WorkingWithLargeTables

Components t:dataModel and t:dataScroller work together nicely to allow a user to "page" through a set of a few dozen to a few hundred records. However the implementation does assume that the entire set of available records are in memory and wrapped up inside a ListDataModel or ArrayDataModel.

When the available dataset is quite large, and the application can have many users, this can lead to memory usage problems.

This page contains discussions on how to handle this scenario.

On-demand loading

A custom DataModel can be used to allow data to be loaded "on demand".

First, a class needs to be defined which your "business methods" (eg EJBs) can use to pass "pages" of data back to the UI. This class needs to be defined in your project, as the "business" level shouldn't be extending MyFaces classes:

```
package example;

import java.util.List;

/**
 * A simple class that represents a "page" of data out of a longer set, ie
 * a list of objects together with info to indicate the starting row and
 * the full size of the dataset. EJBs can return instances of this type
 * when returning subsets of available data.
 */
public class DataPage<T> {

  private int datasetSize;
  private int startRow;
  private List<T> data;

  /**
   * Create an object representing a sublist of a dataset.
   *
   * @param datasetSize is the total number of matching rows
   * available.
   *
   * @param startRow is the index within the complete dataset
   * of the first element in the data list.
   *
   * @param data is a list of consecutive objects from the
   * dataset.
   */
  public DataPage(int datasetSize, int startRow, List<T> data) {
    this.datasetSize = datasetSize;
    this.startRow = startRow;
    this.data = data;
  }

  /**
   * Return the number of items in the full dataset.
   */
  public int getDatasetSize() {
    return datasetSize;
  }

  /**
   * Return the offset within the full dataset of the first
   * element in the list held by this object.
   */
  public int getStartRow() {
    return startRow;
  }

  /**
   * Return the list of objects held by this object, which
   * is a continuous subset of the full dataset.
   */
  public List<T> getData() {
    return data;
  }
}
```

Now a custom DataModel can use this DataPage stuff. Again, it is recommended that you copy this code into your project and change the package name appropriately. This class can't go in the MyFaces libraries as it depends on DataPage, and as noted above, DataPage is accessed by your business level code so it really can't be in the MyFaces libs:

```
package example;

import javax.faces.model.DataModel;

import example.DataPage;
```

```
/**
 * A special type of JSF DataModel to allow a datatable and datascroller
 * to page through a large set of data without having to hold the entire
 * set of data in memory at once.
 * <p>
 * Any time a managed bean wants to avoid holding an entire dataset,
 * the managed bean should declare an inner class which extends this
 * class and implements the fetchData method. This method is called
 * as needed when the table requires data that isn't available in the
 * current data page held by this object.
 * <p>
 * This does require the managed bean (and in general the business
 * method that the managed bean uses) to provide the data wrapped in
 * a DataPage object that provides info on the full size of the dataset.
 */
public abstract class PagedListDataModel<T> extends DataModel {

	private int pageSize;
	private int rowIndex;
	private DataPage<T> page;

	private int lastStartRow = -1;
	private DataPage<T> lastPage;

	/*
	 * Create a datamodel that pages through the data showing the specified
	 * number of rows on each page.
	 */
	public PagedListDataModel(int pageSize) {
		super();
		this.pageSize = pageSize;
		this.rowIndex = -1;
		this.page = null;
	}

	/**
	 * Not used in this class; data is fetched via a callback to the fetchData
	 * method rather than by explicitly assigning a list.
	 */
	@Override
	public void setWrappedData(Object o) {
		throw new UnsupportedOperationException("setWrappedData");
	}

	@Override
	public int getRowIndex() {
		return rowIndex;
	}

	/**
	 * Specify what the "current row" within the dataset is. Note that the
	 * UIData component will repeatedly call this method followed by getRowData
	 * to obtain the objects to render in the table.
	 */
	@Override
	public void setRowIndex(int index) {
		rowIndex = index;
	}

	/**
	 * Return the total number of rows of data available (not just the number of
	 * rows in the current page!).
	 */
	@Override
	public int getRowCount() {
		return getPage().getDatasetSize();
	}

	/**
	 * Return a DataPage object; if one is not currently available then fetch
	 * one. Note that this doesn't ensure that the datapage returned includes
```

```
         * the current rowIndex row; see getRowData.
         */
        private DataPage<T> getPage() {
                if (page != null)
                        return page;

                int rowIndex = getRowIndex();
                int startRow = rowIndex;
                if (rowIndex == -1) {
                        // even when no row is selected, we still need a page
                        // object so that we know the amount of data available.
                        startRow = 0;
                }

                // invoke method on enclosing class
                page = suggestFetchPage(startRow, pageSize);
                return page;
        }

        /**
         * Return the object corresponding to the current rowIndex. If the DataPage
         * object currently cached doesn't include that index then fetchPage is
         * called to retrieve the appropriate page.
         */
        @Override
        public Object getRowData() {
                if (rowIndex < 0) {
                        throw new IllegalArgumentException("Invalid rowIndex for PagedListDataModel; not within
page");
                }

                // ensure page exists; if rowIndex is beyond dataset size, then
                // we should still get back a DataPage object with the dataset size
                // in it...
                if (page == null) {
                        page = suggestFetchPage(rowIndex, pageSize);
                }

                // Check if rowIndex is equal to startRow,
                // useful for dynamic sorting on pages

                if (rowIndex == page.getStartRow()) {
                        page = suggestFetchPage(rowIndex, pageSize);
                }

                int datasetSize = page.getDatasetSize();
                int startRow = page.getStartRow();
                int nRows = page.getData().size();
                int endRow = startRow + nRows;

                if (rowIndex >= datasetSize) {
                        throw new IllegalArgumentException("Invalid rowIndex");
                }

                if (rowIndex < startRow) {
                        page = suggestFetchPage(rowIndex, pageSize);
                        startRow = page.getStartRow();
                }
                else if (rowIndex >= endRow) {
                        page = suggestFetchPage(rowIndex, pageSize);
                        startRow = page.getStartRow();
                }

                return page.getData().get(rowIndex - startRow);
        }

        @Override
        public Object getWrappedData() {
                return page.getData();
        }
```

```
        /**
         * Return true if the rowIndex value is currently set to a value that
         * matches some element in the dataset. Note that it may match a row that is
         * not in the currently cached DataPage; if so then when getRowData is
         * called the required DataPage will be fetched by calling fetchData.
         */
        @Override
        public boolean isRowAvailable() {
                DataPage<T> page = getPage();
                if (page == null)
                        return false;

                int rowIndex = getRowIndex();
                if (rowIndex < 0) {
                        return false;
                }
                else if (rowIndex >= page.getDatasetSize()) {
                        return false;
                }
                else {
                        return true;
                }
        }

        /**
         * the jsf framework can be funky.  It could ask for the same start row multiple times within the
         * same cycle.   Therefore, we cache the results for the startRow and make sure that if the framework
         * asks for them again in very short order that we simply returned the cached value.   What the
         * extended class could do in fetchPage could be costly, so we make sure it is called only when needed!
         *
         * @param startRow
         * @param pageSize
         * @return
         */
        public DataPage<T> suggestFetchPage(int startRow, int pageSize) {
                if(this.lastStartRow == startRow) {
                        return this.lastPage;
                }

                this.lastStartRow = startRow;
                this.lastPage = fetchPage(startRow, pageSize);
                return this.lastPage;
        }

        /**
         * Method which must be implemented in cooperation with the managed bean
         * class to fetch data on demand.
         */
        public abstract DataPage<T> fetchPage(int startRow, int pageSize);
}
```

Finally, the managed bean needs to provide a simple inner class that provides the fetchPage implementation:

```
{{{
  public SomeManagedBean {
    ....


    private DataPage<SomeRowObject> getDataPage(int startRow, int pageSize) {
      // access database here, or call EJB to do so
    }

    public DataModel getDataModel() {
        if (dataModel == null) {
            dataModel = new LocalDataModel(getRowsPerPage());
        }

        return dataModel;
    }
```

```
    private class LocalDataModel extends PagedListDataModel {
        public LocalDataModel(int pageSize) {
            super(pageSize);
        }

        public DataPage<SomeRowObject> fetchPage(int startRow, int pageSize) {
            // call enclosing managed bean method to fetch the data
            return getDataPage(startRow, pageSize);
        }
    }
```

The jsp pages are then trivial; by default the t:dataScroller will update the t:dataTable's "first" property, and that's all that is needed because when the table asks the custom DataModel for the necessary rows callbacks to fetchPage will be made which will fetch exactly the data required. No event listeners, action methods, or anything else is required as glue.

Other approaches

In an email thread on this topic, an alternative was decribed:
http://sebcsaba.fw.hu/scrollablelist.html

So was this:
http://www.mail-archive.com/users%40myfaces.apache.org/msg12597.html

And another example with the DataModel approach
http://www.jroller.com/page/cagataycivici?entry=jsf_datatable_with_custom_paging

And a sortable and scrollable demo with 1.1.1 jars included:
myfaces-cars.zip

Paging large data sets with a LazyList
http://www.ilikespam.com/blog/paging-large-data-sets-with-a-lazylist

Paged datatable with a GenericDataTableHandler:
http://jroller.com/page/jurgenlust?anchor=genericcrudhandler_for_jsf