**DbUtils: JDBC Utility Component Examples**

This page provides examples that show how DbUtils may be used.

**Basic Usage**

DbUtils is a very small library of classes so it won't take long to go through the [javadocs](http://commons.apache.org/proper/commons-dbutils/apidocs/) for each class. The core classes/interfaces in DbUtils are [QueryRunner](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/QueryRunner.html) and [ResultSetHandler](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/ResultSetHandler.html). You don't need to know about any other DbUtils classes to benefit from using the library. The following example demonstrates how these classes are used together.

// Create a ResultSetHandler implementation to convert the  
// first row into an Object[].  
ResultSetHandler<Object[]> h = new ResultSetHandler<Object[]>() {  
    public Object[] handle(ResultSet rs) throws SQLException {  
        if (!rs.next()) {  
            return null;  
        }  
      
        ResultSetMetaData meta = rs.getMetaData();  
        int cols = meta.getColumnCount();  
        Object[] result = new Object[cols];  
  
        for (int i = 0; i < cols; i++) {  
            result[i] = rs.getObject(i + 1);  
        }  
  
        return result;  
    }  
};  
  
// Create a QueryRunner that will use connections from  
// the given DataSource  
QueryRunner run = new QueryRunner(dataSource);  
  
// Execute the query and get the results back from the handler  
Object[] result = run.query(  
    "SELECT \* FROM Person WHERE name=?", h, "John Doe");

You could also perform the previous query using a java.sql.Connection object instead of a DataSource. Notice that you are responsible for closing the Connection in this example.

ResultSetHandler<Object[]> h = ... // Define a handler the same as above example  
  
// No DataSource so we must handle Connections manually  
QueryRunner run = new QueryRunner();  
  
Connection conn = ... // open a connection  
try{  
    Object[] result = run.query(  
        conn, "SELECT \* FROM Person WHERE name=?", h, "John Doe");  
        // do something with the result  
          
} finally {  
    // Use this helper method so we don't have to check for null  
    DbUtils.close(conn);    
}

You can not only fetch data from the database - you can also insert or update data. The following example will first insert a person into the database and after that change the person's height.

QueryRunner run = new QueryRunner( dataSource );  
try  
{  
    // Execute the SQL update statement and return the number of  
    // inserts that were made  
    int inserts = run.update( "INSERT INTO Person (name,height) VALUES (?,?)",  
                              "John Doe", 1.82 );  
    // The line before uses varargs and autoboxing to simplify the code  
  
    // Now it's time to rise to the occation...  
    int updates = run.update( "UPDATE Person SET height=? WHERE name=?",  
                              2.05, "John Doe" );  
    // So does the line above  
}  
catch(SQLException sqle) {  
    // Handle it  
}

For long running calls you can use the AsyncQueryRunner to execute the calls asynchronously. The AsyncQueryRunner class has the same methods as the QueryRunner calls; however, the methods return a Callable.

ExecutorCompletionService<Integer> executor =  
    new ExecutorCompletionService<Integer>( Executors.newCachedThreadPool() );  
AsyncQueryRunner asyncRun = new AsyncQueryRunner( dataSource );  
  
try  
{  
    // Create a Callable for the update call  
    Callable<Integer> callable = asyncRun.update( "UPDATE Person SET height=? WHERE name=?",  
                                                  2.05, "John Doe" );  
    // Submit the Callable to the executor  
    executor.submit( callable );  
} catch(SQLException sqle) {  
    // Handle it  
}  
  
// Sometime later (or in another thread)  
try  
{  
   // Get the result of the update  
   Integer updates = executor.take().get();  
} catch(InterruptedException ie) {  
    // Handle it  
}

**ResultSetHandler Implementations**

In the examples above we implemented the ResultSetHandler interface to turn the first row of the ResultSet into an Object[]. This is a fairly generic implementation that can be reused across many projects. In recognition of this DbUtils provides a set of ResultSetHandler implementations in the [org.apache.commons.dbutils.handlers](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/handlers/package-summary.html) package that perform common transformations into arrays, Maps, and JavaBeans. There is a version of each implementation that converts just the first row and another that converts all rows in the ResultSet.

We'll start with an example using the BeanHandler to fetch one row from the ResultSet and turn it into a JavaBean.

QueryRunner run = new QueryRunner(dataSource);  
  
// Use the BeanHandler implementation to convert the first  
// ResultSet row into a Person JavaBean.  
ResultSetHandler<Person> h = new BeanHandler<Person>(Person.class);  
  
// Execute the SQL statement with one replacement parameter and  
// return the results in a new Person object generated by the BeanHandler.  
Person p = run.query(  
    "SELECT \* FROM Person WHERE name=?", h, "John Doe");

This time we will use the BeanListHandler to fetch all rows from the ResultSet and turn them into a List of JavaBeans.

QueryRunner run = new QueryRunner(dataSource);  
  
// Use the BeanListHandler implementation to convert all  
// ResultSet rows into a List of Person JavaBeans.  
ResultSetHandler<List<Person>> h = new BeanListHandler<Person>(Person.class);  
  
// Execute the SQL statement and return the results in a List of  
// Person objects generated by the BeanListHandler.  
List<Person> persons = run.query("SELECT \* FROM Person", h);

**Custom RowProcessor**

Each of the provided ResultSetHandler implementations accept a [RowProcessor](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/RowProcessor.html) to do the actual conversion of rows into objects. By default the handlers use the [BasicRowProcessor](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/BasicRowProcessor.html)implementation but you can implement a custom version to plug in. Probably the most common customization is to implement the toBean() method to handle custom database datatype issues.

**Custom BeanProcessor**

BasicRowProcessor uses a [BeanProcessor](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/BeanProcessor.html) to convert ResultSet columns into JavaBean properties. You can subclass and override processing steps to handle datatype mapping specific to your application. The provided implementation delegates datatype conversion to the JDBC driver.

BeanProcessor maps columns to bean properties as documented in the [BeanProcessor.toBean()](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/BeanProcessor.html#toBeanjava.sql.ResultSet20java.lang.Class) javadoc. Column names must match the bean's property names case insensitively. For example, the firstname column would be stored in the bean by calling its setFirstName() method. However, many database column names include characters that either can't be used or are not typically used in Java method names. You can do one of the following to map these columns to bean properties:

1. Alias the column names in the SQL so they match the Java names: select social\_sec# as socialSecurityNumber from person
2. Subclass BeanProcessor and override the [mapColumnsToProperties()](http://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/BeanProcessor.html#mapColumnsToPropertiesjava.sql.ResultSetMetaData20java.beans.PropertyDescriptor) method to strip out the offending characters.