Cocoon Forms MODS Editor

1.0beta

Table of Contents

Introduction 1
General Configuration 1
Description of functionality 2
Data handling 2
Design patterns 3
Top-level element 3
Subrepeater 5
Extending the demo: localization 6
Functions 7
Sample Localization File 7
Extending the demo: saving the modified record 8
Saving to a file 8
POSTing to an HTTP interface 9
Reporting Results 9
Extending the demo: authentication 9
Putting MODS Editor into production 10
Desiderata 10

Introduction

MODS Editor is a Cocoon package that generates a web form for editing MODS records. It uses the Cocoon Forms framework [http://cocoon.apache.org/2.1/userdocs/basics/index.html]. It attempts to provide all the elements in the MODS 3.2 [http://www.loc.gov/standards/mods/v3/mods-3-2.xsd]. spec (except `mods:extension`), and it provides a very flexible mechanism for customizing a form to meet the needs of a particular project.

General Configuration

Steps to get MODS Editor working in your instance of Cocoon [http://cocoon.apache.org/] 2.1.10:

1. Unzip the package to some convenient location
2. Add the location of the package to Cocoon's `mount-table.xml` with an entry like this:

```
<mount uri-prefix="modseditor" src="file://path/to/modseditor-directory/"/>
```

3. Access MODS Editor at `http://your.server:8080/cocoon/modseditor/` (adjusting server, port and Cocoon servlet path as necessary)

You should now be able to edit the included sample MODS records. The demo interface allows you to

do any editing you like, but when you submit the changed record will simply be displayed back to you; it will not be saved. (See below for how to add the ability to save records.)

Note

MODS Editor hasn't yet been tested with the latest Cocoon version 2.1.11 or with the 2.2 branch.

Note

Cocoon must be set up to use UTF-8; see the Cocoon wiki [http://wiki.apache.org/cocoon/RequestParameterEncoding] for instructions on how to do this. Steps 1, 2, and 5 in that document are handled within MODS Editor; but you may need to take care of 3 and 4, depending on your servlet container. I use Tomcat 5.5, and found that step 3 is taken care of by editing Cocoon's web.xml to change the form-encoding init-param to UTF-8, and step 4 wasn't needed, but your mileage may vary. If you don't do this, the form will function but non-ISO-8859-1 characters will be garbled.

Description of functionality

The functionality of the interface is fairly simple, but some notes may be useful.

- The Cocoon Forms tabbing feature is used, so each top-level element has a tab in the left column. You can only see one type of top-level element on the screen at a time.
- Some custom javascript (in `mods_template.jx`) updates the counts beside each top-level element in the tabs menu when items are added or deleted from the relevant repeater. Note that the javascript is triggered by an onclick rather than by the successful completion of the operation
- Every top-level MODS element is assumed to be repeatable
- Graphic labels appear to the left of elements in repeaters (both top-level and subrepeaters); these are handled as background images in the CSS stylesheet. They are generated by ImageMagick, which is driven by an Ant script in the images directory; a different colour scheme or different text labels could be implemented by editing that build.xml.
- The out-of-the-box form contains all MODS elements; it can be configured for a particular project using the localization feature (see below)
- Within the recordInfo top-level element, the recordContentSource element will automatically be populated with the user id of the current user (drawn from the Cocoon session), and the recordChangeDate will be populated with the current time. These operations happen at save time and are governed by the `fb:javascript` widgets "lastModified" and "lastModifiedBy" in `mods_bind.xml`. Either or both of these changes can be prevented by suppressing the "lastModified" or "lastModifiedBy" fields in the recordInfo element, using localization.
- The "common-controls-class" in `mods_model.xml` and `mods_template.jx` is used by every repeater, and provides the row of controls (+, -, up arrow, down arrow) that appear in each row. jx:if conditions in `mods_template.jx` control its appearance, suppressing inapplicable controls (e.g. no up arrow for the first row). This class does not appear in `mods_bind.xml` because its widgets do not bind to any values in the MODS record.

Data handling

MODS records are provided to the form by the "mods-data/**" pipeline in sitemap.xmap. Because Cocoon Forms does not handle namespaces properly, this pipeline includes an xsl transformation

(strip-namespaces.xsl) to replace the MODS namespace with a prefix: so "mods:subject" becomes "mods_subject". All of the bindings in `mods_bind.xml` depend on this prefix. The xsl also makes a couple of minor changes (e.g. converting "yes/no" values to "true/false" so they can be handled by boolean widgets). All of these changes are reversed in the success pipeline (which handles the uploaded record when the user submits the form), by a series of transformations. In order to get a clean xml file, the "mods_" prefix is converted to the "mods:" namespace, which is then removed from individual elements and provided in the root element as the default namespace.

Design patterns

MODS Editor imposes rigorous design patterns on similar structures in the model, binding, and template files, especially with regard to repeaters.

Top-level element

A simple top-level element looks like this:

- note the naming conventions: for the "genre" element, the class is named "genre-class" and the repeater is named "genres" (this applies even to tableOfContents, whose repeater is "tableOfContentss"). In the template, the div immediately under ft:repeater-rows has the class "row toplevel genre". The "add-to" action has the name "add-to-genres".
- the top-level element div uses H4; the individual rows use H5
- In the template, note the dynamic elements, using jx variables and functions:
  - in the main div, the class contains "`empty-block-${repeater.getSize() == 0}`". This will give "empty-block-true" for an empty repeater and "empty-block-false" for one that has rows, allowing us to apply different formatting in the CSS
  - the H5 heading in the row contains the number of that row: "`<jx:out value="${repeaterLoop.index + 1}"/>`", giving headings like "Genre 1", "Genre 2", etc.
  - the repeater is wrapped in a condition: "`<jx:if test="${repeater.getSize() &gt; 0}">`". It wll only be rendered if it contains rows
  - the add-to widget is wrapped in the converse condition: "`<jx:if test="${repeater.getSize() == 0}">`". It will only be rendered if the repeater is empty.
- "common-element-*" classes are used to provide elements that are common to many top-level elements

Model

```
<fd:class id="genre-class">
  <fd:widgets>
    <fd:repeater id="genres" orderable="true">
      <fd:label>mods:genre</fd:label>
      <fd:widgets>
        <fd:field id="genre">
          <fd:label>Genre</fd:label>
```

```
          <fd:datatype base="string"/>
        </fd:field>
        <fd:field id="authority">
          <fd:label>Authority</fd:label>
          <fd:datatype base="string"/>
          <fd:selection-list>
            <fd:item value="">
              <fd:label>(none)</fd:label>
            </fd:item>
            <fd:item value="rbgenr">
              <fd:label>rbgenr</fd:label>
            </fd:item>
            <fd:item value="marcgt">
              <fd:label>marcgt</fd:label>
            </fd:item>
          </fd:selection-list>
        </fd:field>
        <fd:new id="common-language-elements-class"/>
        <fd:new id="common-controls-class"/>
      </fd:widgets>
    </fd:repeater>
  </fd:widgets>
</fd:class>
```

Binding

```
<fb:class id="genre-class">
  <fb:repeater id="genres" row-path="mods_genre" parent-path=".">
    <fb:value id="genre" path="."/>
    <fb:value id="authority" path="@authority"/>
    <fb:new id="common-language-elements-class"/>
  </fb:repeater>
</fb:class>
```

Template

```
<ft:class id="genre-class">
  <ft:repeater id="genres">
    <div class="block empty-block-${repeater.getSize() == 0}">
      <h4>
        <ft:widget-label id="../genres"/>
      </h4>
      <jx:if test="${repeater.getSize() &gt; 0}">
        <div class="repeater">
          <ft:repeater-rows>
            <div class="row toplevel genre">
              <h5>
                <ft:widget-label id="../../genres"/>
```

```
              <jx:out value="${repeaterLoop.index + 1}"/>
            </h5>
            <ft:widget-label id="genre"/>
            <ft:widget id="genre">
              <fi:styling size="80"/>
            </ft:widget>
            <jx:out value="${divider}"/>
            <ft:widget-label id="authority"/>
            <ft:widget id="authority"/>
            <ft:new id="common-language-elements-class"/>
            <ft:new id="common-controls-class"/>
          </div>
        </ft:repeater-rows>
      </div>
    </jx:if>
    <jx:if test="${repeater.getSize() == 0}">
      <ft:widget id="../add-to-genres"/>
    </jx:if>
  </div>
</ft:repeater>
</ft:class>
```

Subrepeater

Subrepeaters are repeaters contained within top-level elements, e.g. namePart elements within a name element. They also have a distinctive pattern.

- where the element is named "mods:note", the repeater has the id "notes"; the "add-to" action has the id "add-to-notes".
- the number of the row is added dynamically to the H6 header, as with top-level elements
- the repeater and the add-to button are not controlled dynamically as in top-level elemtns, as the "repeater" object doesn't seem to function in subrepeaters (see to-do list below)
- subsubrepeaters work like subrepeaters
- widgets within subrepeaters and subsubrepeaters must have ids that are unique within their class, so that they can be identified in localization (e.g. see ids like "places_placeterms_authority" in originInfo-class).

Model

```
<fd:repeater id="notes" orderable="true">
  <fd:label>Notes</fd:label>
  <fd:widgets>
    <fd:new id="common-textfield-class"/>
    <fd:new id="common-controls-class"/>
  </fd:widgets>
</fd:repeater>
```

```
<fd:repeater-action id="add-to-notes" action-command="add-row" repeater="notes">
  <fd:label>New note</fd:label>
</fd:repeater-action>the
```

Binding

```
<fb:repeater id="notes" row-path="mods_note" parent-path=".">
  <fb:new id="common-textfield-class"/>
</fb:repeater>
```

Template

```
<div class="block">
  <ft:repeater id="notes">
    <div class="block empty-repeater-${repeater.getSize() == 0}">
      <ft:repeater-rows>
        <div class="row subrepeater physicalDescription-note">
          <h6>
            <ft:widget-label id="../../notes"/>
            <jx:out value="${repeaterLoop.index + 1}"/>
          </h6>
          <ft:new id="common-textfield-class"/>
          <ft:new id="common-controls-class"/>
        </div>
      </ft:repeater-rows>
    </div>
  </ft:repeater>
</div>
<ft:widget id="add-to-notes"/>
```

Extending the demo: localization

MODS Editor includes a basic localization function. This is used to customize the form for a particular project. It can remove specified elements that are not used, and it can add selection lists to specified elements. Any element in a source MODS record that is not included in the form is retained unchanged; this allows an implementation to use different localizations for different purposes. For example, one might provide a simple bare-bones record for quick data entry, while another might provide full coverage for detail work, or just the subject fields for use by a metadata specialized checking consistency.

The localization process works by creating an XSL stylesheet on the fly from the specified localization config file, and applying it to the Cocoon Forms model, binding and template files when the form is built. The dynamic stylesheet contains templates to make all the necessary changes to each of the Cocoon Forms files. It is applied in a cocoon: protocol pipeline.

The dynamic stylesheet depends on the consistent design patterns discussed above, so care must be

taken when modifying the Cocoon Forms files.

Functions

Suppress

The <suppress> element contains classes and fields that are to be removed. Entries may be in these forms:

- <class>common-linking-elements</class> - remove the specified class wherever it occurs, both the declaration and any instantiation
- <field class="common-language-elements">transliteration</field> - removes the specified field from the specified class. The field may be a widget wtihin a repeater or subrepeater.

SelectionList

A <selectionlist> element adds a selection list to a field within a class. There may be any number of <selectionlist> elements in a localization config. The element has a "class" attribute specifying the class to which it applies: e.g. <selectionlist class="common-language-elements">. Each <selectionlist> may contain any number of <field> elements, each specifying a selection list for a given field:

```
<field id="lang">
  <item value="">(no language)</item>
  <item value="eng">English</item>
  <item value="fre">French</item>
  <styling/>
</field>
```

Each <item> element will produce an <fd:item> element in the model with the appropriate value attribute and <fd:label>. The <styling> element will cause all its attributes to be copied into an <fi:styling> element.

Subclass

The <subclass> element is used to force the system to "subclass" a given class, i.e. to declare a copy of it, which is customized for a particular instantiation. For example, the "common-textfield-class" is instantiated in several other classes that contain textareas. If you modify it using <suppress> or <selectionlist> elements, the changes will apply to every intance; so if you want to change only one instance (say the note-class), you need to subclass it. There may be any number of <subclass> elements, and they may contain <suppress> and <selectionlist> elements that will be applied within the subclass. The form is this: <subclass class="common-textfield" userclasses="note">. The class element specifies the class to be subclassed, and userclasses contains a space-delimited list of target classes which are to use the subclass.

Sample Localization File

```
<localize>
  <suppress>
    <class>common-linking-elements</class>
    <class>name</class>
    <field class="common-language-elements">transliteration</field>
    <!-- nested field -->
    <field class="originInfo">places_placeterms_authority</field>
  </suppress>
  <selectionlist class="common-language-elements">
    <field id="lang">
      <item value="">(no language)</item>
      <item value="eng">English</item>
      <item value="fre">French</item>
      <styling/>
    </field>
  </selectionlist>
  <!-- note: userclasses can be a space-delimited list of class names -->
  <subclass class="common-textfield" userclasses="note">
    <suppress>
      <field>xlink</field>
    </suppress>
    <selectionlist>
      <field id="type">
        <item value="">(no type)</item>
        <item value="type1">type 1</item>
        <item value="type2">type 2</item>
        <styling/>
      </field>
    </selectionlist>
  </subclass>
</localize>
```

Extending the demo: saving the modified record

In the demo, the edited record is not saved, but simply displayed back to you. This behaviour is controlled by the "mods-success-pipeline". There are a couple of ways to add the ability to save the file, depending on whether it resides in the file system or was loaded from a network connection and needs to be POSTed back to a WebDAV or other HTTP-based service.

Saving to a file

This process uses the source-writing transformer [http://cocoon.apache.org/2.1/userdocs/sourcewriting-transformer.html] to write the MODS record to a file.

- Uncomment the "write-source" transformer configuration in your `sitemap.xmap`.
- Set the global variable "repo" to the path of the directory where files should be saved. (Make sure this directory is writable by the servlet engine, of course). The path may be absolute or relative to the directory where `sitemap.xmap` resides.

- Uncomment the "write-source" section of the success pipeline:

```
<map:transform src="xsl/source-writer.xsl">
  <map:parameter name="documentURI" value="{flow-attribute:documentURI}"/>
  <map:parameter name="repo" value="{global:repo}"/>
</map:transform>
<map:transform type="write-source"/>
```

- The stylesheet `source-writer.xsl` wraps a copy of the MODS record in write-source directives indicating where and how it should be saved; the write-source transformer executes these directives and inserts the results of the save operatiion (which are therefore available for display at the end of the pipeline).
- If you want the revised record to overwrite the original, it is best to modify the mods-data pipeline to use the global variable "repo" as well (see the comment there in the sitemap).

POSTing to an HTTP interface

In this example we'll use the WebDAV transformer [http://www.wallandbinkley.com/quaedam/?p=104] to POST to a WebDAV server; the method will work for any interface that accepts POSTs (a Solr index, an eXist database, etc. etc.). It would be easy to transform the MODS record into some other format while populating the WebDAV section, e.g. to create a Solr index document.

- Make sure that Cocoon was compiled with the WebDAV block enabled. If so, the necessary transformer should be configured in the main Cocoon sitemap, so you don't need to alter the MODS Editor sitemap.
- Set the global variable "webdav" to the base URL of your WebDAV service.
- Uncomment the WebDAV section of the success pipeline:

```
<map:transform src="xsl/webdav.xsl">
  <map:parameter name="documentURI" value="{flow-attribute:documentURI}"/>
  <map:parameter name="webdav" value="{global:webdav}"/>
</map:transform>
<map:transform type="webdav"/>
```

Reporting Results

As it moves through the success pipeline the MODS record is contained in a <wrapper> element, which also contains the write-source and webdav instructions. When the instructions are executed by appropriate transformers, they are replaced in the XML by statements of the results of the operations. The stylesheet `saveResult.xsl` displays these results in HTML.

Extending the demo: authentication

MODS Editor is configured to use Cocoon's authentication framework [http://cocoon.apache.org/2.1/developing/webapps/authentication.html].

- The authentication and all form continuations are stored in the user's session [http://cocoon.apache.org/2.1/developing/webapps/session.html], which is controlled by the servlet container. (So if you want to change the session time-out, you'll have to do it in the servlet container).
- By default MODS Editor does not ask for or pay attention to passwords: any user id is accepted.
- To add true authentication, you must do the following:
  - Add a password field to the login form, named "password"
  - Modify the "authenticate" pipeline to provide an authoritative answer by testing the id and password. The easiest way to do this is to add users to the `users/users.xml` file, and uncomment the condition in `xsl/authenticate.xsl`:

```
<xsl:if test="/users/user[ID = $ID and @password = $password">
  <ID><xsl:value-of select="$ID"/></ID>
</xsl:if>
```

  - If you want to use some other means of authenticating IDs and passwords, read about "authentication resources [http://cocoon.apache.org/2.1/developing/webapps/authentication/authenticating_user.html]"

Putting MODS Editor into production

- Set up a method for saving the modified records (see above)
- Set up an authentication method (see above)
- In sitemap.xmap, add the attribute "internal-only='true'" to the map:pipeline element that contains all the internal pipelines (it's identified by a comment "Internal Pipelines"). This prevents these pipelines from being accessed by users, on the security principle of not exposing internals needlessly.
- If necessary, localize the form definition and the CSS (see above)
- Don't forget to set up your production instance of Cocoon to handle UTF-8.

Desiderata

- go through the MODS elements in the form with a fine-tooth comb and correct any remaining errors in implementation
- validation [http://cocoon.apache.org/2.1/userdocs/widgetconcepts/validation.html] for various elements, configurable by localization process
- help (populate `fd:help` elements in individual elements in model with text from MODS site)
- update to MODS 3.3 [http://www.loc.gov/standards/mods/changes-3-3.html]; adjust the form according to the MODS version as specified within the record
- test with Cocoon 2.1.11 and Cocoon 2.2

- formatting fix: make subrepeaters suppress their "New" button when the subrepeater isn't empty, the way main repeaters do)
- More localization features:
  - ability to create load-only widgets (e.g. to take the url of a thumbnail image and create an `<img>` tag) - note that you can bind more than one widget to a given MODS field, provided that no more than one of them has a "save" action
  - apply the dynamic stylesheet to `resources/forms-samples-styling.xsl`, which would let us add formatting templates to change the display of selected widgets
  - change labels
  - add or replace links to css stylesheets
  - add arbitrary javascript binding logic to specific fields, like the timestamp binding
- add internationalization: replace all labels in the model and any text in the template with i18n: tags and manage them all in a language file (as well as making it easier to add languages, this would make it easier to customize the labels)