Database Abstraction 1.1 Component Specification

Additions to this document from version 1.1 are in red, changes to existing documentation are in blue.

1. Design

The generated CustomResultSet closely resembles java.sql.ResultSet. This is intentionally done to facilitate familiarity. Method names, where applicable, are identical, and so are the underlying actions. Both classes have metadata objects. This component stores metadata information in the CustomResultSetMetaData object using a collection of Column objects. Data retrieval is also identical. All columns and rows are 1-indexed.

Additional methods for sorting have been added, with robust use of comparators.

Version 1.1 enhances the previous version with the ability to convert values on the fly to the type desired by a user. The new version supports a strict superset of functionality of the old version. That is, the API is completely backwards compatible. By default, the new version behaves exactly as the old version.

Where the previous version required explicit conversion, version 1.1 can convert between compatible types on the fly. The set of conversion that are accessible through this feature can be configured, similar to how the existing explicit conversion can be configured. A large set of default conversions covering all JDBC types are provided.

Version 1.1 seeks to maintain a similar feel to the version 1.0 API. Many methods accept null as a valid value, additionally, invalid values (like a column index that is too large or small) are handled by returning a default value, not throwing an exception. In most cases, this choice is forced by the need to remain consistent with the 1.0 API.

1.1 Design Patterns

- Delegation
- Factory
- Façade
- The Strategy pattern is used for the on-demand mapping functionality.

1.2 Industry Standards

- JDBC 2.0
- java.sql.ResultSet functionality for corresponding methods used in CustomResultSet
- java.sql.ResultSetMetaData functionality for corresponding methods used in CustomResultSetMetaData.

1.3 Required Algorithms

1.3.1 Version 1.1 Algorithms

1.3.1.1 Getting a value from the results set - getObject(int columnIndex, Class desiredType)

In version 1.1, 4 methods are tried (in order) to retrieve a value of the desired type:

- If the explicitly mapped value for the current row and given column is of desiredType, return it.
- If the original value in the current row and given column is of desiredType, return it.
- Try to convert the original value using the on-demand mapper. If this can be done, return the converted value.
- Try to convert the explicitly mapped value using the on-demand mapper. If this can be done, return the converted value.

If none of the above work, null is returned.

1.3.2 Sorting

The developer is free to use whatever sorting algorithm or paradigm he/she desires. When sorting based on column names, retrieve column numbers and call the equivalent method based on column numbers. It goes without saying that the integrity of rows must be preserved in a single- or multiple-column sorting.

1.3.3 Column Value getter methods

When getting column values based on column name, retrieve column number and call the equivalent method based on that column number.

1.3.4 Storing data in the CustomResultSet

Each field in a row is stored in a List which contains all values for the row, and all rows are stored in a List. In each of the row lists, all data values are stored wrapped in RowDataValue objects which maintain both the original and explicitly mapped values for the data. In version 1.0 the items in the lists of lists were the data values themselves.

Thus the CustomResultSet object has a List of Lists of RowDataValues. Because all this storage is internal (the List is never returned to the user), the developer is free to use any implementing class of List, and use its methods explicitly.

1.4 Component Class Overview

CustomResultSet:

The CustomResultSet class is similar to the JDBC ResultSet class in the public API that it exposes. This is intentional, in order to facility familiarity and ease of use of the class.

One note about this class is that exceptions do not follow current TopCoder standards. This is necessitated by the requirement that the API remain the same, so adding exceptions is not an option. For many methods, this means that instead of throwing an exception, either null or some default value (for primitive types) is returned.

Version 1.1 uses a different data structure to store the row data. In version 1.0, data was stored in a rows list, which each entry being a list that contained the column values for the row. In version 1.1, both the currently mapped value and the original JDBC value need to be preserved through the use of the RowDataValue class. By keeping track of the original value, the component allows for on demand conversions to be applied to both the original or mapped value.

Besides updating the sort and getXXX methods for this class (which mostly involves implementing the getObject(int, Class) method and directing the other methods to call it, little should be needed in version 1.1 over the previous version. This class is mutable.

DatabaseAbstractor:

The DatabaseAbstractor class permits a factory type pattern to be used for creation of CustomResultSets. This class is simply a storage facility for a mapper and an on-demand mapper, both of which it passes to any CustomResultSets that is creates. The only changes in version 1.1 to this class are the addition of the OnDemandMapper methods and fields. This class is mutable.

Mapper:

The Mapper class simply maintains a column name to Converter map of the mapping to apply when the remap method of the CustomResultSet class is called. This class is simply a wrapper around a HashMap, and should need no changes from version 1.0 to 1.1. This class is mutable.

Converter:

The Converter interface defines how explicit conversion is done. In order to use the version 1.0 explicit conversion capabilities of this component, a user must create a class implementing the Converter interface, register it with a Mapper and register the Mapper with a DatabaseAbstractor. No changes should be needed from version 1.0 to 1.1. This interface makes no demands on immutability or thread safety.

OnDemandMapper:

The OnDemandMapper class is the main focal point of the version 1.1 additions to this component. The OnDemandMapper is similar to the Mapper class, but contains a set of OnDemandConverters (and since they can be used on any column, no map from column names is needed). Additionally, it exposes a more type-safe set of methods than the Mapper class. This class is quite simply to

implement as all the methods either redirect to the contained set, or iterate over the set, calling the corresponding method on each element of the set. This class is mutable.

OnDemandConverter:

The OnDemandConverter interface is the counterpart of the Converter interface, updated to allow for version 1.1. on-demand conversion. In order to support this, the conversion method takes in an extra parameter, the desired returned type. This interface also exposes a method to determine whether a conversion can be done. This interface makes no thread-safety or mutability requirements.

CustomResultSetMetaData:

The CustomResultSetMetaData class is the counterpart of the JDBC ResultSetMetaData interface. It stores information about the columns in the CustomResultSet and exposes them through an interface very similar to the ResultSetMetaData interface. In addition to imitating the ResultSetMetaData interface, this class allows some pieces of data to for each column to be set in addition to being retrieved. This class is immutable.

Column:

The Column class is used by the CustomResultSetMetaData class to store information about each column in a CustomResultSet. This class is a simple data store for the several related values about a column, and no complicated logic is needed. All the methods are simple getters/setters. This class is mutable.

RowDataValue:

The RowDataValue class is used to store the two data values associated with a row and column – the original data value and the currently mapped value. This class is simply a container for these two variables and contains only simple getters and setters. This class is mutable

BigDecimalConverter:

The BigDecimalConverter class handles on demand conversions from the BigDecimal type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

BlobConverter:

The BlobConverter class handles on demand conversions from the Blob type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

BooleanConverter:

The BooleanConverter class handles on demand conversions from the Boolean type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

ByteConverter:

The ByteConverter class handles on demand conversions from the Byte type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

ByteArrayConverter:

The ByteArrayConverter class handles on demand conversions from byte arrays to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

DateConverter:

The DateConverter class handles on demand conversions from the Date type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

DoubleConverter:

The DoubleConverter class handles on demand conversions from the Double type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

FloatConverter:

The FloatConverter class handles on demand conversions from the Float type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

IntConverter:

The IntConverter class handles on demand conversions from the Integer type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

LongConverter:

The LongConverter class handles on demand conversions from the Long type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

ShortConverter:

The ShortConverter class handles on demand conversions from the Short type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

StringConverter:

The StringConverter class handles on demand conversions from the String type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

TimeConverter:

The TimeConverter class handles on demand conversions from the Time type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

ClobConverter:

The ClobConverter class handles on demand conversions from the Clob type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be one if/else block based on the desired conversion output type. This class is immutable.

TimestampConverter:

The TimestampConverter class handles on demand conversions from the Timestamp type to various other data types. It implements the OnDemandConverter interface and has no state. All the complications in this class lie in the convert and canConvert methods, each of which will essentially be

one if/else block based on the desired conversion output type. This class is immutable.

1.5 Component Exception Definitions

IllegalMappingException:

The IllegalMappingException is thrown when a Converter or OnDemandConverter is not able to make a conversion for an implementation defined reason.

This exception is thrown by implementations of the Converter and OnDemandConverter interfaces in the convert method, and propagates through the OnDemandMapper and CustomResultSet classes.

InvalidCursorStateException:

The InvalidCursorStateException is thrown by a CustomResultSet to indicate that it is not currently positioned on any row. This could be because it is positioned before the first row or after the last row.

This exception is thrown by CustomResultSet in the getXXX methods when data can not be retrieved because of the current row position of the CustomResultSet.

1.6 Thread Safety

This component is not thread safe. First, almost all of the classes are mutable, so this makes thread safety difficult. Second, version 1.0 was not thread safe, and adding thread safety to the existing API would be difficult. Third, this component is designed to mirror the JDBC ResultSet which is not thread-safe. Not having a thread-safe component will not come as a surprise to the primary users of this component.

To make the component thread safe is a matter of synchronizing the methods of the various mutable classes.

2. Environment Requirements

2.1 Environment

Java 1.4 is needed for compilation, testing, and use of this component.

2.2 TopCoder Software Components

None.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

None.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location

.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.sql.databaseabstraction
com.topcoder.util.sql.databaseabstraction.ondemandconversion

3.2 Configuration Parameters

None.

Parameter	Description	Values

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

4.2.1 Data Type Conversion

The java.sql.ResultSetMetaData and the CustomResultSetMetaData contain three key parameters: ***columnClassName***, ***columnType***, and ***columnTypeName***. These are important when dealing with data type conversions.

columnClassName refers to the fully-qualified java class name of this data type. The data in this column had been converted to objects of this java class. Any custom remapping will need to be able to remap from this java class.

columnType refers to the JDBC 2.0 SQL data type. Table 1 shows some of the common SQL to Java mappings in JDBC 2.0. These are not relevant to the conversion process.

columnTypeName refers to the database-specific types. The mapping of data types is based on these types, which the ResultSet contains in its metadata, and not JDBC 2.0 SQL data types, which are ubiquitous.

The user needs to implement Converter objects to handle mapping, and a Mapper object to coordinate remapping.

The Converter class accepts as input the Object to be converter, the column number, and the metadata for any column-specific information, such as the precision. This may be of relevance, in spite of the Object already being a Java Object.

Table 1 shows the default mappings, where applicable, of the three database types to JDBC 2.0 SQL types and Java types.

JDBC 2.0 SQL Type	**Java Type**	**SQL Server 2000 Type**	**Informix Type**	**Oracle Type**
CHAR	String	char, nchar, uniqueidentifier	`CHAR`(*n*)	CHAR
VARCHAR	String	Nvarchar, sql_variant, sysname	`VARCHAR`(*m*, *r*)	VARCHAR2
LONGVARCHAR	String	ntext, text	`TEXT`	LONG
NUMERIC	java.math.BigDecimal	numeric, numeric() identity	`DECIMAL`	NUMBER
DECIMAL	java.math.BigDecimal	decimal, decimal() identity, money, smallmoney	`DECIMAL`	NUMBER
BIT	boolean	bit,	Not supported	NUMBER
TINYINT	byte	tinyint, tinyint identity	`SMALLINT`	NUMBER
SMALLINT	short	smallint, smallint identity	`SMALLINT`	NUMBER
INTEGER	int	int, int identity	`INTEGER`	NUMBER
BIGINT	long	bigint, bigint identity	`INT8`	NUMBER
REAL	float	real	`SMALLFLOAT`	NUMBER
FLOAT	double	float	`SMALLFLOAT`	NUMBER

DOUBLE	double		FLOAT	NUMBER
BINARY	byte[]	binary, timestamp	BYTE	RAW
VARBINARY	byte[]		BYTE	RAW
LONGVARBINARY	byte[]	image	BYTE	LONGRAW
DATE	java.sql.Date		DATE	DATE
TIME	java.sql.Time		DATETIME	DATE
TIMESTAMP	java.sql.Timestamp	datetime, smalldatetime	DATETIME	DATE
BLOB	java.sql.Blob			BLOB
CLOB	java.sql.Clob			CLOB
STRUCT	java.sql.Struct			user-defined object
REF	java.sql.Ref			user-defined reference
ARRAY	java.sql.Array			user-defined collection

4.2.2 Sorting

When sorting with an array of comparators, the component can accept null comparators, upon which it will attempt to sort the column using the natural ordering of the column's elements.

After sorting, the cursor is set before the first row (i.e. cursor=0). See section 4.1.3, below.

Multiple row sorts will be done in descending importance. Therefore the first element in the array will be sorted first, then the next element, etc. Note that column sorts will not change the order of the previous sorts, except when it is ambiguous.

4.2.3 Cursor state

The user should not guess where the cursor is, because the position is not always guaranteed. When retrieving data, the use of the relative next() method is ok. When performing sorting, the cursor will be reset to zero. Once the remapping process is complete, the cursor is reset to zero.

The robust navigation methods are used for exact cursor positioning. Recall that rows are 1-based.

4.2.4 *Getter methods*

The most generic way to access data is through the getObject methods. However, there are getter methods for most Java datatypes that the user can use in lieu of type casting.

4.2.5 *Method summary*

CustomResultSet

- sortXXX methods
  These methods are use for sorting. Sorting can be done my column name or index, with single or multiple columns to be sorted, and for either descending or ascending sorts.

- navigation methods
  absolute, beforeLast, afterFirst, first, isAfterLast, isBeforeFirst, isFirst, isLast, next, previous, relative.

  These methods will act in the same fashion as they do in java.sql.ResultSet. That is, as the name of method suggest. Recall that row numbers are 1-based.

- getXXX methods
  These methods will act in the same fashion as they do in java.sql.ResultSet. That is, to retrieve data of the give type.

- other methods
  remap – used for remapping
  findColumn – used to find column index given a column name

4.3 Demo

```
// Database configuration(In Oracle):
// table: ABSTRACTION_TABLE
// Column      name      type         can-be-null
//             ID        NUMBER(10)   no
//             NAME      VARCHAR2(10) no
//             AGE       LONG         yes
//             BLOB_T    BLOB         yes
//             CLOB_T    CLOB         yes
//             DATE_T    DATE         yes
//
// Values in table ABSTRACTION_TABLE
//   50 TOPCODER 10 EMPTY_BLOB EMPTY_CLOB 2006-06-18
//   100 TEST 20 NOT-EMPTY NOT-EMPTY 2006-06-19
// NOT-EMPTY above means that it contains value

// Create database connection.
Connection conn = UnitTestHelper.getDatabaseConnection();
// Retrieve ResultSet for Database.
ResultSet rs = UnitTestHelper.getResultSet(conn);
// Create DatabaseAbstractor
```

```
DatabaseAbstractor dbAbstractor = new DatabaseAbstractor();

// Initialize mapper
HashMap map = new HashMap();
// Class MockConverterBoolean which converts BigDecimal
// value to Boolean(true) and throw IllegalMappingException
// if it is not BigDecimal value.
class MockConverterBoolean implements Converter {
    // simple implementation.
    /**
     * Convert value to Boolean(true), if it is a BigDecimal.
     *
     * @return the converted result (may be null)
     * @param value the original object value
     * @param column the column the value came from
     * @param metaData metadata for the result set
     * @throws IllegalMappingException when value is not
     *         BigDecimal
      */
     public Object convert(Object value, int column,
                           CustomResultSetMetaData metaData)
                 throws IllegalMappingException {
           if (value.getClass().equals(BigDecimal.class)) {
                  return new Boolean(true);
           } else {
                  throw new
                  IllegalMappingException("illegalException.");
            }
      }
 }
 // map instance of Converter to the column type named "number"
 map.put("number", new MockConverterBoolean());
 // create mapper with one converter which aims to convert number
  // column to boolean.
 Mapper mapper = new Mapper(map);

 // set mapper of DatabaseAbstractor.
 dbAbstractor.setMapper(mapper);

 // Get CustomResultSet.
 CustomResultSet crs = dbAbstractor.convertResultSet(rs);
 while (crs.next()) {
       // Get original value of CustomResultValue.
       crs.getBigDecimal("ID");

       // Get mapped value of CustomResultSet.
       crs.getBoolean(1);

       // Get not exist value will result in ClassCastException.
       // crs.getByte(1);

  }

 // Here demos improved function of version 1.1
  DatabaseAbstractor dbAbstractor2 = new DatabaseAbstractor();

 // set On-Demand Mapper of DatabaseAbstractor.
```

```
    dbAbstractor2.setOnDemandMapper(OnDemandMapper.createDefaultOnDema
    ndMapper());

    // Set Mapper.
    dbAbstractor2.setMapper(mapper);

    // Get CustomResultSet.
    CustomResultSet crs2 = dbAbstractor2.convertResultSet(rs);
    while (crs2.next()) {
          // Get original value of CustomResultValue.
          crs.getBigDecimal("ID");

          // Get mapped value of CustomResultSet.
          crs.getBoolean(1);

          // Get not exist value in original value or mapped value,
          //but was converted from BigDecimal to Byte.
          crs.getByte("ID");
          // Get not exist value in original value or mapped value,
          //but was converted from BigDecimal to String.
          crs.getString(1);
    }

    // Demo the sort method of CustomResultSet.
    crs2.sortAscending("ID");

    // Retrieve value of sorted CustomResultSet.
    while (crs2.next()) {
           crs.getBigDecimal("ID");
    }

    // Close connection.
    conn.close();
}
```

5. Future Enhancements

Provide additional on demand conversions.