XMLFormViewAbstractionDraft

About

revision 0.0.1 **this is work in progress, the author appreciates any constructive critique or ideas**

Initial Author: Jakob Praher

I created this proposal, as a result of a larger web application I did (and do) with the Cocoon XMLForm components.

Described is a cleaner specification of a framework I have deployed successfully.

- It's all about the further abstraction of XMLForm View navigation, without the need to subclass the AbstractXMLFormAction.

In my experience, view navigation can be abstracted to a much larger degree, enabling more rapidly built webapps.

Unfortunately I started working on the idea, before XMLForm flow extensions were available, but I think it would make sense with cocoon-flow too.

Description

Just a rough documentation for now.

Please look at the components to get an idea of the architecture.

It would be also interesting to use jxpath to do transitions, based on the state of the bean,

this would lead to much more powerful transitons.

Components

FormViews

FormViews abstracts all the views of one form. FormViews is the state machine that knows, for a given input (the command, or action) the next or previous view.

As of this writing, FormViews is implemented using constants, but I am working on a version that uses xml format, which can be configured in the sitemap, like the schematron validation.

The FormViews xml document could easily be generated from an existing xml form definition.

- The views are defined in xf:view.
- The commands are defined with

```
public interface FormViews {

  /* well known actions */

  String CMD_PREV = &amp;quot;prev&amp;quot;,
         CMD_NEXT = &amp;quot;next&amp;quot;,
         CMD_CANCEL = &amp;quot;cancel&amp;quot;
  ;




  /**
   *
   *
   * this is the state transition function, as in traditional
   * automata theory.
   * f: View x Input -&amp;gt; View
   *
   *
   * @param  view the current view
   * @param  command the input
   * @return the next view, if everything works fine.
   *         so the xmlforms logic will ask the next view only
   *         if there is no error.
   */
  String getView( String view, String command );



  String[] getViewChain( ) ;
  String[] getViewChain( String[] commands );


  boolean isView( String view ) ;
  boolean isCommand( String cmd ) ;


  // -------------------------------------------------------
  boolean isLastView( String view );
  boolean isLastButOneView( String view );
  boolean isFirst( String view );
}
```

the xml-version would look sth like this

```
&lt;xfv:views xmlns:xfv=&quot;anuri&quot;&gt;

  &lt;xfv:view name=&quot;start&quot;&gt;
     &lt;!--
     * the default next is the next sibling in the xml form
     * the default prev is the prev sibling in the xml form
     * cancel is also defined, for all views
     --&gt;
     &lt;xfv:transition cmd=&quot;acmd&quot;  &gt;edit-main&lt;/xfv:transition&gt;

     &lt;!-- this would also be a very interesting way --&gt;
     &lt;xfv:transition cmd=&quot;acmd&quot; &gt;
       &lt;xfv:rule context=&quot;/&quot;&gt;
          &lt;!-- better use the jstl format for ifs here ... --&gt;
          &lt;xfv:when test=&quot;string(inputType)='2'&quot;&gt;super-next-view&lt;/xfv:when&gt;
       &lt;/xfv:rule&gt;
     &lt;/xfv:transition&gt;


  &lt;/xfv:view&gt;

&lt;/xfv:views&gt;
```

FormLifecycle

About the Lifecycle

Now in order to get in touch with your business logic, you need to handle certain events in the life cycle of a form application.

I abstracted the following states:

- starting
- view switching
- completing
- completed
- aborted

Notes:

the -ing form denotes an event that is ongoing - it is not finished yet. Some -ing events can be vetoed, vetoing is done by throwing an Exception of a certain type. The framework will check this type of exception and handle it appropriately. Another form of veto in view switching and completing can be done by introducing new input violations, in which case the error is returned to the user like when having an input problem.

Also interesting is the notion of a PreConditionException, that is an error which is worser than an Input Violation, but which when thrown, will result in a *nice* error information getting displayed (using the null-return feature of actions).

The Lifecycle Listener

The FormLifecycleListener is a simple interface, that get called when the above mentioned lifecycle stages occur.

By using this abstraction, form logic writing gets much cleaner, easier and less prune to changes , since the view transitions are handled by the base framework.

Every FormLifecycleListener event gets a FormInvocationContext object passed, which is represents all the state of the current Form plus request. There is room for changing this.

The FormLifecycleListener looks like:

```
public interface FormLifecycleListener {


   /**
    * this is just an informatino that the user, for instance has
    * cancelled the form application, it is not possible to step out of
    * this, but can be useful, for business logic maintenance for instance.
    *
    * @param ctx the Form Ctx, all the data needed ...
    */
   public void formLifecycleAborted( FormInvocationContext ctx );

   /**
    * this is vetoable, the form is starting, a new Model has been initialized,
    * configuration through business logic may be performed in this stage.
    */
   public void formLifecycleStarting( FormInvocationContext ctx )
     throws PreConditionException, FormLifecycleVetoException, FormLifecycleException
   ;

   public void formLifecycleViewSwitching( FormInvocationContext, String fromView, String toView )
   throws PreConditionException, FormLifecycleVetoException, FormLifecycleException
   ;

   /**
    * gets called when the form is in the completing stage. Errors and vetos are possible.
    * And should be raised in such a condition ...
    */
   public void formLifecycleCompleting( FormInvocationContext ctx )
   throws PreConditionException, FormLifecycleVetoException, FormLifecycleException
   ;

   /**
    * the form has completed successfully, is called before the form model is cleared
    * and the session data gets retained. Useful for transaction management and such ...
    */
   public void formLifecycleCompleted( FormInvocationContext ctx )
   ;

}
```

TODO

- an overview of a sitemap using this framework
- a sample application walkthrough

Further Enhancements

On thing that is very important enhancement point is the notion of dependent data, where the depency is known at load time. Currently, binding static data (using the xf:itemset) is great, but when the itemset depends on other form controls, you have to dig in and write the javascript layer.