SYBASE®

Data Filtering API Cookbook

Sybase Unwired Platform 1.2

Contents

Result Set Filters

A result set filter is a custom java class an experienced developer writes in order to specifically manipulate the rows or columns of data returned from a read operation for an MBO. To write a filter, developers must have previous experience with Java programming — particularly with the reference implementations for `javax.sql.RowSet`, which is used to implement the filter interface.

Note: For more information on the javadocs for the Filter API, go to either:

```
%SUP_HOME%\Servers\UnwiredServer\tomcat\webapps\onepage\docs\javadoc
\index.html
```

or

```
http://<host>:port/onepage/docs/javadoc/index.html
```

When a read operation returns data that does not completely suit the business requirements for your MBO, you can write and add a filter to the MBO to customize the data into the form you need. A filter can be written to add, delete, or change columns as well as to add and delete rows.

You can chain multiple filters together. Multiple filters are processed in the order they are added, each applying an incremental change to the data. Consequently, Sybase recommends that you always preview the results. After you preview the MBO, notice that the MBO has a different set of attributes than it would have had directly from the read operation. You can map and use the altered attributes in the same way you would a regular attribute from an unfiltered read operation.

Note: The filter interfaces are defined in terms of `java.sql.ResultSet` and `java.sql.ResultSetMetaData`, but these standard JDBC interfaces tend to be read-only implementations. To change data, use a `CachedRowSetImpl` object instead. This object implements ResultSet but also allows you to modify row data.

Example: a simple SELECT statement filter

Suppose you have an MBO based on the following query:

```
SELECT * FROM sampledb.customer
```

However, you do not want to have "fname" and "lname" displayed in separate columns. To avoid separation, write a filter that replaces these columns with a single concatenated "commonName" column.

Example: two separate data sources filter

Suppose you have customer data in two data sources: basic customer information is in an SAP repository, and more complete details are contained in another database on your network. You can use a result set filter to combine the SAP customer data with additional database customer data, so that the MBO displays a complete set of information in a single view. In this case, you would use a JDBC connection to each data source, and perhaps use a `JoinRowSetImpl` to merge the columns together, thereby joining the two sets of data against the customerID column in the SAP source.

Filtering Result Sets Returned by Attributes

Configure a result set filter when you define the attributes used to read data from your data source and create a result set (that is, a set of rows and metainformation). You can configure attributes when you create a mobile business object or when you edit the object's properties.

Before adding a result set filter to an MBO from Unwired WorkSpace, perform these tasks:

Adding Result Set Filters

Choose and add a result set filter when you edit Attribute properties for a mobile business object from the **Definitions** tab. You can also configure result set filters when you create an object. You can choose a predefined or a custom filter, if you have created one.

1. (Optional) If you have not yet created the classes, then in the **Resultset Filters** area of the **New Attributes** dialog, click **Create** to run the New Java Class wizard.
2. If prompted, add a Java nature.
   - (Recommended) Click **Yes** to add a Java nature. In Eclipse, a Java nature adds Java-specific behavior to projects. A Java nature is recommended because you are creating a new Java project and adding a Java class to it. By default an Unwired Platform project does not include all the required behaviors for Java development. Clicking **Yes** automates this process.

     If you clicked **Yes**, a wizard appears allowing you to enter the java package name and java class name. Click **Finish** in the wizard to compile the java skeleton source file and add the skeleton java filter class to the MBO. If you are adding the filter from the **Definition** tab of Attribute Properties view, you are prompted to refresh the data source definition. Choose **Yes** in response to this prompt. Other wise, choose **No** and only refresh after the logic has been put implemented and the java class has been built.

     Note: Only the wizard generates a skeleton java source file.
   - Click **No** if you do not want to add the Java nature to the selected Mobile Application Project.
3. Implement the new class by writing the real implementation on top of the skeleton created as documented in the next topic, *Writing a Custom Result Set Filter*.
4. Add the filter you require to your mobile business object. In the **Resultset Filters** area of the **New Attributes** dialog, click **Add**. If you are creating an MBO, you can also perform similar action with the corresponding wizard.
5. Select an existing filter class that you imported into Unwired WorkSpace.

   Only valid filters (that is, filters implemented with com.sybase.uep.eis.ResultFilter) appear in the filters table.
6. (Optional) Repeat steps 3 and 4 to add more filters to the mobile object.
7. (Optional) If the filters are not in the order you require, reorder them with either the **Up** or **Down** button. The order of multiple filters affects the actual Result set and metadata.
8. Test and preview the Result of your filter settings:

   a) To reuse input values you have already saved for previous previews, select **Existing Configuration**. Otherwise, load defaults, or create a new set of input values expressly for this preview instance.

   b) To run the preview, click **Preview**.

   If the data filters successfully, `Execution Succeeded` appears at the top of the Preview dialog and data appears in the **Preview Result** window.

Writing a Custom Result Set Filter

Writing a custom result set filter allows you to define the application processing logic you require. A custom filter requires that you write a Java filter class and save the compiled Java class file to location that is accessible from Unwired Workspace.

A custom result set filter allows you to process the record set returned by the attribute properties you configure so that it can be better consumed by the device client application you create. Sometimes, a result set returned from a data source requires unique processing and the filter you create performs that function before the information is downloaded to the client.

Note: For more information on the javadocs for the Filter API, go to either:

`%SUP_HOME%\Servers\UnwiredServer\tomcat\webapps\onepage\docs\javadoc\index.html`

or

`http://<host>:port/onepage/docs/javadoc/index.html`

1. (Required) Create a record set filter class that implements the `com.sybase.uep.eis.ResultSetFilter` interface.

   This interface defines how a custom filter for the data is called.

   For example, the following code segment sets the package name and import the required classes:

   ```
   package com.mycompany.myname;
   import java.sql.ResultSet;
   import java.util.Map;
   ```
2. (Recommended) Implement the `sybase.uep.eis.ResultSetFilterMetaData` interface on your filter class as required by your business requirements.

   Without this interface, you need to execute a chain of mobile business object operations and filters with real data before you can get actual results of the output columns and their data types. This can impact information on the data source that may need to be reverted in that system. By implementing this interface, the operation does not need to be executed first. Instead, the getMetaData method is used to obtain the necessary column or data type information.

   The following example sets the package name but uses a different combination of classes than in the example for step 1:

   ```
   package com.mycompany.myname;
   import java.sql.ResultSetMetaData;
   import java.util.Map;
   ```
3. Depending on the interfaces you implement, call the appropriate method.

   ResultSetFilter performs the filtering of data in the first option documented in step 1. Each filter defines a distinct set of arguments for its own use. Therefore, use the only arguments with the specific filter that defines these arguments in the `getArguments()` method, rather than use all filters and data source operations.

   The ResultSet passed in contains the grid data. Data should be considered to be read only. Do not use operations that change or transform data.

   ```
   public interface ResultSetFilter {
      ResultSet filter(ResultSet in, Map<String, Object> arguments) throws
        Exception;
      Map <String, Class> getArguments();
   }
   ```

   ResultSetFilterMetaData should then be used to show the how filtered data of the second option documented in step 1 appears. Use this interface to avoid executing a data source operation to generate a sample data set returned and thereby avoid extraneous executions.

   ```
   public interface ResultSetFilterMetaData {
      ResultSetMetaData getMetaData(ResultSetMetaData in, Map<String,
         Object> arguments) throws Exception;
   }
   ```

   Note: If the filter returns different columns depending on the argument values supplied, then the filter may not work reliably. If arguments affect the metadata, ensure that the arguments have constant values in the final mobile business object definition, so the schema does not change dynamically.

4. Implement the class you have created, defining any custom processing logic at this time.

   For example, to create a filter that adds an integer valued column called "rowCount" as the first column in the result set, you might implement the class as follows:

```
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.util.HashMap;
import java.util.Map;

import com.sun.rowset.CachedRowSetImpl;
import javax.sql.RowSetMetaData;

import javax.sql.rowset.RowSetMetaDataImpl;

public class AddRowCountFilter
    implements com.sybase.eis.ResultSetFilter,
com.sybase.eis.ResultSetFilterMetaData
{

      public ResultSet filter(ResultSet in, Map<String, Object> arguments)
            throws Exception
         {
            CachedRowSetImpl crsi = new CachedRowSetImpl();
            ResultSetMetaData rsmd = in.getMetaData();
            crsi.setMetaData(createMetaData(rsmd));
            in.moveToCurrentRow();
            in.beforeFirst();

            for (int i = 0; in.next(); i++)
            {
                crsi.moveToInsertRow();
                crsi.updateInt(1, i + 1);

            for (int j = 0; j < rsmd.getColumnCount(); j++)
            {
                 crsi.updateObject(j + 2, in.getObject(j + 1));
            }
            crsi.insertRow();
         }
         crsi.moveToCurrentRow();
         crsi.beforeFirst();

         return crsi;
}

     public Map<String, Class> getArguments()
     {
          return new HashMap<String, Class>();
     }

      public ResultSetMetaData getMetaData(ResultSetMetaData in, Map<String,
Object> arguments)
               throws Exception

      {
           return createMetaData(in);
      }

       private RowSetMetaData createMetaData(ResultSetMetaData rsmd) throws
Exception
       {
            RowSetMetaData md = new RowSetMetaDataImpl();
            md.setColumnCount(rsmd.getColumnCount() + 1);
            md.setColumnLabel(1, "rowCount");
            md.setColumnName(1, "rowCount");
            md.setColumnType(1, java.sql.Types.INTEGER);

            for (int i = 0; i < rsmd.getColumnCount(); i++)
            {
                md.setColumnLabel(i + 2, rsmd.getColumnLabel(i + 1));
                md.setColumnName(i + 2, rsmd.getColumnName(i + 1));
                md.setColumnType(i + 2, rsmd.getColumnType(i + 1));
            }
             return md;
        }
}
```

5. Save the classes you have created to an accessible Unwired WorkSpace location. This allows you to select the class when you configure result set filters for your mobile business object.

Debugging Filter Classes

You can set debugging options to view the output stream when using System.out.printIn in filter classes to help you debug your filter classes.

1. Go to `<SUP Installation Root>\Eclipse` and open the `UnwiredWorkSpace.bat` file with a text editor.
2. If the -vm options is specified, replace `javaw.exe` with `java.exe`.

   Note: The `javaw.exe` command is the same as the `java.exe` command except that with `javaw.exe`, there is no associated console window.
3. After the line `%ECLIPSE_ROOT%\eclipse.exe" %ADDITIONAL_ARGS%` add either `-debug` or `-consoleLog`.
4. Start Eclipse.

   A Java console window appears with the output.
5. View the debugging statement through `System.out.println` on the server side.

   On the Unwired Server side, all debug statements are saved to the `ml.log` file.

Deploying Custom Filters to Unwired Server

Before deploying mobile business objects that use custom filter classes, you must copy the filter classes to the Unwired Server.

If your Unwired Servers are running in a cluster, you must copy the filter classes to the "primary" server in the cluster. From Sybase Control Center, you can identify which server is the current primary on the left-hand panel that lists your Unwired Servers.

1. (Optional) Create a Java archive of all your classes if you would like to maintain and deploy only a single file.
   For example, running this JAR command generates these messages:

   ```
   C:\workspace.2\549543\bin>jar -cvf testFilter.jar *

   added manifest adding: test/(in = 0) (out= 0)(stored 0%)

   adding: test/TestFilter.class(in = 3409) (out= 1596)(deflated 53%)
   ```
2. Choose the target location for the filter classes and their dependency/third-party `.JAR`s on the Unwired Server:

   - To avoid having to restart Unwired Server, copy the filter classes (either in a `.JAR` file or the tree of filter classes) to:

     `<Unwired-Server-install>\lib\filters\<DeploymentPackageName>`

For example, if you have a FilterTest_1.0.0 package with a filter called com.acme.filters.TestFilter, you would create a `<Unwired-Server-install>\lib\filters\FilterTest_1.0.0\com\acme\filters\TestFilter.class` directory. Or, you can jar up the com.acme.filters.TestFilter class into a `acmeFilters.jar` file and copy it to `<Unwired-Server-install>\lib\filters\FilterTest_1.0.0\acmeFilters.jar`.

- To restart Unwired Server post-deployment, copy the filter classes to:

  `<Unwired-Server-install>\lib\filters\`

Whenever a result set filter is deleted, renamed, or moved (from WorkSpace Navigator for example), all it's references are automatically updated.

3. Deploy the mobile business object to the Unwired Server.

The custom classloader for FilterTest_1.0.0. is refreshed during deployment so there is no need to restart Unwired Server.

Validating Result Set Filter Performance

Once you have deployed the filters to Unwired Server, you should synchronize data and ensure that filters are performing as you expect.

1. Confirm that the columns appear correctly after the filter has been added to the mobile business object.
   a) Refresh the object.
   b) In the Properties view, select the **Attribute Mapping** tab.
   c) Check the **Map to** column to see that columns appear in the list correctly.

2. From the device client or the device simulator, open the mobile object, and check that the new column appears.

3. Synchronize the object from the device client or simulator.

4. Review the appropriate log to troubleshoot the filters if issues arise:

- While the data synchronization operation is being performed, all `system.out` statements are printed to the `ml.log` file on Unwired Server.
- If you started Unwired Workspace with the -consoleLog and java.exe options, `system.out` statements are also printed to the console window.