ReinhardHorn

Modular Database Actions

This How-To is based on components included in the Cocoon 2.1.3 distribution. If you do not have this version, you can obtain it from the Apache Cocoon web site.

Overview

This How-To shows you how to use Modular Database Actions introduced in Cocoon 2.1.x. It requires a basic knowledge of Cocoon, XSLT, XSP and SQL. You should also have a basic understanding of professional SQL databases.

Purpose

You will learn how to maintain database data with Cocoon.

Intended Audience

Cocoon users who need database data in their web applications.

Prerequisites

Cocoon must be running on your system. The steps below have been tested with Cocoon 2.1.3.

You will need the following:

- A servlet engine such as Tomcat.
- JDK 1.4 or later
- A SQL database integrated into Cocoon
- The Cocoon 2.1.3 source distribution

If don't know how to build the source distribution you should read this paper first. An overview of the basic Cocoon database concepts is also helpful. An introduction how to integrate your database into Cocoon can be found here. To understand the main page based on XSP have look at the ESQLdocumentation. And last but not least read the introduction to database actions

---

Steps

Create an small Web Application

Create the files and name them as specified below. Create also a database table with two fields called band_id with an *integer* type and band with a type of *varchar*. You can of cause also create a different example and learn something about the configuration. To keep the first example as simple as possible I created a very simple HTML page for data input.

band.html

```
<html>
 <body>
  <form action="insert">
   <h4>Please insert a band</h4>
   <table>
    <tr>
     <td>Number</td>
     <td><input type="text" name="bands.band_id"/>
    </tr>
    <tr>
     <td>Name</td>
     <td><input type="text" name="bands.band"/>
    </tr>
    <tr>
     <td><input type="submit" name="action" value="Insert"/>
    </tr>
   </table>
  </form>
 </body>
</html>
```

The parameter names of the text fields (bands.band_id, bands.band)will become very important in the action configuration (see below). To display the input results I created A XSL stylesheet with ESQL support.

bands.xsp
{{{<?xml version="1.0" encoding="ISO-8859-1"?>
<xsp:page language="java"
xmlns:xsp="http://apache.org/xsp"
xmlns:esql="http://apache.org/cocoon/SQL/v2">
<page>
<esql:connection>
<esql:pool>kaffee</esql:pool>
<esql:execute-query>
<esql:query>SELECT band_id, band FROM bands</esql:query>
<esql:results>
<bands>
<esql:row-results>
<band>
<esql:get-columns/>
</band>
</esql:row-results>
</bands>
</esql:results>
</esql:execute-query>
</esql:connection>
</page>
</xsp:page>
}}}

bands.xsl
{{{<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:esql="http://apache.org/cocoon/SQL/v2">
<xsl:template match="bands">
<html>
<body>
<form action="bands" method="get">
<table border="1">
<tr>
<th>Band ID</th>
<th>Name</th>
</tr>
<xsl:apply-templates/>
</table>
</form>
</body>
</html>
</xsl:template>
<xsl:template match="band">
<tr>
<td><xsl:value-of select="BAND_ID"/></td>
<td><xsl:value-of select="BAND"/></td>
</tr>
</xsl:template>
</xsl:stylesheet>
}}}

Attention: If you don't see any data keep in mind that XML-Tags are case sensitive. Some database systems return fieldnames in lower cases. In this case replace *BAND_ID* in the value-of-tag with *band_id* and do the same with bands.

Create a Configuration File

The configuration file maps your database table to your input params in the page.

insert.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <bands>
  <connection>kaffee</connection>
  <table name="bands">
   <values>
    <value name="band_id" type="int"/>
    <value name="band" type="string"/>
   </values>
  </table>
 </bands>
```

The connection-tag refers to your datasource in the cocoon.xconf file. The table-tag is self explaining. The name attribute must be identical to the name of your database table. In this case the table name is *bands*. The name attribute inside the value-tag must also must match the fieldnames of your database table. Make sure that the type attribute matches the datatype of your table. Typically *int* is used to map numeric fields and *string* to map character fields like *char* or *varchar*.

How does the database action find your input parameters from band.html? According to the Cocoon documention modular database actions default to *table name.columnname*. In this case *bands.band_id* and *bands.band* (see band.html).

Register your Database Action in the Sitemap

sitemap.xmap

```
...
<map:action name="mod-db-add" src="org.apache.cocoon.acting.modular.DatabaseAddAction">
 <descriptor>xml/insert.xml</descriptor>
 <throw-exception>true</throw-exception>
</map:action>
...
```

Since Modular Database Actions are not located in Cocoon core package it is necessary to register them in the sitemap configuration file. The source attribute refers to the corresponding Java class incapsulating the action. It is important to make a correct entry. Note that the modular package is a subpackage of the original database action package. There are also delete and update action classes. You can register those classes in the same manner. We will need the in the later examples. The descriptor file contains the action configuration described in the previous step.

Match your Application in the Sitemap

sitemap.xmap

```
...
<map:match pattern="insert">
 <map:act type="mod-db-add">
  <map:generate type="serverpages" src="xsp/bands.xsp"/>
  <map:transform type="xslt" src="xsl/bands.xsl"/>
  <map:serialize type="html"/>
 </map:act>
</map:match>
...
```

The database action is executed immediately after the request. If the execution has been successful the act method of the database action class will return a filled map that contains the return value or an empty map if the execution has failed. In the letter case actions normally return the execution to the outer block and the request processing continues. However in this simple example that is not the case because a missing entry in the band_id input field will cause an exception to be thrown.

In a real world example it would be necessary to validate the request parameters before using the database action. In addition to that there would be an outer block to be executed in case that the validation fails. Because in this example that is not the case we need exceptionally a second matcher that maps the HTML page.

```
...
<map:match pattern="bands">
 <map:read mime-type="text/html" src="html/band.html"/>
</map:match>
...
```

Execute the Example

Create the following subdirectories under your Cocoon application directory:

- *html* for the html page
- *xml* for the config file
- *xsp* for the band view
- *xslt* for the stylesheet

If you have not subsitemap call entry page using the following URL in your Browser: *http://localhost:8080/cocoon/bands_. If you have a subsitemap call _http://localhost:8080/cocoon/subsite_name/bands_. For example my subsitemap is called _kaffee*, the german word for coffee 🙂. The correct URL would be _http://localhost:8080/cocoon/kaffee/bands_.

Auto Values

In a real world database application fields with primary key values are inserted by using sequences or auto values. Normally not the user but the developer is responsible for key values. Auto values depend on the DMBS. We cannot cover them here. Look into your DMBS manual to learn how to use auto-values. In the following I show you how to prepare Cocoon for auto values.

Cocoon 2.1.x provides some autoincrement modules for several database systems. Look into the cocoon.xconf file located in the WEB-INF directory and find the autoincrement-modules-tag. By default there is only one autoincrement module activated for the HSQL database. If you want to use more than on module you have to uncomment an rename them. For example if you need the MySQL module change the name attribute to *mysql* and reload your Cocoon application.

Configuration of Autoincrement Modules

cocoon.xconf

```
...
<autoincrement-modules>
...
 <component-instance class=
  "org.apache.cocoon.components.modules.database.HsqlIdentityAutoIncrementModule"
   logger="core.modules.auto" name="auto"/>
 <component-instance logger="core.modules.auto"
 name="mysql" class="org.apache.cocoon.components.modules.database.MysqlAutoIncrementModule"/>
 <component-instance logger="core.modules.auto"
  name="manual" class="org.apache.cocoon.components.modules.database.ManualAutoIncrementModule"/>
...
</autoincrement-modules>
...
```

Consider the last entry. The manual autoincrement module is different form the others because it increments the table's key value by using a simple sql statement. This module cannot prevent two users making insert from using the same key value. This module should be your second choice. On the other hand Cocoon currently provides only modules for some DMBS.

Changes to insert.xml

Add an autoincrement attribute to the key tag in the action configuration file. In this case we use the manual mode with the corresponding name in the configuration file.

insert.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <bands>
  <connection>kaffee</connection>
  <table name="bands">
   <keys>
     <key name="band_id" type="int" autoincrement="true">
      <mode name="manual" type="autoincr"/>
     </key>
   </keys>
   <values>
    <value name="band" type="string"/>
   </values>
  </table>
 </bands>
```

Changes in the HTML File

Remove the ID entry field from band.html.

band.html

```
<html>
 <body>
  <form action="insert">
   <h4>Please insert a Band</h4>
   <table>
    <tr>
     <td>Name</td>
     <td><input type="text" name="bands.band"/>
    </tr>
    <tr>
     <td><input type="submit" name="action" value="Insert"/>
    </tr>
   </table>
  </form>
 </body>
</html>
```

That's it. Execute the example. If you want to use a different input module, for example the MySQL module, you just have to change the name attribute in the mode tag of the config file to *mysql*. However do not forget to prepare the field type in the database table with the "auto_increment" attribute. Otherwise it won't work.

Table Updates

So far I have been covering database inserts. Now I want to talk about modifications on existing values. To make it a little more complicated I show you an example with indexed params. Indexed params are useful if you want to update multiple rows.

Changes to bands.xsl

In this example the updates are made in the XSP/XSL page directly. The stylesheet got a text field for data entries and a submit button to invoke the update action. Attention must be paid to the param names. Indexed params are concatenated by *tablename.columnname[*]*. The asterisk is a token for the row index. As you can see I have parameterized the name of the input fields with the value of the ID.

bands.xsl

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:esql="http://apache.org/cocoon/SQL/v2">
<xsl:template match="bands">
 <html>
  <body>
   <form action="bands" method="get">
    <table border="1">
      <tr>
       <th>Band ID</th>
       <th>Name</th>
      </tr>
       <xsl:apply-templates/>
    </table>
    <input type="submit" name="action" value="Save"/>
   </form>
  </body>
 </html>
</xsl:template>
<xsl:template match="band">
  <xsl:variable name="band_id" select="BAND_ID"/>
  <xsl:variable name="band" select="BAND"/>
  <tr>
    <td><xsl:value-of select="$band_id"/><input type="hidden" name="bands.band_id[{$band_id}]" value="
{$band_id}"/></td>
    <td><input type="text" name="bands.band[{$band_id}]" value="{$band}"/></td>
  </tr>
 </xsl:template>
</xsl:stylesheet>
```

The Update Configuration

The set attribute in the config file indicates that the parameter name can vary. A value of "master" determines the number of rows. All other fields need a corresponding value of *slave*. In this case I added a parameter tag for the indexed request params. Notice the mode attributes. The name has a value of *request-param* and a type value of *all*. This means that the entry page is expected to have varying request params with the names *bands.band_id[*]* and *bands.band[*]*. These params contain the key values of the database tables. Each row is identified by a primary key value to make sure that the correct datarow will be modified. The index represented by the asterisk must contain the *distinct* key values of the primary key field. Internally the indexes vary between 0 and rows - 1.

update.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bands>
 <connection>kaffee</connection>
 <table name="bands">
  <keys>
   <key name="band_id" type="int" set="master">
    <mode name="request-param" type="all">
     <parameter>bands.band_id[*]</parameter>
    </mode>
   </key>
  </keys>
  <values>
   <value name="band" type="string" set="slave">
    <mode name="request-param" type="all">
     <parameter>bands.band[*]</parameter>
    </mode>
   </value>
  </values>
 </table>
</bands>
```

Register the Update Action

Add the following entry to your sitemap:

sitemap.xmap

```
...
<map:action name="mod-db-upd" src="org.apache.cocoon.acting.modular.DatabaseUpdateAction">
 <descriptor>xml/update.xml</descriptor>
 <throw-exception>true</throw-exception>
</map:action>
...
```

The Matcher

In the last step we configure the action mapping. When the page is executed the first time the database update fails. The execution continues with the pipeline after the closing act tag. The inner pipeline will be executed when you submit your changes pressing the save button.

sitemap.xmap

```
...
<map:match pattern="bands">
 <map:act type="mod-db-upd">
  <map:generate type="serverpages" src="xsp/bands.xsp"/>
  <map:transform type="xslt" src="xsl/bands.xsl"/>
  <map:serialize type="html"/>
 </map:act>
 <map:generate type="serverpages" src="xsp/bands.xsp"/>
 <map:transform type="xslt" src="xsl/bands.xsl"/>
 <map:serialize type="html"/>
</map:match>
...
```

Deleting Table Rows

The handling of delete actions is very similar to table updates.

The stylesheet

bands.xsl

```
<?xml version="1.0"?>
 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:esql="http://apache.org/cocoon/SQL/v2">
 <xsl:template match="bands">
  <html>
   <body>
    <form action="bands" method="get">
     <table border="1">
       <tr>
        <th>Band ID</th>
        <th>Name</th>
       </tr>
        <xsl:apply-templates/>
     </table>
     <input type="submit" name="action" value="Delete"/>
    </form>
   </body>
  </html>
 </xsl:template>
 <xsl:template match="band">
   <xsl:variable name="band_id" select="BAND_ID"/>
   <xsl:variable name="band" select="BAND"/>
   <tr>
     <td><xsl:value-of select="$band_id"/></td>
     <td><input type="checkbox" name="bands.band_id[{$band_id}]" value="{$band_id}<xsl:value-of select="$band"
/></input></td>
   </tr>
 </xsl:template>
</xsl:stylesheet>
```

You can use checkboxes to mark the rows to be deleted. The value of the checkbox is set to the value of the key. When the checkbox is checked, its value is submitted to the action as a normal request parameter.

The Configuration

delete.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <bands>
  <connection>kaffee</connection>
  <table name="bands">
   <keys>
    <key name="band_id" type="int" set="master">
     <mode name="request-param" type="all">
      <parameter>bands.band_id[*]</parameter>
     </mode>
    </key>
   </keys>
  </table>
 </bands>
```

The configuration is a minimal one. In fact you can copy the update configuration and remove the values tag.

The Sitemap

sitemap.xmap

```
...
<map:match pattern="bands>
 <map:act type="request">
  <map:parameter name="parameters" value="true"/>
  <map:select type="parameter">
   <map:parameter name="parameter-selector-test" value="{action}"/>
   <map:when test="Delete">
    <map:act type="mod-db-del">
     <map:generate type="serverpages" src="xsp/bands.xsp"/>
     <map:transform type="xslt" src="xsl/bands.xsl"/>
     <map:serialize type="html"/>
    </map:act>
   </map:when>
  </map:select>
 </map:act>
 <map:generate type="serverpages" src="xsp/bands.xsp"/>
 <map:transform type="xslt" src="xsl/bands.xsl"/>
 <map:serialize type="html"/>
</map:match>
...
```

To prevent a complete deletion of the whole table the delete action is wrapped in a request action. The delete action is only executed when the delete button is pressed. The parameter selector decides whether the action parameter has a value of *Delete*, otherwise the outer pipeline will be executed. The map:parameter tag makes the request params available in the sitemap. The value attribute must be set to a value of *true*.

Putting it all together

1. The Stylesheet

The last example shows how the insert, delete and update action can be combined. When the page is request the first time the date is read only. A press on the link activates the update mode. The Java Script snippet in the link submits the user changes to the server. The button activates the an edit field for inserts. This example works only with MySQL since the MySQL autoincrement module is used. The stylesheet uses the bands.xsp to load the data from the DMBS.

bands.xsl

```
<?xml version="1.0"?>
 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                               xmlns:esql="http://apache.org/cocoon/SQL/v2">
 <xsl:param name="action"/>
 <xsl:param name="bands.band_id"/>
 <xsl:param name="bands.band"/>
 <xsl:template match="bands">
  <html>
   <body>
    <h1>Bands</h1>
    <form name="f" action="update" method="get">
     <table border="1">
      <xsl:apply-templates/>
      <xsl:if test="$action='Edit'">
       <tr>
        <td>#</td>
        <td><input type="text" name="bands.band" value=""/></td>
        <td><a href="javascript:document.f.submit()">Insert</a>
            <input type="hidden" name="action" value="Insert"/>
        </td>
        <td colspan="2"><a href="update?action=Cancel">Cancel</a>
        </td>
       </tr>
      </xsl:if>
      </table>
      <input type="submit" name="action" value="Edit"/>
    </form>
   </body>
  </html>
 </xsl:template>
 <xsl:template match="band">
   <tr>
     <xsl:variable name="band" select="band"/>
     <xsl:variable name="band_id" select="band_id"/>
     <xsl:choose>
      <xsl:when test="$action='Select'">
       <xsl:choose>
        <xsl:when test="$bands.band_id=$band_id">
         <td>
          <xsl:value-of select="$band_id"/></td>
         <td><input type="text" name="bands.band" value="{$band}"/>
             <input type="hidden" name="bands.band_id" value="{$band_id}"/>
         </td>
         <td><a href="javascript:document.f.submit()">Update</a></td>
         <td><a href="update?action=Cancel">Cancel</a>
             <input type="hidden" name="action" value="Update"/>
         </td>
        </xsl:when>
        <xsl:otherwise>
         <td><xsl:value-of select="$band_id"/></td>
         <td><xsl:value-of select="$band"/></td>
         <td><a href="update?action=Delete&amp;bands.band_id={$band_id}">Delete</a></td>
         <td><a href="update?action=Select&amp;bands.band_id={$band_id}">Select</a></td>
        </xsl:otherwise>
       </xsl:choose>
      </xsl:when>
      <xsl:otherwise>
       <td>
        <xsl:value-of select="$band_id"/>
       </td>
       <td><xsl:value-of select="$band"/></td>
         <td><a href="update?action=Delete&amp;bands.band_id={$band_id}">Delete</a></td>
         <td><a href="update?action=Select&amp;bands.band_id={$band_id}">Select</a></td>
      </xsl:otherwise>
    </xsl:choose>
   </tr>
 </xsl:template>
</xsl:stylesheet>
```

2. The Configuration

As mentioned above the insert configuration uses the MySQL autoincrement module. If you want try this example with an different DMBS just change the autoincrement module.

insert.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bands>
 <connection>kaffee</connection>
 <table name="bands">
  <keys>
   <key name="band_id" type="int" autoincrement="true">
    <mode name="mysql" type="autoincr"/>
   </key>
  </keys>
  <values>
   <value name="band" type="string"/>
  </values>
 </table>
</bands>
```

The update configuration is similar to the above update example. This time I did not use indexed params because this page only allows single row updates.

update.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bands>
 <connection>kaffee</connection>
 <table name="bands">
  <keys>
   <key name="band_id" type="int">
    <mode name="request-param" type="request">
     <parameter>bands.band_id</parameter>
    </mode>
   </key>
  </keys>
  <values>
   <value name="band" type="string">
    <mode name="request-param" type="request">
     <parameter>bands.band</parameter>
    </mode>
   </value>
  </values>
 </table>
</bands>
```

The delete configuration uses the band.band_id param to determine the primary key value of the row to be deleted. The name of the parameter must be found in the XSL page (see band.xsl).

delete.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bands>
 <connection>kaffee</connection>
 <table name="bands">
  <keys>
   <key name="band_id" type="int">
    <mode name="request-param" type="request">
     <parameter>bands.band_id</parameter>
    </mode>
   </key>
  </keys>
 </table>
</bands>
```

The database action are wrapped in a request action to control the user selection. Depending on the user action either the delete, insert or update action will be executed. All other actions will reload the page in a different mode. The parameter selector makes the request params available to the sitemap.

sitemap.xmap

```
<map:match pattern="update">
  <map:act type="request">
   <map:parameter name="parameters" value="true"/>
   <map:select type="parameter">
    <map:parameter name="parameter-selector-test" value="{action}"/>
     <map:when test="Update">
      <map:act type="mod-db-upd">
        <map:generate type="serverpages" src="xsp/bands.xsp"/>
        <map:transform type="xslt" src="xsl/bands.xsl">
         <map:parameter name="use-request-parameters" value="true"/>
        </map:transform>
       <map:serialize type="html"/>
      </map:act>
     </map:when>
     <map:when test="Delete">
      <map:act type="mod-db-del">
       <map:generate type="serverpages" src="xsp/bands.xsp"/>
       <map:transform type="xslt" src="xsl/bands.xsl">
        <map:parameter name="use-request-parameters" value="true"/>
       </map:transform>
       <map:serialize type="html"/>
      </map:act>
     </map:when>
     <map:when test="Insert">
      <map:act type="mod-db-add">
       <map:generate type="serverpages" src="xsp/bands.xsp"/>
       <map:transform type="xslt" src="xsl/bands.xsl">
        <map:parameter name="use-request-parameters" value="true"/>
       </map:transform>
       <map:serialize type="html"/>
      </map:act>
     </map:when>
   </map:select>
  </map:act>
   <map:generate type="serverpages" src="xsp/bands.xsp"/>
    <map:transform type="xslt" src="xsl/bands.xsl">
     <map:parameter name="use-request-parameters" value="true"/>
    </map:transform>
   <map:serialize type="html"/>
  </map:match>
```

Conclusion

I hope that this introduction has helped you to get started with modular database actions. Of cause the is a lot more you can do. For example it is possible to update more than one table. This is done with table sets. For complex data handling you can use an create input and output modules that cooperate with the action modules. This offers you the flexibility to handle almost any data.