A JSTL primer, Part 4: Accessing SQL and XML content

Custom tag libraries for exchanging XML and database content in JSP pages

Level: Intermediate

Mark A. Kolb (mak@taglib.com)
Software Engineer
May 20, 2003

A hallmark of Web-based applications is the integration of multiple subsystems. Two of the most common mechanisms for exchanging data between such subsystems are SQL and XML. In this article, Mark Kolb concludes his coverage of JSTL with an introduction to the `sql` and `xml` libraries for accessing database and XML content in JSP pages.

Related content:

Building Web-based applications with JDBC
Take control of your JSP pages with custom tags
JSP taglibs: Better usability by design
Subscribe to the developerWorks newsletter
developerWorks Toolbox subscription

Also in the Java zone:

Tutorials
Tools and products
Code and components
Articles

The stereotypical architecture for a Web-based application calls for three tiers: a Web server for handling requests, an application server for implementing business logic, and a database for managing persistent data. The linkage between the application and database tiers typically takes the form of SQL calls into a relational database. When the business logic is written in the Java language, JDBC is used to implement these calls.

If the application calls for integration with additional servers (either local or remote), further mechanisms for exchanging data between the various subsystems will be required. An increasingly common approach to communicating data both within and between Web applications is the exchange of XML documents.

So far in our tour of JSTL, we've examined the JSTL expression language (EL) and both the `core` and `fmt` tag libraries. In this final installment, we'll consider the `sql` and `xml` libraries that -- as their names suggest -- provide custom tags for accessing and manipulating data retrieved from SQL databases and XML documents.

Don't miss the rest of this series

Part 1, "The expression language" (February 2003)

Part 2, "Getting down to the core" (March 2003)

Part 3, "Presentation is everything" (April

The xml library

By design, XML provides a flexible means for representing structured data that is at the same time readily amenable to validation. As a result, it is particularly well suited for exchanging data between loosely coupled systems. This in turn makes it an attractive integration technology for Web-based applications.

The first step in interacting with data represented as XML is to retrieve it as an XML document and parse it to yield a data structure for accessing the contents of the document. After the document has been parsed you can then optionally transform it to yield a new XML

document, to which the same operations can again be applied. Finally, the data from the document can be extracted and then displayed or used as input for performing additional operations.

2003)

These steps are mirrored in the JSTL tags used for manipulating XML. XML documents are retrieved using the `<c:import>` tag from the `core` library, as we discussed in Part 2, Getting down to the core. The `<x:parse>` tag is then used to parse the document, with support for standard XML parsing technologies such as the Document Object Model (DOM) and Simple API for XML (SAX). The `<x:transform>` tag is available for transforming XML documents and relies on the standard technology for transforming XML data: the *Extensible Stylesheet Language* (XSL). Finally, several tags are provided for accessing and manipulating parsed XML data, all of which rely on yet another standard, the *XML Path Language* (XPath), for referencing the contents of parsed XML documents.

Parsing XML

The `<x:parse>` tag actually takes several forms, depending upon the type of parsing desired. The most basic form of this action uses the following syntax:

```
<x:parse xml="expression" var="name" scope="scope"
    filter="expression" systemId="expression"/>
```

Of these five attributes, only the `xml` attribute is required, and its value should be either a `String` containing the XML document to be parsed or an instance of `java.io.Reader` through which the document to be parsed can be read. Alternatively, you can specify the document to be parsed as the body content of the `<x:parse>` tag, using this syntax:

```
<x:parse var="name" scope="scope"
    filter="expression" systemId="expression">
  body content
</x:parse>
```

The `var` and `scope` attributes specify a scoped variable for storing the parsed document. This variable can then be used by the other tags in the `xml` library to perform additional operations. Note that when the `var` and `scope` attributes are present, the type of data structure used by JSTL to represent the parsed document is implementation-specific, allowing for vendor optimization.

If your application needs to perform operations on the parsed document that is provided by JSTL, then an alternate form of `<x:parse>` can be used, which requires that the parsed document adhere to a standard interface. In this case the syntax for the tag is as follows:

```
<x:parse xml="expression" varDom="name" scopeDom="scope"
    filter="expression" systemId="expression"/>
```

When you use this version of `<x:parse>`, the object representing the parsed XML document must implement the `org.w3c.dom.Document` interface. You can also use the `varDom` and `scopeDom` attributes in place of `var` and `scope` when the XML document is specified as the body content of `<x:parse>`, as follows:

```
<x:parse varDom="name" scopeDom="scope"
    filter="expression" systemId="expression">
  body content
</x:parse>
```

The remaining two attributes, `filter` and `systemId`, enable more fine-grained control of the parsing. The

`filter` attribute specifies an instance of the `org.xml.sax.XMLFilter` class for filtering the document prior to parsing. This attribute is particularly useful if the document to be parsed is very large, but only a small subset is of interest for the task at hand. The `systemId` attribute indicates the URI for the document being parsed and resolves any relative paths present in the document. This attribute is required if the XML being parsed uses relative URLs to refer to other documents or resources that need to be accessed during the parsing process.

Listing 1 demonstrates the use of the `<x:parse>` tag, including its interaction with `<c:import>`. Here, the `<c:import>` tag is used to retrieve the RDF Site Summary (RSS) feed for the well-known Slashdot Web site. The XML document representing the RSS feed is then parsed by `<x:parse>`, and an implementation-specific data structure representing the parsed document is stored in a variable named `rss` with `page` scope.

Learning about RSS
RDF Site Summary (RSS) is an XML document format published by many news-oriented sites, which lists their current headlines and provides URLs for linking to the corresponding articles. As such, it provides a simple mechanism for syndicating news items over the Web. For further details on RSS, see Resources.

Listing 1. Interaction of the <x:parse> and <c:import> actions

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="${rssFeed}"/>
```

Transforming XML
XML is transformed by means of XSL stylesheets. JSTL supports this operation through use of the `<x:transform>` tag. As was the case for `<x:parse>`, the `<x:transform>` tag supports several different forms. The syntax for the most basic form of `<x:transform>` is:

```
<x:transform xml="expression" xslt="expression"
    var="name" scope="scope"
    xmlSystemId="expression" xsltSystemId="expression">
  <x:param name="expression" value="expression"/>
  ...
</x:transform>
```

Here, the `xml` attribute specifies the document to be transformed, and the `xslt` attribute specifies the stylesheet defining that transformation. These two attributes are required; the others are optional.

Like the `xml` attribute of `<x:parse>`, the value of the `xml` attribute of `<x:transform>` can be either a `String` containing an XML document or a `Reader` for accessing such a document. In addition, however, it can also take the form of an instance of either the `org.w3c.dom.Document` class or the `javax.xml.transform.Source` class. Finally, it can also be the value of a variable assigned using either the `var` or `varDom` attribute of the `<x:parse>` action.

Alternatively, you can include the XML document to be transformed as the body content of the `<x:transform>` action. In this case, the syntax for `<x:transform>` is:

```
<x:transform xslt="expression"
    var="name" scope="scope"
    xmlSystemId="expression" xsltSystemId="expression">
  body content
  <x:param name="expression" value="expression"/>
  ...
</x:transform>
```

In both cases, the `xslt` attribute specifying the XSL stylesheet should be either a `String`, a `Reader`, or an instance of `javax.xml.transform.Source`.

If the `var` attribute is present, the transformed XML document will be assigned to the corresponding scoped variable as an instance of the `org.w3c.dom.Document` class. As usual, the `scope` attribute specifies the scope for this variable assignment.

The `<x:transform>` tag also supports storing the result of the transformation in an instance of the `javax.xml.transform.Result` class, rather than as an instance of `org.w3c.dom.Document`. If the `var` and `scope` attributes are omitted and a `Result` object is specified as the value of the `result` attribute, the `<x:transform>` tag will use that object to hold the results of applying the stylesheet. The two syntax variations for using the `result` attribute of `<x:transform>` appear in Listing 2:

Listing 2. Syntax variations for the <x:transform> action when using the result attribute to supply a javax.xml.transform.Result instance

```
<x:transform xml="expression" xslt="expression"
    result="expression"
    xmlSystemId="expression" xsltSystemId="expression">
  <x:param name="expression" value="expression"/>
  ...
</x:transform>

<x:transform xslt="expression"
    result="expression"
    xmlSystemId="expression" xsltSystemId="expression">
  body content
  <x:param name="expression" value="expression"/>
  ...
</x:transform>
```

When you employ either of these two forms of `<x:transform>`, the `javax.xml.transform.Result` object must be created independently from the custom tag. The object itself is supplied as the value of the `result` attribute.

If neither the `var` attribute nor the `result` attribute is present, then the results of the transformation will simply be inserted into the JSP page as a result of processing the `<x:transform>` action. This is particularly useful when a stylesheet is being used to transform data from XML into HTML, as illustrated in Listing 3:

Listing 3. Directly displaying transformed XML data in a JSP page

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<c:import var="rssToHtml" url="/WEB-INF/xslt/rss2html.xsl"/>
<x:transform xml="${rssFeed}" xslt="${rssToHtml}"/>
```

In this example, both the RSS feed and an appropriate stylesheet are read in using the `<c:import>` tag. The output of the stylesheet is HTML, which is directly displayed by omitting both the `var` and `result` attributes of `<x:transform>`. Figure 1 shows a sample result:

Figure 1. Output of Listing 3

Like the `systemId` attribute of `<x:parse>`, the `xmlSystemId` and `xsltSystemId` attributes of `<x:transform>` are used to resolve relative paths within XML documents. In this case, the `xmlSystemId` attribute applies to the document provided as the value of the tag's `xml` attribute, while the `xsltSystemId` attribute is used to resolve relative paths within the stylesheet specified by the tag's `xslt` attribute.

If the stylesheet driving the document transformation takes parameters, they are specified using the `<x:param>` tag. When present, these tags must appear inside the body of the `<x:transform>` tag. If the XML document being transformed is also specified as body content, then it must precede any `<x:param>` tags.

The `<x:param>` tag has two required attributes -- `name` and `value` -- just like the `<c:param>` and `<fmt:param>` tags discussed in Part 2 and Part 3 of this series.

Working with XML content

Parsing and transformation act upon XML documents in their entirety. After you've massaged the document into a usable form, however, often only certain elements of the data contained in the document will be of interest to a particular application. For this reason, the `xml` library includes several tags for accessing and manipulating individual pieces of content from XML documents.

If you've read Part 2 of this series (Getting down to the core) then the names of these `xml` tags will be familiar. They are based on corresponding tags from the JSTL `core` library. Whereas these `core` library tags access data from the JSP container through their `value` attributes using EL expressions, their counterparts in the `xml` library access data from XML documents through `select` attributes using XPath expressions.

XPath is a standardized notation for referencing the elements of XML documents and their attributes and body content. As its name suggests, this notation resembles file system paths in the sense that the components of an XPath statement are delimited by slashes. These components map to the nodes of an XML document, with successive components matching nested elements. In addition, asterisks can be used as wildcards to match multiple nodes, and bracketed expressions can be used to match attribute values and specify indices. There are several online references describing XPath and its use (see Resources).

To display an element of data from an XML document, then, use the `<x:out>` action, which is the XML analog to the `core` library's `<c:out>` tag. Whereas `<c:out>` has attributes named `value` and `escapeXml`, however, the attributes of `<x:out>` are `select` and `escapeXml`:

```
<x:out select="XPathExpression" escapeXml="boolean"/>
```

The difference, of course, is that the value of the `select` attribute must be an XPath expression, while the `value` attribute of `<c:out>` must be an EL expression. The meaning of the `escapeXml` attribute is the same for both tags.

Listing 4 demonstrates use of the `<x:out>` action. Note that the XPath expression specified for the `select` attribute is prefaced by an EL expression for a scoped variable, specifically `$rss`. This EL expression identifies the parsed XML document against which the XPath statement is to be evaluated. The statement here searches the document for elements named `title` whose parent nodes are named `channel`, selecting the first such element it finds (as specified by the `[1]` index at the end of the expression). The `<x:out>` action causes the body content of this element to be displayed, with XML character escaping turned off.

Listing 4. Using the <x:out> action to display the body content of an XML element

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="${rssFeed}"/>

<x:out select="$rss//*[name()='channel']/*[name()='title'][1]"
  escapeXml="false"/>
```

In addition to `<x:out>`, the JSTL `xml` library includes the following tags for manipulating XML data:

- `<x:set>` for assigning the value of an XPath expression to a JSTL scoped variable
- `<x:if>` for conditionalizing content based on the boolean value of an XPath expression
- `<x:choose>`, `<x:when>`, and `<x:otherwise>` for implementing mutually exclusive conditionalization based on XPath expressions
- `<x:forEach>` for iterating over multiple elements matched by an XPath expression

Each of these tags behaves similarly to the corresponding tag from the `core` library. Use of `<x:forEach>`, for example, is illustrated in Listing 5, in which the `<x:forEach>` action is used to iterate over all of the elements named `item` in an XML document representing an RSS feed. Note that the XPath expressions in the two `<x:out>` actions nested in the body content of `<x:forEach>` are relative to the nodes over which the `<x:forEach>` tag is iterating. They are used to retrieve the `link` and `title` child nodes of each `item` element.

Listing 5. Using the <x:out> and <x:forEach> actions to select and display XML data

```
<c:import var="rssFeed" url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="${rssFeed}"/>

<a href="<x:out select="$rss//*[name()='channel']/*[name()='link'][1]"/>"
  ><x:out select="$rss//*[name()='channel']/*[name()='title'][1]"
    escapeXml="false"/></a>

<x:forEach select="$rss//*[name()='item']">
  <li> <a href="<x:out select="./*[name()='link']"/>"
        ><x:out select="./*[name()='title']" escapeXml="false"/></a>
</x:forEach>
```

The output resulting from the JSP code in Listing 5 is identical to that of Listing 3, which appears in Figure 1. The `xml` library's XPath-oriented tags thus provide an alternative to stylesheets for transforming XML content, particularly in cases where the final output is HTML.

The sql library

The fourth and final set of JSTL actions is the `sql` custom tag library. As its name suggests, this library provides tags for interacting with relational databases. More specifically, the `sql` library defines tags for

specifying datasources, issuing queries and updates, and grouping queries and updates into transactions.

Datasources
Datasources are factories for obtaining database connections. They often implement some form of connection pooling to minimize the overhead associated with the creating and initializing connections. Java 2 Enterprise Edition (J2EE) application servers typically provide built-in support for datasources, which are made available to J2EE applications through the Java Naming and Directory Interface (JNDI).

JSTL's `sql` tags rely on datasources for obtaining connections. Several, in fact, include an optional `dataSource` attribute for explicitly specifying their connection factory, either as an instance of the `javax.sql.DataSource` interface or as a JNDI name.

You can obtain instances of `javax.sql.DataSource` using the `<sql:setDataSource>` tag, which takes the two following forms:

```
<sql:setDataSource dataSource="expression"
    var="name" scope="scope"/>

<sql:setDataSource url="expression" driver="expression"
    user="expression" password="expression"
    var="name" scope="scope"/>
```

For the first form, only the `dataSource` attribute is required. For the second, only the `url` attribute is required.

Use the first form to access a datasource associated with a JNDI name, by providing that name as the value of the `dataSource` attribute. The second form causes a new datasource to be created, using the JDBC URL provided as the value of the `url` attribute. The optional `driver` attribute specifies the name of the class implementing the database driver, while the `user` and `password` attributes provide login credentials for accessing the database, if needed.

For either of the two forms of `<sql:setDataSource>`, the optional `var` and `scope` attributes assign the specified datasource to a scoped variable. If the `var` attribute is not present, however, then the `<sql:setDataSource>` action has the effect of setting the default datasource for use by `sql` tags that don't specify an explicit datasource.

You can also use the `javax.servlet.jsp.jstl.sql.dataSource` context parameter to configure the `sql` library's default datasource. In practice, placing an entry such as the one in Listing 6 in your application's web.xml file is the most convenient way to specify a default datasource. Using `<sql:setDataSource>` to do so requires the use of a JSP page to initialize the application, and therefore some way to run that page automatically.

Listing 6. Using a JNDI name to set JSTL's default datasource in the web.xml deployment descriptor

```
<context-param>
  <param-name>javax.servlet.jsp.jstl.sql.dataSource</param-name>
  <param-value>jdbc/blog</param-value>
</context-param>
```

Submitting queries and updates
After access to a datasource is established, you can use the `<sql:query>` action to execute queries, while database updates are performed using the `<sql:update>` action. Queries and updates are specified as SQL statements, which may be parameterized using an approach based on JDBC's `java.sql.PreparedStatement` interface. Parameter values are specified using nested `<sql:param>` and `<sql:dateParam>` tags.

Three variations of the <sql:query> action are supported, as follows:

```
<sql:query sql="expression" dataSource="expression"
    var="name" scope="scope"
    maxRows="expression" startRow="expression"/>

<sql:query sql="expression" dataSource="expression"
    var="name" scope="scope"
    maxRows="expression" startRow="expression">
  <sql:param value="expression"/>
  ...
</sql:query>

<sql:query dataSource="expression"
    var="name" scope="scope"
    maxRows="expression" startRow="expression">
  SQL statement
  <sql:param value="expression"/>
  ...
</sql:query>
```

For the first two forms, only the `sql` and `var` attributes are required. For the third, only `var` is required.

The `var` and `scope` attributes specify a scoped variable for storing the results of the query. The `maxRows` attribute can be used to limit the number of rows returned by the query, while the `startRow` attribute allows some initial number of rows to be ignored (such as being skipped over when the result set is being constructed by the database).

After you execute the query, the result set is assigned to the scoped variable as an instance of the `javax.servlet.jsp.jstl.sql.Result` interface. This object provides properties for accessing the rows, column names, and size of the query's result set, as summarized in Table 1:

Table 1. Properties defined by the javax.servlet.jsp.jstl.sql.Result interface

Property	Description
rows	An array of `SortedMap` objects, each of which maps column names to a single row in the result set
rowsByIndex	An array of arrays, each corresponding to a single row in the result set
columnNames	An array of strings naming the columns in the result set, in the same order as used for the rowsByIndex property
rowCount	The total number of rows in the query result
limitedByMaxRows	True if the query was limited by the value of the `maxRows` attribute

Of these properties, `rows` is particularly convenient, because you can use it to iterate through the result set and access the column data by name. This is demonstrated in Listing 7, where a query's results are assigned to a scoped variable named `queryResults`, the rows of which are then iterated over using the `core` library's `<c:forEach>` tag. Nested `<c:out>` tags take advantage of the EL's built-in support for `Map` collections to look up row data corresponding to column names. (Recall from Part 1 that `${row.title}` and `${row["title"]}` are equivalent expressions.)

Listing 7 also demonstrates the use of `<sql:setDataSource>` to associate a datasource with a scoped variable, which is subsequently accessed by the `<sql:query>` action through its `dataSource` attribute.

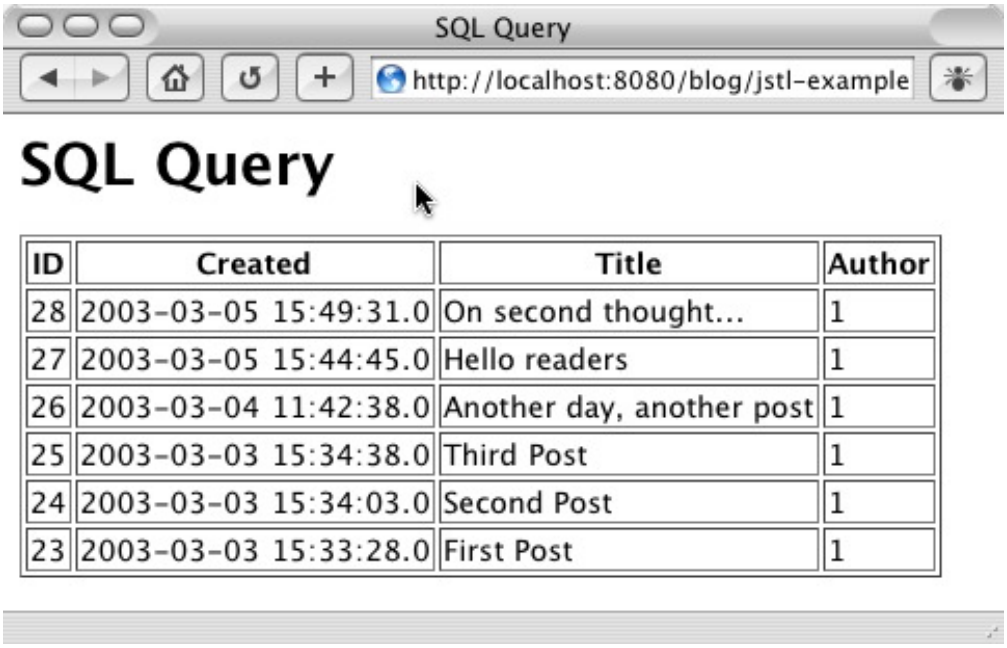
Listing 7. Using <sql:query> to query a database, and using <c:forEach> to iterate through the result set

```
<sql:setDataSource var="dataSrc"
    url="jdbc:mysql:///taglib" driver="org.gjt.mm.mysql.Driver"
    user="admin" password="secret"/>
    <sql:query var="queryResults" dataSource="${dataSrc}">
  select * from blog group by created desc limit ?
  <sql:param value="${6}"/></sql:query>

<table border="1">
  <tr>
    <th>ID</th>
    <th>Created</th>
    <th>Title</th>
    <th>Author</th>
  </tr>
<c:forEach var="row" items="${queryResults.rows}">
  <tr>
    <td><c:out value="${row.id}"/></td>
    <td><c:out value="${row.created}"/></td>
    <td><c:out value="${row.title}"/></td>
    <td><c:out value="${row.author}"/></td>
  </tr>
</c:forEach>
</table>
```

Figure 2 shows sample page output corresponding to the JSTL code in Listing 7. Note also that the SQL statement appearing in the body of the `<sql:query>` action in Listing 7 is parameterized.

Figure 2. Output of Listing 7



Within an `<sql:query>` action, SQL statements specified either as body content or through the `sql` attribute can be parameterized using the `?` character. For each such parameter in the SQL statement, there should be a corresponding `<sql:param>` or `<sql:dateParam>` action nested in the body of the `<sql:query>` tag. The `<sql:param>` tag takes a single attribute -- `value` -- for specifying the parameter value. Alternatively, when the value for the parameter should be a character string, you can omit the `value`

attribute and provide the parameter value as the body content of the <sql:param> tag.

Parameter values representing dates, times, or time stamps are specified using the <sql:dateParam> tag, using the following syntax:

```
<sql:dateParam value="expression" type="type"/>
```

For <sql:dateParam>, the expression for the value attribute must evaluate to an instance of the java.util.Date class, while the value of the type attribute must be either date, time, or timestamp, depending upon which of these three types of time-related values is required by the SQL statement.

Like <sql:query>, the <sql:update> action supports three forms:

```
<sql:update sql="expression" dataSource="expression"
    var="name" scope="scope"/>

<sql:update sql="expression" dataSource="expression"
    var="name" scope="scope">
  <sql:param value="expression"/>
  ...
</sql:update>

<sql:update dataSource="expression"
    var="name" scope="scope">
  SQL statement
  <sql:param value="expression"/>
  ...
</sql:update>
```

The sql and dataSource attributes have the same semantics for <sql:update> as they do for <sql:query>. Similarly, the var and scope attributes are again used to specify a scoped variable, but in this case the value assigned to the scoped variable will be an instance of java.lang.Integer indicating the number of rows that were changed as a result of executing the database update.

Managing transactions

Transactions are used to protect a sequence of database operations that must either succeed or fail as a group. Transaction support is built into JSTL's sql library, which makes it trivial to wrap a series of queries and updates into a transaction simply by nesting the corresponding <sql:query> and <sql:update> actions in the body content of a <sql:transaction> tag.

The syntax for <sql:transaction> is as follows:

```
<sql:transaction dataSource="expression" isolation="isolationLevel">
  <sql:query .../> or  <sql:update .../>
  ...
```

The <sql:transaction> action has no required attributes. If you omit the dataSource attribute, then the JSTL default datasource is used. The isolation attribute is used to specify the isolation level for the transaction and may be either read_committed, read_uncommitted, repeatable_read, or serializable. If you do not specify this attribute, the transaction will use the datasource's default isolation level.

As you might expect, all nested queries and updates must use the same datasource as the transaction itself. In

fact, a `<sql:query>` or `<sql:update>` nested inside a `<sql:transaction>` action is not allowed to specify a `dataSource` attribute. It will automatically use the datasource associated (either explicitly or implicitly) with the surrounding `<sql:transaction>` tag.

Listing 8 shows an example of how `<sql:transaction>` is used:

Listing 8. Using <sql:transaction> to combine database updates into a transaction

```
<sql:transaction>
  <sql:update sql="update blog set title = ? where id = ?">
    <sql:param value="New Title"/>
    <sql:param value="${23}"/>
  </sql:update>
  <sql:update sql="update blog set last_modified = now() where id = ?">
    <sql:param value="${23}"/>
  </sql:update>
</sql:transaction>
```

A word of caution

JSTL's `xml` and `sql` libraries enable complex functionality to be implemented in JSP pages using custom tags. At the same time, however, implementing this sort of functionality in your presentation layer may not necessarily be the best approach.

For large applications being written by multiple developers over a long period of time, strict segregation between the user interface, the underlying business logic, and the data repository has proven to simplify software maintenance over the long term. The popular Model-View-Controller (MVC) design pattern is a formalization of this "best practice." In the domain of J2EE Web applications, the model is the business logic of an application, and the JSP pages comprising the presentation layer are the view. (The controllers are the form handlers and other server-side mechanisms for enabling browser actions to initiate changes to the model and subsequently update the view.) MVC dictates that the three major elements of an application -- model, view, and controller -- have minimal dependencies upon one another, restricting their interactions with each other to consistent, well-defined interfaces.

An application's reliance on XML documents for data exchange and relational databases for data persistence are characteristics of the application's business logic (that is, its model). Adherence to the MVC design pattern would suggest, therefore, that these implementation details should not be reflected in the application's presentation layer (that is, its view). When JSP is used to implement the presentation layer, then, use of the `xml` and `sql` libraries would be a violation of MVC, because their use would mean exposing elements of the underlying business logic within the presentation layer.

For this reason, the `xml` and `sql` libraries are best suited to small projects and prototyping efforts. Dynamic compilation of JSP pages by the application server also makes the custom tags in these libraries useful as debugging tools.

Summary

In this series, we have examined the capabilities of the four JSTL custom tag libraries and their usage. In Part 1 and Part 2, we saw how you can avoid JSP scripting elements in many common situations through use of the EL and the tags of the `core` library. Part 3 focused on using the `fmt` library to localize Web content.

In this final installment, we reviewed the functionality of the `xml` and `sql` libraries. If you're willing to accept the consequences of including business logic in the presentation layer, the tags in these two libraries make it very easy to incorporate content from XML documents and relational databases into JSP pages. These two libraries also demonstrate how the JSTL libraries build upon one another and interoperate when integrating `<sql:query>` and `<c:forEach>`, as well as the ability of the `xml` library to leverage the `<c:import>` action.

Resources

- Download the source code for the Web log example application.
- The official JSTL Web site is a good starting point to learn more about JSTL.
- The JSTL 1.0 Specification is the final authority on the EL and the four JSTL tag libraries.
- The Jakarta Taglibs project is home to the reference implementation for JSTL 1.0.
- *JSTL in Action*, by Shawn Bayern (Manning, 2002), provides excellent coverage of all JSTL features, having been written by the reference implementation lead.
- Popular Java programming author David Geary has also written a book on JSTL, entitled *Core JSTL* (Prentice-Hall and Sun Microsystems Press, 2002).
- JSPTags.com is a directory of JSP resources, focusing particularly on custom tag libraries.
- Coverage of JSTL is included as part of Sun's Java Web Services Tutorial.
- To find out more about RSS, read James Lewin's "An introduction to RSS news feeds" (*developerWorks*, November 2000).
- Mark Colan provides an introductory overview of XSL in "Putting XSL transformations to work" (*developerWorks*, October 2001).
- For a better understanding of XPath and its relationship to the Document Object Model (DOM), take a look at "Effective XML processing with DOM and XPath in Java" (*developerWorks*, May 2002) by Parand Tony Darugar.
- Get up to speed with JDBC with this hands-on tutorial, "Building Web-based applications with JDBC" (*developerWorks*, December 2001).
- "Using JSPs and custom tags within VisualAge for Java and WebSphere Studio" (WebSphere Developer Domain) is a WBOnline hands-on workshop demonstrating the use of servlets, JSP pages, and custom tag libraries.
- Learn all about custom tag libraries with Jeff Wilson's excellent article, "Take control of your JSP pages with custom tags" (*developerWorks*, January 2002).
- Noel Bergman's article, "JSP taglibs: Better usability by design" (*developerWorks*, December 2001), shows you how declarative tags help improve the usability of your JSP pages.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.

About the author

Mark Kolb is a Software Engineer working in Austin, Texas. He is a frequent industry speaker on server-side Java platform topics and the co-author of *Web Development with JavaServer Pages, 2nd Edition*. You can contact Mark at mak@taglib.com.

What do you think of this document?

Killer! (5) Good stuff (4) So-so; not bad (3) Needs work (2) Lame! (1)

Comments?

Submit feedback