Dynamic Web-based data access using JSP and JDBC technologies

Level: Intermediate

Noel Bergman (noel@jspdevguide.com), CTO, Development Technologies, Inc./DevTech

01 Sep 2001

This article discusses using the JSP and JDBC technologies to integrate static, dynamic, and database content in Web sites. For the purposes of simplicity and illustration, the JSP pages here use short scriptlets to expose the JSP developer to the underlying JDBC concepts instead of hiding them in custom tags. The author introduces a key design approach that integrates JavaBeans components with JDBC, similar to the way that JavaServer Pages technology already uses beans with HTTP. He also provides code for implementing this integration.

Building on the Java Servlet technology, JavaServer Pages (JSP) technology is the core server-side Java architecture for generating dynamic content. One source of dynamic content is the relational database. To manage everything from online communities to e-commerce transactions, Web sites use relational databases to store all sorts of information: catalog items, images, text, data about registered members, and so on. This article discusses the application of JSP technology to relational databases through Java Database Connectivity (JDBC). JDBC is the means by which Java programs work with relational databases.

To get the most out of this article, you should be familiar with JDBC and SQL.

**JDBC basics**

JDBC is the bridge between Java code and SQL databases. The primary JDBC objects represent connections to a database and the statements performed using those connections. The two basic kinds of statements used with a relational database are queries and updates. As a prerequisite to each, you first need to establish a connection to the database, which is done with the java.sql.DriverManager class. Connections take a long time (in computer time) to establish, so in a transaction-intensive environment like a Web server, you want to reuse connections whenever possible. Such reuse is called connection pooling.

If your JDBC skills are a bit rusty, the code snippet in Listing 1 illustrates how to establish a connection with a test database, create a statement object to use with that connection, issue an SQL query, process the results, and release the JDBC resources:

**Listing 1. Simple JDBC code**

```java
Connection connection = DriverManager.getConnection(URL, user, password);
Statement statement = connection.createStatement();
ResultSet results = statement.executeQuery(sqlQuery);

while (results.next())
{
    ... process query results ...
    logSQLWarnings(results.getWarnings());
}
results.close();
statement.close();
connection.close();
```

In real life, JDBC code is not this simple; exceptions and warning conditions need to be handled. Listing 2 illustrates the same JDBC example but adds handling for JDBC exceptions and warnings. In this example, exceptions and warnings are simply logged and, in the case of exceptions, we abort the operation. However, the finally{} clauses ensure that resource cleanup proceeds.

The actual processing of the results is only hinted at here; we'll be looking at it more closely later on in this
article. If we were performing a database update instead of a query, we would replace the `while` loop with the following:

```java
int count = statement.executeUpdate(sqlUpdate);
```

The `executeUpdate()` method returns the number of rows affected by the `update` statement.

If the material in these code listings seems unfamiliar, you may want to spend some time reviewing some of the JDBC tutorial information found in the Resources section.

---

### Using JDBC with JSP pages

So how do we combine JDBC and JSP technologies so that our dynamic content comes from a database?

As a general rule, good JSP practice suggests that you separate presentation from model behavior. This is analogous to the Model-View-Controller (MVC) paradigm in object-oriented programming. One reason for the separation is that applications based on JSP technology are likely to have the Model and Controller components authored by programmers, whereas the View components will be authored by page designers. In the case of JSP application architectures, the role of View, whose responsibility is presentation, is handled by a JSP page. The role of Controller, whose responsibility is reacting to requests, is often played by a servlet, but many JSP practitioners are coming to realize the advantages of using a JSP page in the Controller role. The role of Model, whose responsibility is modeling the behavior of application entities, is typically played by JavaBeans components.

In addition to deciding where in the MVC paradigm to interact with the database, you have several choices for integrating JDBC technology into your JSP pages. For example, you can insert JDBC using scriptlets, insert it using a tag library, or hide it within custom tags or other classes. We'll next look at examples of several approaches and discuss their use.

### A JSP scriptlet example

The first thing a new JSP programmer is likely to do is write a scriptlet to access JDBC. Perhaps it will be something like this example in Listing 3, which uses JDBC to implement a "hit counter" for the page. (A live version of this page is located on the JavaServer Pages Developers Guide Web site.)

#### Listing 3. JSP page using JDBC in a scriptlet

```html
<jsp:directive.page import="java.sql.*" />
<jsp:scriptlet>
Class.forName("org.gjt.mm.mysql.Driver");
Connection connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", ", ", ");
Statement statement = connection.createStatement();
int changed = statement.executeUpdate("update counters set hitCount = hitCount + 1 " + 
  "where page like "+ request.getRequestURI() + ");
if (changed == 0) statement.executeUpdate("insert counters(page) values(" + 
  request.getRequestURI() + "));
ResultSet rs = statement.executeQuery("select hitCount from counters where page like "+ 
  request.getRequestURI() + ");
rs.next();
int hitCount = rs.getInt(1);
statement.close();
connection.close();
</jsp:scriptlet>
</HTML>
```
This page has a scriptlet in it (the first highlighted section), which connects to the database, creates a statement, and attempts to update a counter record keyed by the page's URL. If the update affects no rows, this example assumes there is no such record and adds one. Finally, the scriptlet queries the database for the current hit count and assigns the result to a local variable. Further down, in the "presentation" portion of this JSP page, the highlighted JSP expression is used to render the value of the hit counter.

Although it is functional, this JSP page has a number of problems. First, the scriptlet is not something that a nonprogramming page designer is going to want in the page. Frankly, it isn't even something that programmers should want in the page, because it clutters up the page's real content. Second, to keep the example simple, this page lacks the exception handling that should be part of any real-world JDBC code. Third, the implementation of the hit counter is literally embedded in the JSP page, so any change to the hit counter would need to be propagated to every JSP page for which we maintain a hit counter.

So how can we fix this JSP page? One oft-touted solution is to use a tag library to eliminate scriptlets. In our next example, we'll look at this alternative solution.

### A tag library example using DBTags

One of the first things that a new JSP programmer hears, usually from well-intentioned friends and experts, is not to use scriptlets. Instead, they tell the new JSP programmer to use custom tags. Custom tags are a means by which the JSP platform's capabilities are extended: custom XML-style tags, tied to code libraries, implement the desired functionality. We'll see how well they work, in our next example.

The Jakarta TagLibs Project is a subproject of the Jakarta Project (see Resources), the official reference implementation of the Java Servlet and JavaServer Pages technologies.

One of the packages developed under the auspices of the Jakarta TagLibs Project is the DBTags custom tag library (formerly known as the JDBC tag library). The JSP page in Listing 4 implements the same hit counter as in Listing 3, replacing the scriptlet with custom tags.

#### Listing 4. JSP page using DBTags

```
<%@ taglib uri="http://jakarta.apache.org/taglibs/dbtags" prefix="sql" %>
<sql:connection id="conn1">
  <sql:url>jdbc:mysql://localhost/test</sql:url>
  <sql:driver>org.gjt.mm.mysql.Driver</sql:driver>
</sql:connection>
```
I don't know about you, but I'm feeling a bit let down. That seems even less clear to me than the scriptlet example, and I don't know any nonprogramming HTML page designers who would be pleased with it, either. But what went wrong? After all, we followed people's advice: we got rid of the scriptlet and replaced it with custom tags.

What they don't tell you about tag libraries is this: **tag design is language design.** Most tag libraries written to date have been written by programmers for programmers; the semantics of those tags are geared toward other programmers. Furthermore, remember the separation of model and presentation? That isn't well supported by DBTags. The `sql:getColumn` tag is analogous to the `jsp:getProperty` action: it emits the tag's result directly into the output stream. That makes it difficult to separate using DBTags from rendering output into the desired form. Finally, notice that the logic differs between Listing 3 and Listing 4. The DBTags `execute` tag consumes the update count from any `update` statement sent via JDBC; only query results can be retrieved. That means we cannot find out how many rows were updated by the `update` statement. So we have to switch the `update` and `insert` statements; we always try to insert a new record, force DBTags to ignore any error, and then perform the update.

In fairness to the DBTags tag library, it is not a bad tag library for programmers. Aside from its consumption of the update count, the code provides a fairly good mapping to JDBC. Therein lies the

---

Developing custom tag libraries is relatively straightforward, but it does take some thought and it is time consuming. I often recommend that tag library authors first prototype the tag behavior using scriptlets, and then convert those scriptlets into tags.

An alternative is to use Allaire's JRun Server Tags (JST), which enables you to prototype tag libraries by authoring each tag as a JSP page (with a `.jst` extension). The JST converts that page into a tag handler at run time, so the JST technology is transparent to the client pages. Although Allaire claims that the "goal is to establish JST as a portable technology so that all members of the J2EE community can leverage its benefits" JST is currently only available in JRun. Time will tell whether JST becomes a more common means to...
problem, however: the tags provide little more than a direct translation of the JDBC package. Other than hiding some exception handling, the tag library doesn't really provide any abstraction over scriptlets. It certainly doesn't help separate presentation from function.

So, the real issue is not whether to use scriptlets or tags; that question is a consequence, not a cause, of the problem of separating function from presentation. The solution is to provide higher-level functionality to presentation-page authors at an appropriate level of discourse. The reason tags are considered better than scriptlets is that scriptlets, by definition, are programming, whereas tags can represent high-level concepts.

### Hiding JDBC from presentation pages

When integrating JDBC with JSP technology, we want to hide as much of that integration from the presentation author as possible. Where we do expose database concepts, we want to expose them at a suitable level of abstraction. This approach leads to our next example.

In the example in Listing 5, we hide the JDBC integration from the presentation page. (A live version of this page is located on the JavaServer Pages Developers Guide Web site.)

#### Listing 5. JSP page with hidden JDBC

```html
<jsp:directive.include file="/pagelets/hitCounter.jsp" />

<HTML>
<HEAD>
<TITLE>JDBC hidden example</TITLE>
</HEAD>
<BODY>

<P>This page has been hit
<jsp:getProperty name="hitCounter" property="int" />
times. The page counter is implemented indirectly: a JavaBeans component containing the hit count is inserted into the environment and referenced within the page using the JSP getProperty action. The JSP page doesn't have any exposure to JDBC.</P>
</BODY>
</HTML>
```

The included hitCounter.jsp file takes care of setting up the environment. The contents can be a scriptlet, tags, or nothing more than a taglib directive; the contents can be anything that establishes the desired environment for the presentation page. If you wish, you can replace the `getProperty` action with a custom tag; for example:

```html
This page has been hit
<page:hitcounter />
times.
```

As I observed earlier, these hit-counter examples are purely for illustration; performing such database operations for each page would be an unnecessary expense. The above example shows how you might actually want to expose a hit counter. By hiding it in a custom tag, we've completely hidden the implementation. Now we can aggregate the hit-count information at run time and update the database periodically (at the end of each session, for example). Even the means of storage (database or other) is hidden from the presentation-page author. That is how we implement hit counters at DevTech: we have bean classes implementing hit-counter model behavior. Tags tie that behavior into our pages.
Integrating with JavaBeans components

The examples so far have been fairly simple, but most database operations are going to be more sophisticated than these simple queries and updates. So now that we've covered some basic principles of using JDBC with JSP pages, let's conclude with a slightly more complex, and certainly more common, type of application.

The example for this section (Listing 9, below) will show one way to support visitor-supplied content on a Web site. In other words, we want to allow visitors to read database content associated with a URI and to contribute additional content. Such content is fairly common on modern Web sites. The same basic parts can be used to construct:

- Review pages, such as those found on Amazon.com
- Links pages
- Bulletin boards
- Wikiwebs

An only slightly more elaborate version of the JSP components in this example can implement Web pages that seem very different, authored by designers of varying technical backgrounds. All that the pages would appear to have in common is a provision for visitor-contributed content.

Our annotation example uses an HTML form. When using HTML forms with JSP, it is convenient to use a bean whose properties map to the form fields. This allows the `setProperty` tag to do its magic:

Listing 6. Bean instance that maps to a form

```html
<%-- setup a bean instance that matches our form --%>
<jsp:useBean id="instance-name" class="bean-class" ... />
<%-- set all bean properties that match a form field --%>
<jsp:setProperty name="instance-name" property="*" />
```

Integration with JavaBeans components is one of the better-designed aspects of JSP technology. Unfortunately, the integration between beans and JDBC is not seamless at all, so for our JDBC work at DevTech, we developed a package that provides not only integration between beans and JDBC but also the necessary exception handling, relieving the programmer from having to deal with the details.

The annotation example uses two of the query and update methods from the `com.devtech.sql` package. The particular query method used passes a `bean` class, an SQL query, and an `Object` array to fill in the placeholders in the query. In this case, the only placeholder is for the page's URL. The result is a database cursor object, which is essentially a type of iterator.

Listing 7. Database cursor object

```java
database.queryCursor(AnnotationDBBean.class, new String[ ] { URL },
    "select page, author, annotation, DATE_FORMAT(whenPosted, '%W %d%b%y %T')" +
    " as whenPosted from annotations where page like ?");
```

What makes this query method interesting is that the specified type of bean will be instantiated for you, and any bean properties whose names match column names in the ResultSet will have their values set automatically. Each time you use the cursor to select the next row, the bean's properties are automatically set from the ResultSet.

The particular update method used takes a `bean` instance, a `String` array, and an `update` statement. The values of the `String` array specify the desired bean properties to be used to fill in the placeholders in the update. In this case, the `page`, `author`, and `annotation` properties are selected from the bean.

Listing 8. Update method

```java
<%@-- setup a bean instance that matches our form --%>
<jsp:useBean id="instance-name" class="bean-class" ... />
<%@-- set all bean properties that match a form field --%>
<jsp:setProperty name="instance-name" property="*" />
```
Our example JSP page, *annotations.jsp*, is shown in Listing 9. The highlighted sections indicate a couple of scriptlets that could be replaced with custom tags, as shown in Listing 10. The remainder of the page consists of some JSP comments provided to assist the page designer, `getProperty` actions to place dynamic content onto the page, and standard HTML. JSP comments are used because they are private and won't appear in the output stream.

**Listing 9. JSP page for annotations**

```jsp
int count = database.update(annotationBean,
    new String[] { "page", "author", "annotation" },
    "insert into annotations(page, author, annotation) values(?, ?, ?)");
```

By the time we arrive here, the annotation bean has been established, and if the form is submitted, the contents will be posted to the database. The page property is initialized. If the author is known during this session, that property is also initialized.

Bean:  
```
String page;
String author;
String annotation;
String whenPosted;
```

Properties:  
```
"annotation"
```

Access to any bean property follows the format:
```
<jsp:getProperty name="annotation" property="property-name" />
```

```html
<html>
    <head>
        <title>Comments for <jsp:getProperty name="annotation" property="page" /></title>
    </head>

    <body>
        <p align="left"><font size="+1">Comments for <i><jsp:getProperty name="annotation" property="page" /></i></font>.</p>

        <center><hr width="100%"></center>

        <!-- Annotation Submission Form -->
        <form method="POST">
            <table>
                <tr>
                    <th align="left">Name:</th>
                    <td><input type="text" name="author" size=50 maxlength=60 
                        value="<jsp:getProperty name="annotation" property="author" />"></td>
                </tr>

                <tr>
                    <th align="top" align="left">Note:</th>
                    <td><textarea name="annotation" cols=40 rows=5 wrap=virtual>
                        <jsp:getProperty name="annotation" property="annotation" 
                    </textarea></td>
                </tr>

                <tr>
                    <td align="center" colspan="2"><input type="submit" value="Add Comment"></td>
                </tr>
            </table>
        </form>

        <!-- End of Annotation Submission Form -->

        <jsp:scriptlet>
            Database.Cursor annotations = annotation.getCursor();
            while (annotations.next(annotation) != null)
            {
                <jsp:scriptlet>
                    <jsp:scriptlet>
                    <jsp:scriptlet>
                    <jsp:scriptlet>
                        database.insert(annotations.get("page"), annotations.get("author"), annotations.get("annotation"));
                    </jsp:scriptlet>
                </jsp:scriptlet>
            }
        </jsp:scriptlet>
```

The following section iterates through all annotations in the database for the requested page. To change the look of the page, just change anything in the demarcated area.

```
The custom tags equivalent is clear but uninformative:

**Listing 10. Custom tags equivalent**

```xml
<sql:results queryName="annotations" bean="annotation">
  <CENTER><HR WIDTH="100%"></CENTER>
  From: <jsp:getProperty name="annotation" property="author" />
  at <jsp:getProperty name="annotation" property="whenPosted" /><BR>
  <jsp:getProperty name="annotation" property="annotation" /><BR>
  <!-- end of annotation change area --%>
  <jsp:scriptlet>
    annotations.close();
  </jsp:scriptlet>
  <!-- end of annotations -->
</sql:results>
```

We've used scriptlets in this example only to show you, a programmer, what is happening. If they were replaced with declarative tags, they would be clear to the page designer, but uninformative to you.

The logic is straightforward. The `annotation.getCursor()` call acquires a connection to the server, issues the query, and sets up a database cursor object, `annotations`, on the result set. Each time `annotations.next()` is called, a new row is fetched from the result set, and its values moved into a bean whose reference is returned from the method. The particular `next()` method being used takes a bean parameter to populate. Although we could have run the cursor instantiate a new bean for each row, reusing the bean is more efficient.

Notice that neither the actual query, nor the update, is present in the presentation page. The included page, which sets up the environment for the presentation page, also includes the `setProperty` and `update` actions. Those actions are independent of the presentation page; only the contract embodied by the `annotation` bean's properties is significant. This is in keeping with a policy to separate presentation from model behavior. The page designer is fully able to change how the presentation is rendered but has no access to how the database is integrated. If a change is to be effected in updating or querying the database, it is delegated to a JSP programmer.

---

**Summary**

This concludes an introduction to combining the JavaServer Pages, JavaBeans, and JDBC technologies to generate dynamic content through relational databases. We started with the most obvious approach for the new JSP programmer: scriptlets. We saw how the uncontrolled use of scriptlets intertwines logic and presentation, making both of them hard to maintain. We also saw that tag libraries don't necessarily improve MVC separation, and that the pages using them may not be understandable to page designers if the tags are expressed in programming terms. Finally, we looked at more complex examples that illustrate a few ways to separate database access from the presentation of content.

You should now have some basic ideas about how to integrate database content into a Web site while hiding the actual database access from page designers. Note, too, that the least informative examples for you, a programmer, are the ones most appropriate for a page designer. When you plan your JSP solutions, keep your page designers in mind.
Resources

- The official home page for JavaServer Pages technology is an excellent starting point for locating tremendous amounts of information, including all official documents, about JSP technology.

- The official home page for the Jakarta Project contains the reference implementation for Java Servlet and JavaServer Pages technologies and serves as the hub for many other related open-source activities.

- The Java Developer Connection has an article covering several Jakarta Taglibs Project libraries, including database tags.

- The IBM developerWorks Java technology zone offers vendor-neutral tutorials and articles on all things Java, including JavaServer Pages technology.

- "Introduction to JavaServer Pages technology" (developerWorks, August 2001), by Noel J. Bergman, is an introductory tutorial that illustrates the fundamentals of JavaServer Pages (JSP) technology.

- "An easy JDBC wrapper" (developerWorks, August 2001), by Greg Travis, describes a simple wrapper library that makes basic database usage a snap.

- "What's new in JDBC 3.0?" (developerWorks, July 2001) provides an overview of the new features and enhancements found in the Java Database Connectivity 3.0 specification.

- The official home page for JDBC offers helpful resources.

- Allaire's JRun Server Tags technology enables the rapid prototyping of custom tags.

About the author

Noel Bergman's background in object-oriented programming spans more than 20 years, including participation on the original CORBA and Common Object Services task forces. He has consistently received high marks as a favored speaker at the Colorado Software Summit as well as other industry conferences; he is also in demand as a mentor and consultant, providing customized services for each client.

At present, Noel is involved in programming for handheld devices and in developing interactive database-backed Web sites using open-source freeware. To that end, Noel is also a coauthor of GNUJSP, an open-source implementation of JavaServer Pages technology, and an originator of the JSP Developers Guide Web site.