Apache Camel SQL Component

 javarticles.com/2015/08/apache-camel-sql-component.html

Ram Satish August 13, 2015

In this article, we will see some examples of Camel's SQL Component. `sql:` component is used to work with databases using JDBC queries.

It internally depends on Camel's JDBC Component. The main difference between SQL component and JDBC component is that in case of SQL the query is part of the endpoint and the body contains the parameters passed to the query. You may also pass the query parameters as header properties.

In case of JDBC component, SQL query is itself the payload.

This example uses the following frameworks:

Dependencies

We are just relying camel's core components, the spring based components and the logger component in case you want to log something so our `pom.xml` consists of:

1. `camel-core` – camel core components like timer, bean etc
2. `slf4j-api` – in case you want to use log
3. `slf4j-log4j12` – if you want to use log4j as the slf4j implementation
4. `camel-stream` – for printing the messages to console
5. `spring-context` and `spring-core` – for spring support
6. `camel-spring` – include it if you want to define route in spring
7. `camel-sql` – To access SQL component from Camel route
8. `mysql-connector-java` – MySQL Driver

pom.xml:

?

```
< modelVersion >4.0.0</ modelVersion >
< groupId >com.javarticles.camel</ groupId >
< artifactId >camelContentBasedRouting</ artifactId >
< version >0.0.1-SNAPSHOT</ version >
< dependencies >
< dependency >
< groupId >org.apache.camel</ groupId >
< artifactId >camel-core</ artifactId >
< version >2.15.2</ version >
</ dependency >
< dependency >
< groupId >org.slf4j</ groupId >
< artifactId >slf4j-api</ artifactId >
< version >1.7.12</ version >
</ dependency >
< dependency >
< groupId >org.slf4j</ groupId >
< artifactId >slf4j-log4j12</ artifactId >
< version >1.7.12</ version >
```

```
</ dependency >
< dependency >
< groupId >org.springframework</ groupId >
< artifactId >spring-context</ artifactId >
< version >4.1.5.RELEASE</ version >
</ dependency >
< dependency >
< groupId >org.apache.camel</ groupId >
< artifactId >camel-spring</ artifactId >
< version >2.15.2</ version >
</ dependency >
< dependency >
< groupId >org.apache.camel</ groupId >
< artifactId >camel-stream</ artifactId >
< version >2.15.2</ version >
</ dependency >
< dependency >
< groupId >org.apache.camel</ groupId >
< artifactId >camel-sql</ artifactId >
< version >2.15.2</ version >
</ dependency >
< dependency >
< groupId >org.springframework</ groupId >
< artifactId >spring-core</ artifactId >
< version >4.1.5.RELEASE</ version >
</ dependency >
< dependency >
< groupId >org.springframework</ groupId >
< artifactId >spring-context</ artifactId >
< version >4.1.5.RELEASE</ version >
</ dependency >
< dependency >
< groupId >mysql</ groupId >
< artifactId >mysql-connector-java</ artifactId >
< version >5.1.26</ version >
</ dependency >
< dependency >
< groupId >org.springframework</ groupId >
< artifactId >spring-jdbc</ artifactId >
< version >4.1.5.RELEASE</ version >
</ dependency >
</ dependencies >
</ project >
```

DataSource Set Up

We will be providing the database schema script and the same data to create.

db-schema.sql:

?

```
drop table if exists `articles`;
CREATE TABLE `articles` (
`ID` INT (10) UNSIGNED NOT NULL AUTO_INCREMENT,
` NAME ` VARCHAR (100) NOT NULL ,
`CATEGORY` VARCHAR (50) NOT NULL ,
`TAGS` VARCHAR (100) NOT NULL ,
`AUTHOR` VARCHAR (50) NOT NULL ,
PRIMARY KEY (`ID`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

db-test-data.sql:

?

```
insert into articles(id, name , category, tags, author) values
(1, "JdbcTemplate Example" , "spring" , "spring,jdbcTemplate" ,
"Joe" );
insert into articles(id, name , category, tags, author) values
(2, "Camel JMS Example" , "camel" , "camel,jms" , "Sam" );
insert into articles(id, name , category, tags, author) values
(3, "Camel JDBC Example" , "camel" , "camel,jdbc" , "Joe" );
```

Configure `DataSource` in spring XML context. Initialize database using <jdbc:initialize-database>.

applicationContext.xml:

?

```
<? xml version = "1.0" encoding = "UTF-8" ?>
< beans xmlns = "http://www.springframework.org/schema/beans"
xmlns:jdbc = "http://www.springframework.org/schema/jdbc"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">
< jdbc:initialize-database data-source = "dataSource"
enabled = "true" >
< jdbc:script location = "classpath:db-schema.sql" />
< jdbc:script location = "classpath:db-test-data.sql" />
</ jdbc:initialize-database >
< bean id = "dataSource"
class = "org.springframework.jdbc.datasource.DriverManagerDataSource" >
< property name = "driverClassName" value = "com.mysql.jdbc.Driver"
/>
< property name = "url" value = "jdbc:mysql://localhost/test" />
< property name = "username" value = "root" />
< property name = "password" value = "mnrpass" />
</ bean >
</ beans >
```

Camel SQL URI

The URI starts with `sql:` and then follows the SQL query. '#' symbol is the placeholder for the SQL parameter. The standard symbol is '?' but since it also used to denote options '#' is used instead.

?

```
sql: select * from table where id =
```

The parameters are provided as a `java.util.List` or `java.util.map` object.

Camel SQL Component Example

Our first example consists of a simple sql query.

?

```
'sql:select * from articles where category = #'
```

SQL component just like JDBC Component requires a `DataSource`. You can set the data source using the below statement.

?

```
camelContext.getComponent( "sql" , SqlComponent. class )
.setDataSource(dataSource);
```

You can also provide the `DataSource` as an option. We will see an example in our next section.
The queried result appears as an instance of List<Map<String, Object>> type.
We trigger the routing by passing the parameter value to `direct:sqlParam` endPoint. It retrieves all the articles with category='camel'.

?

```
template.sendBody( "direct:sqlParam" , "camel" );
```

CamelSqlSelectExample:

?

```
package com.javarticles.camel;
import javax.sql.DataSource;
import org.apache.camel.CamelContext;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.component.sql.SqlComponent;
import org.apache.camel.impl.DefaultCamelContext;
import org.apache.camel.impl.DefaultProducerTemplate;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class CamelSqlSelectExample {
public static void main(String[] args) throws Exception {
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
"applicationContext.xml" );
final DataSource dataSource = (DataSource) context
.getBean( "dataSource" );
CamelContext camelContext = new DefaultCamelContext();
camelContext.getComponent( "sql" , SqlComponent. class )
.setDataSource(dataSource);
try {
camelContext.addRoutes( new RouteBuilder() {
public void configure() {
from( "direct:sqlParam" )
.to( "sql:select * from articles where category = #" )
.process( new Processor() {
public void process(Exchange exchange)
throws Exception {
System.out.println(exchange.getIn()
.getBody().getClass());
System.out.println(exchange.getIn()
.getBody());
}
});
}
});
camelContext.start();
ProducerTemplate template = new DefaultProducerTemplate(
camelContext);
template.start();
template.sendBody( "direct:sqlParam" , "camel" );
} finally {
camelContext.stop();
context.close();
}
}
}
```

As you can see below, the processor attached, prints the payload type and payload.

Output:

?

```
class java.util.ArrayList
[{ID=2, NAME=Camel JMS Example, CATEGORY=camel, TAGS=camel,jms,
AUTHOR=Sam}, {ID=3, NAME=Camel JDBC Example, CATEGORY=camel,
TAGS=camel,jdbc, AUTHOR=Joe}]
```

DataSource in SQL Options

You can also provide the `DataSource` as a registry object and then refer to it in the option 'dataSource=dataSourceName'.
First bind the `DataSource` object do that it can be looked up from the context.

?

```
final DataSource dataSource = (DataSource)
context.getBean( "dataSource" );
JndiContext jndiContext = new JndiContext();
jndiContext.bind( "ds" , dataSource);
CamelContext camelContext = new DefaultCamelContext(jndiContext);
```

Refer to the `DataSource` in the SQL URI options.

?

```
'sql:select * from articles where category=# and author=#?
dataSource=ds'
```

If you note the query now has two parameters. We provide the parameters as 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.

?

```
List<String> params = new ArrayList<String>();
params.add( "camel" );
params.add( "Joe" );
template.sendBody( "direct:sqlParam" , params);
```

CamelSqlSelectDataSourceExample:

?

```
package com.javarticles.camel;
import java.util.ArrayList;
import java.util.List;
import javax.sql.DataSource;
import org.apache.camel.CamelContext;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.DefaultCamelContext;
import org.apache.camel.impl.DefaultProducerTemplate;
import org.apache.camel.util.jndi.JndiContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class CamelSqlSelectDataSourceExample {
public static void main(String[] args) throws Exception {
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
"applicationContext.xml" );
final DataSource dataSource = (DataSource) context
.getBean( "dataSource" );
JndiContext jndiContext = new JndiContext();
jndiContext.bind( "ds" , dataSource);
CamelContext camelContext = new DefaultCamelContext(jndiContext);
try {
camelContext.addRoutes( new RouteBuilder() {
public void configure() {
from( "direct:sqlParam" )
.to( "sql:select * from articles where category=# and author=#?
dataSource=ds" )
.log( "${body}" );
}
});
camelContext.start();
ProducerTemplate template = new DefaultProducerTemplate(
camelContext);
template.start();
List<String> params = new ArrayList<String>();
params.add( "camel" );
params.add( "Joe" );
template.sendBody( "direct:sqlParam" , params);
} finally {
camelContext.stop();
context.close();
}
}
}
```

Output:

?

```
12:45| INFO | MarkerIgnoringBase.java 95 | [{ID=3, NAME=Camel JDBC
Example, CATEGORY=camel, TAGS=camel,jdbc, AUTHOR=Joe}]
```

Use of Named Parameters in Query

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

Named parameter 'cat' is repeated in both the message body and headers still the one in message body takes precedence.

?

```
1  Map<String, String> params =  new HashMap<String, String>();
2  params.put( "cat" ,  "camel" );
3
4  Map<String, Object> headers =  new HashMap<String, Object>();
5  headers.put( "authr" ,  "Joe" );
6  headers.put( "cat" ,  "spring" );
7  template.sendBodyAndHeaders( "direct:sqlParam" , params, headers);
```

CamelSqlSelectNamedParametersExample:

?

```
package com.javarticles.camel;
import java.util.HashMap;
import java.util.Map;
import javax.sql.DataSource;
import org.apache.camel.CamelContext;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.impl.DefaultCamelContext;
import org.apache.camel.impl.DefaultProducerTemplate;
import org.apache.camel.util.jndi.JndiContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class CamelSqlSelectNamedParametersExample {
public static void main(String[] args) throws Exception {
ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext(
"applicationContext.xml" );
final DataSource dataSource = (DataSource) context
.getBean( "dataSource" );
JndiContext jndiContext = new JndiContext();
jndiContext.bind( "ds" , dataSource);
CamelContext camelContext = new DefaultCamelContext(jndiContext);
try {
camelContext.addRoutes( new RouteBuilder() {
public void configure() {
from( "direct:sqlParam" )
.to( "sql:select * from articles where category=:#cat and
author=:#authr?dataSource=ds" )
.log( "${body}" );
}
});
camelContext.start();
ProducerTemplate template = new DefaultProducerTemplate(
camelContext);
template.start();
Map<String, String> params = new HashMap<String, String>();
params.put( "cat" , "camel" );

Map<String, Object> headers = new HashMap<String, Object>();
headers.put( "authr" , "Joe" );
headers.put( "cat" , "spring" );
template.sendBodyAndHeaders( "direct:sqlParam" , params, headers);
} finally {
camelContext.stop();
context.close();
}
}
}
```

Camel SQL Component Example using Spring DSL

We will now configure the camel context using spring. We will build on our previous examples to convert the rows into a POJOs.
Instead of hard coding the SQL in the XML, will use property place holder and provide the

actual SQL in a property file.

sql.properties:

?

```
sql.articles= select * from articles where category =
```

`RowProcessor` will process the list of rows into list of POJOs and then set it as the new payload.

camelContext.xml:

?

```
<? xml version = "1.0" encoding = "UTF-8" ?>
< beans xmlns = "http://www.springframework.org/schema/beans"
xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">
< import resource = "applicationContext.xml" />

< bean id = "rowProcessor"
class = "com.javarticles.camel.RowProcessor" >
</ bean >
< camelContext xmlns = "http://camel.apache.org/schema/spring" >
< propertyPlaceholder id = "placeholder"
location = "classpath:sql.properties" />
< route >
< from uri = "direct:sqlParam" />
< to uri = "sql:{{sql.articles}}?dataSource=dataSource" />
< process ref = "rowProcessor" />
< log message = "${body}" />
</ route >
</ camelContext >
</ beans >
```

Converts the row data into POJOs.

RowProcessor:

?

```
package com.javarticles.camel;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import org.apache.camel.Exchange;
import org.apache.camel.Processor;
public class RowProcessor implements Processor {
public void process(Exchange exchange) throws Exception {
List<Map<String, Object>> rows =
exchange.getIn().getBody(List. class );
System.out.println( "Processing " + exchange.getIn().getBody());
List<Article> articles = new ArrayList<Article>();
for (Map<String, Object> row : rows) {
Article article = new Article();
article.setAuthor((String) row.get( "AUTHOR" ));
article.setCategory((String) row.get( "CATEGORY" ));
article.setName((String) row.get( "NAME" ));
article.setTags((String) row.get( "TAGS" ));
article.setId((Long) row.get( "ID" ));
articles.add(article);
}
exchange.getOut().setBody(articles);
}
}
```

Article:

?

```
1  package com.javarticles.camel;
2  public class Article {
3  private long id;
4  private String name;
5  private String author;
6  private String category;
7  private String tags;
8  public long getId() {
9  return id;
10 }
11 public void setId( long id) {
12 this .id = id;
13 }
14 public String getName() {
15 return name;
16 }
17 public void setName(String name) {
18 this .name = name;
19 }
20 public String getAuthor() {
21 return author;
22 }
23 public void setAuthor(String author) {
24 this .author = author;
25 }
26 public String getCategory() {
27 return category;
28 }
29 public void setCategory(String category) {
30 this .category = category;
31 }
32 public String getTags() {
33 return tags;
34 }
35 public void setTags(String tags) {
36 this .tags = tags;
37 }
38
39 public String toString() {
40 return "Article(id:" + id + ":" + "name:" + name + ":" +
41 "author:" + author + ":" + "category:" + category + ")" ;
42 }
43 }
44
```

CamelJdbcSelectExampleUsingSpring:

?

```
package com.javarticles.camel;
import java.util.List;
import org.apache.camel.CamelContext;
import org.apache.camel.ProducerTemplate;
import org.apache.camel.impl.DefaultProducerTemplate;
import org.apache.camel.spring.SpringCamelContext;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
public class CamelJdbcSelectExampleUsingSpring {
public static final void main(String[] args) throws Exception {
ClassPathXmlApplicationContext appContext = new
ClassPathXmlApplicationContext(
"camelContext.xml" );
CamelContext camelContext = SpringCamelContext.springCamelContext(
appContext, false );
try {
camelContext.start();
ProducerTemplate template = new DefaultProducerTemplate(
camelContext);
template.start();
List<Article> articles = (List<Article>)
template.requestBody( "direct:sqlParam" , "camel" );
for (Article article : articles) {
System.out.println(article);
}
Thread.sleep( 2000 );
} finally {
camelContext.stop();
appContext.close();
}
}
}
```

Output:

?

```
Processing [{ID=2, NAME=Camel JMS Example, CATEGORY=camel,
TAGS=camel,jms, AUTHOR=Sam}, {ID=3, NAME=Camel JDBC Example,
CATEGORY=camel, TAGS=camel,jdbc, AUTHOR=Joe}]
14:14| INFO | MarkerIgnoringBase.java 95 | [Article( id :2:name:Camel
JMS Example:author:Sam:category:camel), Article( id :3:name:Camel JDBC
Example:author:Joe:category:camel)]
Article( id :2:name:Camel JMS Example:author:Sam:category:camel)
Article( id :3:name:Camel JDBC Example:author:Joe:category:camel)
14:14| INFO | DefaultCamelContext.java 2660 | Apache Camel 2.15.2
```

Download the source code

This was an example about Apache Camel SQL Component.

You can download the source code here: **camelSqlComponentSelectExample.zip**