Java generic paged lazy List with JSF/JPA example implementation

by Benny Bottema - Wednesday, July 22, 2009

http://www.bennybottema.com/2009/07/22/java-generic-paged-lazy-list-with-jsfjpa-example-implementation/

November 4, 2012 – PageList has now been released on GitHub!

Here's a list implementation I came up with to enable true paged data fetching completely transparent to any List user. It works independent of persistence layers, such as JPA implementations etc and it can be used by anything that only needs a List, such as a .

- PagedList source files on GitHub

Contents

Lazy loading with the PagedList

Originally I made the PagedList because I needed paged data fetching using JMS queues for a JSF Rich datatable and datascroller; a common problem in JSF but less commonly solved, even less documented and less still solved in an elegant way. Most people are able to solve this problem by using an OpenSessionInView antipattern (keeping Hibernate-oid session open until the view requests the lazy loaded data) but since I'm using JMS queue, this was not an option. Alternative solutions include using contrived JSF specific extended data table models and what not; I decided to circumvent the entire JSF to JMS queue integration by providing my data in a lazy paging list. It works beautifully.

- PagedList.java

It is different from Apache's LazyList (and many other implementations) in that it relies on an external dataprovider to provide the dataset's total size, pagesize and data provision itself. For example, using my

lazy list version the *size()* method actually returns the total size of data potentially available, not just physically available in the list at the present. This way the list works completely transparent to its users. Also, there is no simple factory being passed in that creates dummy objects or something: actual remote data is being fetched in pages by a data provider. Pages being fetched are independent of each other; when requesting page 5, page 1 through 4 are not needed or fetched.

The PagedList works in combo with a *PagedDataProvider* interface that external dataproviders (some dao for example) could implement. So it has no dependencies whatsoever on some specific implementation. The paged data provider provides the total size of the dataset, the page size and pages of the dataset itself when needed.

Example implementation for JSF and JPA dao

Here's an example implementation which ties a JSF Richfaces datatable to a JPA dao using a PagedList. I cut out the service layer, view handlers/controllers for sake of simplicity.

```
...
```

```
/**
 * JPA dao which acts as paged data provider for the paged list.
 *
 * Relies on two JPA named queries specified on the Entity being fetch
ed.
 *
 * @author Benny Bottema
 * @see PagedDataProvider
 */
public class AppleJPADao implements AppleDao, PagedDataProvider<AppleD
to, AppleQueryParameters> {

    // entitymanager stuff omitted for example (real world: injected b
y Spring)

    private static final int PAGESIZE = 50;

    /**
     * This method is used to provide a lazy pagin List of apples used
 in the rich datatable.
     * The query parameters will be reused in subsequent provide() cal
ls from the list.
     *
     * @see AppleDao#getAllApples(String, String)
```

```
     */
    public List<AppleDto> getAllApples(String type, String quality) {
        // optional query parameters: can be null
        AppleQueryParameters params = new AppleQueryParameters();
        params.setAppleType(type);
        params.setAppleQuality(quality);
        return new PagedList<AppleDto, AppleQueryParameters>(this, par
ams);
    }

    /**
     * Called by the PagedList instance. The query criteria is the sam
e instance as initially provided in getAllApples().
     *
     * @see PagedDataProvider#provide(int, Object).
     */
    public List<AppleDto> provide(int page, AppleQueryParameters query
Criteria) {
        Query query = entityManager.createNamedQuery("Apple.findAll");
        query.setParameter("appleType", queryCriteria.getAppleType());
        query.setParameter("appleQuality", queryCriteria.getAppleQuali
ty());
        // JPA's paging mechanism
        query.setFirstResult(PAGESIZE * page);
        query.setMaxResults(PAGESIZE);
        return (List<AppleDto>) query.getResultList();
    }

    /**
     * Returns the dataset's total size.
     *
     * @see PagedDataProvider#getDataSize()
     */
    public int getDataSize() {
        Query countQuery = entityManager.createNamedQuery("Apple.count
");
        return ((Long) countQuery.getSingleResult()).intValue();
    }

    /**
     * @see PagedDataProvider#getPageSize()
     */
    public int getPageSize() {
        return PAGESIZE;
    }
}
```

About *TDto*

This is a generic type for the *PagedList* and *PagedDataProvider*. To the paging list is doesn't matter what objects it contains / returns and, being a *List* itself, the paging list uses this information to type itself as a *List<Dto>* instance.

```
// AppleDto example, with entity annotations removed
public class AppleDto {
    private String type;
    private String quality;

    // setters and getters omitted
}
```

In the above example, for brevity purposes, the AppleDto object returned by the DAO is directly used in in the paged list. This can be dangerous if the entity is actually a JPA lazy loading proxy: a service class could be sitting in between to convert the JPA entity into a POJO version ready for use in the JSF view. The paged list doesn't care where the dto's come from, since a data provider transparently provides them.

About *TQueryParameters*

This also is a generic type for the *PagedList* and *PagedDataProvider*. It can be anything and is only meant to be used when initially creating the list and when providing new data. This way, the data provider can be sure to use the same criteria over and over again when returning a new page of data. Otherwise a page from a completely different dataset might be returned or the order might have been switched around.

```
public class AppleQueryParameters {
    private String appleType;
    private String appleQuality;

    // setters and getters omitted
}
```

This example is very basic, but you can ofcourse include which column to sort on, which direction etc. Here's a real world example where the search criteria are explicitely defined, because the are

communicated over JMS:

```
public abstract class SearchCriteria {
    private int pageNumber; // the DAO on the other end of the queue w
ill need this
    private int pageSize; // as well as this
    private String sortBy; // column name or a flag
    private String direction; // ascending / descending

    // setters and getters omitted
}

public class CustomerSearchCriteria extends SearchCriteria {
    // as with the AppleQueryParameters, customer specific fields will
 be here
}
```

Stale data detection

There is an issue with lazy loading in general: synchronizing the lazy loading list with the remote dataset to avoid becoming stale when changes happen to the dataset.

To solve this, the PagedList provides three levels of stale data detection, or rather, three different resolutions, as each option looks for differences more precise and often. You can provide this option by passing in a *DataIntegrityCheckingMode* flag into the constructor:

```
// never check, assume the dataset never changes
new PagedList<AppleDto, AppleQueryParameters>(this, params, DataIntegr
ityCheckingMode.OFF);

// check each time we fetch a new page of data
new PagedList<AppleDto, AppleQueryParameters>(this, params, DataIntegr
ityCheckingMode.ON_FETCH_PAGE);

// check each time an item is being 'get()' from the list
new PagedList<AppleDto, AppleQueryParameters>(this, params, DataIntegr
ityCheckingMode.ON_GET);
```

However, there is a limit to this solution. As long as the dataset’s size doesn’t change, we won’t know if the data itself has changed or not. Example: when the dataset changes but the same number of rows is

being returned from the database, the size of the list is not being updated and the *get()* method may return the wrong item.
