Using Data Access Beans in WebSphere Studio

JavaBeans provide quick and easy access to data

Skill Level: Intermediate

Jeff K. Wilson (wilsonje@us.ibm.com)
E-business Architect
IBM

11 Mar 2003

In this tutorial, we will build an application using Data Access Beans. Data Access Beans are a set of JavaBeans that provide quick and easy access to data sources when the cost of implementing EJBs outweighs the benefits. The application we will build will access the SAMPLE database and run a few SQL commands against it to select and display data, and update and delete records. This tutorial is intended for developers who need to access data from back-end data sources, particularly those who would like to know when and whether it is appropriate to do so without EJBs.

Section 1. Introduction

Which Data Access Beans?

Let me start by saying that when I refer to Data Access Beans, I am not referring to Microsoft's Data Access Objects. The Microsoft part of that might tip you off there.

Nor am I talking about Data Class Access Beans, which are a type of EJB designed to make connecting to EJBs easier. That's right -- more EJBs to make connecting to EJBs easier. Confused? Doesn't matter. That's not what this tutorial is about either. The "No EJBs" in the title might have clued you in on that one.

The Data Access Beans I'm actually going to lead you through are a set of JavaBeans that provide quick and easy access to data sources when the cost of
implementing EJBs outweighs the benefits. IBM WebSphere Studio also provides a series of tag libraries that give JSP developers easy access to those beans from JSP pages.

We will examine both accessing the beans directly in the first part of the tutorial as well as interfacing with them through some of the JSP tags provided by WebSphere Studio.

On with the show...

Uh oh...are you sure you want to do this?

Data access without EJBs? Heresy! Or is it? Actually, you're probably thinking, "Should I be using EJBs? 'Cuz I haven't been..."

When I first mentioned to others on my team that I was thinking about doing a tutorial on accessing data without EJBs -- and, for that matter, even directly from JSPs -- I got looks like I might expect had I presented a Tofurky platter to my Thanksgiving dinner guests.

For those of you who aren't familiar with a traditional American Thanksgiving Day celebration or the carnivores I generally entertain, the look was not a pleasant one.

If you haven't figured it out yet, the problem here is that accessing data from a client is bad. Very, very bad.

Jay-walking bad.

Sticking-your-tongue-out-at-your-teacher-when-she-turns-around bad.

So why would I write a tutorial on such a subject?

The answer is quite simple. People do it all the time. You've probably done it. Even I admit to experimenting with it once or twice.

Why do we generally recommend using EJBs to access data from a Web-based application? Two words: stability and reuse. You may have heard of this in terms of the Model, View, Controller or MVC code design concept.

MVC, proper code, and the path to sainthood

MVC dictates that certain procedures be isolated to certain types of components. If
data access is isolated to EJBs, then multiple JSPs, servlets, JavaBeans, even legacy and various other non-Java systems can easily access that data source with relatively little overhead and, more importantly, without replicating the same steps. Sounds good, right? And it is. However, let's be honest, it can also be overkill for some applications.

For developers of enterprise-wide applications, coding standards and strict principles are essential. These kinds of applications tend to be long-lasting, they evolve over time, and they usually require access to other systems in different environments. By sticking to strict standards, integration becomes a far less daunting task.

For those of us who consult with developers of some of the world's most complex and frequently convoluted systems, sometimes it is easy to forget that most of the applications developed are not all that complicated and may well need little or no updating or integration. It takes a lot of time and effort to develop a completely streamlined application chock-full of reusable components. If the components don't need to be reused, is that time and effort well spent?

The ultimate goal is always to develop code efficiently.

If the project is large, these strict tactics will save time in down-the-road edits. If the project is small and ephemeral, the extra steps can be more of a hindrance than a help.

EJBs require a lot of time and skill to develop and are, therefore, not always appropriate.

While WebSphere Studio provides some very impressive EJB tools that make development easier, it also provides other interesting data access features that, if used properly, can greatly enhance the development process while maintaining some level of stability. Using those data access features is what this tutorial is about.

EJBs or data access beans: simply put

Both EJBs and data access beans provide some nice features for accessing data but certainly do have their specific places within application development. I don't want to say there are negatives for each as much as I would stress that they both provide different strengths and, depending on your application, one will likely appear more appropriate than the other.

There are those that believe that you should never access data from JSPs. "I don't care why it's bad. It just breaks the spirit of the MVC separation of presentation and business logic -- ergo, evil." Because of the great many people who hold this ideal, I
wanted to bring the issue to the table. One needs to release oneself from this heavenly doctrine and look at things more realistically.

EJBs and data access beans both provide an easy interface to data. Both simplify using dynamic content. Both are relatively stable. Both are just fine. Relax. Breathe in. Say it. Just fine.

The main differences, as I see it, come down to flexibility, ease of implementation, and reusability. Data access beans are definitely flexible. But using "bean-managed" EJBs can allow a developer to use transaction and persistence models that the application server does not support. However, EJBs are certainly far, far more difficult to implement -- especially bean-managed EJBs over container managed EJBs. Data access beans are incredibly easy to set up and use. From a business perspective, JSP/servlet developers cost a lot less than EJB developers. And finally, and perhaps most importantly, reusability. If several different systems require access to the same data, providing the interface to that data in JSP form does no good. Each system would need to re-create their own JSPs, servlets, beans, or whatever requires access. If that data interface is in the form of an EJB, multiple components can easily access the data source with little additional effort.

Given that, one can make the determination which is more appropriate for their given application.

What is this tutorial about?

In this tutorial, we will take a quick, introductory look at the data access features available in WebSphere Studio. To forewarn you, we will not end up with a very MVC-like application. WebSphere Studio will, of course, allow you to develop an unstable, non-MVC application, if you choose. We will, however, look at available tools, discuss the steps required for MVC compliance, and implement a few along the way. It will be up to you to ultimately build your applications to whatever level of compliance you feel suits your needs.

The application we'll build will access the SAMPLE database available from DB2 and run a few SQL commands against it, showing us how to select and display data, and update and delete records.

The main features we will use to access the database come from custom tag libraries available in WebSphere Studio and could also be coded completely by hand.

In addition, we will use the Data Access Web Pages and SQL Statement wizards to
facilitate some of the basic steps. If you were developing a real-world application, you would likely be required to do some additional coding to customize it to your specific needs. And certainly your HTML designers would want to come in and pretty up the site after the fact. We won't concern ourselves with that part for now.

Who should take this tutorial?

This tutorial is intended for developers who need to access data from back-end data sources, particularly those who would like to know when and whether it is appropriate to do so without EJBs.

It may also be helpful to developers who find themselves in occasional situations where quick and dirty data access is required, but who don't want to compromise too much stability and reuse.

Tools

- WebSphere Studio Application Developer
- DB2 UDB V8.1 Personal Edition

Install both WebSphere Studio and DB2 as directed and then load the SAMPLE database from DB2's First Steps.

About Data Access Beans

Data Access Beans are WebSphere Studio components designed to help developers access data more easily, and, well, they're beans. Thus, the name. Primarily, they take the form of custom JSP tags that can be used directly from JSPs. There are also controller and data beans that are generated in some circumstances for added functionality. We will use some of these.

Custom JSP tags are specific XML or HTML-like tags that have been developed to extend the functionality of a JSP page. Custom tags are made up of three components: the tag itself; an XML file with the extension .tld (Tag Library Descriptor) that functions as a library of tags that lists the available tags and details about them; and finally a tag handler Java class, which is what ultimately does the work.

The tag is used within JSP code that passes information to the tag handler class to
perform the task through a series of attributes or from text between opening and closing tags. Data may or may not be sent back to the client.

Say you had a custom tag that looked like this:

```
<mytag:highlight level="warning">I want this bold</mytag:highlight>
```

The text "I want this bold" is sent to the tag handler, which returns the text wrapped with the appropriate HTML tags like this:

```
<font color="orange"><b>I want this bold</b></font>
```

We will see custom tags used to both set information back on the server where no data is sent back, as well as to retrieve information, such as data from a specific table column.

You can always make your own custom tags -- that’s why they are called custom tags. However, WebSphere Studio comes with several sets of these tags that you can use in your JSPs, as we will in this tutorial. Below are some of the more useful tags that we will use:

- `<dab:dataSourceSpec/>` - specifies a data source to be used for connecting to a database
- `<dab:driverManagerSpec/>` - specifies connection information, but cannot utilize connection pooling like using a data source
- `<dab:select/>` - executes an SQL statement and processes its result set
- `<dab:sql/>` - specifies an SQL statement
- `<dab:modify/>` - executes an insert, update, or delete statement
- `<dab:setColumn/>` - sets the value of a column
- `<dab:repeat/>` - iterates; for example, through a result set
- `<dab:batch/>` - nests or groups several commands into one unit of work
- `<dab:deleteRow/>` - deletes a row in a result set produced using the `dab:select` action from the database
Section 2. Database Web Pages wizard

Overview

WebSphere Studio provides a convenient wizard for getting started on what we want to accomplish.

The Database Web Pages wizard generates a variety of resources that will access a data source, edit data, and display results. The files generated by the wizard will not likely be what you will end up deploying to your site without further customization. However, the wizard is a great starting point for developing a data access application.

For this tutorial, we won't be too concerned with the look and feel of the final pages. While the wizard has some limited functionality for editing the user interface, typically an HTML designer will be responsible for this task after the fact. We will simply accept most of the minimal look and feel given to us under the assumption that a more competent designer will come to our aid prior to final deployment.

In this section, we'll learn about:

• The basic functionality of the Database Web Pages wizard.
• The tag libraries that are packaged with WebSphere Studio and some of the tags within them.
• Some of the things you should keep in mind when trying to maintain a solid MVC-based application while taking advantage of this type of functionality.

Create a Web project

Let's give the Database Web Pages wizard a try.

1. Open WebSphere Studio by selecting Start => IBM WebSphere Studio => Application Developer.
2. Accept the default workspace and click OK.
3. Select Window => Open Perspective => Other => Web and click OK.

4. Right-click in the J2EE Navigator on the left and select New => Project to create a new Web project.

5. On the New Project dialog, select Web on the left and Web Project on the right, then click Next.
6. On the Define the Web Project page, type **DataAccessTutorial** for the **Project name**.

7. In the Web Project features list, check the tag libraries for accessing **JSP Objects** and **database access** then click **Finish**.
Run the Database Web Pages wizard

1. Expand **DataAccessTutorial** => **Web Content**.
2. Right-click **Web Content** and select **New => Other**.
3. On the Select page of the New dialog window, select **Web** on the left and **Database Web Pages** on the right, then click **Next**.
Choose the data access beans model

1. On the Create Database Web Pages page, make sure the Destination folder is set to /DataAccessTutorial/Web Content and leave the SQL Statement Type dropdown set to **Select Statement**, but note the other options available.

2. From the Model drop-down list, select the Java Beans choice **IBM Database Access Java Beans - Master Details Pattern** (not the Tag Library - Master Details Pattern).
Note: The other two options use custom JSP tags to access the database. For this section, however, we will use a JavaBean to execute our database call. Notice that you can see a description of each file used displayed in the File Description pane by selecting them individually in the Files list. This lets you see what types of files are generated when you select different options.

3. Click **Next**.

4. On the Choose SQL Method page, select the following options:
   - Be guided through creating an SQL statement
   - Connect to a database and import a new database model
Create the connection and define the SQL statement

1. On the Connection Page, make sure that the default settings are as follows:
   - Save Database Metadata should be checked.
   - Database should be set to SAMPLE.

2. Set the Userid and Password to something that will access the database. I used db2admin/db2admin.

3. Set the database type and drivers to DB2 and the Class location to the location of your db2java.zip (mine is at C:\Program Files\SQLLIB\java\db2java.zip).

4. Click **Connect to Database**.
5. On the Tables tab of the Construct an SQL Statement page, expand the **schema name** (which will likely be your username) and then expand **Tables**.

6. Select the Employees and Department tables and click the > button to move them over to the right-hand pane.
7. Click the Columns tab.

8. Expand both the Employee and Department tables and select the following columns by holding down the CTRL key:

   1. EMPNO
   2. FIRSTNME
   3. MIDINIT
   4. LASTNAME
   5. PHONENO
   6. DEPTNAME
   7. DEPTNO
9. Click > to move them all over to the right-hand pane.

10. Click the Joins tab.

11. Drag WORKDEPT in the Employee table and drop it onto DEPTNO in the Department table. This creates the join between the two tables based on the department number of the employee -- and, no, in case you were wondering, they don't need to both be checked.
12. Click the Order tab.

13. Expand the Department table, select **DEPTNAME** and move it over to the right panel to order the result set by department.

14. Click **Next**.

15. The SQL Statement Page displays the actual SQL statement, which can be tested by clicking **Execute** on this page, then the Execute button on the window that pops up. (Click **Close** to close the SQL test window.)

16. Click **Next**.

**Set up the driver manager**

1. On the Runtime Connection Page, select **Use driver manager connection**.
   Note: Using a data source connection will make control of creating the connection object the responsibility of the Web container. As a result, this approach takes advantage of connection pooling for better performance. Using the driver manager connection does not give us that ability. We will use the data source in a little bit, but for the purposes of this particular
exercise of the tutorial, let's see how the driver manager is set up.

2. Click **Next**.

3. On the View Bean Data page, select **Session**, so we can access our result set from multiple pages without looking the information back up from the database.
   Note: If this were data that could be changed by other users while our result set was stored in the session, you would not want to keep it in memory for long. Instead, you likely would re-collect the data every time you changed pages or needed to update the data, to ensure data integrity. The downside of this is, of course, extra trips to the database and, therefore, more time to process. It is your responsibility as you develop your application to weigh the benefits and problems in the relationship between performance and data integrity.

4. Select **Create a new Front Controller**.
   Note: A front controller is a servlet that is called just before the JSP file is accessed. This provides an opportunity to do logging or perform some other action like authentication when the data is being accessed by our little application. We will not do anything with this controller beyond look at how we might use one.
5. Click **Next**. The next few pages give us the opportunity to edit the look and feel of the input and result pages. We will edit only a few items.

**Design the JSPs**

1. On the Input Form page, simply click **Next**, as there is nothing we will change. Note the aspects of the page you can change here as well as in
the HTML after the fact.

2. We will change a few things on the Design the Master View page, which displays our initial result set. Notice the table headers. We can change these to something more readable.

3. From the Result set columns list, uncheck [schema_name].DEPARTMENT.DEPTNO, [schema_name].EMPLOYEE.MIDINIT, and [schema_name].EMPLOYEE.PHONENO.

4. Select [schema_name].EMPLOYEE.EMPNO from the Result set columns list.

5. Notice the Property/Value fields below the result set columns and change Label to ID. Note that in the preview pane the first column header got a little bit shorter and a little more readable.

6. Set its Key Value to true.
7. Select [schema_name].EMPLOYEE.FIRSTNME from Result set columns and change Label to First name.

8. Select the rest of the checked result set columns and change their labels to something more friendly. Don't reset the Key Value on anything other than the EMPNO column.

9. Click Next.

10. On the Details View page, leave all the fields checked, but revise the labels in the same manner as the previous page.
    Note: The Details page displays information specific to a single selected item from the initial result set. It is generally intended that a small subset of the collected data would be displayed on the Master page and further details would be displayed here. That is why we unchecked the middle initial and phone number on the Master page, but will display everything here. These generated files also prepare a method for calling this Details page by passing it a specific ID of one of the result set items.

11. Click Next.

12. On the Specify Prefix page, set the Prefix to select_. This is merely for ease of locating the files generated by this wizard. It makes no real difference what the prefix is; we will have select, update, and delete pages by the end.
13. Click **Finish**.

Examine the output

Take a look at the files that were generated for us. In the J2EE Navigator view, we can see that there are six files beginning with `select_` -- three Java files and three JSPs:

- **DataAccessTutorial/Java Source/select_Controller.java** - the controller where we can process data, do logging, or just pick our nose if we want to before the JSP file is displayed.

- **DataAccessTutorial/Java Source/select_MasterViewBean.java** - the bean that manages the full result set returned from the database.

- **DataAccessTutorial/Java Source/select_DetailsViewBean.java** - the bean that manages an individual item returned.

- **DataAccessTutorial/Web Content/select_InputForm.jsp** -
the JSP file that initiates this action. In our case, it is simply going to call the controller. However, if we were required to pass in some arguments, this page would include an HTML form to input data before submitting and calling the controller.

- `DataAccessTutorial/Web Content/select_MasterView.jsp` - the page that formats the result list data.

- `DataAccessTutorial/Web Content/select_DetailsView.jsp` - the page that formats the details page.

Open up the files to see how they use the data being passed around.

Test the application

Since WebSphere Studio comes with an application server built into it as a test
environment, we can run our application directly from our tool. To run anything in the test environment, we first need to configure a server instance. Luckily, when we do not have anything set up (as we do at this point) and we try to run something, WebSphere Studio prompts us and provides a wizard to configure a basic server.

For the application we now have, this simple server is all we need to see what we just built.

1. Right-click DataAccessTutorial/Web Content/select_InputForm.html and select Run on server.
2. Select **Create a new server**.

3. For Server type, select **WebSphere version 5.0 => Test Environment** and click **Next**.

4. Click **Yes** to create the new server project.

5. Click **Finish**.

6. The server will take a minute or two to finish loading. When it loads the input form, click the Submit link. Now you can see what I mean when I say this might not be the look and feel you will likely be shooting for in your final design!

7. On the Master Table View page, you can see the major listing returned from the database. To select an item for which to see a details view, select the radio button on that item and click the Details link at the top.

**Summary**

In this particular section, much of the work was handled by the SQL Statement and Database Web Pages wizards. Notice most importantly that we started with nothing -- no components, no beans, no JSP files, nothing. Just data we wanted exposed to users.

In just a matter of minutes we were able to generate resources, such as beans and JSP files, to connect to our data source and extract some specific information. In the next sections, we will see what specific controls we have at our disposal for accessing and using our data, but here we have been able to demonstrate the ease of setting up some basic connections from the user to the data source.

As I said before, the resulting files are not intended for any live site. This process does however create the core for what we will eventually need. At this point, we have created some front-end JSP files that have retrieved specific necessary data. At the very least, HTML developers and graphic artists will need to come in and clean up this mess. It is also likely that we will need to incorporate this data into some other process -- not just display on the screen, though it may be just that simple.
Section 3. Database update

Overview

Now that we had a look at the wizard that can generate resources needed to access a data source, let's look a little closer into the tag libraries that give us the functionality to customize our application beyond what the wizard will provide.

In this section, we'll quickly run the wizard again to generate resources to run updates on our employee table. The wizard will provide the initial JSP files, but we'll be looking for excuses to try out some of the custom tags to incorporate the select list we just generated with the new update files.

Since the focus in this section is more on the custom coding, we'll pay close attention to ways to make our JSP file work to access the database, while still maintaining the MVC underlying architecture.

Run the wizard again

1. Right-click **DataAccessTutorial/Web Content** and select **New => Other => Web => Database Web Pages**.
2. Click **Next** on the dialog window.
3. Change SQL Statement type to **Update Statement**.
4. In this case, we will simply be using the Database Access Tag Library Model.
5. Click **Next**.

6. Select **Be guided through creating an SQL statement**.

7. Select **Use existing database model** and click **Browse**.

8. On the Data resource selection dialog, select **DataAccessTutorial/Web Content/WEB-INF/databases/SAMPLE** and click **OK**.

9. On the Choose SQL Method page, click **Next**.

10. On the Tables tab of the Construct an SQL Statement page, select the Employee table and click the > button to make it appear in the Selected Table field.
11. Click the Update tab to specify which fields to update.

12. Enter :FNAME in the Value column of the [+FIRSTNAME] row. :FNAME is a host variable that will be dynamically replaced with a real employee first name on the fly.


<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+EMPNO</td>
<td>CHARACTER</td>
<td>:EMPNO</td>
</tr>
<tr>
<td>+FIRSTNAME</td>
<td>VARCHAR</td>
<td>:FNAME</td>
</tr>
<tr>
<td>+MIDINIT</td>
<td>CHARACTER</td>
<td>:LNAME</td>
</tr>
</tbody>
</table>

14. Click the Conditions tab.

15. In the Column column (yep, that's right), click in the top cell to display the drop-down list, then select [schema_name].EMPLOYEE.EMPNO.

16. In the Operator column, select =.

17. In the Value column, type in :EMPNO.

<table>
<thead>
<tr>
<th>Column</th>
<th>Operator</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[JEFFWILSON.EMPLOYEE.EMP...]</td>
<td>=</td>
<td>:EMPNO</td>
</tr>
</tbody>
</table>

18. Click Next.

19. If you want to execute a test, you will need to use an existing ID, such as 000200, and remember that all String values must be surrounded by
single quotes when prompted for the dynamic data (see below).

![Specify Variable Values Window]

20. Close the Execute window and click **Next** on the SQL Statement page.

21. Use the Driver Manager again and click **Next**.

22. Select **Do not use a Front Controller** on this set of files and click **Next**.

23. Select **EMPNO** and set the Input type property to **hidden** so the user cannot enter a new employee ID, but we can still pass the ID along.

24. And while we are here, you can make the labels a little friendlier, if you like, then click **Next**.

25. Click **Next** on the confirmation page.
26. On the Specify Prefix page, specify the prefix `update_`, and click **Finish**.

**Add the tag libraries to the JSPs**

The wizard provided us with a couple of pages to work from. We will need to make a few edits to incorporate these new files with those we generated in the previous exercise. In particular, the input form the wizard created is a static HTML form which we will make dynamic, including setting and using a data source JNDI name for performance enhancements that we, at least in our little tutorial, will surely never notice.

We will create an edit link from the result set we just ran to our new, dynamic form, which will load data from a given employee that we can then edit and submit.

We will also play with the code on the JSP file that executes the update to make it a little more MVC compliant. In particular, we will strip out some of the scriptlets the wizard put on the page and replace them with custom tags. If you're really sharp, you may notice that we won't make any significant performance enhancements here, but will use the excuse of removing scriptlets simply to see what kinds of tags are at our disposal.

1. First, change the `update_InputForm.html` file to a JSP by doing the following:
   1. Right-click `DataAccessTutorial/WebContent/update_InputForm.html` and select **Rename**.
   2. Change `.html` to `.jsp` and click **OK**.
   3. Click **OK** to fix the links to moved or renamed files.
2. Double-click `update_InputForm.jsp` to open it in the main editor.
3. Click the Source tab.
4. From the main menu, select **JSP => Insert Custom**.
5. On the Insert Custom dialog, click the Add button.

6. On the Select a Tag Library dialog, select jspsql in the Tag Libraries list. This gives us tags for accessing data sources.

7. Set the Prefix to dab.

8. Select http://Jakarta.apache.org/taglibs/request-1.0. This gives us tags for accessing attributes and functions of the request object.

9. Set the Prefix to req.

10. Make sure both tag libraries are checked and click OK.
11. On the Insert Custom Tag dialog, click **Close**.
    Note: The calls to the tag libraries were added to the top of the JSP source.

**Edit the code**

In other places in this tutorial we have purposely avoided "optimizing" our code in the interest of time. In this section you may notice that we go one step further and make an edit or two that will actually worsen the performance -- not that there will be any way of noticing that in this tutorial.

It may be necessary to point out that in general the parameterized variables in SQL statements we will be building have some benefits to performance that we will, well, to be honest, break. We want to take a moment to show you how another set of tags can be used. The tag library we will be using gives us access to certain request information that does not require scriptlets.

However, for reference, the SQL statements themselves do have a couple of important benefits. For one thing, using parameterized variables as you will see generated by the wizards means that the database can prepare the statement once and pass in dynamic content. Not requiring the database to prepare a new statement every time from scratch can save processing time which can make a big difference
under larger loads. Another thing to consider is that statements can also simply be cached by the database and therefore can be used over and over again completely.

While this might not be the most appropriate place to use our "request" tag library, it is ultimately something that can be used to simplify accessing information from the passed-in request object. Generally speaking, this should be considered a better method for accessing that information.

1. Locate the following block of code that begins the form in the JSP file:

```html
<!--Navigation-->
<FORM name="myForm" method="POST" action="update_UpdateView">
<A href="javascript:submitForm('update_UpdateView')" >Submit</A>
</FORM>
</form>
```

2. Just below it, add the following JNDI reference (in case you are wondering, we will create this JNDI object in the server configuration in a moment):

```html
<dab:dataSourceSpec dataSource="jdbc/sample" id="update_Connection" />
```

3. Below the JNDI reference, add the following to find the specified employee data (where `[schemaName]` is the schema name on your computer -- likely your username):

```html
<dab:select id="update_master" scope="session" connectionSpec="update_Connection">
    <dab:sql>
        SELECT [schemaName].EMPLOYEE.FIRSTNME, [schemaName].EMPLOYEE.LASTNAME, [schemaName].EMPLOYEE.EMPNO
        FROM [schemaName].EMPLOYEE
        WHERE [schemaName].EMPLOYEE.EMPNO='<req:parameter name="EMPNO"/>
    </dab:sql>
</dab:select>
```

Note: The `<req:parameter />` tag is the equivalent of `request.getAttribute()`. We will be passing the employee ID from the main select list to this page and will use it to pull other employee data to pre-populate the form.

4. Locate and revise the following:

```html
<INPUT NAME="EMPNO" TYPE="hidden" VALUE=""/>
```

To
Note: Be careful of conflicting quotes! Notice that once we nest a custom tag in the value attribute, we must change the value attribute double quotes to single quotes.

The `<dab:repeat>` tag will iterate through a result set. Here, we just put in the opening tag. The specific result set is identified by the `name` attribute. Notice the `name` value is set to the `id` attribute of the select block above it. You may have multiple selects on a given page to iterate through. The closing tag will be added below the input fields. This way we have one block of input elements for every employee returned, though in this case we should end up with a single user per ID.

The `<dab:getColumn />`, as you might expect, retrieves the data from a specified column, identified by a `colName` attribute. If you are dynamically scrolling through columns and you do not know the specific column names, you can also identify them as indexed fields, referred to with an `index` attribute.

5. In the same way we edited the EMPNO input field `value` attribute, add the `<dab:getColumn>` calls to the `value` attributes of the FIRSTNME (note the spelling) and LASTNAME input fields to pull in the appropriate columns. FIRSTNME and LASTNAME are the column names in the result set.

6. Locate the following HTML tags: `</TABLE></FORM>`.

7. Put `<dab:repeat>` between the closing table and form tags. Note: Placement is VERY important. We need the entire table in which the fields are nested to repeat for every employee, but we do not want to repeat the closing form tag.

8. Save (CTRL+S) and close the file.

9. Double-click `update_UpdateView.jsp` to open it in the main editor.

10. From the main menu, select JSP => Insert Custom.


12. Select the `request-1.0` tag library and enter `req` for the Prefix.

13. Click OK.
14. Click **Close**.

15. There are several things here that we could tweak, but in the interest of time, locate the `<dab:modify>` tag. Notice that we will run the update with an SQL update statement chock-full of host variables and that those variables are set with the `<dab:parameter>` tags. The only issue with this method is that there is no convenient way to avoid setting the dynamic value attributes without a scriptlet. The ones here are defined above and their values are simply pulled in from the request object. What we can do is modify the SQL statement to use `<req:parameter>` tags instead of the host variables, thus avoiding the use of scriptlets.

16. Replace :EMPNO with `<req:parameter name="EMPNO"/>`. Make sure you include the single quotes.

17. Replace :FNAME with `<req:parameter name="FIRSTNME"/>`.

18. Replace :LNAME with `<req:parameter name="LASTNAME"/>`.

19. Delete the `<dab:parameter>` tags.

20. Add the following code below the line that reads, **Number of rows updated:** to return to the master result set of employees:

   <p> <a href="select_MasterView.jsp">Show all employees</a> </p>

21. Save (CTRL+S) and close the file.

22. Open `select_MainView.jsp` and view its source.

23. Locate and revise the following line from (replace `[schemaName]` with your `schemaName` -- likely your username):

   `<%=masterViewDBBean.get[schemaName]_EMPLOYEE_EMPNO()%>`

   to (put all this on one line):

   `<a href="update_InputForm.jsp?EMPNO=
   <%=masterViewDBBean.get[schemaName]_EMPLOYEE_EMPNO()%>"/>
   <%=masterViewDBBean.get[schemaName]_EMPLOYEE_EMPNO()%></a>`

24. Save (CTRL+S) and close the file.
Set up the server

Before we can run our latest changes, we need to edit some server configurations to specify the JNDI name we want to access.

1. In the J2EE Navigator, click on the Server Configuration tab and locate **Servers/WebSphere v5.0 Test Environment**.

2. Double-click **WebSphere v.5.0 Test Environment** to open it in the main editor.

3. Click on the Security tab.

4. Click the Add button for the JAAS Authentication Entries.

<table>
<thead>
<tr>
<th>Alias</th>
<th>User ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database user alias</td>
<td>db2admin</td>
<td>alias used for database access security</td>
</tr>
</tbody>
</table>

5. Enter the following information:
   - **Alias**: Database user alias
   - **User ID**: db2admin (or other user who can access the database)
   - **Password**: db2admin (or other user who can access the database)
   - **Description**: alias used for database access security

6. Click on the Data source tab.
7. Scroll down to Server settings.

8. In the JDBC provider list, select the Default DB2 JDBC entry.

9. Next to the Data source defined list, click Add.

10. Select the Version 5.0 data source and click Next.

11. On the Modify Data Source page, enter Data Access data source in the Name field and jdbc/sample in the JNDI name field. Then choose com.ibm.websphere.rsadapter.DB2DataStoreHelper from the Data source helper class name drop-down list and Database user alias in the Component-managed authentication alias drop-down list.
12. Click **Next**.

13. Make sure **sample** is specified as the value of the Value field.

14. Click **Finish**.

15. Save (CTRL+S) and close the server configuration editor.

16. At the bottom of the tool, click on the Servers tab, then right-click the server and select **Stop**.
Test the update pages

1. On the Sever Configuration view, click the J2EE Navigator tab.

2. Right-click on `select_InputForm.jsp` and select Run on server.

3. Select Use an existing server, make sure that WebSphere v5.0 Test Environment is selected, and click Finish.

4. Click the Submit link.

5. Select one of the records to edit by clicking one of the ID hyperlinks that now point to our updated `update_InputForm.jsp` page (passing the ID in the query string).

<table>
<thead>
<tr>
<th>Select</th>
<th>ID</th>
<th>Department</th>
<th>First name</th>
<th>Last name</th>
</tr>
</thead>
<tbody>
<tr>
<td>☑</td>
<td>000030</td>
<td>INFORMATION CENTER</td>
<td>SALLY</td>
<td>KWAN</td>
</tr>
<tr>
<td>☑</td>
<td>000130</td>
<td>INFORMATION CENTER</td>
<td>DOLORES</td>
<td>QUINTANA</td>
</tr>
<tr>
<td>☑</td>
<td>000140</td>
<td>INFORMATION CENTER</td>
<td>HEATHER</td>
<td>NICHOLLS</td>
</tr>
<tr>
<td>☑</td>
<td>000060</td>
<td>MANUFACTURING SYSTEMS</td>
<td>IRVING</td>
<td>STERN</td>
</tr>
<tr>
<td>☑</td>
<td>000150</td>
<td>MANUFACTURING SYSTEMS</td>
<td>BRUCE</td>
<td>ADAMSON</td>
</tr>
<tr>
<td>☑</td>
<td>000160</td>
<td>MANUFACTURING SYSTEMS</td>
<td>ELIZABETH</td>
<td>PLANKA</td>
</tr>
</tbody>
</table>

Note: If you did the previous steps correctly, the first and last name input fields should now display the employee’s name.

6. Edit the name and click the Submit link.

7. One record should have been updated.

8. Click the Show all employees link to see the changes we made.

Summary

In this section, we were able to take some time to play in the code and get a feel for some of the components available in the custom tag library used for our data access beans.

While we only made a few simple programmatic additions, you can see that we have plenty of flexibility outside of what is managed by the various wizards.
Section 4. Database delete

Overview

Now that we have been exposed to both the wizard and some of the custom tags available, let's try one more exercise to play with a few more of the tags and concepts. In this final section, we'll run a couple of delete actions on our dataset. What will be interesting with this run is setting up a transaction and the benefits of using a <dab:batch> tag set.

If you are not familiar with transactions, the concept is rather simple and straightforward. A transaction is considered to be a single unit of work, even if multiple processes are run. If one of the processes fails, then everything should be rolled back to its original state. Committing all changes happens only after everything is considered to have completed successfully.

The classic example is that of transferring money from your checking account to your savings account. If you move $1000 dollars from one account to another, the full process would entail subtracting $1000 from one account and adding the same amount to the other. If one of those processes failed and the other account wasn't rolled back, well, that could be either really good or really bad, depending on your perspective.

With EJBs, one can let the container manage the transaction -- that is, verifying successful process completion, committing changes, and rolling back if necessary. When we are using custom tags, the <dab:batch> tag will give us this ability.

The batch tag has a couple of other benefits. First, a batch tag specifies the connection object that database calls use. Wrapping several calls to the database in a batch tag can share that same call to the object. Exciting, isn't it?

Perhaps a little more useful is that the batch holds off on the final commit until the entire process is complete. For example, executing a delete re-indexes the cached result set.

If you were scrolling through a result set, deleting some of the rows without a batch tag, the delete call would re-index the result set you were scrolling through.

That means that if you need to delete the third AND fourth record, when you commit
the delete of the third, the entire set is re-indexed and the fourth record moves up into the third position. The looping would move you to the fourth position on the record that just a moment before was in the fifth, effectively skipping over the item you wanted to delete. Confused? Just wrap it in a batch and forget about it!

By the way, insert, update, and delete calls wrapped in a batch tag can have their processing deferred as described above, but the modify tag that we used in the last exercise cannot be. It can be included -- it just is not deferred. Something to keep in mind.

Wrapping that deletion loop in a batch means it waits until the end to commit all the deletes and, therefore, does not move things around prematurely.

Add the delete pages

Let's have a little fun, shall we? In the spirit of our less than secure industry, let's wipe out an entire division of workers.

What we'll do is delete both a given department from the DEPARTMENT table, as well as all the employees in the EMPLOYEE table.

The transaction will ensure that if something goes haywire in deleting the data in one or the other tables, the other is spared -- that is, until we can fix the bug that jammed the slaying.

1. In the J2EE Navigator view, right-click update_UpdateView.jsp and select Copy.
2. Right-click it again, select Paste, and rename the file to delete_DeleteView.jsp.
3. Next, we need to set the same configuration settings for the delete JSP file that the update JSP file we just copied uses.
   2. Click on the Source tab. We could use the GUI, but since we are just copying a block of XML, the source view is a bit easier.
   3. Locate the <servlet> block where the servlet-name is update_UpdateView.
4. Copy and paste that block as a new block of XML.

5. Change the servlet-name, display-name, and jsp-file to reflect the delete_DeleteView.jsp.

6. Locate the servlet-mapping block for the update_UpdateView. Copy and edit that in the same way as for delete_DeleteView.

7. Save (CTRL+S) and close web.xml.

4. Open delete_DeleteView.jsp to its source view.

5. Revise the line:

   <H1>Update Form</H1>

   to read:

   <H1>Delete Form</H1>

6. Locate the <dab:driverManagerSpec> block and change the id from update_Connection to delete_Connection.

7. Revise the line:

   dsSpec = update_Connection;

   to read:

   dsSpec = delete_Connection;

   Note: I know, I know, this is a scriptlet! In the interest of time, We are only going so far to clean up our code. Good MVC says no scriptlets in JSP files.

8. Delete the <dab:modify> block and replace it with the following select statement (changing [schemaName] to your schema name -- likely your username):

   <!-- GET ALL DOOMED EMPLOYEES WITH SPECIFIED DEPARTMENT NUMBER -->
   <dab:select id="get_doomed_employees" connectionSpec="delete_Connection">
     SELECT * FROM [schemaName].EMPLOYEE
WHERE [schemaName].EMPLOYEE.WORKDEPT='<req:parameter name="DEPTNO"/>
</dab:sql>
</dab:select>

9. Add another select statement below it to delete from multiple tables using this select statement block:

<!-- GET SPECIFIED DOOMED DEPARTMENT -->
<dab:select id="get_doomed_department" connectionSpec="delete_Connection">
  <dab:sql>
    SELECT * FROM [schemaName].DEPARTMENT
    WHERE [schemaName].DEPARTMENT.DEPTNO='<req:parameter name="DEPTNO"/>
  </dab:sql>
</dab:select>

10. Add the following <dab:batch> block that will scroll through both record sets deleting rows willy-nilly:

<dab:batch connectionSpec="delete_Connection">
  <!-- dab:repeat SCROLLS THROUGH RESULT SET NAMED get_doomed_employees -->
  <dab:repeat name="get_doomed_employees" over="rows">
    <!-- dab:repeat DELETES CURRENT ROW AS WE SCROLL THROUGH -->
    <dab:deleteRow/>
  </dab:repeat>
  <!-- dab:repeat SCROLLS THROUGH RESULT SET NAMED get_doomed_department -->
  <dab:repeat name="get_doomed_department" over="rows">
    <!-- dab:repeat DELETES CURRENT ROW AS WE SCROLL THROUGH -->
    <dab:deleteRow/>
  </dab:repeat>
</dab:batch>

11. Finally, delete the line that outputs the update:

Number of rows updated:  
<jsp:getProperty name="update_statement" property="updateCount"/>

12. Save (CTRL+S) and close the file.

13. Open select_MasterView.jsp again so we can put the death link on it.

14. Locate the table cell that displays the Department name:

<%=masterViewDBBean.get[schemaName]_DEPARTMENT_DEPTNAME()%>

15. Add the following below that line (put this all on one line):
16. Save (CTRL+S) and close the file.

Test the delete pages

1. In the Servers view, right-click the server instance and select Restart to pick up the changes we made to the web.xml file.

2. When the console reads: Server server1 open for e-business, right-click select_inputForm.html and select Run on Server.

3. Click the Submit link.

4. Click the Bye Bye link. (Make note of the department name.)

5. When the delete_DeleteView.jsp page loads, it won’t look too terribly impressive.

6. Click the Show all employees link to see what damage we did. (Hint: the department we just whacked and all the employees in it should no longer be there.)

Summary

In this case, we simply copied the update page, since we knew that much of the
code we needed was already on it. We only needed to tweak it to delete instead of update.

This exercise gave us an opportunity to test the batch tag to set up a transaction. It is likely that everything deleted as expected, but had something gone wrong with one part of the deletion process, everything would have been rolled back to its original state. That isn't to say we handled that in any way so that the user will have errors thrown back, but at least we can be relatively certain that the data in database is safe and sound.

Section 5. Conclusion

What we built

What we eventually built was, as I said from the beginning, not a finely-tuned, well-coded, particularly attractive application. The purpose of this exercise was simply to demonstrate the functionality of the data access tags and wizards available in WebSphere Studio.

While some of what we built may seem a little complex the first time through, I encourage you to experiment further with the various tags and other tag libraries available for data access (as well as for other controls such as XML and user interface functionality). They are intended specifically for quick and easy development and can be a great help in application development.
Resources

Learn

- More fun connection to databases: Creating a Web Application using the Data Access Beans in WebSphere Studio V5
- Another perspective on data access from JSPs: Dynamic Web-based data access using JSP and JDBC technologies
- In case you want to know how to do it (properly) with EJBs: Easy, breezy EJB tooling in WebSphere Studio
- The full dragonSlayer story: The Go-ForIt Chronicles: Memoirs of eXtreme DragonSlayers
- Stay current with developerWorks technical events and webcasts.

Get products and technologies

- Download DB2 UDB from the developerWorks downloads page.
- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in the discussion forum for this content.

About the author

Jeff K. Wilson

Jeff Wilson, a self-proclaimed dot com refugee, has for the past two years been an e-business architect for the DragonSlayers, IBM’s Developer Relations Technical Consulting team in Austin, Texas. It is their goal to excite, evangelize, educate, and enable developers on the latest tools and technologies available. Jeff welcomes any and all questions, comments, recipes, insider stock tips, cash, prizes, and good juicy gossip. He can be reached at wilsonje@us.ibm.com. See Resources for more articles from the DragonSlayer team.