SqlC tutorial

12th may 2006

Revision 1.0

openbravo™
opening ERP's future!

Visit us at www.openbravo.com

Table of Contents

I. Introduction

SqlC is a tool that builds a java class from a file with SQL statements. The java class contains the code necessary to connect it to a database, format the statement, execute it and return the resultset from the database. This is a common function for any statement execution. SqlC avoids this repetitive task building the java code from the definitions of the desired statements.

II. The Xml input file

The SQL statements are defined in an XML file. These statements are directly executable on a database, so they are not defined for any specific programming language.

The next code is the full content of one of these files, *Hours_data.xsql*:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<SqlClass name="HoursData" package="data.hours">
  <SqlClassComment>Class HoursData</SqlClassComment>
  <SqlMethod name="select" type="preparedStatement"
return="multiple">
    <SqlMethodComment>Hours list</SqlMethodComment>
    <Sql>
        SELECT Employee, Year, Month, Day, Hours FROM Hours
    </Sql>
  </SqlMethod>
</SqlClass>
```

The first line indicates that it is an xml file version 1.0:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

The next line defines the name of the class that will be created and the package that it belongs to; in this case, the class will be *HoursData* in the *data.hours* package:

```
<SqlClass name="HoursData" package="data.hours">
```

The user can input a comment (optional) for the class; in this case, the text is *"Class HoursData"*:

```
  <SqlClassComment>Class HoursData</SqlClassComment>
```

In one file the user can define some methods. Each one will produce a method of the class. Each method will execute an SQL statement. In this example there is only one method.

The first line of the method defines the name of the method as "*select*" as well as two more attributes. The first attribute is the type of the method; in this case, it is a *preparedStatement*, and the second attribute is what returns the statement. In this case, it is *"multiple"* because it can have several rows. *"Multiple"* returns a vector with all the rows that are returned in the statement.

```
<SqlMethod name="select" type="preparedStatement"
return="multiple">
```

The user can input one statement for each method:

```
<SqlMethodComment>Hours list</SqlMethodComment>
```

The SQL statement is defined inside the tag <*Sql*>:

```
<Sql>
  SELECT Employee, Year, Month, Day, Hours FROM Hours
</Sql>
```

The next lines closes the method and the class.

III. Executing SqlC

SqlC is a java class. To execute SqlC, it is necessary to include the following classes in the *classpath*:

- SqlC
- Log4j (for debugging purposes)
- SaxParser (to read the xml file)

```
>set classpath=C:\AppsTecnicia\dist\classes;C:\Jakarta-Tomcat-
4.1.24\common\lib\log4j.jar;C:\AppsTecnicia\lib\xml4j_1_1_9.jar
```

We execute Sqlc with the following command:

```
>java org.tecnicia.data.Sqlc C:\Apps\manualApp\dbCon5.xml .xsql
. .
```

It accepts four arguments:

- **Configuration file**, where the user defines the driver and url to connect to a database.
- The **extension** of the files that SQL will read.
- The **directory** where Sqlc will **read the XML files**.
- The **directory** where Sqlc will **create the java classes**.

In this case the configuration file is *dbCon5.xml*, which has the following content:

```
<?xml version="1.0"?>
<data-base>
  <connection name="myPool" type="pool" driver =
"sun.jdbc.odbc.JdbcOdbcDriver" URL = "jdbc:odbc:tutorial">
  </connection>
</data-base>
```

In this example we use the *jdbc-odbc* bridge driver and the DSN of the ODBC has the name *tutorial*.

The extension that sqlc will read is XSQL like *Hours_data.xsql*.

It will read from the current directory “.” and the output will be in the same current directory “.”.

IV. The output

After executing Sqlc we get the file *HoursData.java*. Its content is this:

```
//Sqlc generated V1.O00-1
package data.hours;

import java.sql.*;
import java.util.*;
import java.lang.System;

import org.apache.log4j.Category;
import org.apache.log4j.PropertyConfigurator;

import javax.servlet.ServletException;

import org.openbravo.data.FieldProvider;
import org.openbravo.base.ConnectionProvider;
import org.openbravo.data.UtilSql;

/**Class HoursData
 */
public class HoursData implements FieldProvider {
static Category log4j = Category.getInstance(HoursData.class);
  private String InitRecordNumber="0";
  public String employee;
  public String year;
  public String month;
  public String day;
  public String hours;

  public String getInitRecordNumber() {
    return InitRecordNumber;
  }

  public String getField(String fieldName) {
    if (fieldName.toUpperCase().equals("EMPLOYEE") ||
fieldName.equals("employee"))
      return employee;
    else if (fieldName.toUpperCase().equals("YEAR") ||
fieldName.equals("year"))
      return year;
    else if (fieldName.toUpperCase().equals("MONTH") ||
fieldName.equals("month"))
      return month;
    else if (fieldName.toUpperCase().equals("DAY") ||
fieldName.equals("day"))
      return day;
    else if (fieldName.toUpperCase().equals("HOURS") ||
fieldName.equals("hours"))
      return hours;
    else {
      log4j.warn("No existe el campo: " + fieldName);
      return null;
    }
```

```
  }

/**Hours list
 */
  public static HoursData[] select(ConnectionProvider
connectionProvider)
    throws ServletException {
    return select(connectionProvider, 0, 0);
  }

/**Hours list
 */
  public static HoursData[] select(ConnectionProvider
connectionProvider, String keyValue, String keyName, int
numberRegisters)
    throws ServletException {
    boolean existsKey = false;
    String strSql = "";
    strSql = strSql + "";
    strSql = strSql + "        SELECT Employee, Year, Month,
Day, Hours FROM Hours";
    strSql = strSql + "    ";

    PreparedStatement st =
connectionProvider.getPreparedStatement(strSql);
    ResultSet result;
    Vector vector = new Vector(0);

    int iParameter = 0;
    try {

      result = st.executeQuery();
      long countRecord = 0;
      long initRecord = 0;
      boolean searchComplete = false;
      while(result.next() && !searchComplete) {
        countRecord++;
        HoursData objectHoursData = new HoursData();
        objectHoursData.employee = UtilSql.getValue(result,
"Employee");
        objectHoursData.year = UtilSql.getValue(result, "Year");
        objectHoursData.month = UtilSql.getValue(result,
"Month");
        objectHoursData.day = UtilSql.getValue(result, "Day");
        objectHoursData.hours = UtilSql.getValue(result,
"Hours");
        objectHoursData.InitRecordNumber =
Long.toString(initRecord);
        if (!existsKey) existsKey =
(objectHoursData.getField(keyName).equalsIgnoreCase(keyValue));
        vector.addElement(objectHoursData);
        if (countRecord == numberRegisters) {
          if (existsKey) searchComplete=true;
          else {
            countRecord = 0;
            initRecord += numberRegisters;
```

```
            vector.clear();
          }
        }
      }
      result.close();
    } catch(SQLException e){
      log4j.error("Error of SQL in query: " + strSql +
"Exception:"+ e);
      throw new
ServletException(Integer.toString(e.getErrorCode()));
    } finally {
      connectionProvider.releasePreparedStatement(st);
    }
    if (existsKey) {
      HoursData objectHoursData[] = new
HoursData[vector.size()];
      vector.copyInto(objectHoursData);
      return(objectHoursData);
    }
    return(new HoursData[0]);
  }

/**Hours list
 */
  public static HoursData[] select(ConnectionProvider
connectionProvider, int firstRegister, int numberRegisters)
    throws ServletException {
    String strSql = "";
    strSql = strSql + "";
    strSql = strSql + "        SELECT Employee, Year, Month,
Day, Hours FROM Hours";
    strSql = strSql + "    ";

    PreparedStatement st =
connectionProvider.getPreparedStatement(strSql);
    ResultSet result;
    Vector vector = new Vector(0);

    int iParameter = 0;
    try {

      result = st.executeQuery();
      long countRecord = 0;
      long countRecordSkip = 1;
      boolean continueResult = true;
      while(countRecordSkip < firstRegister && continueResult) {
        continueResult = result.next();
        countRecordSkip++;
      }
      while(continueResult && result.next()) {
        countRecord++;
        HoursData objectHoursData = new HoursData();
        objectHoursData.employee = UtilSql.getValue(result,
"Employee");
        objectHoursData.year = UtilSql.getValue(result, "Year");
        objectHoursData.month = UtilSql.getValue(result,
```

```
"Month");
        objectHoursData.day = UtilSql.getValue(result, "Day");
        objectHoursData.hours = UtilSql.getValue(result,
"Hours");
        objectHoursData.InitRecordNumber =
Integer.toString(firstRegister);
        vector.addElement(objectHoursData);
        if (countRecord >= numberRegisters && numberRegisters !=
0) {
          continueResult = false;
        }
      }
      result.close();
    } catch(SQLException e){
      log4j.error("Error of SQL in query: " + strSql +
"Exception:"+ e);
      throw new
ServletException(Integer.toString(e.getErrorCode()));
    } finally {
      connectionProvider.releasePreparedStatement(st);
    }
    HoursData objectHoursData[] = new HoursData[vector.size()];
    vector.copyInto(objectHoursData);
    return(objectHoursData);
  }
}
```

This file contents the java code of the *HoursData* class. This class has one private data member and five data members:

```
  public String employee;
  public String year;
  public String month;
  public String day;
  public String hours;
```

The same as the fields of the Sql statement:

```
  SELECT Employee, Year, Month, Day, Hours FROM Hours
```

This is a rule for all the classes generated with SqlC – the first method defined the data members of the class. In subsequent methods, you cannot use fields that have not been defined in the first method.

This class implements the *FieldProvider* interface. The *FieldProvider* interface only has one method:

```
package org.openbravo.data;

public interface FieldProvider  {
  public String getField(String fieldName);
}
```

In the *HoursData* class it returns with an “if” structure one of the five data members of the class.

The other important method implements the *SqlMethod* of the XML file:

```
  public static HoursData[] select()
```

Because the return is multiple, the return type is a vector: *HoursData[]*.

Looking at the method we recognize the usual functions to perform a query: format a string with the SQL statement, create a *PreparedStatement* from the connection, execute the query, read the resultset, create the vector and return it. (The class has two implementations of the method, one return all the records and a second that returns from a key value a defined number of records).

V. Types of methods

The types of return supported by Sqlc are:

- **constant**: returns an empty object of the class
- **preparedStatement**: execute a prepared statement on the database
- **callableStatement**: execute a callable statement on the database

In the example we add a method that return a constant:

```
<SqlMethod name="selectConstant" type="constant"
return="multiple">
    <SqlMethodComment>Hours list</SqlMethodComment>
    <Sql></Sql>
  </SqlMethod>
```

The code generated is an empty element:

```
  public static HoursDataEx2[] selectConstant()
    throws ServletException {
    HoursDataEx2 objectHoursDataEx2[] = new HoursDataEx2[1];
    objectHoursDataEx2[0] = new HoursDataEx2();
    objectHoursDataEx2[0].employee = "";
    objectHoursDataEx2[0].year = "";
    objectHoursDataEx2[0].month = "";
    objectHoursDataEx2[0].day = "";
    objectHoursDataEx2[0].hours = "";
    return objectHoursDataEx2;
  }
```

VI. Returns

The possible returns of a method are:

- **STRING**: return a string with the first field of the statement
- **BOOLEAN**: return a boolean with the first field of the statement
- **DATE**: return a date as a string with the first field of the statement
- **SINGLE**: return a unique object of the class
- **MULTIPLE**: return a vector of objects of the class
- **ROWCOUNT**: return the number of rows of the recordset
- **SEQUENCE**: return the key of the sequence created
- **OBJECT**: return an object of the type specified by the attribute “object”

The next is an example of a method that return a rowCount:

```
  <SqlMethod name="delete" type="preparedStatement"
return="rowCount">
    <Sql>
        DELETE FROM Hours
        WHERE Employee = ?

    </Sql>
    <Parameter name="param1"></Parameter>

  </SqlMethod>
```

The file now has another function:

```
  public static int delete(ConnectionProvider
connectionProvider, String param1)
    throws ServletException {
    String strSql = "";
    strSql = strSql + "";
    strSql = strSql + "        DELETE FROM Hours";
    strSql = strSql + "        WHERE Employee = ? ";
    strSql = strSql + "        ";
    strSql = strSql + "    ";

    PreparedStatement st =
connectionProvider.getPreparedStatement(strSql);
    int updateCount = 0;

    int iParameter = 0;
    try {
      iParameter++; UtilSql.setValue(st, iParameter, 12, null,
param1);

      updateCount = st.executeUpdate();
    } catch(SQLException e){
      log4j.error("Error of SQL in query: " + strSql +
"Exception:"+ e);
      throw new
```

```
ServletException(Integer.toString(e.getErrorCode()));
    } finally {
      connectionProvider.releasePreparedStatement(st);
    }
    return(updateCount);
  }
```

VII. Attributes of SqlMethod

In our example, there were three attributes: name, type and return. The complete list of attributes is:

- **name**: the name of the *SqlMethod*
- **return**: the type of return, as explained above
- **default**: the default value of return for the returns (*STRING, BOOLEAN* and *DATE*)
- **static**: to define if the methods are static or not, by default they are static
- **connection**: if the connection is provided in the call to the method
- **type**: type of method, as explained above
- **object**: the type of object, when the type is “OBJECT”
- **package**: the name of the package if the class needs to belong to a package
- **import**: if a specific import must be declared in the java file

VIII. Parameters

In an Sql method, it is customary to define parameters. These are the values for the “?” of the prepared statements or the parameters of a callable statement.

The attributes of parameters are:

- **name**: the name of the parameter
- **default**: the value, if not set in the call to the method
- **type**: if it is a “in”, “out” or “inout” parameter in a callable statement (in, out, inOut, replace, argument)
- **optional**: to define that it is optional (*true* or *false*)
- **after**: if it is optional (the attribute *optional* is set to *true*), this parameter defines where is going to be inserted the SQL stament defined in the parameter *text*. If *after* is not set, the text is placed after the WHERE clause, and the text is set to “*parameter_name = ? AND* ”
- **text**: if it is optional, the text to add to the sql statement