org.jax.mgi.shr.dbutils Design Document

Author: Mike Walker

Created: March 20, 2003

Last Modified: September 17, 2004 00:12

1 Purpose of Document

This document describes the classes belonging to the org.jax.mgi.shr.dbutils package and provides source code examples for common usage patterns.

2 Introduction

The purpose of this package is to provide encapsulation of database access and functionality through JDBC. Additionally there are sub packages belonging to this package but are discussed in separate documents. These include the bcp and dao packages.

This product integrates with the org.jax.mgi.shr.exception package (see lib_java_exception), the org.jax.mgi.shr.log package (see lib_java_log), the org.jax.mgi.shr.config package (see lib_java_config), and org.jax.mgi.shr.types (see lib_java_types).

It was written for the Java1.4 JVM release.

3 Configuration

Configuration is accomplished by use of the org.jax.mgi.shr.config package. The configurator class included in this package is the **DatabaseCfg** class. This class is used to handle the accessing of configuration parameters needed for the **SQLDataManager** class. Details of how this is done can be found in the documentation for the config package, but in brief, the configuration parameters are read in from configuration files and java system properties. The configuration file names are indicated on the java command line by use of the CONFIG system property (see example 1). The configurator class, **DatabaseCfg**, looks up the values for these parameters and provides accessor methods for them (see Example 2). If you have more than one object that needs different configurations, for example, two **SQLDataManagers** need to be pointing to two different databases, then you can prefix the base parameter names in the configuration file with an additional string of choice. An example configuration file is shown in Example 3 where the prefixes ‘RADAR’ and ‘MGD’ are used. That prefix string can then be passed to a **DatabaseCfg** object in its constructor, forcing it to lookup those parameters prefixed by that string first and, if not found, search on the un-prefixed string secondly (see Example 4).

EXAMPLE 1. setting the configuration

```
java -DCONFIG=configfile -DPARM1=OVERRIDE -DPARM2=OVERRIDE2 <app>
```

EXAMPLE 2. using configuration objects

```
// the constructor will cause all configuration files and java
// properties to be read for the known parameters

DatabaseCfg dbConfig = new DatabaseCfg();

// values are now accessable

String database = dbConfig.getDatabase();
```

EXAMPLE 3. configuration using prefixes

```
# Primary database values
RADAR_DBSERVER=DEV_MGI
RADAR_DBNAME=radar

# Secondary database values
MGD_DBSERVER=DEV_MGI
MGD_DBNAME=mgd_jsam
```

EXAMPLE 4. using configuration objects with prefixing

```
// the constructor will cause all configuration files and java
// properties to be read for the known parameters prefixed by
// the given string

DatabaseCfg radarConfig = new DatabaseCfg('RADAR');
databaseCfg mgdConfig = new DatabaseCfg('MGD');
```

The **SQLDataManager** has configuration parameters for setting database connection parameters and for setting the install directory for the DBSchema product.

The following is a list of all the configuration parameters.

DBSERVER

The database server name. The default is DEV_MGI. This parameter is mainly used for bcp and script execution.

DBURL

The url used for obtaining a JDBC connection to the database. The default is rohan.informatics.jax.org:4100

DBNAME

The database name. The default is mgd.

DBUSER

The database user to log in as. The default is mgd_dbo.

DBPASSWORD

The password for the database user to log in as. There is no defualt value.

DBPASSWORDFILE

The name of the file containing the database password. The default is /usr/local/mgi/live/dbutils/mgd/mgddbschema.

DBSCHEMADIR

The directory name where the DBSchema product is installed. The defualt is /usr/local/mgi/live/dbutils/mgd/mgddbschema

DBCONNECTION_MANAGER

The name of the class used for obtaining a database connection. The default is org.jax.mgi.shr.dbutils.MGIDriverManager.

All these parameters can be overridden at runtime with setter methods within the **SQLDataManager**.

4 Overview of Classes

The **SQLDataManager** class encapsulates a database connection and provides methods for executing SQL in a manner consistent with other in-house java classes in exception handling, data types, logging and configuration.

The **SQLDataManagerFactory** class provides a way to share a reference to a **SQLDataManager** amongst multiple client classes. This factory stores named references to **SQLDataManager** instances and allows for reuse of database connections.

The **MGIDriverManager** class is used internally by the **SQLDataManager** for obtaining a JDBC connection to the Sybase database. This class implements the **ConnectionManager** interface and can be interchanged with other **ConnectionManagers** for other database connections through the configuration parameter DBCONNECTION_MANAGER.

The **DatabaseCfg** class is used to configure the **SQLDataManager** from parameters within configuration files and command line arguments.

The **ResultsNavigator** interface is used to navigate through the SQL query results. The **SQLDataManager** returns a **ResultsNavigator** when a query is executed. It has methods next() and getCurrent() or getRowReference() as the means of navigating through the results and accessing the data.

The **RowReference** class is a wrapper around the ResultSet class form JDBC. It points to the current row in the ResultSet and allows access to that row, but it does not allow you to change the ResultSet pointer. Changing the ResultSet pointer is done through the **ResultsNavigator**. The **RowReference** object is returned from the call to getRowReference() on the **ResultsNavigator** and is returned as the default object type from a call to the getCurrent() method of the **ResultsNavigator**.

The **RowDataInterpreter** interface is used for "plugging-in" a class to the **ResultsNavigator** so that any java data object can be instantiated and returned instead of a **RowReference** during the call to the getCurrent() method. The new objects are created using a callback method on the **RowDataInterpreter**. This method is interpret(**RowReference**). Each newly created object will be based on data from the current row. The **RowDataInterpreter** is provided to the **ResultsNavigator** by way of a setter method.

The **MultiRowInterpreter** is an extension of the **RowDataInterpreter** and it is used for creating java data objects based on multiple rows of database records. These results sets have been sorted in some way so that the multiple records are consecutive to one another. It is used by the **MultiRowIterator** when iterating through multiple rows of data which represent a more complex java data type.

The **MultiRowIterator** iterates through a query results set and returns a java data object based on more than one row data for each logical groupings of rows. This approach assumes that there is some ordering of rows so that each grouping is made of contiguous rows.

The **Table** class is responsible for providing meta data on a given database table as the form of a **ColumnDef** object. It also is responsible for caching the next key value for tables which have an incremental primary key. It is also used for validating data specific to the table.

The **ColumnDef** class encapsulates the column meta data for the columns of a database table, including the column names and types. It is obtained from an instance of the **Table** class.

The **BindableStatement** class is a wrapper class for the JDBC PreparedStatement class. The main purpose is for integrating the other in-house java classes written for handling data types, exceptions and logging.

The **BatchProcessor** is a class for handling JDBC batch execution. An instance of this class is obtained by calling getBatchProcessor() on the **SQLDataManager** class. The main purpose is for integrating the other in-house java classes written for handling data types, exceptions and logging.

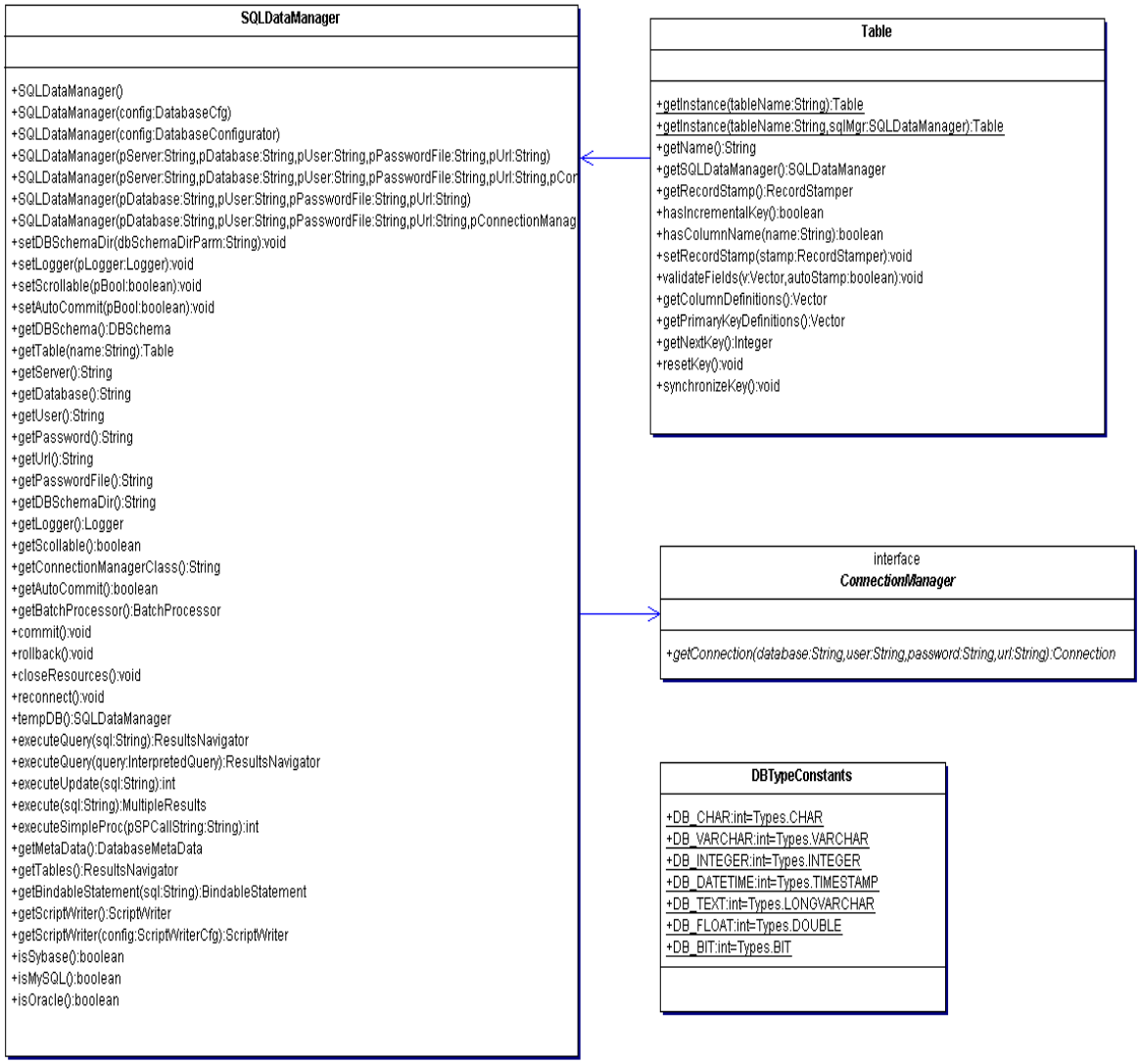
The **ScriptWriter** is a class for executing sql scripts into Sybase through the isql command. An instance of this class is obtained by calling getScriptWriter() on the **SQLDataManager** class.

The **DBSchema** class is used to execute DDL commands to the database. An instance of this class is obtained by calling getDBSchema() on the **SQLDataManager** class.

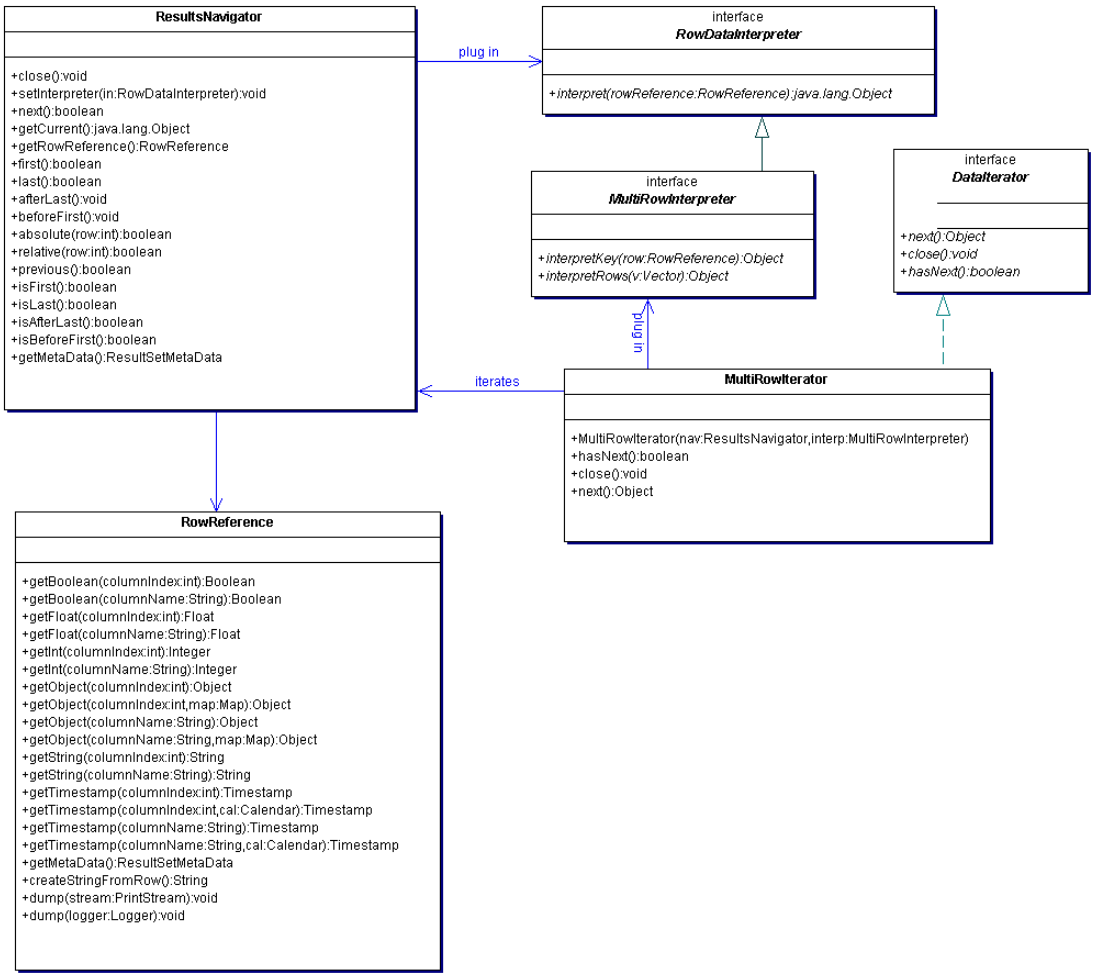
The **DBTypeConstants** is a static class containing static final definitions for constants which map the Sybase data types conventionally used in MGI to their corresponding JDBC types as defined in java.sql.Types. These mappings are used by the **Table** class (already defined in Section 4) when performing data validation during bcp processing.

5 Class Diagrams

5.1 SQLDataManager Classes



5.2 ResultsNavigator Classes



6 Executing Queries and Parsing Results

The classes which work together to perform query execution and results parsing are as follows: **SQLDataManager**, **ResultsNavigator**, and **RowReference**. When parsing data which logically extends across multiple contiguous rows, the classes that work together for query execution and parsing are **SQLDataManager**, **MultiRowIterator** and **MultiRowInterpreter**. Whereas, for single row data processing, the **ResultsNavigator** allows navigation through the query results in forward and backward directions, for multiple row processing, the **MultiRowIterator** only proceeds in a forward direction. The basic premise is that the **SQLDataManager** is used for executing the query. The **ResultsNavigator** is returned from the query and is used for navigating through the results. The **RowReference** is the object which is being pointed to within the **ResultsNavigator** (see example 5). Other patterns which involve the **SQLDataManager**, **MultiRowIterator**, and **MutiRowInterpreter** are given at the end of this section.

EXAMPLE 5. common use of a SQLDataManager, ResultsNavigator, RowReference

```
ResultsNavigator nav = sqlDataManager.executeQuery(some_sql);
while (nav.next())
{
  RowReference row = navigator.getRowReference();
  // process row
}
nav.close();
```

The **RowReference** is a reference to the current row. Whenever a call to next() is made on the **ResultsNavigator**, the **RowReference** is updated to the next row. So, alternative to example 5, another usage pattern is presented in example 6.

EXAMPLE 6. alternative to example 5

```
ResultsNavigator nav = sqlDataManager.executeQuery(some_sql);
RowReference row = nav.getRowReference();
while (nav.next())
{
  // process row
}
nav.close();
```

You can also have the **ResultsNavigator** return other types of objects other than **RowReference** objects by “plugging-in” a **RowDataInterpreter**. An interpreter object is used to take the values of the data in the current row and create a new java data object based on that data. For example, when querying the PRB_Source table, you may want the results of the query to return a series of MolecularSource objects. This can be done using the **RowDataInterpreter** interface. The interface is defined as follows:

```
public interface RowDataInterpreter
{
  public java.lang.Object interpret(RowReference row)
  throws DBException, InterpretException;
}
```

There is one method defined for this interface, which is the interpret() method. Note that this method takes a **RowReference** as input and returns a java object of type Object as output. The **ResultsNavigator** will call the interpret() method each time the getCurrent() method is executed. Since all java objects are derived from the Object class, the return type from getCurrent() can be any java type. The object would get cast to the appropriate data type within the user code.

To "plug-in" an interpreter, you would call the setInterpreter() method on the **ResultsNavigator** object (see Example 7).

EXAMPLE 7. using a RowDataInterpreter

```
// note that the classes in this example, InterpreterClass and
// IterpretedClass are devised for this example and do not pertain
// to any actual classes with the frameworks
```

```
SQLDataManager sqlman = new SQLDataManager();
InterpreterClass interpreterClass = new InterpreterClass();

// get a navigator and pass in the interpreter class
ResultsNavigator nav = sqlDataManager.executeQuery(some_sql);
nav.setInterpreter(interpreterClass);

// iterate through the results
// which is now returning some other object other than RowReference
while (nav.next())
{
  // the object returned is detremined by the interpreter class
  // and must be cast to that object
  InterpretedObject o = (InterpretedObject)rn.getCurrent();
}
```

A similar mechanism is available for creating a pluggable interpreter class for results which logically spans across multiple rows. The interface used for this is the **MultiRowInterpreter** class. The aim of this interface is to be able to support creating a java object based on contiguous rows from a query result set. So it will have to provide a way to interrogate any row to determine whether the row should be grouped with any previous rows. It is assumed that the query for which this interface is being applied contained an order by clause which controls the grouping of the logically similar rows in a contiguous manner. The interpretKey(**RowReference**) method is used for this purpose. It simply returns a value which would be consistent for every member of a groups set of rows. It is an extension of **RowDataInterpreter**. So it then supports the interpret(**RowReference**) method. This method returns a java object based on the given **RowReference**. Additionally, it will have to support a method which takes all the objects created for each row within the logical group and create a final compound object from them. The interpretRows(Vector) is used for this purpose. The vector is a vector of objects which were previously created from calls to the interpret(**RowReference**) methods. The interface is shown as follows:

```
public interface MultiRowInterpreter
extends RowDataInterpreter
{
  public Object interpretKey(RowReference row)
  throws DBException, InterpretException;

  public Object interpretRows(Vector v)
  throws InterpretException;
}
```

When processing rows in this way, so that a given data object is represented by more than one row of data, a new navigation class is required. It is the **MultiRowIterator**. The basic difference between this class and the **ResultsNavigator** is that this class can only navigate in a forward direction. Hence the term iterator is used in lieu of navigator in the class name. See example 8 for usage of this approach.

EXAMPLE 8. processing multiple rows in a query result set

```
// note that the classes in this example, InterpreterClass and
// IterpretedClass are devised for this example and do not pertain
// to any actual classes with the frameworks

SQLDataManager sqlman = new SQLDataManager();
```

```
InterpreterClass interpreterClass = new InterpreterClass();

// create a MultiRowIterator
ResultsNavigator nav = sqlDataManager.executeQuery(some_sql);
MultiRowIterator mri = new MultiRowIterator(nav, interpreterClass);

// iterate through the results
// which is now returning some other object based on multiple rows
while (mri.hasNext())
{
  // the object returned is detremined by the interpreter class
  // and must be cast to that object
  InterpretedObject o = (InterpretedObject)mri.next();
}
mri.close();
```

7 DBSchema Class

The **DBSchema** class is used for executing SQL statements which are part of the Data Definition Language (DDL). It is closely tied with the existing in-house DBSchema products as all DDL statements are pulled from the definitions defined within this product. Currently, there is support for creating and dropping tables, creating and dropping indexes, creating and dropping primary and foreign keys, and truncating tables and logs. The configuration variable DBSCHEMADIR is used to locate the installed version of the DBSchema product you will be referring to when executing DDL statements. You can set this parameter either through the configuration parameters or through setter methods on the **SQLDataManager**. To obtain an instance you would call the getDBSchema() method on the **SQLDataManager** (see Example 9).

EXAMPLE 9. Using DBShema

```
SQLDataManager sqlManager = new SQLDataManager();
DBSchema schema = sqlManager.getDBSchema();

schema.dropIndexes('RPCI_Cloneload');
```