poll. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
onConsume	String	null	**Camel 2.11: SQL consumer only:** After processing each row then this query can be executed, if the Exchange was processed successfully, for example to mark the row as processed. The query can have parameter. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
onConsumeFailed	String	null	**Camel 2.11: SQL consumer only:** After processing each row then this query can be executed, if the Exchange failed, for example to mark the row as failed. The query can have parameter. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
onConsumeBatchComplete	String	null	**Camel 2.11: SQL consumer only:** After processing the entire batch, this query can be executed to bulk update rows etc. The query cannot have parameters. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
expectedUpdateCount	int	-1	**Camel 2.11: SQL consumer only:** If using consumer.onConsume then this option can be used to set an expected number of rows being updated. Typically you may set this to 1 to expect one row to be updated. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.

breakBatchOnConsumeFail	boolean	false	**Camel 2.11: SQL consumer only:** If using consumer.onConsume and it fails, then this option controls whether to break out of the batch or continue processing the next row from the batch. Notice in Camel 2.15.x or older you need to prefix this option with consumer., eg consumer.useIterator=true.
alwaysPopulateStatement	boolean	false	**Camel 2.11: SQL producer only:** If enabled then the populateStatement method from org.apache.camel.component.sql.SqlPrepareStatementStrategy is always invoked, also if there is no expected parameters to be prepared. When this is false then the populateStatement is only invoked if there is 1 or more expected parameters to be set; for example this avoids reading the message body/headers for SQL queries with no parameters.
separator	char	,	**Camel 2.11.1:** The separator to use when parameter values is taken from message body (if the body is a String type), to be inserted at # placeholders. Notice if you use named parameters, then a Map type is used instead.
outputType	String	SelectList	**Camel 2.12.0:** outputType='SelectList', for consumer or producer, will output a List of Map. SelectOne will output single Java object in the following way: a) If the query has only single column, then that JDBC Column object is returned. (such as SELECT COUNT(*) FROM PROJECT will return a Long object. b) If the query has more than one column, then it will return a Map of that result. c) If the outputClass is set, then it will convert the query result into an Java bean object by calling all the setters that match the column names. It will assume your class has a default constructor to create instance with. d) If the query resulted in more than one rows, it throws an non-unique result exception. From **Camel 2.14.1** onwards the SelectList also supports mapping each row to a Java object as the SelectOne does (only step c). From **Camel 2.18** onwards there is a new StreamList outputType that streams the result of the query using an Iterator. It can be used with the Splitter EIP in streaming mode to process the ResultSet in streaming fashion. This StreamList do not support batch mode, but you can use outputClass to map each row to a class.
outputClass	String	null	**Camel 2.12.0:** Specify the full package and class name to use as conversion when outputType=SelectOne.

outputHeader	String	null	**Camel 2.15:** To store the result as a header instead of the message body. This allows to preserve the existing message body as-is.
parametersCount	int	0	**Camel 2.11.2/2.12.0** If set greater than zero, then Camel will use this count value of parameters to replace instead of querying via JDBC metadata API. This is useful if the JDBC vendor could not return correct parameters count, then user may override instead.
noop	boolean	false	**Camel 2.12.0** If set, will ignore the results of the SQL query and use the existing IN message as the OUT message for the continuation of processing
useMessageBodyForSql	boolean	false	**Camel 2.16:** Whether to use the message body as the SQL and then headers for parameters. If this option is enabled then the SQL in the uri is not used. The SQL parameters must then be provided in a header with the key `CamelSqlParameters`. This option is only for the producer.
transacted	boolean	false	**Camel 2.16.2: SQL consumer only:** Enables or disables transaction. If enabled then if processing an exchange failed then the consumer break out processing any further exchanges to cause a rollback eager

Treatment of the message body

The SQL component tries to convert the message body to an object of `java.util.Iterator` type and then uses this iterator to fill the query parameters (where each query parameter is represented by a # symbol (or configured placeholder) in the endpoint URI). If the message body is not an array or collection, the conversion results in an iterator that iterates over only one object, which is the body itself.

For example, if the message body is an instance of `java.util.List`, the first item in the list is substituted into the first occurrence of # in the SQL query, the second item in the list is substituted into the second occurrence of #, and so on.

If `batch` is set to `true`, then the interpretation of the inbound message body changes slightly – instead of an iterator of parameters, the component expects an iterator that contains the parameter iterators; the size of the outer iterator determines the batch size.

From Camel 2.16 onwards you can use the option useMessageBodyForSql that allows to use the message body as the SQL statement, and then the SQL parameters must be provided in a header with the key SqlConstants.SQL_PARAMETERS. This allows the SQL component to work more dynamic as the SQL query is from the message body.

Result of the query

For `select` operations, the result is an instance of `List<Map<String, Object>>` type, as returned by the JdbcTemplate.queryForList() meth od. For `update` operations, the result is the number of updated rows, returned as an `Integer`.

By default, the result is placed in the message body. If the outputHeader parameter is set, the result is placed in the header. This is an alternative to using a full message enrichment pattern to add headers, it provides a concise syntax for querying a sequence or some other small value into a header. It is convenient to use outputHeader and outputType together:

```
from("jms:order.inbox")
    .to("sql:select order_seq.nextval from dual?
outputHeader=OrderId&outputType=SelectOne")
    .to("jms:order.booking");
```

Using StreamList

From **Camel 2.18** onwards the producer supports outputType=StreamList that uses an iterator to stream the output of the query. This allows to process the data in a streaming fashion which for example can be used by the Splitter EIP to process each row one at a time, and load data from the database as needed.

```
from("direct:withSplitModel")
   .to("sql:select * from projects order by id?
outputType=StreamList&outputClass=org.apache.camel.component.sql.
ProjectModel")
   .to("log:stream")
   .split(body()).streaming()
       .to("log:row")
       .to("mock:result")
   .end();
```

Header values

When performing `update` operations, the SQL Component stores the update count in the following message headers:

Header	Description
`CamelSqlUpdateCount`	The number of rows updated for `update` operations, returned as an `Integer` object. This header is not provided when using outputType=StreamList.
`CamelSqlRowCount`	The number of rows returned for `select` operations, returned as an `Integer` object. This header is not provided when using outputType=StreamList.
`CamelSqlQuery`	**Camel 2.8:** Query to execute. This query takes precedence over the query specified in the endpoint URI. Note that query parameters in the header *are* represented by a ? instead of a # symbol

When performing `insert` operations, the SQL Component stores the rows with the generated keys and number of these rown in the following message headers (**Available as of Camel 2.12.4, 2.13.1**):

Header	Description
`CamelSqlGeneratedKeysRowCount`	The number of rows in the header that contains generated keys.
`CamelSqlGeneratedKeyRows`	Rows that contains the generated keys (a list of maps of keys).

Generated keys

Available as of Camel 2.12.4, 2.13.1 and 2.14

If you insert data using SQL INSERT, then the RDBMS may support auto generated keys. You can instruct the SQL producer to return the generated keys in headers.
To do that set the header `CamelSqlRetrieveGeneratedKeys=true`. Then the generated keys will be provided as headers with the keys listed in the table above.

You can see more details in this unit test.

Configuration

You can now set a reference to a `DataSource` in the URI directly:

```
  select * from table where id=# order by name?dataSource=myDS
```

Sample

In the sample below we execute a query and retrieve the result as a `List` of rows, where each row is a `Map<String, Object>` and the key is the column name.

First, we set up a table to use for our sample. As this is based on an unit test, we do it in java:

```
db = new EmbeddedDatabaseBuilder()
    .setType(EmbeddedDatabaseType.DERBY).addScript("sql
/createAndPopulateDatabase.sql").build();
```

The SQL script `createAndPopulateDatabase.sql` we execute looks like as described below:

```
create table projects (id integer primary key, project varchar(10),
license varchar(5));
insert into projects values (1, 'Camel', 'ASF');
insert into projects values (2, 'AMQ', 'ASF');
insert into projects values (3, 'Linux', 'XXX');
```

Then we configure our route and our `sql` component. Notice that we use a `direct` endpoint in front of the `sql` endpoint. This allows us to send an exchange to the `direct` endpoint with the URI, `direct:simple`, which is much easier for the client to use than the long `sql:` URI. Note that the `DataSource` is looked up up in the registry, so we can use standard Spring XML to configure our `DataSource`.

```
from("direct:simple")
    .to("sql:select * from projects where license = # order by id?
dataSource=#jdbc/myDataSource")
    .to("mock:result");
```

And then we fire the message into the `direct` endpoint that will route it to our `sql` component that queries the database.

```
MockEndpoint mock = getMockEndpoint("mock:result");
mock.expectedMessageCount(1);

// send the query to direct that will route it to the sql where we will
execute the query
// and bind the parameters with the data from the body. The body only
contains one value
// in this case (XXX) but if we should use multi values then the body will
be iterated
// so we could supply a List<String> instead containing each binding value.
template.sendBody("direct:simple", "XXX");

mock.assertIsSatisfied();

// the result is a List
List<?> received = assertIsInstanceOf(List.class, mock.
getReceivedExchanges().get(0).getIn().getBody());

// and each row in the list is a Map
Map<?, ?> row = assertIsInstanceOf(Map.class, received.get(0));

// and we should be able the get the project from the map that should be
Linux
assertEquals("Linux", row.get("PROJECT"));
```

We could configure the `DataSource` in Spring XML as follows:

```
<jee:jndi-lookup id="myDS" jndi-name="jdbc/myDataSource"/>
```

Using named parameters

Available as of Camel 2.11

In the given route below, we want to get all the projects from the projects table. Notice the SQL query has 2 named parameters, :#lic and :#min. Camel will then lookup for these parameters from the message body or message headers. Notice in the example above we set two headers with constant value
for the named parameters:

```
from("direct:projects")
   .setHeader("lic", constant("ASF"))
   .setHeader("min", constant(123))
   .to("sql:select * from projects where license = :#lic and id > :#min
order by id")
```

Though if the message body is a `java.util.Map` then the named parameters will be taken from the body.

```
from("direct:projects")
   .to("sql:select * from projects where license = :#lic and id > :#min
order by id")
```

Using expression parameters

Available as of Camel 2.14

In the given route below, we want to get all the project from the database. It uses the body of the exchange for defining the license and uses the value of a property as the second parameter.

```
from("direct:projects")
   .setBody(constant("ASF"))
   .setProperty("min", constant(123))
   .to("sql:select * from projects where license = :#${body} and id > :
#${property.min} order by id")
```

Using IN queries with dynamic values

Available as of Camel 2.17

From Camel 2.17 onwards the SQL producer allows to use SQL queries with IN statements where the IN values is dynamic computed. For example from the message body or a header etc.

To use IN you need to:

- prefix the parameter name with `in:`
- add ( ) around the parameter

An example explains this better. The following query is used:

```
select * from projects where project in (:#in:names) order by id
```

In the following route:

```
from("direct:query")
  .to("sql:classpath:sql/selectProjectsIn.sql")
  .to("log:query")
  .to("mock:query");
```

Then the IN query can use a header with the key names with the dynamic values such as:

```
// use an array
template.requestBodyAndHeader("direct:query", "Hi there!", "names", new
String[]{"Camel", "AMQ"});

// use a list
List<String> names = new ArrayList<String>();
names.add("Camel");
names.add("AMQ");
template.requestBodyAndHeader("direct:query", "Hi there!", "names", names);

// use a string separated values with comma
template.requestBodyAndHeader("direct:query", "Hi there!", "names", "Camel,
AMQ");
```

The query can also be specified in the endpoint instead of being externalized (notice that externalizing makes maintaining the SQL queries easier)

```
from("direct:query")
  .to("sql:select * from projects where project in (:#in:names) order by
id")
  .to("log:query")
  .to("mock:query");
```

Using the JDBC based idempotent repository

Available as of Camel 2.7: In this section we will use the JDBC based idempotent repository.

Abstract class From Camel 2.9 onwards there is an abstract class `org.apache.camel.processor.idempotent.jdbc.AbstractJdbcMessageIdRepository` you can extend to build custom JDBC idempotent repository.

First we have to create the database table which will be used by the idempotent repository. For **Camel 2.7**, we use the following schema:

```
 sqlCREATE TABLE CAMEL_MESSAGEPROCESSED (
   processorName VARCHAR(255),
   messageId VARCHAR(100) )
```

In **Camel 2.8**, we added the createdAt column:

```
 CREATE TABLE CAMEL_MESSAGEPROCESSED (
   processorName VARCHAR(255),
   messageId VARCHAR(100),
   createdAt TIMESTAMP )
```

The SQL Server **TIMESTAMP** type is a fixed-length binary-string type. It does not map to any of the JDBC time types: **DATE**, **TIME**, or **TIMESTAMP**.

We recommend to have a unique constraint on the columns processorName and messageId. Because the syntax for this constraint differs for database to database, we do not show it here.

Second we need to setup a `javax.sql.DataSource` in the spring XML file:

```
<jdbc:embedded-database id="dataSource" type="DERBY" />
```

And finally we can create our JDBC idempotent repository in the spring XML file as well:

```
 <bean id="messageIdRepository" class="org.apache.camel.processor.
idempotent.jdbc.JdbcMessageIdRepository">
   <constructor-arg ref="dataSource" />
   <constructor-arg value="myProcessorName" />
 </bean>
```

Customize the JdbcMessageIdRepository

Starting with **Camel 2.9.1** you have a few options to tune the `org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository` for your needs:

Parameter	Default Value	Description
createTableIfNotExists	true	Defines whether or not Camel should try to create the table if it doesn't exist.
tableExistsString	SELECT 1 FROM CAMEL_MESSAGEPROCESSED WHERE 1 = 0	This query is used to figure out whether the table already exists or not. It must throw an exception to indicate the table doesn't exist.
createString	CREATE TABLE CAMEL_MESSAGEPROCESSED (processorName VARCHAR(255), messageId VARCHAR(100), createdAt TIMESTAMP)	The statement which is used to create the table.
queryString	SELECT COUNT(*) FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ? AND messageId = ?	The query which is used to figure out whether the message already exists in the repository (the result is not equals to '0'). It takes two parameters. This first one is the processor name (`String`) and the second one is the message id (`String`).
insertString	INSERT INTO CAMEL_MESSAGEPROCESSED (processorName, messageId, createdAt) VALUES (?, ?, ?)	The statement which is used to add the entry into the table. It takes three parameter. The first one is the processor name (`String`), the second one is the message id (`String`) and the third one is the timestamp (`java.sql.Timestamp`) when this entry was added to the repository.
deleteString	DELETE FROM CAMEL_MESSAGEPROCESSED WHERE processorName = ? AND messageId = ?	The statement which is used to delete the entry from the database. It takes two parameter. This first one is the processor name (`String`) and the second one is the message id (`String`).

A customized `org.apache.camel.processor.idempotent.jdbc.JdbcMessageIdRepository` could look like:

```
```

```
<bean id="messageIdRepository" class="org.apache.camel.processor.
idempotent.jdbc.JdbcMessageIdRepository">
   <constructor-arg ref="dataSource" />
   <constructor-arg value="myProcessorName" />
   <property name="tableExistsString" value="SELECT 1 FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE 1 = 0" />
   <property name="createString" value="CREATE TABLE
CUSTOMIZED_MESSAGE_REPOSITORY (processorName VARCHAR(255), messageId
VARCHAR(100), createdAt TIMESTAMP)" />
   <property name="queryString" value="SELECT COUNT(*) FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?" />
   <property name="insertString" value="INSERT INTO
CUSTOMIZED_MESSAGE_REPOSITORY (processorName, messageId, createdAt) VALUES
(?, ?, ?)" />
   <property name="deleteString" value="DELETE FROM
CUSTOMIZED_MESSAGE_REPOSITORY WHERE processorName = ? AND messageId = ?" />
</bean>
```

Using the JDBC based aggregation repository

Available as of Camel 2.6

Using JdbcAggregationRepository in Camel 2.6 In Camel 2.6, the JdbcAggregationRepository is provided in the `camel-jdbc-aggregator` com ponent. From Camel 2.7 onwards, the `JdbcAggregationRepository` is provided in the `camel-sql` component.

`JdbcAggregationRepository` is an `AggregationRepository` which on the fly persists the aggregated messages. This ensures that you will not loose messages, as the default aggregator will use an in memory only `AggregationRepository`.

The `JdbcAggregationRepository` allows together with Camel to provide persistent support for the Aggregator.

It has the following options:

Option	Type	Description
dataSource	DataSource	**Mandatory:** The `javax.sql.DataSource` to use for accessing the database.
repositoryName	String	**Mandatory:** The name of the repository.
transactionManager	TransactionManager	**Mandatory:** The `org.springframework.transaction.PlatformTransactionManager` to mange transactions for the database. The TransactionManager must be able to support databases.
lobHandler	LobHandler	A `org.springframework.jdbc.support.lob.LobHandler` to handle Lob types in the database. Use this option to use a vendor specific LobHandler, for example when using Oracle.
returnOldExchange	boolean	Whether the get operation should return the old existing Exchange if any existed. By default this option is `false` to optimize as we do not need the old exchange when aggregating.
useRecovery	boolean	Whether or not recovery is enabled. This option is by default `true`. When enabled the Camel Aggregator automatic recover failed aggregated exchange and have them resubmitted.

recoveryInterval	long	If recovery is enabled then a background task is run every x'th time to scan for failed exchanges to recover and resubmit. By default this interval is 5000 millis.
maximumRedeliveries	int	Allows you to limit the maximum number of redelivery attempts for a recovered exchange. If enabled then the Exchange will be moved to the dead letter channel if all redelivery attempts failed. By default this option is disabled. If this option is used then the `deadLetterUri` option must also be provided.
`deadLetterUri`	String	An endpoint uri for a Dead Letter Channel w here exhausted recovered Exchanges will be moved. If this option is used then the `max imumRedeliveries` option must also be provided.
storeBodyAsText	boolean	**Camel 2.11:** Whether to store the message body as String which is human readable. By default this option is `false` storing the body in binary format.
headersToStoreAsText	List<String>	**Camel 2.11:** Allows to store headers as String which is human readable. By default this option is disabled, storing the headers in binary format.
jdbcOptimisticLockingExceptionMapper	jdbcOptimisticLockingExceptionMapper	**Camel 2.12:** Allows to plugin a custom `org. apache.camel.processor.aggregate. jdbc. JdbcOptimisticLockingExceptionMap per` to map vendor specific error codes to an optimistick locking error, for Camel to perform a retry. This requires `optimisticL ocking` to be enabled.

Optimistic Locking

Optimistic locking is set to on by default. If two exchanges attempt to insert at the same time an exception will thrown, caught, converted to an OptimisticLockingException, and rethrown.

What is preserved when persisting

`JdbcAggregationRepository` will only preserve any `Serializable` compatible data types. If a data type is not such a type its dropped and a `WARN` is logged. And it only persists the `Message` body and the `Message` headers. The `Exchange` properties are **not** persisted.

From Camel 2.11 onwards you can store the message body and select(ed) headers as String in separate columns.

Recovery

The `JdbcAggregationRepository` will by default recover any failed Exchange. It does this by having a background tasks that scans for failed Exchanges in the persistent store. You can use the `checkInterval` option to set how often this task runs. The recovery works as transactional which ensures that Camel will try to recover and redeliver the failed Exchange. Any Exchange which was found to be recovered will be restored from the persistent store and resubmitted and send out again.

The following headers is set when an Exchange is being recovered/redelivered:

Header	Type	Description
Exchange.REDELIVERED	Boolean	Is set to true to indicate the Exchange is being redelivered.
Exchange.REDELIVERY_COUNTER	Integer	The redelivery attempt, starting from 1.

Only when an Exchange has been successfully processed it will be marked as complete which happens when the `confirm` method is invoked on the `AggregationRepository`. This means if the same Exchange fails again it will be kept retried until it success.

You can use option `maximumRedeliveries` to limit the maximum number of redelivery attempts for a given recovered Exchange. You must also set the `deadLetterUri` option so Camel knows where to send the Exchange when the `maximumRedeliveries` was hit.

You can see some examples in the unit tests of camel-sql, for example this test.

Database

To be operational, each aggregator uses two table: the aggregation and completed one. By convention the completed has the same name as the aggregation one suffixed with `"_COMPLETED"`. The name must be configured in the Spring bean with the `RepositoryName` property. In the following example aggregation will be used.

The table structure definition of both table are identical: in both case a String value is used as key (**id**) whereas a Blob contains the exchange serialized in byte array.
However one difference should be remembered: the **id** field does not have the same content depending on the table.
In the aggregation table **id** holds the correlation Id used by the component to aggregate the messages. In the completed table, **id** holds the id of the exchange stored in corresponding the blob field.

Here is the SQL query used to create the tables, just replace `"aggregation"` with your aggregator repository name.

```
CREATE TABLE aggregation (
 id varchar(255) NOT NULL,
 exchange blob NOT NULL,
 constraint aggregation_pk PRIMARY KEY (id)
);

CREATE TABLE aggregation_completed (
 id varchar(255) NOT NULL,
 exchange blob NOT NULL,
 constraint aggregation_completed_pk PRIMARY KEY (id)
);
```

Storing body and headers as text

Available as of Camel 2.11

You can configure the `JdbcAggregationRepository` to store message body and select(ed) headers as String in separate columns. For example to store the body, and the following two headers `companyName` and `accountName` use the following SQL:

```
CREATE TABLE aggregationRepo3 (
 id varchar(255) NOT NULL,
 exchange blob NOT NULL,
 body varchar(1000),
 companyName varchar(1000),
 accountName varchar(1000),
 constraint aggregationRepo3_pk PRIMARY KEY (id)
);

CREATE TABLE aggregationRepo3_completed (
 id varchar(255) NOT NULL,
 exchange blob NOT NULL,
 body varchar(1000),
 companyName varchar(1000),
 accountName varchar(1000),
 constraint aggregationRepo3_completed_pk PRIMARY KEY (id)
);
```

And then configure the repository to enable this behavior as shown below:

```
<bean id="repo3" class="org.apache.camel.processor.aggregate.jdbc.
JdbcAggregationRepository">
    <property name="repositoryName" value="aggregationRepo3"/>
    <property name="transactionManager" ref="txManager3"/>
    <property name="dataSource" ref="dataSource3"/>
    <!-- configure to store the message body and following headers as text
in the repo -->
    <property name="storeBodyAsText" value="true"/>
    <property name="headersToStoreAsText">
       <list>
           <value>companyName</value>
           <value>accountName</value>
       </list>
    </property>
 </bean>
```

Codec (Serialization)

Since they can contain any type of payload, Exchanges are not serializable by design. It is converted into a byte array to be stored in a database BLOB field. All those conversions are handled by the `JdbcCodec` class. One detail of the code requires your attention: the `ClassLoadingAware ObjectInputStream`.

The `ClassLoadingAwareObjectInputStream` has been reused from the Apache ActiveMQ project. It wraps an `ObjectInputStream` and use it with the `ContextClassLoader` rather than the `currentThread` one. The benefit is to be able to load classes exposed by other bundles. This allows the exchange body and headers to have custom types object references.

Transaction

A Spring `PlatformTransactionManager` is required to orchestrate transaction.

Service (Start/Stop)

The `start` method verify the connection of the database and the presence of the required tables. If anything is wrong it will fail during starting.

Aggregator configuration

Depending on the targeted environment, the aggregator might need some configuration. As you already know, each aggregator should have its own repository (with the corresponding pair of table created in the database) and a data source. If the default lobHandler is not adapted to your database system, it can be injected with the `lobHandler` property.

Here is the declaration for Oracle:

```
<bean id="lobHandler" class="org.springframework.jdbc.support.lob.
OracleLobHandler">
  <property name="nativeJdbcExtractor" ref="nativeJdbcExtractor"/>
</bean>
<bean id="nativeJdbcExtractor" class="org.springframework.jdbc.support.
nativejdbc.CommonsDbcpNativeJdbcExtractor"/>
<bean id="repo" class="org.apache.camel.processor.aggregate.jdbc.
JdbcAggregationRepository">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="repositoryName" value="aggregation"/>
    <property name="dataSource" ref="dataSource"/>
    <!-- Only with Oracle, else use default -->
```

```
    <property name="lobHandler" ref="lobHandler"/>
</bean>
```

Optimistic locking

From **Camel 2.12** onwards you can turn on `optimisticLocking` and use this JDBC based aggregation repository in a clustered environment where multiple Camel applications shared the same database for the aggregation repository. If there is a race condition there JDBC driver will throw a vendor specific exception which the `JdbcAggregationRepository` can react upon. To know which caused exceptions from the JDBC driver is regarded as an optimistick locking error we need a mapper to do this. Therefore there is a `org.apache.camel.processor.aggregate.jdbc.JdbcOptimisticLockingExceptionMapper` allows you to implement your custom logic if needed. There is a default implementation `org.apache.camel.processor.aggregate.jdbc.DefaultJdbcOptimisticLockingExceptionMapper` which works as follows:

The following check is done:

If the caused exception is an `SQLException` then the SQLState is checked if starts with 23.

If the caused exception is a `DataIntegrityViolationException`

If the caused exception class name has "ConstraintViolation" in its name.

optional checking for FQN class name matches if any class names has been configured

You can in addition add FQN classnames, and if any of the caused exception (or any nested) equals any of the FQN class names, then its an optimistick locking error.

Here is an example, where we define 2 extra FQN class names from the JDBC vendor.

```
<bean id="repo" class="org.apache.camel.processor.aggregate.jdbc.
JdbcAggregationRepository">
   <property name="transactionManager" ref="transactionManager"/>
   <property name="repositoryName" value="aggregation"/>
   <property name="dataSource" ref="dataSource"/>
   <property name"jdbcOptimisticLockingExceptionMapper" ref="
myExceptionMapper"/>
</bean>
<!-- use the default mapper with extra FQN class names from our JDBC
driver -->
<bean id="myExceptionMapper" class="org.apache.camel.processor.aggregate.
jdbc.DefaultJdbcOptimisticLockingExceptionMapper">
   <property name="classNames">
     <util:set>
       <value>com.foo.sql.MyViolationExceptoion</value>
       <value>com.foo.sql.MyOtherViolationExceptoion</value>
     </util:set>
   </property>
</bean>
```

See Also

Endpoint

SQL Stored Procedure

JDBC