(/)

Introduction to the JDBC RowSet Interface in Java

Last modified: August 15, 2018

by Michael Good (/author/michael-good/)

Java (/category/java/) + Persistence (/category/persistence/)

I just announced the new *Spring Boot 2* material, coming in REST With Spring:

>> CHECK OUT THE COURSE (/rest-with-spring-course#new-modules)

1. Overview

In this article, we're reviewing the JDBC *RowSet* interface**. A JDBC *RowSet* object holds tabular data in a style that makes it more adaptable and simpler to use than a result set.**

Oracle has defined five *RowSet* interfaces for the most frequent uses of a *RowSet:*

- *JdbcRowSet*
- *CachedRowSet*

- *WebRowSet*
- *JoinRowSet*
- *FilteredRowSet*

In this tutorial, we'll review how to use these *RowSet* interfaces.

2. *JdbcRowSet*

Let's start with the *JdbcRowSet* – we'll simply create one by passing a *Connection* object to the *JdbcRowSetImpl*:

```
JdbcRowSet jdbcRS = new JdbcRowSetImpl(conn);
jdbcRS.setType(ResultSet.TYPE_SCROLL_INSENSITIVE);
String sql = "SELECT * FROM customers";
jdbcRS.setCommand(sql);
jdbcRS.execute();
jdbcRS.addRowSetListener(new ExampleListener());
while (jdbcRS.next()) {
    // each call to next, generates a cursorMoved event
    System.out.println("id = " + jdbcRS.getString(1));
    System.out.println("name = " + jdbcRS.getString(2));
}
```

In the above example, *jdbcRs* contained no data until we defined the SQL statement with the method *setCommand* and then ran the method *execute*.

Also notice how, in order to perform event handling, we added a *RowSetListener* into the *JdbcRowSet.*

JdbcRowSet is different than the other four *RowSet* implementations – because **it's always connected to the database** and because of this it's most similar to the *ResultSet* object.

3. *CachedRowSet*

A *CachedRowSet* object is unique because it can operate without being connected to its data source. We call this a "disconnected *RowSet* object".

CachedRowSet gets its name due to the fact it caches its data in memory so that it can operate on its own data instead of the data stored in a database.

As *CachedRowSet* interface is the **super interface for all disconnected RowSet objects**, the code we review below is also applicable to a *WebRowSe*t, *JoinRowSet*, or *FilteredRowSe*t just as well:

```
CachedRowSet crs = new CachedRowSetImpl();
crs.setUsername(username);
crs.setPassword(password);
crs.setUrl(url);
crs.setCommand(sql);
crs.execute();
crs.addRowSetListener(new ExampleListener());
while (crs.next()) {
    if (crs.getInt("id") == 1) {
        System.out.println("CRS found customer1 and will remove the rec
        crs.deleteRow();
        break;
    }
}
```

4. *WebRowSet*

Next, let's have a look at the *WebRowSet*.

This is also unique because, in addition to offering the capabilities of a *CachedRowSet* object, **it can write itself to an XML documen**t and can also read that XML document to convert itself back to a *WebRowSet*:

```
WebRowSet wrs = new WebRowSetImpl();
wrs.setUsername(username);
wrs.setPassword(password);
wrs.setUrl(url);
wrs.setCommand(sql);
wrs.execute();
FileOutputStream ostream = new FileOutputStream("customers.xml");
wrs.writeXml(ostream);
```

Using the *writeXml* method, we write the current state of a *WebRowSet* object to an XML document.

By passing the *writeXml* method an *OutputStream* object, we write in bytes instead of characters, which can be quite helpful to handle all forms of data.

5. *JoinRowSet*

JoinRowSet lets us create a SQL *JOIN* between *RowSet* objects when these are in memory. This is significant because it saves us the overhead of having to create one or more connections:

```
CachedRowSetImpl customers = new CachedRowSetImpl();
// configuration of settings for CachedRowSet
CachedRowSetImpl associates = new CachedRowSetImpl();
// configuration of settings for this CachedRowSet
JoinRowSet jrs = new JoinRowSetImpl();
jrs.addRowSet(customers,ID);
jrs.addRowSet(associates,ID);
```

Because each *RowSet* object added to a *JoinRowSet* object needs a match column, the column on which the SQL *JOIN* is based, we specify *"id"* in the *addRowSet* method.

Note that, rather than using the column name, we could have also used the column number.

6. *FilteredRowSet*

Finally, the *FilteredRowSet* **lets us cut down the number of rows that are visible** in a *RowSet* object so that we can work with only the data that is relevant to what we are doing.

We decide how we want to "filter" the data using an implementation of the *Predicate* interface:

```
public class FilterExample implements Predicate {

    private Pattern pattern;

    public FilterExample(String regexQuery) {
        if (regexQuery != null && !regexQuery.isEmpty()) {
            pattern = Pattern.compile(regexQuery);
        }
    }

    public boolean evaluate(RowSet rs) {
        try {
            if (!rs.isAfterLast()) {
                String name = rs.getString("name");
                System.out.println(String.format(
                  "Searching for pattern '%s' in %s", pattern.toString(
                  name));
                Matcher matcher = pattern.matcher(name);
                return matcher.matches();
            } else
                return false;
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    // methods for handling errors
}
```

Now we apply that filter to a *FilteredRowSet* object:

```
RowSetFactory rsf = RowSetProvider.newFactory();
FilteredRowSet frs = rsf.createFilteredRowSet();
frs.setCommand("select * from customers");
frs.execute(conn);
frs.setFilter(new FilterExample("^[A-C].*"));

ResultSetMetaData rsmd = frs.getMetaData();
int columncount = rsmd.getColumnCount();
while (frs.next()) {
    for (int i = 1; i <= columncount; i++) {
        System.out.println(
          rsmd.getColumnLabel(i)
          + " = "
          + frs.getObject(i) + " ");
        }
    }
```

7. Conclusion

This quick tutorial covered the five standard implementations of the *RowSet* interface available in the JDK.

We discussed the configuration of each implementation and mentioned the differences between them.

As we pointed out, only one of the *RowSet* implementations is a connected *RowSet* object – the *JdbcRowSet*. The other four are disconnected *RowSet* objects.

And, as always, the full code for this article can be found over on Github (https://github.com/eugenp/tutorials/tree/master/core-java-persistence).

I just announced the new Spring Boot 2 material, coming in REST With Spring:

>> CHECK OUT THE LESSONS (/rest-with-spring-course#new-modules)