JIDE Data Grids Developer Guide

Contents

Purpose of This Document

JIDE Data Grids is built on top of the JIDE *Grids* to provide additional support for the database related features. This developer guide is designed for developers who want to learn how to use *JIDE Data Grids* in their applications.

JIDE Data Grids heavily depends on features and components provided by *JIDE Grids*. If you never used *JIDE Grids* before, we strongly recommend you read *JIDE Grids Developer Guide* first or at least refer to it while reading this developer guide.

Overview

JIDE Grids brought many advanced features to JTable. Among them, the two mostly used features are filtering and sorting. Both the filtering and sorting feature are implemented inside the table models, which are *FilterableTableModel* and *SortableTableModel* respectively. However, all database systems can handle filtering (using WHERE statement) and sorting (using ORDER BY statement). When the database table is huge, it is preferred to let the database handles the filtering and sorting so that the data doesn't need to be downloaded to the client side. Thus, to let the database handling filtering and sorting is one of the most important features in *JIDE Data Grids*. Since Hibernate is one of the most widely used Java relational persistence layer, we also want to support Hibernate in *JIDE Data Grids*.

Tables in database could be huge. If a table has millions of rows and we load all the rows into memory, it will consume a huge amount of memory and will of course be very slow. As a matter of facts, user will probably only look at 10 or maybe 20 rows depending on the viewport size, there is no need to download all the rows locally. That's the reason we introduced the paging feature in *JIDE Data Grids*. In fact, the paging mechanism is not limited to the database or to the *JTable*. It is generic enough to support any non-database *JTable* or even *JList*.

Database Support

There are two table models related to the database. The first one is the *ResultSetTableModel* that deals with a JDBC *ResultSet*. The second one is the *DatabaseTableModel*. It uses *ResultSetTableModel* internally but it supports the database filtering and sorting.

ResultSetTableModel

ResultSetTableModel uses an existing *ResultSet* and converts it into a *TableModel* so that *JTable* or tables in *JIDE Grids* can use it. If you have a *ResultSet* available, you can use this table model to display it. *ResultSetTableModel* will not load all the records at once but loading it when *TableModel#getValueAt* is called. In the other word, we will only load the first several rows that are currently visible in the current viewport. While users page up/down or scroll up/down in the table, *ResultSetTableModel* will then load those rows just before they are displayed. We will also cache all the rows that are ever loaded so that we don't need to ask the database for the same rows again. Of course, if the database table is changed, you can tell the *ResultSetTableModel* to invalidate the cache so that next time it will ask the database for the rows again.

To create a ResultSetTableModel, you call

```
ResultSetTableModel  tableModel = new ResultSetTableModel(resultSet);
```

Or

```
ResultSetTableModel  tableModel = new ResultSetTableModel(resultSet, recordCount);
```

The reason we need a second parameter for the *recordCount* is because it is very slow to find out how many records in a *ResultSet*. The only way we knew is to use

```
resultSet.last();
int recordCount = resultSet.getRow();
```

It works but very slow if the table is large. So if you happen to know the record count for the *ResultSet*, you can pass it in to the constructor so that we don't need to use the code above to figure it out.

Since the *ResultSet* contains the metadata for the columns, most table model methods such as *getColumnCount*, *getColumnName* and *getColumnClass* are implemented based on the metadata in the *ResultSet*.

The *getValueAt* is implemented as well. We added an extra layer of cache to cache the *ResultSet*. So the first thing we did in *getValueAt* is to look at the cache and check if we have the record. If not cached, we will jump to the record using *ResultSet*'s *absolute(recordIndex)* method, load the record to the cache and return the value for the specific column. Note that not all *ResultSet*s support scrollable cursor, which means it will throw exception when the *absolute* method is called. In this case, we will have to keep calling *next()* method from the cursor position until it reaches the specified row index. It won't be an issue if user page down page by page. If user uses the vertical scroll bar to scroll down, it will take time to fetch all the records till reach the page. So please keep this in mind and use a scrollable *ResultSet* if possible.

The *ResultSetTableModel* doesn't know how to handle filtering and sorting. It still needs *FilterableTableModel* and *SortableTableModel* to do it which means it will not leverage the database filtering and sorting. Here comes the *DatabaseTableModel*.

DatabaseTableModel

We started to implement this class with a test case of a *SortableTable* using *ResultSetTableModel* with 200k rows using Derby JDBC. To sort these 200k rows, it took about 12 seconds. To filter this table using a filter similar to the SQL IN statement, it took about 12 seconds as well. What happen is that most of 12 seconds is spent on loading the records from the database. Our goal here is to how much we improve in these two areas with the *DatabaseTableModel*. Of course, the code to use this *DatabaseTableModel* needs to be as simple as possible.

Let's start with the constructors of the *DatabaseTableModel*.

```
public DatabaseTableModel(Connection connection, String fromStatement) throws SQLException
public DatabaseTableModel(Connection connection, String selectStatement, String fromStatement) throws
SQLException
```

If you ever used JDBC before, you should be familiar with the *Connection*. You can easily get a *Connection* instance using *DriverManager.getConnection* method as long as you have the database url and user name/password if any. See below for an example to get a *Connection* to Derby database. Without getting into too much detail, you can refer to the document for the database you are using to figure out how to do it.

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
Connection connection = DriverManager.getConnection("jdbc:derby:MyDatabase;create=true"); // MyDatabase is
the database name
```

The next thing is to create a DatabaseTableModel. Let's say we have a "sales" table in the database. All you need to do is

```
DatabaseTableModel tableModel = new DatabaseTableModel(connection, "*", " sales");
```

The second parameter here is the select statement. The "*" means select all the columns from the table "sales". In the other word, all columns will be included in the *DatabaseTableModel* as the table columns. The third parameter is the FROM statement. Putting it together, it is the same as the SQL statement "SELECT * FROM sales".

If you only want to include some columns to this table model, you could use the following code to only select two columns.

```
DatabaseTableModel tableModel = new DatabaseTableModel(connection, "ProductName, ProductSales", " sales");
```

Or if you want to join several tables in the database, you can use list tables in the FROM statement just like in SQL. See below for an example to join both "sales" table and the "items" table.

```
DatabaseTableModel tableModel = new DatabaseTableModel(connection, "*", " sales, items");
```

You can even include a JOIN statement as part of the FROM statement.

```
DatabaseTableModel tableModel = new DatabaseTableModel(connection, "*", "sales LEFT JOIN items on
sales.ProductName = items.ProductName");
```

Again, please refer to any SQL documentation to find out the details of the SQL syntax.

Sorting

Even though creating a *DatabaseTableModel* is simple, it has all the features built in. *DatabaseTableModel* implements *ISortableTableModel* which means it is sortable. If you set *DatabaseTableModel* to a *SortableTable*, clicking on the table header can sort the column. Multiple column sorting is supported as well. Please note, the sorting is happening in the database. Because it doesn't have to load the records, it is much faster than sorting using Java as in our JIDE Grids' *SortableTableModel*. For the same test case with 200k rows, the sorting only took 0.7 seconds. If you know user will sort certain columns more often than other columns, you can index those columns using the database indexing feature to future improve the speed.

Filtering

DatabaseTableModel implements *IFilterableTableModel*. It has *addFilter* method. The Filter actually has the logic to filter values. However we introduced an interface called *SqlFilterSupport*. As long as the filter implements this interface, *DatabaseTableModel* can understands it and knows how to translate the filter to the SQL WHERE statement.

See below from a simple comparison of the built-in Filters and the corresponding WHERE statement.

Filter	SQL WHERE	Description
EqualFilter	=	Equal
NotEqualFilter	<>	Not equal
GreaterThanFilter	>	Greater than
LessThanFilter	<	Less than

GreaterOrEqualFilter	>=	Greater than or equal
LessOrEqualFilter	<=	Less than or equal
BetweenFilter	BETWEEN	Between an inclusive range
NotBetweenFilter	NOT BETWEEN	Not between an inclusive range
LikeFilter	LIKE	Search for a pattern. % is the wildcard for any chars and _ is the wildcard for one char.
NotLikeFilter	NOT LIKE	Search for opposite of a pattern
InFilter	IN	If you know the exact value you want to return for at least one of the columns
NotInFilter	NOT IN	If you know the exact values you want to exclude

See below for an example of adding an *InFilter* for the column 3 of a *DatabaseTableModel*.

```
databaseTableModel.addFilter(3, new InFilter(new Object[]{45, 46, 77}));
databaseTableModel.setFiltersApplied(true);
```

Because the filtering is happening in the database, the speed is a lot faster. For the same test case with 200k rows, the filtering only took 0.7 seconds, the time spent on running one SQL query.[1]

Caching and Performance

You may wonder how much memory or how many seconds it will use to create a new *DatabaseTableModel* that has a lot of rows. The answer is it is instant and uses almost no memory for most JDBC drivers[2]. The reason is obviously. It is just one SQL call to the database. No records are loaded. The record for a row is loaded only before the table displays that row. We will only load the first several rows that are currently visible. While users page up/down or scroll up/down in the table, *DatabaseTableModel* will then load those rows just before they are displayed. We also cache all the rows that are ever loaded so that we don't need to ask the database for the same rows again. Of course, if the database table is changed, you can tell the *DatabaseTableModel* to invalidate the cache so that next time it will ask the database for the rows again.

[1] It is not the FilterableTableModel's filtering algorithm or the SortableTableModel's sorting algorithm are slower than the algorithms used in the database. Our test indicates, once the records are loaded in memory, the performance is of FilterableTableModel and SortableTableModel is actually 10% to 20% faster. The slowness comes from loading the records. The DatabaseTableModel will not load all the records while doing filtering and sorting. That's why it is faster.

[2] The statement is actually depending on the JDBC driver. We tested with four JDBC drivers – Derby, Hsql, MySql and PostgreSql. It shows for Derby and Hsql don't load the whole record when the executeQuery is called. However for PostgreSql and MySql, it will load the all the records. Please read the link below for more information.

http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-implementation-notes.html

Paging Support

Even though *DatabaseTableModel* provides the caching feature to prevent using too much memory when loading a large database table, it is still not easy to use scroll bar to navigate through several thousands or millions of rows. That's why we introduce the paging support.

PageNavigationSupport

PageNavigationSupport is an interface to support paging on a table model or a list model. It has methods to get/set the page size, the current page index, the total row count and methods to navigate the pages. It also supports *PageNavigationListener* which will tell you when the current page index changes, the page size changes or total page count changes.

```
public interface PageNavigationSupport {
   int getPageSize();
   void setPageSize(int pageSize);
   int getCurrentPage();
   void setCurrentPage(int page);
   int getPageCount();
   int getTotalRecordCount();
   void setTotalRecordCount(int totalRecordCount);
   void nextPage();
   void previousPage();
   void firstPage();
   void lastPage();
   void addPageNavigationListener(PageNavigationListener l);
   void removePageNavigationListener(PageNavigationListener l);
   PageNavigationListener[] getPageNavigationListeners();
   void firePageNavigationEvent(Object source, int id, int oldValue, int newValue);
}
```

The methods defined on *PageNavigationSupport* can be divided into three categories. The first category is for the pagination, for example, how many records in the model (*setTotalRecordCount*) and what's the page size (*setPageSize*). The *PageNavigationSupport* knows how to paginate the model once it has the information. The second category is the navigation part. You can use the *setCurrentPage* method to jump to any page or use *firstPage/lastPage/nextPage/previousPage* to go back and forth among the pages. The third category is the support for the *PageNavigationListener*. It will tell you what happen so that you can load/unload pages in/from memory or display the current page index on UI etc.

AbstractPageTableModel and DefaultPageTableModel

AbstractPageTableModel provides the abstract implementation for the interface *PageNavigationSupport*. It implements all the methods on *PageNavigationSupport* but introduces two abstract methods – *pageCountChanged* and *pageIndexChanged* to let its subclasses to implement. Most likely, you don't need to use this abstract implementation directly in your code but use one of its concrete subclasses.

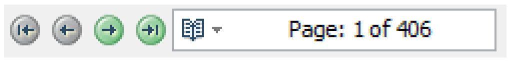
DefaultPageTableModel is the concrete implementation. *DefaultPageTableModel* uses the table model wrapper design pattern as we used in JIDE Grids to implement *SortableTableModel* and *FilterableTableModel*. It can wrap any *TableModel* and paginate it. The current page is nothing but a row index mapping maps from the visual row index to the actual row index in the actual *TableModel*. When the current page index changes, it will recreate a new row index mapping that maps to another group of the rows in the actual *TableModel*.

We also introduced *AdvancePageTableModel* to support sorting and filtering in a paged model scenario. We will cover it in the Hibernate section.

PageNavigationBar

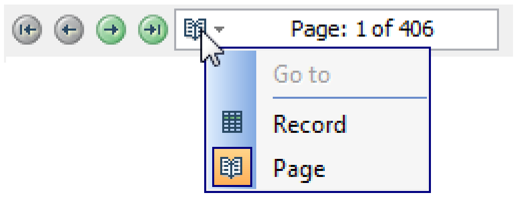
To make it easy to use the *PageNavigationSupport*, we created *PageNavigationBar* to provide a user interface to do the navigation.

See below for a screenshot.

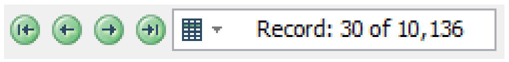


The first button is to jump to the first page. The next button is to jump to the previous page. In the middle, it is the page index field. It shows the current page index and the total page count. Then there is the next page button and the last page button.

Note the page index field can also be used as the record index field. User can click on the page icon to show a context menu to switch to the record navigation mode.



In the record navigation mode, the display in the field will be the current record index and the total record count.



You can type in the page index in the field directly and press ENTER to jump to the page. All known hot keys are supported in the page/record index field. PageUp and PageDown is the same as clicking on the previous page button and the next page button, respectively. UP and DOWN key will go to the previous record or the next record. CTRL-PageUp will go the first record in the current page. CTRL-PageDown will go to the last record in the current page. CTRL-Home and CTRL-End is the same as clicking on the first page button and the last page button, respectively. The Left/Right/Home/End keys are not used as they are taken by the text field. To avoid the confusion, the hotkeys work the same way in the page navigation mode and the record navigation mode.

Creating a *PageNavigationBar* is very easy. You just need to call

```
PageNavigationBar bar = new PageNavigationBar(table);
```

The table can be any JTable or one of JIDE tables. Although *PageNavigationBar* only works with *PageNavigationSupport*, you don't have to create a table model that supports *PageNavigationSupport* beforehand because *PageNavigationBar* will create a

DefaultPageTableModel for you if it doesn't see a model in the table that supports *PageNavigationSupport*. As I mentioned earlier that *DefaultPageTableModel* works with any table model, thus *PageNavigationBar* can work with any tables.

In fact, since we mostly discuss about tables, we tend to ignore the fact that *PageNavigationBar* supports JList as well. *PageNavigationSupport* is a general interface that doesn't tie to *TableModel*. There are *AbstractPageListModel* and *DefaultPageListModel* as well to support *PageNavigationSupport* for *JList*. Thus *PageNavigationBar* can be used on *JList* like this.

```
JList list = …;
PageNavigationBar bar = new PageNavigationBar(list);
```

Hibernate Support

Although using JDBC and SQL to access the database is still the most efficient way, it is easy to get tedious and difficult to maintain over the time. That's why people introduced relational database mapping to map from the database tables and records to Java objects. Obviously, manipulating a Java object is much simpler than changing a record in a database table. There are several projects in this area and Hibernate is probably the most popular one.

There are a bunch of setup tasks you need to do before you can start to use Hibernate. You can refer to http://www.hibernate.org for more information. Once you set it up, the first thing you need to do in your code is

```
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
```

The *Session* class to Hibernate is the same as the *Connection* class to JDBC. In order to get all the objects from a table called "sales", you can do

```
Criteria criteria = session.createCriteria(Sales.class);
List list = criteria.list();
```

Note that we use Sales.class as an example. In order to work with Hibernate, we will have to create a class called *Sales* and maps to the "sales" table. The list is a java.util.List that contains all the Sales objects. You can consider it as POJO which has getters and setters for properties. Once you have that, you can use the following code to create a table model.

```
HibernateTableModel  tableModel = HibernateTableModel(session, list, Sales.class);
```

The table model created using the code above is just a regular table model that has all the Objects. So if you set it to a SortableTable or wrap into a DefaultPageTableModel to paginate it, it won't benefit anything from Hibernate. If the list is small, you can certainly use it this way. However if the list is huge, we suggest you to use *HibernatePageTableModel* instead of *HibernateTableModel*.

HibernatePageTableModel extends *AdvancePageTableModel* which supports filtering, sorting and paginating. *AdvancePageTableModel* implements all the features and only leaves the *refreshData* method to its subclasses to implement. The *refreshData* method looks like this.

```
void refreshData(SortItemSupport sortItem, FilterItemSupport filterItem, int page, int pageSize);
```

The *refreshData* method takes the sort orders, the filters, the page size and the current page etc information. All the information together is enough to determine the data for the current page. Hibernate provides a class called *Criteria*. *Criteria* has *addOrder(Order)* which can be created from *SortItemSupport*. It also has *add(Criterion)* method which can be created from *FilterItemSupport*. It also has *setFirstResult* and *setMaxResult* methods which can be used to fetch a certain page. At the end, criteria.list() to get the List of *Objects* for the current page and create a *HibernateTableModel* for it.

What's Next

JIDE Data Grids is still in beta. It is the first product in JIDE that deals with the database. There are still many features that we want to add to this product but haven't got a chance to do so.

1. Editable table model: allow adding/removing/modifying records directly from the *DatabaseTableModel* and *HibernateTableModel*. *PageNavigationBar* can be enhanced to add more buttons for this purpose.
2. Use a special row header to indicate the cursor and newly inserted row position and in the *RecordSet*.
3. Use the information from the *ResultSetMetaData* to format the column using ObjectConverter and CellRenderer. Use special icon to indicate a column is autoIncrement, searchable, nullable etc.
4. Support grouping using the database.

If you have any feedbacks and suggestions, please feel free to email us or post on the forum.