Vaadin SLQContainer 0.8 User Manual

Teppo Kurki
© 2010 Vaadin Ltd.

Table of Contents

1. Introduction

Vaadin SQLContainer is a Vaadin container implementation that allows easy and customizable access to data stored in various SQL-speaking databases (see appendix A for details). SQLContainer supports two types of database access. Using `TableQuery`, the pre-made query generators will enable fetching, updating and inserting data directly from the container into a database table - automatically, whereas `FreeformQuery` allows the developer to use their own, probably more complex query for fetching data and their own optional implementations for writing, filtering and sorting support - item and property handling as well as lazy loading will still be handled automatically.

In addition to the customizable database connection options, SQLContainer also extends the Vaadin container interface to implement a bit more advanced and more database oriented filtering rules. Finally, the add-on also offers connection pool implementations for JDBC connection pooling and JEE connection pooling, as well as integrated transaction support; auto-commit mode is also provided.

The purpose of this manual is to briefly explain the architecture and some of the inner workings of SQLContainer. It will also give the readers some examples on how to use SQLContainer in their own applications. The requirements, limitations and further development ideas are also discussed.

2. Architecture

The architecture of SQLContainer is relatively simple, and it is described in detail in this section.

`SQLContainer` is the class implementing the Vaadin container interfaces and providing access to most of the functionality of this add-on. The standard Vaadin Properties and Items have been extended by the `ColumnProperty` and `RowItem` classes. Item IDs are represented by `RowId` and `TemporaryRowId` classes. The RowId class is built based on the primary key columns of the connected database table or query result.

In the connection package the `JDBCConnectionPool` interface defines the requirements for a connection pool implementation. Two implementations of this interface are provided: `SimpleJDBCConnectionPool` provides a simple´yet very usable implementation to pool and access JDBC connections. `J2EEConnectionPool` provides means to access J2EE DataSources.

The query package contains the `QueryDelegate` interface which defines everything the SQLContainer needs to enable reading and writing data to and from a database. As discussed earlier, two implementations of this interface are provided: `TableQuery` for automatic read-write support for a database table, and `FreeformQuery` for customizing the query, sorting, filtering and writing; this is done by implementing relevant methods of the `FreeformQueryDelegate` interface.

The query package also contains `Filter` and `OrderBy` classes which have been written to provide an alternative to the standard Vaadin container filtering and make sorting non-String properties a bit more user friendly.

Finally, the generator package contains a SQLGenerator interface which defines the kind of queries that are required by the `TableQuery` class. The provided implementations include support for HSQLDB, MySQL, PostgreSQL (`DefaultSQLGenerator`), Oracle (`OracleGenerator`) and Microsoft SQL Server (`MSSQLGenerator`). A new or modified implementation may be provided to gain compatibility with older versions or other database servers.

A class diagram of SQLContainer and its supporting classes as well as connections to Vaadin interfaces is provided in figure 1. Note that not every method is shown in the diagram due to space constraints. For further detail, refer to the SQLContainer API documentation.

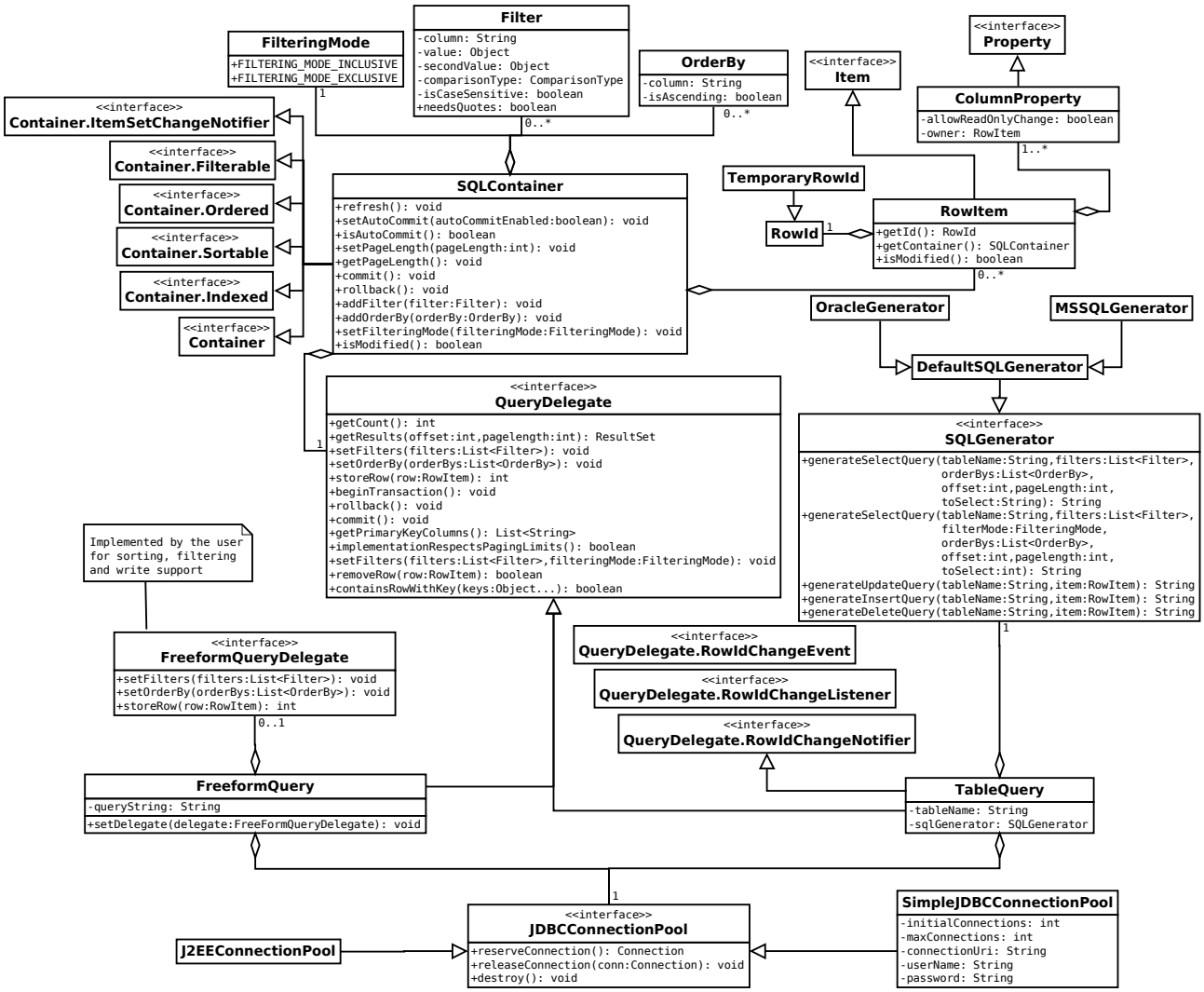

Figure 1. Detailed class diagram of SQLContainer and its supporting classes

3. Getting started with SQLContainer

Gettting development going with the SQLContainer is easy and quite straight-forward. The purpose of this chapter is to describe how to create the required resources and how to fetch data from and write data to a database table attached to the container.

3.1. Creating a connection pool

First, we need to create a connection pool to allow the SQLContainer to connect to a database. Here we will use the `SimpleJDBCConnectionPool`, which is a basic implementation of connection pooling with JDBC data sources. In the following code we create a connection pool that uses the HSQLDB driver together with an in-memory database. The initial amount of connections is 2 and the maximum amount is set at 5. Note that the database driver, connection url, username and password parameters will vary depending on the database you are using.

```
JDBCConnectionPool connectionPool = new SimpleJDBCConnectionPool(
                        "org.hsqldb.jdbc.JDBCDriver",
                        "jdbc:hsqldb:mem:sqlcontainer", "SA", "", 2, 5);
```

3.2. Creating the TableQuery query delegate

After the connection pool is created, we'll need a query delegate for the SQLContainer. The simplest way to create one is by using the built-in TableQuery class. The TableQuery delegate provides access to a defined database table and supports reading and writing data out-of-the-box. We create the TableQuery with the following statement:

```
TableQuery tq = new TableQuery("tablename", connectionPool);
```

If we need to enable the write support, we must set a version column to the TableQuery as well. The version column is an integer or timestamp typed column which will either be incremented or set to the current time on each modification of the row. TableQuery assumes that the database will take care of updating the version column; it just makes sure the column value is correct before updating a row. The following code will set the version column:

```
tq.setVersionColumn("OPTLOCK");
```

3.3. Creating the container

Finally we may create the container itself. This is as simple as stating:

```
SQLContainer container = new SQLContainer(tq);
```

After this statement the SQLContainer is connected to the table `tablename` and is ready to use for example as a data source for a Vaadin Table or a Vaadin Form.

4. Using the SQLContainer

4.1. Filtering and Sorting

Filtering and sorting the items contained in an SQLContainer is by design always performed in the database. In practice this means that whenever the filtering or sorting rules are modified, at least some amount of database communication will take place (the minimum is to fetch the updated row count using the new filtering/sorting rules).

4.1.1. Filtering

Filtering can be performed either using the Vaadin-provided means implemented from `Container.Filterable` using the following method, where propertyId means column name in the SQLContainer context. More information on the standard filtering can be found in Book of Vaadin.

```
public void addContainerFilter(Object propertyId, String filterString,
            boolean ignoreCase, boolean onlyMatchPrefix)
```

Note! Using the standard method only allows filtering on String-typed properties. Numeric types, dates etc. must be filtered using the Filter class (see below).

In addition to the standard method, it is also possible to directly add a `Filter` to the container via the `addFilter(Filter filter)` method. This enables the developer to take advantage of a few more features, including:

- More comparison methods via Filter.ComparisonType
  - EQUALS, GREATER, LESS, GREATER_OR_EQUAL, LESS_OR_EQUAL
  - STARTS_WITH, ENDS_WITH, CONTAINS
  - BETWEEN
- Filtering of numeric and other non-String types
- Two-valued filtering via Filter.setSecondValue(Object secondValue)
- Implicit setting for need of quoting via setNeedsQuotes(boolean needsQuotes)

Removing the filtering rules is also done via the standard Vaadin methods:

```
public void removeContainerFilters(Object propertyId)
public void removeAllContainerFilters()
```

These methods will remove filters added with either `addFilter` or `addContainerFilter` method.

4.1.2. Filtering mode

Currentle the SQLContainer has limited support for two filtering modes. The modes are defined in the `FilteringMode` enum which is located in the query package. The two modes are called exclusive and inclusive. Exlusive mode means that in the generated query all the filtering rules will be joined with an OR. Inclusivu mode means that AND will be used.

The default filtering mode is inclusive (AND) filtering.

4.1.3. Sorting

Sorting can be performed either using the Vaadin-provided means implemented from `Container.Sortable` using the following method, where the propertyIds again refer to column names. More information on the standard filtering can be found in Book of Vaadin.

```
public void sort(Object[] propertyId, boolean[] ascending)
```

In addition to the standard method, it is also possible to directly add an `OrderBy` to the container via the `addOrderBy(OrderBy orderBy)` method. This enables the developer to insert sorters one by one without providing the whole array of them at once.

Sorting rules can be cleared by calling the `sort` method with null or an empty array as the first argument.

4.2. Editing

Editing the items (RowItems) of SQLContainer can be done similarly to editing the items of any Vaadin container. ColumnProperties of a RowItem will automatically notify SQLContainer to make sure that changes to the items are recorded and will be applied to the database immediately or on commit, depending on the state of the autocommit mode.

4.2.1. Adding items

Adding items to the SQLContainer can only be done via the `addItem()` method. This method will create a new Item based on the connected database table column properties. The new item will either be buffered by the container or commited to the database through the querydelegate depending on whether the auto commit mode (see 4.2.3.) has been enabled.

When an item is added to the container it is impossible to precisely know what the primary keys of the row will be, or will the row insertion succeed at all. This is why the SQLContainer will assign an instance of `TemporaryRowId` as a RowId for the new item. See 4.2.2. on how to fetch the actual key after the row insertion has succeeded.

Note! Currently the addItem() method will return a temporary row ID for the added item even if auto commit mode is enabled.

4.2.2. Fetching generated row keys

Since it is a common need to fetch the generated key of a row right after insertion, a listener/notifier has been added into the `QueryDelegate` interface. Currently only the `TableQuery` class implements the `RowIdChangeNotifier` interface, and thus can notify interested objects of changed row IDs. The events fill be fired after TableQuery.commit() has finished; this method is called by SQLContainer when necessary.

To receive updates on the row IDs, you might use the following code (assuming `container` is an instance of SQLContainer):

```
app.getDbHelp().getCityContainer().addListener(
    new QueryDelegate.RowIdChangeListener() {
        public void rowIdChange(RowIdChangeEvent event) {
            System.err.println("Old ID is: " + event.getOldRowId());
            System.err.println("New ID is: " + event.getNewRowId());
        }
    });
```

4.2.3. Version column requirement

If you are using the TableQuery class as the query delegate to the SQLContainer and need to enable write support, there is an enforced requirement of specifying a version column name to the TableQuery instance. The column name can be set to the TableQuery using the following statement:

```
tq.setVersionColumn("OPTLOCK");
```

The version column is preferrably an integer or timestamp typed column in the table that is attached to the TableQuery. This column will be used for optimistic locking; before a row modification the TableQuery will check before that the version column value is the same as it was when the data was read into the container. This should ensure that no one has modified the row inbetween the current user's reads and writes.

Note! TableQuery assumes that the database will take care of updating the version column by either using an actual VERSION column (if supported by the database in question) or by a trigger or a similar mechanism.

If you are certain that you do not need optimistic locking but do want to enable write support, you may point the version column to e.g. a primary key column of the table.

4.2.3. Autocommit mode

SQLContainer is by default in transaction mode, which means that actions that edit, add or remove items are recorded internally by the container. These actions can be either committed to the database by calling `commit()` or discarded by calling `rollback()`.

The container can also be set to auto commit mode. When this mode is enabled, all changes will be committed to the database immediately. To enable or disable the auto commit mode call the following method:

```
public void setAutoCommit(boolean autoCommitEnabled)
```

It is recommended to leave the auto commit mode disabled, since it ensures that the changes can be rolled back if any problems are noticed within the container items. Using the auto commit mode will also lead to failure in item addition if the database table contains non-nullable columns.

4.2.4. Modified state

When used in the transaction mode it may be useful to determine whether the contents of the SQLContainer have been modified or not. For this purpose the container provides an `isModified()` method which will tell the state of the container to the developer. This method will return true if any items have been added to or removed from the container, as well as if any value of an existing item has been modified.

Additionally, each RowItem and each ColumnProperty have isModified() methods to allow for a more detailed view over the modification status. Do note that the modification statuses of RowItems and ColumnProperties only depend on whether or not the actual Property values have been modified. That is, they do not reflect situations where the whole RowItem has been marked for removal or has just been added to the container.

4.3. Caching, paging and refreshing

To decrease the amount of queries made to the database, SQLContainer uses internal caching for database contents. The caching is implemented with a size-limited LinkedHashMap containing a mapping from RowIds to RowItems. Typically developers do not need to modify caching options,

although some fine-tuning can be done if required.

4.3.1. Container size

The SQLContainer keeps continuously checking the amount of rows in the connected database table in order to detect external addition or removal of rows. By default, the table row count is assumed to remain valid for 10 seconds. This value can be altered from code; class SQLContainer, field `sizeValidMilliSeconds`.

If the size validity time has expired, the row count will be automatically updated on:

- A call to `getItemIds()` method
- A call to `size()` method
- Some calls to `indexOfId(Object itemId)` method
- A call to `firstItemId()` method
- When the container is fetching a set of rows to the item cache (lazy loading)

4.3.2. Page length and cache size

The page length of the SQLContainer dictates the amount of rows fetched from the database in one query. The default value is 100, and it can be modified with the `setPageLength` method. To avoid constant queries it is recommended to set the page length value to at least 5 times the amount of rows displayed in a Vaadin Table; obviously this is also dependent on the cache ratio set for the Table component.

The size of the internal item cache of the SQLContainer is calculated by multiplying the page lenght with the cache ratio set for the container. The cache ratio can only be set from the code, and the default value for it is 2. Hence with the default page length of 100 the internal cache size becomes 200 items. This should be enough even for larger Tables while ensuring that no huge amounts of memory will be used on the cache.

4.3.3. Refreshing the container

Normally the SQLContainer will handle refreshing automatically when required. However there may be situations where an implicit refresh is needed, e.g. to make sure that the version column is up-to-date prior to opening the item for editing in a form. For this purpose a `refresh()` method is provided. This method simply clears all caches, resets the current item fetching offset and sets the container size dirty. Any item-related call after this will inevitably result into row count and item cache update.

Note that a call to the refresh method will not affect or reset the following properties of the container:

- The QueryDelegate of the container
- Autocommit mode
- Page length
- Filters
- Sorting

5. Using FreeformQuery and FreeformQueryDelegate

In most cases the provided TableQuery will be enough to allow a developer to gain effortless access to an SQL data source. However there may arise situations when a more complex query with e.g. joins is needed. Or perhaps you need to redefine how the writing or filtering should be done. The FreeformQuery query delegate is provided for this exact purpose. Out of the box the FreeformQuery supports read-only access to a database, but it can be extended to allow writing also.

5.1. Getting started

Getting started with the FreeformQuery may be done as shown in the following. The connection pool initialization is similar to the TableQuery example so it is omitted here. Note that the name(s) of the primary key column(s) must be provided to the FreeformQuery manually. This is required because depending on the query the result set may or may not contain data about primary key columns. In this example there is one primary key column with a name 'ID'.

```
FreeformQuery query = new FreeformQuery("SELECT * FROM SAMPLE",
                                        connectionPool, "ID");
SQLContainer container = new SQLContainer(query);
```

5.2. Limitations

While this looks just as easy as with the TableQuery, do note that there are some important caveats here. Using FreeformQuery like this (without providing FreeformQueryDelegate implementation) it can only be used as a read-only window to the resultset of the query. Additionally filtering, sorting and lazy loading features will not be supported, and the row count will be fetched in quite an inefficient manner. Bearing these limitations in mind, it becomes quite obvious that the developer is in reality meant to implement the FreeformQueryDelegate interface.

5.3. Creating your own FreeformQueryDelegate

To create your own delegate for FreeformQuery you must implement some or all of the methods from the FreeformQueryDelegate interface, depending on which ones your use case requires. The interface contains eight methods which are shown below. For more detailed requirements, see the JavaDoc documentation of the interface.

```
/* Read-only queries */
public String getCountQuery()
public String getQueryString(int offset, int limit)
public String getContainsRowQueryString(Object... keys)

/* Filtering and sorting */
public void setFilters(List<Filter> filters)
public void setFilters(List<Filter> filters, FilteringMode filteringMode)
public void setOrderBy(List<OrderBy> orderBys)

/* Write support */
public int storeRow(Connection conn, RowItem row)
public boolean removeRow(Connection conn, RowItem row)
```

A simple demo implementation of this interface can be found in the SQLContainer package, more specifically in the class `com.vaadin.addon.sqlcontainer.demo.DemoFreeformQueryDelegate`.

6. Non-implemented methods of Vaadin container interfaces

Due to the database connection inherent to the SQLContainer, some of the methods from the container interfaces of Vaadin can not (or would not make sense to) be implemented. These methods are listed below, and they will throw an `UnsupportedOperationException` on invocation.

```
public boolean addContainerProperty(Object propertyId, Class<?> type,
                                    Object defaultValue)
public boolean removeContainerProperty(Object propertyId)
public Item addItem(Object itemId)
public Object addItemAt(int index)
public Item addItemAt(int index, Object newItemId)
public Object addItemAfter(Object previousItemId)
public Item addItemAfter(Object previousItemId, Object newItemId)
```

Additionally, the following methods of the Item interface are not supported in the RowItem class:

```
public boolean addItemProperty(Object id, Property property)
public boolean removeItemProperty(Object id)
```

6.1. About the `getItemIds()` method

To properly implement the Vaadin Container interface, a `getItemIds()` method has been implented in the SQLContainer. By definition this method returns a collection of all the item IDs present in the container. What this means in the SQLContainer case is that the container has to query the database for the primary key columns of all the rows present in the connected database table. It is obvious that this could potentially lead to fetching tens or even hundreds of thousands of rows in an effort to satisfy the method caller. This will effectively kill the lazy loading properties of SQLContainer and therefore the following warning is expressed here:

*It is highly recommended **not** to call the `getitemIds()` method, unless it is known that in the use case in question the item ID set will always be of reasonable size.*

7. Appendices

A. Supported databases

The following databases are supported by SQLContainer and TableQuery classes by default:

- HSQLDB [1.8 or newer]
- MySQL [5.1 or newer]
- PostgreSQL [8.4 or newer]
- Oracle Database [10g or newer]
- Microsoft SQL Server [2005 or newer]

B. Known issues and limitations of SQLContainer

At this point, there are still some known issues and limitations affecting the use of SQLContainer in certain situations. The **known issues** and brief explanations are listed below:

- The `addItem()` method will return a temporary row ID even if auto commit mode is enabled.
- Some **SQL data types** do not have write support when using TableQuery:
  - All binary types
  - All custom types
  - CLOB (if not converted automatically to a String by the JDBC driver in use)
- When using **Oracle or MS SQL** database, the column name '`rownum`' can not be used as a column name in a table connected to SQLContainer.
  - This limitation exists because the databases in question do not support limit/offset clauses required for paging. Instead, a generated column named '`rownum`' is used to implement paging support.

The **permanent limitations** are listed below. These can not or most probably will not be fixed in future versions of SQLContainer.

- The `getItemIds()` method is very inefficient - avoid calling it unless absolutely required!
- When using **FreeformQuery** without providing a FreeformQueryDelegate, the row count query is very inefficient - avoid using FreeformQuery without implementing at least the count query properly.
- When using **FreeformQuery** without providing a FreeformQueryDelegate, writing, sorting and filtering will not be supported..
- When using **Oracle** database most or all of the numeric types are converted to `java.math.BigDecimal` by the Oracle JDBC Driver.
  - This is a feature of how Oracle DB and the Oracle JDBC Driver handles data types.

C. Planned features

- Create a method of easily joining two SQLContainers with TableQueries in cases where e.g. foreign key is used in one of the tables.
  - This would greatly reduce mapping code which currently has to be written to the applications using the SQLContainer
- Provide support for all SQL data types
- Provide means for the developer to specify their own database type to Java type mapping