DATABASE ACCESS FROM JAVASERVER PAGES

Julie Basu, Oracle Corporation

1. INTRODUCTION

This paper examines the issues in accessing the Oracle database from *JavaServer Pages*, also known as JSPs[1]. JSP is a recent language specification developed by Sun Microsystems (cooperatively with other software companies including Oracle) to allow the generation of dynamic content in the HTML pages of a web application. For example, a JSP may perform a query against the database and report the results as an HTML table. The JSP translator converts JSP files to *servlets*, which can be executed on a webserver that supports a servlet runner. Once deployed, a JSP can be invoked from a browser via an http URL to return the dynamically generated page.

The industry-leading Oracle8*i* database products provide integrated and comprehensive Java support[1], including support for execution of servlets and JSPs. JSPs can be run on the Oracle Application Server, on WebDB, and on upcoming releases of Oracle Lite and the JServer. Access to object-relational data in SQL tables is provided through JDBC and SQLJ, which are standard frameworks for database connectivity in Java. Our focus in this paper is on accessing the database from a JSP using JDBC and SQLJ, with emphasis on resource management and formatting of query results.

The paper is organized as follows. In Section 2 we give an overview of a web application and the JSP execution model, and illustrate the steps in writing basic JSPs. Section 3 provides a brief review of JDBC and SQLJ. In Section 4 we discuss the programming and performance issues for JSPs that interact with the database. Detailed code samples are used to explain the various strategies, including the usage of JavaBeans. Section 5 deals with formatting of SQL query results and explains how XML output can be dynamically generated in a JSP using the Oracle XML-SQL and related utilities. We conclude with a technology preview of Oracle's JSP extensions, and of the embedded web server hosted by Oracle JServer that will allow servlets and JSPs to be executed on the Java Virtual Machine running inside the database.

2. CONCEPTS AND TERMS

In this section we present an overview of the basic concepts and terms that will be used in the paper.

The Internet has spawned the concept of web applications with browser-based interfaces. These application employ simple point-and-click graphical user interfaces to successfully hide much of their complexity. However, their goal is not just ease of use but wide and uniform accessibility across different types of machines and networked geographical areas. New and innovative technology has prompted the rapid adoption of this elegant yet powerful class of applications. Java plays an important role in this area – it is the language of choice for developing web-based applications. Likewise, HTML (Hyper Text Markup Language) is the standard annotation scheme for presentation of web content to the user. JSPs are right in this programming space – they allow web output to be generated on the fly using HTML with Java scripting. They are based on the servlet programming model [4]. Before we look at JSPs, let us delve a little more into what constitutes a web application.

2.1 WEB APPLICATION ARCHITECTURE

The central entity in web applications is the web server. The basic idea of a web server is quite simple – it understands the HTTP request-response protocol and returns *pages* that are requested by the user through an HTTP URL. Around this simple concept have grown powerful Java-based technologies such as applets, servlets, and JSPs. Applets exploit the *write-once-run-anywhere* design philosophy of the Java language, downloading code from the web server into the local browser for execution and presentation. In contrast, servlets and JSP are executed by the web server upon invocation through a URL. The generated results are returned to the browser as part of the response to the HTTP request. Servlets

[1] *JavaServer Pages are not to be confused with Java Stored Procedures, which are often referred to with the same JSP acronym.*

and JSPs are relatively new technologies but they have gained quick popularity due to their flexible yet powerful programming model.

In addition to static files with predetermined content, a web application may contain a mixture of dynamic pages generated using servlets, JSPs that are compiled to servlets before execution, other types of dynamic pages such as DHTML, plus stylesheets for presentation, and supporting Java libraries. Figure 1 shows the general setup of a web application.

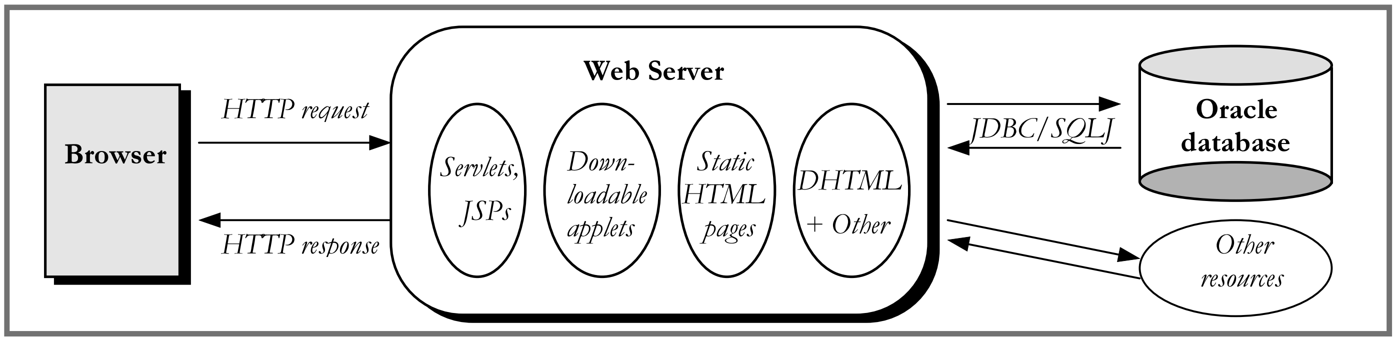


Figure1. The Architecture of a Web Application

2.2 WHAT IS A JAVASERVER PAGE(JSP) ?

The JSP programming model allows web content to be generated dynamically during program execution through Java *scriptlets*, declarations, and expressions interleaved with the static content in a HTML page [1]. A scriptlet is a block of Java code that is run as part of the servlet that is generated from JSP translation. Compared to servlets, JSPs provide a convenient shorthand, making them simpler and easier to author. Their main intent is to support clean separation of content generation and presentation logic. In particular, the integration of reusable modular components such as JavaBeans is emphasized.

As we shall see later in this paper, emerging standards like XML and XSL [9] are usable within the JSP framework. In contrast to competing technology such as the Active Server Pages from Microsoft Corporation, the JSP model is based almost entirely on Java. The specification does not rule out the possibility of other scripting languages but they are all required to support manipulation of Java objects, invocation of methods on them, and handling of Java exceptions. We will see an example of such an extension later in our paper to permit the use of SQLJ in scriptlets.

Without further ado, let us look at some examples of JSP code. We will start with a very basic page and move on to more advanced usage.

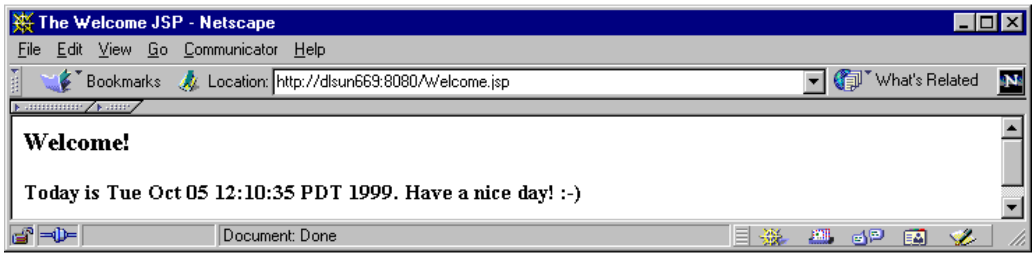
2.2.1 A SIMPLE JSP

Let us write a very simple JSP named `Welcome.jsp` to print out a greeting and the current date.

```
<HTML>
<HEAD> <TITLE> The Welcome JSP  </TITLE> </HEAD>
<BODY  BGCOLOR=”white”>

<H3> Welcome! </H3>
<P><B> Today is <%= new java.util.Date() %>.  Have a nice day! :-) </B></P>
</BODY>
</HTML>
```

Figure 2. A Simple JavaServer Page Welcome.jsp

The first three lines of the Welcome JSP are usual HTML code that define the title and body color of the page. Following the welcome greeting is a new paragraph that prints the date and current time using a Java expression within the `<%=` and `%>` tags. This expression creates a new instance of the `java.util.Date` type which is set to the current time by the Java runtime when the page executes. The result of this expression is coerced to a `String` and returned tothe browser as part of the HTTP response. Below is the result of invoking the Welcome JSP from the Netscape browser:


The Welcome JSP - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://dlsun669:8080/Welcome.jsp
What's Related
Document: Done

Welcome!

Today is Tue Oct 05 12:10:35 PDT 1999. Have a nice day! :-)

Figure3. Output of running the Welcome JSP

2.2.2 Using Http Request Parameters

It is often the case that a dynamic page needs to use a parameter that is entered interactively by the user on the screen. This data can be passed to the JSP as part of the HTTP request. Let us now write a JSP `WelcomeUser.jsp` that prints out a greeting for such a user name parameter in the HTTP request. Figure 4 shows what such a JSP could look like.

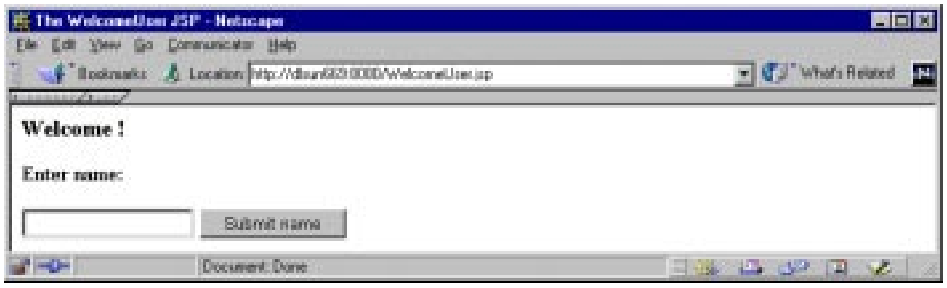
In this JSP we are using an HTML form to enter the name of the user. When the "Submit name" button is clicked, the form invokes the GET method of the same JSP. The input string (if any) is assigned to a parameter named `user` and passed to the JSP via the HTTP request object. This object is named `request` and is implicitly available in a JSP. It provides a method `getParameter(String paramName)` for retrieving parameter values by their name. The WelcomeUser JSP calls this method to read the value of the parameter `user`, and if the returned value is not null it prints out a welcome message with the specific user name.

Figure 5(a) shows the result of invoking this JSP with no parameters – it prints a welcome message without a user name since none was supplied in the HTTP request. If now a value 'Julie' is entered in the input field and the 'Submit name' button is clicked, the WelcomeUser JSP returns the output shown in Figure 5(b). Notice in Figure 5(b) that the URL includes the value of the name parameter that was entered in the HTML form. Since the method associated with the form is GET, the HTTP URL is augmented with ? followed by key-value pairs representing the form fields, in this case just `user`.

```
<HTML>
<HEAD> <TITLE> The WelcomeUser JSP  </TITLE> </HEAD>
<BODY  BGCOLOR="white">

<% String user = request.getParameter("user"); %>
<H3> Welcome  <%=  (user == null)? "" : user %> ! </H3>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name");
</FORM>
</BODY>
</HTML>
```

Fig 4. The WelcomeUser JSP uses a parameter in the HTTP request object

(a) First invoked without parameters

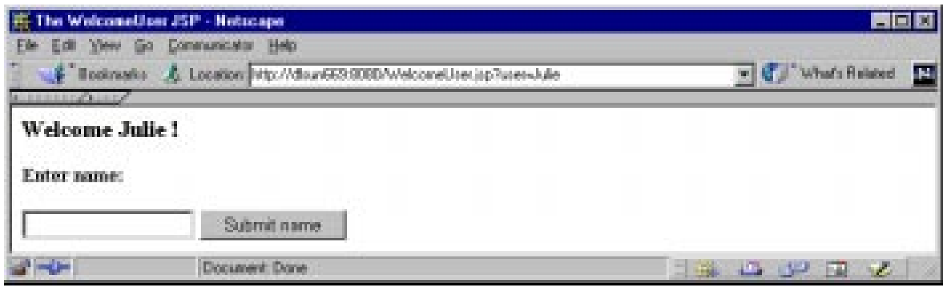

The WelcomeUser JSP - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://dlsun663:8080/WelcomeUser.jsp
What's Related
Document: Done

Welcome !

Enter name:

Submit name

(b) Next hit after after submitting the name 'Julie' through the form


The WelcomeUser JSP - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://dlsun663:8080/WelcomeUser.jsp?user=Julie
What's Related
Document: Done

Welcome Julie !

Enter name:

Submit name

Figure 5. Output of WelcomeUser JSP

2.2.3 Using JavaBeans and Lifecycle Scoping

JSP supports the use of the `<jsp:usebean>` tag to invoke a modular program entity known as a JavaBean. JavaBeans function as reusable elements of *component programming* [6]. A JavaBean is a Java program that conforms to certain design rules with well-defined semantics that permit dynamic discovery and manipulation of the bean. For example, a bean can have *properties* with accessor methods. By default, accessor methods can be associated with their respective properties simply by their naming convention – accessors for a property `int x` are named `getX()` and `setX(int newX)`. Such implicit rules as well as the facility for explicitly providing information about a bean through a `BeanInfo` class support well-defined behavior of the bean. This information can be effectively used by environments in which the bean is embedded.

In the Java file below we have defined a very simple bean called `NameBean` with a single String-valued property `newName` and its `get` and `set` accessor methods that conform to the implicit naming rules. The bean lives in a Java package called `mybeans`.

```
package mybeans;
public class NameBean {
  String newName="";
  public void NameBean() {  }

  public String getNewName() {
    return newName;
  }
  public void setNewName(String newName) {
    this.newName = newName;
  }
}
```

The JSP framework supports easy invocation of such a bean through the `<jsp:usebean id=… scope=… />` tag. This tag specifies an identifier for the bean instance and optionally its *lifetime*, i.e., the duration of its availability, through an attribute named `scope`. This attribute can have one of four different values: `page`, `request`, `session`, or `application`, with the default value being `page`. If the scope is `page`, then an instance of the bean is created when the usebean command is encountered in the page and its reference is associated with the identifier given in the id attribute. Reference to the bean is no longer possible once execution of the page body has been completed. Re-invoking the page will create and associate with the identifier an entirely new instance of the bean. Likewise, the session scope specifies that the bean is associated with the HTTP session that causes the JSP page to be invoked, and it is available for use through the given identifier until the HTTP session is explicitly terminated or implicitly times out.

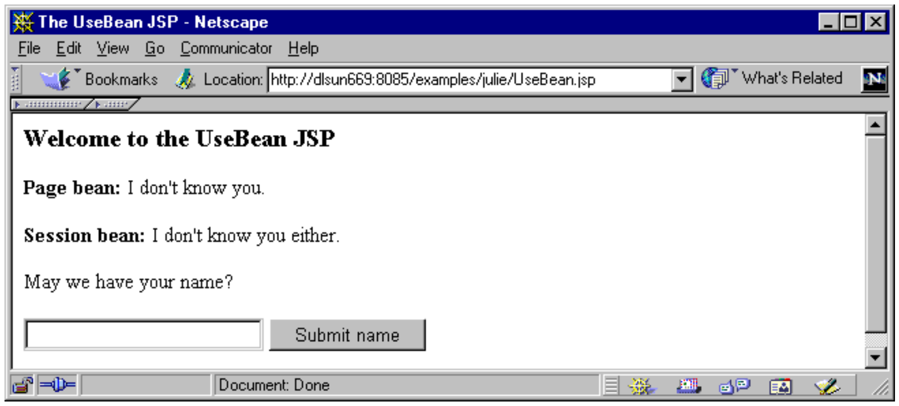
Using JavaBeans in a JSP allows clear separation of Java logic that generates dynamic content and its presentation. Let us see how the NameBean we defined above can be used in a JSP. The intent of the example is to illustrate not only the use of modular beans but also the different scoping options for their lifetime. Hence we have two usebean tags that use NameBean in two different scopes, namely page and session. Following the two usebean commands are <jsp:setproperty> tags that set the properties of the specified beans from the HTTP request parameters. The property="*" clause implies that the properties of the bean will be set to the HTTP request parameter values by matching their names. In our case, the HTML form field is called newName, which is a parameter in the HTTP GET request, and it is no coincidence that its name matches the property newName of NameBean.

```
<%@ page import="mybeans.NameBean"  %>

<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
<jsp:setProperty name="pageBean" property="*" />

<jsp:useBean id="sessionBean" class="mybeans.NameBean" scope="session" />
<jsp:setProperty name="sessionBean" property="*" />

<HTML>
<HEAD> <TITLE> The UseBean JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">

<H3> Welcome to the UseBean JSP </H3>
<P><B>Page bean: </B>
<% if (pageBean.getNewName().equals("")) { %>
  I don't know you.
<% } else { %>
  Hello <%= pageBean.getNewName() %> !
<%  } %>

<P><B>Session bean: </B>
<% if (sessionBean.getNewName().equals("")) { %>
  I don't know you either.
<% } else {
       if ((request.getParameter("newName") == null) ||
           (request.getParameter("newName").equals(""))) { %>
         Aha, I remember you.
<%     } %>
       You are <%= sessionBean.getNewName() %>.
<% } %>

<P>May we have your name?
<FORM METHOD="get">
<INPUT TYPE="TEXT" name="newName" size=20>
<INPUT TYPE="SUBMIT" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

Figure 6. The UseBean JSP with page-scoped and session-scoped beans

Let us now examine what happens when this JSP page is invoked. The behavior is shown in Figure 7. In case (a) the JSP is invoked without any arguments. The usebean tags cause instances of both beans to be created. However, there is no matching HTTP parameter for their `newName` property and hence calls to their `getNewName()` method returns the initial value of the empty string. Hence both beans are unaware of the name of current user. Next, suppose that I enter the name Julie through the HTML form and submit the request. The properties of both the page bean (but a different instance!) and the same session bean are set from this input value, and both beans output a greeting message with the supplied user name. In case (c) I hit the same page again with no arguments after a few seconds but within the limit of session timeout (a configuration parameter of the webserver). The new instance of the page-scoped bean is unaware of the user name but the note that *the session-scoped bean has retained the setting of its newName property from the previous page hit!*

(a) First invoked with no parameters


The UseBean JSP - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://dlsun669:8085/examples/julie/UseBean.jsp
What's Related
Submit name
Document: Done

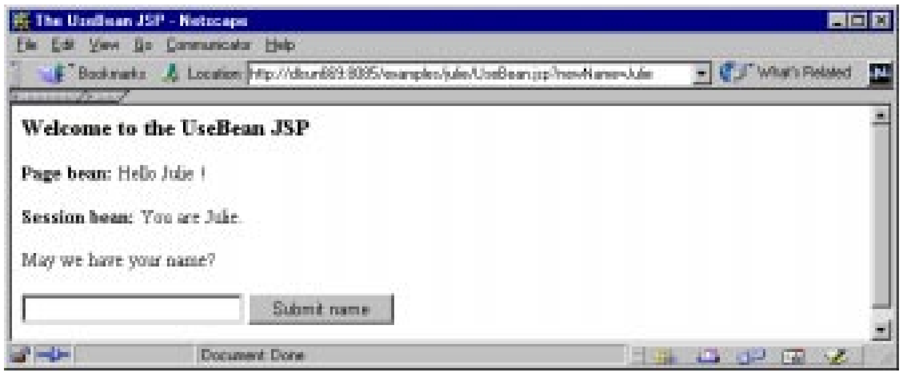
Welcome to the UseBean JSP

Page bean: I don't know you.

Session bean: I don't know you either.

May we have your name?

(b) After entering user name 'Julie'


The UseBean JSP - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://dlsun669:8085/examples/julie/UseBean.jsp?newName=Julie
What's Related
Submit name
Document: Done

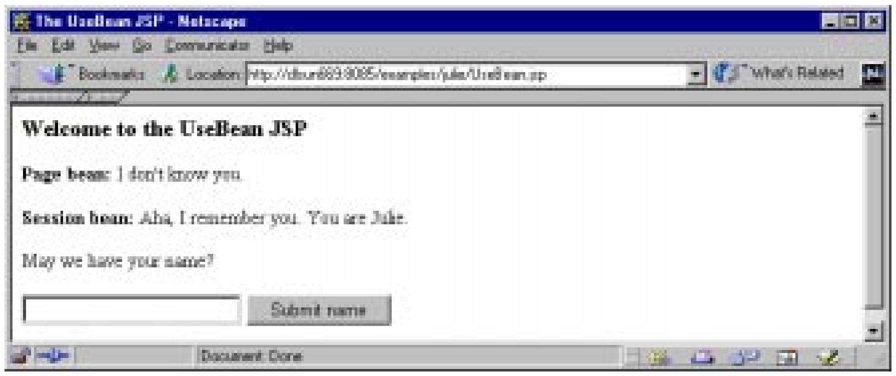
Welcome to the UseBean JSP

Page bean: Hello Julie !

Session bean: You are Julie.

May we have your name?

(c) Again invoked with no parameters within the same HTTP session.


The UseBean JSP - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://dlsun669:8085/examples/julie/UseBean.jsp
What's Related
Submit name
Document: Done

Welcome to the UseBean JSP

Page bean: I don't know you.

Session bean: Aha, I remember you. You are Julie.

May we have your name?

Figure 7. Output of page-scoped and session-scoped beans in the UseBean JSP

3. Database Access Schemes in Java

In this section we will provide a brief overview of the database access mechanisms in Java. Two different frameworks, JDBC and SQLJ, are available for connecting to database in a Java program.

3.1.1 JDBC

The JDBC specification from Sun Microsystems defines a set of interfaces for SQL-based database access from Java. JDBC is a call-level *application program interface* (API), which means that SQL statements are executed on a database by calling methods in the JDBC library from the Java program. Database vendors can provide different implementations of the JDBC APIs for their databases. For example, Oracle provides three JDBC *drivers* [3], namely the JDBC-OCI driver which uses the client-side OCI installation to interact with the Oracle database, the JDBC thin driver that is written purely in Java and communicates directly with the database using Java sockets over TCP/IP, and a JDBC server-side driver packaged as part of the Oracle 8*i* JServer for executing Java stored procedures inside the database. Which JDBC driver is appropriate for the application depends on the deployment requirements. The important point is that all JDBC drivers support the same standard set of interfaces, thereby promoting portability across different JDBC drivers as well as databases.

The JDBC programming model is based on ODBC. Interfaces defined in the `java.sql` package represent database connections (`java.sql.Connection`), statement execution handles (`java.sql.Statement`), and query result sets (`java.sql.ResultSet`), among others. Resultsets provide row-by-row access to the results returned by a SQL query. We will not provide details of JDBC programming here; it is assumed that you are familiar with these basic JDBC concepts.

3.1.2 SQLJ

SQLJ is a recent ANSI standard for embedding static SQL statements directly in Java code [5]. The mixed code is converted to Java by the SQLJ translator, and can be executed on a database using the SQLJ runtime library and an underlying JDBC driver. Oracle is a major participant in the design and development of SQLJ, and supports SQLJ wherever JDBC runs.

SQL statements in SQLJ are *static*; that is, they must be known at program time and cannot change as the program executes. Data values passed to SQL operations (i.e., the values of *bind parameters*) can be determined at runtime but the SQL operation is known a priori. In contrast, the JDBC API is fully dynamic – the SQL statement itself can be formulated "on the fly". Most SQL operations in a typical database application are static. SQLJ provides a simpler model for static SQL statements compared to JDBC, and provides a higher-level interface by automatically managing JDBC statement handles. Additionally, the SQLJ translator can check the SQL statements against a database for syntax and semantic errors. This checking is performed at compile-time unlike just at runtime as in JDBC, and it is independent of the actual flow of program logic. Compared to JDBC, SQLJ programs are therefore more robust, much quicker to write and easier to maintain.

A SQLJ source file uses the short-hand **`#sql`** notation to embed SQL statements inline. For example, to insert a new employee in our database we can directly write:

```
#sql [ctx]{ INSERT INTO emp (ename, sal) VALUES (:newEmp, :(getSal(newEmp))};
```

The Java variable `ctx` in the above statement represents the database connection on which the SQL statement is to be executed. Within the curly braces is the SQL command, and the Java arguments (called *host expressions*) for the operation are embedded directly in their appropriate positions. Execution of this statement automatically evaluates the host expressions, acquires a JDBC statement handle on the given database connection, binds the parameters, and releases the statement after execution is complete. Contrast the compact syntax to the sequence of JDBC calls that are required to set up the statement handle and bind parameter values.

Querying the database requires the use of SQLJ *iterators,* which are strongly-typed equivalents of JDBC result sets. The strong typing aids compile-time checking of SQL queries, and is defined in terms of the types and optionally the names of the SQL columns fetched. Further details about SQLJ programming can be obtained from [5].

Usage Note:

JDBC or SQLJ? An important point is that SQLJ and JDBC are complementary approaches. If your application has some static SQL statements and some dynamic ones, you can mix-and-match SQLJ and JDBC in a single source file as necessary. SQLJ is designed to inter-operate nicely with JDBC; for example, a single database connection can be easily shared across the two programming models. Likewise, JDBC resultsets can be converted to SQLJ iterators and vice-versa.

4. QUERYING THE DATABASE FROM A JSP

In this section we illustrate through detailed code examples how to query a database using JDBC and SQLJ in a JSP. Other SQL operations such as insert and update are permitted of course, but our focus in this paper is on fetching and formatting data from SQL tables.

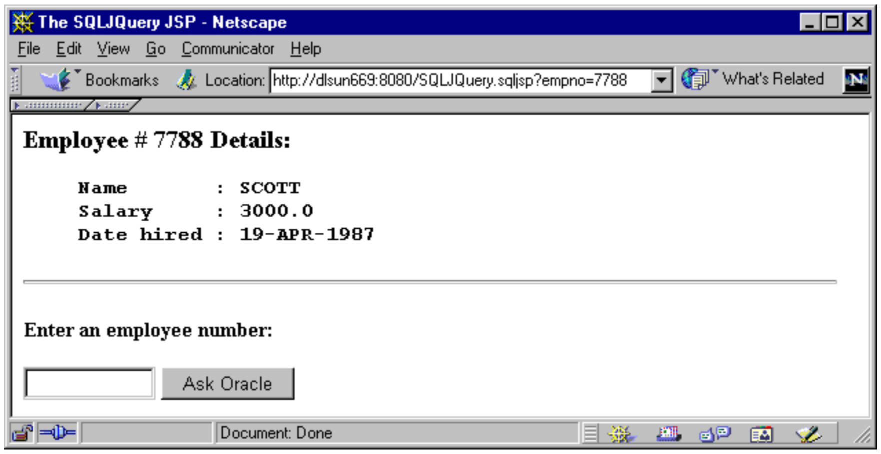
4.1 USING SQLJ

Since JDBC is a set of Java interfaces, its use within a JSP scriptlet is directly supported. Using SQLJ in JSP scriptlets is a convenient Oracle extension to this model. The JSP `page` directive has a language attribute which can be set to `sqlj`, as in the JSP shown in Figure 8 below. Equivalently, the JSP file may be given the `.sqljsp` extension.

```
<%@ page language="sqlj"
         import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>
<HTML>
<HEAD> <TITLE> The SQLJQuery JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String empno = request.getParameter("empno");
   if (empno != null) { %>
      <H3> Employee # <%=empno %> Details: </H3>
      <%= runQuery(empno) %>
      <HR><BR>
<% }  %>

<B>Enter an employee number:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%! private String runQuery(String empno) throws java.sql.SQLException {
     DefaultContext dctx = null;
     String ename = null;  double sal = 0.0;  String hireDate = null;
     StringBuffer sb = new StringBuffer();
     try {
        dctx = Oracle.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");
        #sql [dctx] {  SELECT ename, sal, TO_CHAR(hiredate, 'DD-MON-YYYY')
                       INTO :ename, :sal, :hireDate
                       FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
      };
      sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
      sb.append("Name       : " + ename + "\n");
      sb.append("Salary     : " + sal + "\n");
      sb.append("Date hired : " + hireDate);
      sb.append("</PRE></B></BIG></BLOCKQUOTE>");

     } catch (java.sql.SQLException e) {
        sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
     } finally {
        if (dctx!= null) dctx.close();
     }
     return sb.toString();
  }
%>
```

Figure 8. The SQLJQuery JSP with embedded static SQL in scriptlets

The first part of the SQLJQuery JSP defines the HTML code for the page. A request parameter named `empno` can be entered through the form and submitted through the "Ask Oracle" button. If the presence of this parameter is detected in the request object then the `runQuery()`method is executed with `empno` as input. This method is a private routine defined in the latter part of the JSP itself using the tags `<%!` … `%>` for class-level declarations. A database connection to the `scott/tiger` account is first established in this method using the JDBC-OCI driver for the Oracle database. Then, a static SQL query is performed to fetch the details of an employee based on the input `empno` argument. Notice the use of the compact **`SELECT..INTO`** statement to fetch a single record into Java variables. This construct is a special feature of SQLJ and is not directly available in JDBC where query results must always be handled via ResultSets irrespective of whether they contain one or more rows. If the query succeeds in finding a row then the data is formatted using standard HTML tags and inserted at the place where the `runQuery()` method was invoked as a Java expression. The database connection is closed in the `finally` block of the `try` statement that performs the SQL query. Output of executing this page for `empno` = 7788 is shown in Figure 9 below.


The SQLJQuery JSP - Netscape
File Edit View Go Communicator Help
Bookmarks
Location: http://dlsun669:8080/SQLJQuery.sqljsp?empno=7788
What's Related
Ask Oracle
Document: Done

Employee # 7788 Details:

```
Name       : SCOTT
Salary     : 3000.0
Date hired : 19-APR-1987
```

Enter an employee number:

Figure 9. The result of executing a static query through SQLJQuery JSP

Performance Notes:

There are two important performance notes about the database access logic in the SQLJQuery JSP:

1) **Database connections**: Each time it is invoked with a new search condition, a new database connection is opened by the `runQuery()` method. In practice, a real web environment would have performance optimizations such as database connection pooling in place. These are usually applicable in a transparent fashion, e.g., the `close()` method on a connection may simply return the connection to a shared pool. Thus the JSP would still have the same code but underlying layers would cause available connections to be shared effectively among multiple concurrent users of the database. In Section 4.2 we will look at how resource management can be explicitly programmed in beans used by a JSP, which may be desirable for some web applications.

2) **Query re-parsing**: Another apparent source of inefficiency is the re-parsing of the SQL statement if the JSP is invoked multiple times to perform the same SQL query with different parameters. This situation is not as bad as it seems. In Oracle 8.1.6 the SQLJ runtime will automatically cache and re-use JDBC statement handles as long as the underlying database connection is open. If connection pooling is in place then these statement handles may even be shared across different users. An alternative to this scheme is to use a session-scoped query bean that explicitly re-uses a prepared statement handle for the duration of the HTTP session.

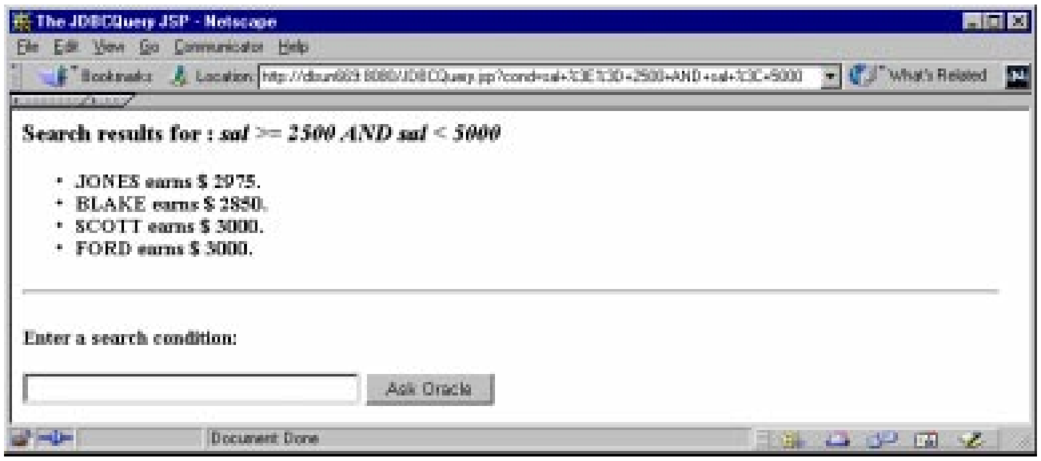
4.2 Using JDBC

SQLJ is intended for static SQL, i.e., when the SQL command is known at program time. In some cases it is required to formulate the SQL command dynamically as the program executes, for example if a general search condition can be specified by the user. One must use JDBC in these cases.

Let us program a JSP called JDBCQuery that performs a simple dynamic query against the Oracle database. The code is shown below in Figure 10. The `runQuery()` method first logs on to the `scott` schema using password `tiger` and obtains a statement handle for executing the query. The `SELECT` statement looks up employees and their salaries in the `emp` table based on the `searchCondition` parameter obtained through the form. Notice that the `WHERE` clause in the SQL query is being constructed at runtime depending on user input and it is therefore not static SQL programmable in SQLJ. The `finally` block closes the result set, the underlying statement handle and the database connection upon exit from the `try` block of SQL operations. The method `formatResult(ResultSet)` takes the result of a successful query and generates a bulleted list of employee data.

```
<%@ page import="java.sql.*" %>

<HTML>
<HEAD> <TITLE> The JDBCQuery JSP  </TITLE> </HEAD>
<BODY BGCOLOR=”white”>

<% String searchCondition = request.getParameter("cond");
   if (searchCondition != null) { %>
      <H3> Search results for  <I> <%= searchCondition %> </I> </H3>
      <B> <%= runQuery(searchCondition) %> </B> <HR><BR>
<% }  %>
<B>Enter a search condition:</B>
<FORM METHOD=”get”>
<INPUT TYPE="text" NAME="cond" SIZE=30>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%! private String runQuery(String cond) throws SQLException {
     Connection conn = null;
     Statement stmt = null;
     ResultSet rset = null;
     try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                           "scott", "tiger");
        stmt = conn.createStatement();
        // dynamic query
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                                  (cond.equals("") ? "" : "WHERE " + cond ));
       return (formatResult(rset));
     } catch (SQLException e) {
         return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
     } finally {
         if (rset!= null) rset.close();
         if (stmt!= null) stmt.close();
         if (conn!= null) conn.close();
     }
  }
  private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
      sb.append("<P> No matching rows.<P>\n");
    else {  sb.append("<UL>");
          do {  sb.append("<LI>" + rset.getString(1) +
                            " earns $ " + rset.getInt(2) + ".</LI>\n");
            } while (rset.next());
          sb.append("</UL>");
    }
    return sb.toString();
  }
%>
```

When invoked with no arguments the JDBCQuery JSP simply displays a form for entering a search condition and a submit button named “Ask Oracle”. After the search condition `sal >= 2500 AND sal < 5000` is entered on the screen and the ‘Ask Oracle’ button is clicked, the output of the JDBCQuery JSP appears as follows:


The JDBCQuery JSP - Netscape
File Edit View Go Communicator Help
Bookmarks
Location:
What's Related
Ask Oracle
Document: Done

Search results for : *sal >= 2500 AND sal < 5000*

- JONES earns $ 2975.
- BLAKE earns $ 2850.
- SCOTT earns $ 3000.
- FORD earns $ 3000.

Enter a search condition:

Figure 11. Result of doing a dynamic query through the JDBCQuery JSP

4.3 Resource Management in a Web Application

A web application must manage resources acquired during its execution, as such database connections and JDBC statement handles. In the case of JSPs, there are basically two ways to handle web application resources:

1) **Build the management logic into beans called from the JSP.** For example, a session-scoped query bean could acquire a database cursor when it is instantiated and release it when the HTTP session is terminated (either explicitly or implicitly via timeout). However, in this scheme a bean needs to be aware that it is running in a servlet environment; for example, the `javax.servlet.http.HttpSessionEventListener` interface would have to be implemented by the bean so that it can be notified by the servlet execution engine and release the resources upon expiration of the HTTP session. Since beans are often designed by third parties as reusable and pluggable components, such explicit knowledge of the execution environment may not be available or be even desirable within the bean. A workaround in this case would be to create a proxy “environment-aware” bean that wraps an existing bean to provide the necessary resource management logic in the HTTP execution scenario.

2) **Have the JSP code manage the resources itself.** JSPs and servlets in the web application know that they will be running in an HTTP environment, hence they can allocate and de-allocate resources at appropriate times. For example, a JSP can have explicit logic to associate a resultset with the HTTP session. For Oracle JSPs we have built an extension that lets a JSP easily control resource lifetimes using well-known lifecycle events. This is done through special scripting code that is executed when the application events such as session start and session end occur, along the lines of the `globals.asa` facility for Active Server Pages.

Which of the above two methods is employed for resource management is upto the preference of the JSP programmer and design requirements of the web application.

4.4 Using Session-Scoped JavaBeans for Queries

The SQLJQuery and JDBCQuery JSPs have a drawback that Java logic is interspersed with the HTML code. While this approach may be adequate for simple database operations, in practice the majority of JSP authors are likely to be graphic designers and scripters rather than full-fledged Java programmers with knowledge of JDBC or SQLJ. Thus, it makes sense to cleanly separate the logic for generation of dynamic content from its presentation. In accordance with its anticipated usage, JSPs support convenient creation and usage of modular JavaBeans through the `<jsp:usebean>` tag described earlier in Section 2.2.3. We will now illustrate the use of beans to perform SQL queries and manage database resources.

If in a web application the same query is re-executed with different parameters then re-parsing the same SQL statement each time is inefficient. As we have mentioned earlier, underlying cursor caching and connection pooling mechanisms may help reduce such costs. Let us assume for illustrative purposes that there is no connection pooling, i.e., closing a database connection actually terminates it instead of returning it to a shared pool. In Oracle 8.1.6 SQLJ will provide automatic caching of JDBC prepared statement handles; however, the caching is only applicable until the connection is closed. Thus, to obtain the benefits of SQLJ statement caching for repeated query execution in a no-pooling environment, the database connection must be left open for the duration of the HTTP session. In other words, the lifetime of the `java.sql.PreparedStatement` representing the parsed SQL query should be the same as that of the HTTP session. The `<jsp:usebean>` tag provides an easy way of doing this type of lifetime scoping using a modular query bean.

Below we define a JavaBean named `mybeans.QueryBean` that performs a SQLJ query based on the value of its `empNum` property.

```
package mybeans;
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
import oracle.sqlj.runtime.Oracle;
import javax.servlet.http.*;

public class QueryBean implements HttpSessionBindingListener {
  String empNum;
  String result = null;
  public void QueryBean() {  }

  public String getResult() {
    if (result != null) return result; else return runQuery();
  }
  public void setEmpNum(String empNum) {
      result = null; this.empNum = empNum;
  }

  DefaultContext dc = null;
  private String runQuery() {
    StringBuffer sb = new StringBuffer();
    try {
      if (dc == null)
          dc = Oracle.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");
      String ename = null;  double sal = 0.0;  String hireDate = null;
      #sql [dc] {  SELECT ename, sal, TO_CHAR(hiredate, 'DD-MON-YYYY')
                   INTO :ename, :sal, :hireDate  FROM emp
                  WHERE UPPER(empno) = UPPER(:empNum) ORDER BY ename };
      sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
      sb.append("Name       : " + ename + "\n");
      sb.append("Salary     : " + sal + "\n");
      sb.append("Date hired : " + hireDate);
      sb.append("</PRE></B></BIG></BLOCKQUOTE>");
      return sb.toString();
    } catch (SQLException e) {
       return ("SQL Error: " + e.getMessage());
    }
  }
  // Event listener methods for HttpSession binding
  public void valueBound(HttpSessionBindingEvent event) {
    // do nothing -- we know the bean is session-scoped and so already bound
  }
  public synchronized void valueUnbound(HttpSessionBindingEvent event) {
    if (dc != null) {
      try { dc.close();
      } catch (SQLException ignored) {}
    }
  }
}
```

There are two important points to note about the logic in this bean:

a) It is intended to be used in a session scope and uses the instance variable dc to remember the database connection.

b) It implements the `javax.servlet.http.HttpSessionBindingListener` interface.

This interface has two methods, `valueBound()` and `valueUnbound()`, that are called at HTTP session startup and shutdown respectively. These methods can be used to perform the necessary allocation and de-allocation of session-based resources.

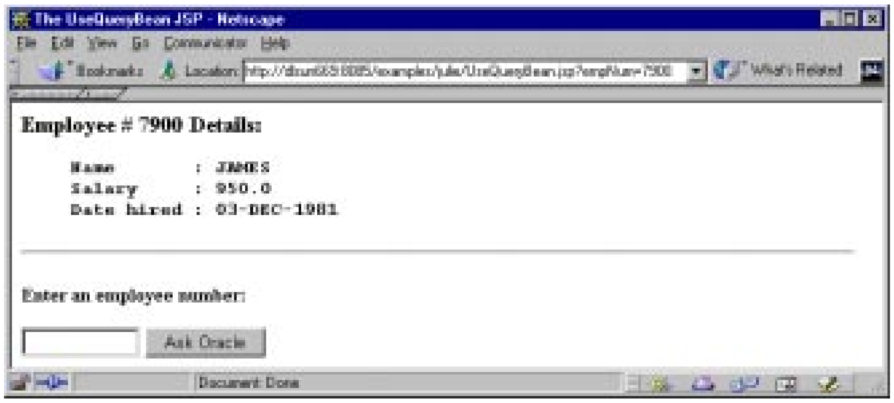
Programming Notes on Session Scoping:

1) In our case QueryBean will be invoked through a session-scoped `<jsp:usebean>` tag so that at session startup the bean will be automatically placed on the list of objects listening for session events. Our JSP (defined below) that uses this bean calls its `getResult()` method only when the user provides a search parameter in the form. So, we will not connect to the database in the `valueBound()` method in order to minimize the connection interval. Rather, the `runQuery()` method will establish the connection when invoked for the first time via `getResult()`. Therefore, the `valueBound()` method does nothing (an alternative approach is to place the `usebean` tag under conditional logic and open the database connection upon instantiation of the bean). We use the `valueUnbound()` method to close the database connection if it is not null.
2) A valid question is why do we need to implement the special session listerner methods? Could we instead close the connection in a `finalize` method for the bean that would be called when it is garbage collected? The answer is that the session listener interface has a much more well-defined behavior compared to the `finalize` method. It is true that at the end of the HTTP session the reference to a session-scoped bean is no longer available and presumably the bean becomes gargage. However, garbage collection frequency depends on the memory consumption pattern of the application. The database connection would be held open until the garbage collector runs and invokes the `finalize` method for the bean. Holding on to a connection for such an unpredictable interval is simply not a good idea. The session listener interfaces provide well-defined event-based hooks precisely for this purpose.

The UseQueryBean JSP shown in Figure 12 below uses `mybeans.QueryBean` to select the details for a particular employee. The bean is invoked with session scope, and its property is set to the value of the `empNum` field entered through the HTML form. Output of executing the JSP appears in Figure 13.

```
<jsp:useBean id="queryBean" class="mybeans.QueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="empNum" />

<HTML>
<HEAD> <TITLE> The UseQueryBean JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">

<%  String empNum = request.getParameter("empNum");
    if (empNum != null) { %>
      <H3>Employee # <%= empNum %> Details: </H3>
      <%=  queryBean.getResult() %>
      <HR>
<%  }  %>

<P><B>Enter an employee number:</B></P>
<FORM METHOD=”get”>
<INPUT TYPE=”text” NAME="empNum" SIZE=10>
<INPUT TYPE=”submit” VALUE="Ask Oracle">
</FORM>

</BODY>
</HTML>
```

Figure 12. The UseQueryBean JSP uses mybeans.QueryBean to get employee data


The UseQueryBean JSP - Netscape
File Edit View Go Communicator Help
Bookmarks
Location:
What's Related
Ask Oracle
Document Done

Employee # 7900 Details:

```
Name       : JAMES
Salary     : 950.0
Date hired : 03-DEC-1981
```

Enter an employee number:

Figure 13. Output of executing the UseQueryBean JSP

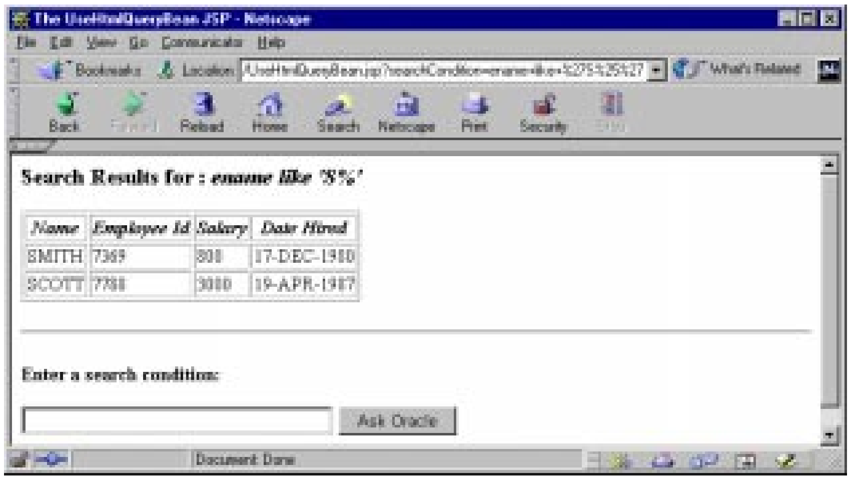
5. Formatting SQL Query Results

A common issue in web applications is visually-pleasing representation of SQL query results. In this section we will look at three different mechanisms by which query results can be formatted in a JSP.

5.1 Generating Basic HTML Tables

The following class `HtmlTable` defines a static method `format(ResultSet rs)` that accepts a JDBC resultset parameter and uses the associated metadata to generate an HTML table with column headers and data rows. This example is based on the corresponding sample in the Java Servlet Programming book [4]. The JDBC `ResultSetMetaData` object contains the names of SQL columns selected by the query, and these names are used as column headers. A programmer can easily get a desired column header by using the AS clause to alias a SQL column or expression in a JDBC or SQLJ query. Once a query has been successfully executed, the `format` method can be invoked from a JavaBean or directly from a JSP scriptlet. Figure 14 displays the output of executing this method on a SQLJ iterator for employee data. The JSP is omitted for brevity.

```
/**********  File HtmlTable.java  **********/
import java.sql.*;

public class HtmlTable {

  public static String format(ResultSet rs) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (rs==null) || !rs.next()) sb.append("<P> No matching rows.<P>\n");
    else {
      sb.append("<TABLE BORDER>\n");
      ResultSetMetaData md = rs.getMetaData();
      int numCols = md.getColumnCount();
      for (int i=1; i<= numCols; i++) {
         sb.append("<TH><I>" + md.getColumnLabel(i) + "</I></TH>");
      }
      do {  sb.append("<TR>\n");
            for (int i = 1; i <= numCols; i++) {
               sb.append("<TD>");
               Object obj = rs.getObject(i);
               if (obj != null)  sb.append(obj.toString());
               sb.append("</TD>");
            }
            sb.append("</TR>");
      } while (rs.next());
      sb.append("</TABLE>");
    }
    return sb.toString();
  }
}
```



Search Results for : *ename like 'S%'*

Name	*Employee Id*	*Salary*	*Date Hired*
SMITH	7369	800	17-DEC-1980
SCOTT	7788	3000	19-APR-1987

Enter a search condition:

Figure 14. *Output of formatting using the HtmlTable.format(ResultSet) method*

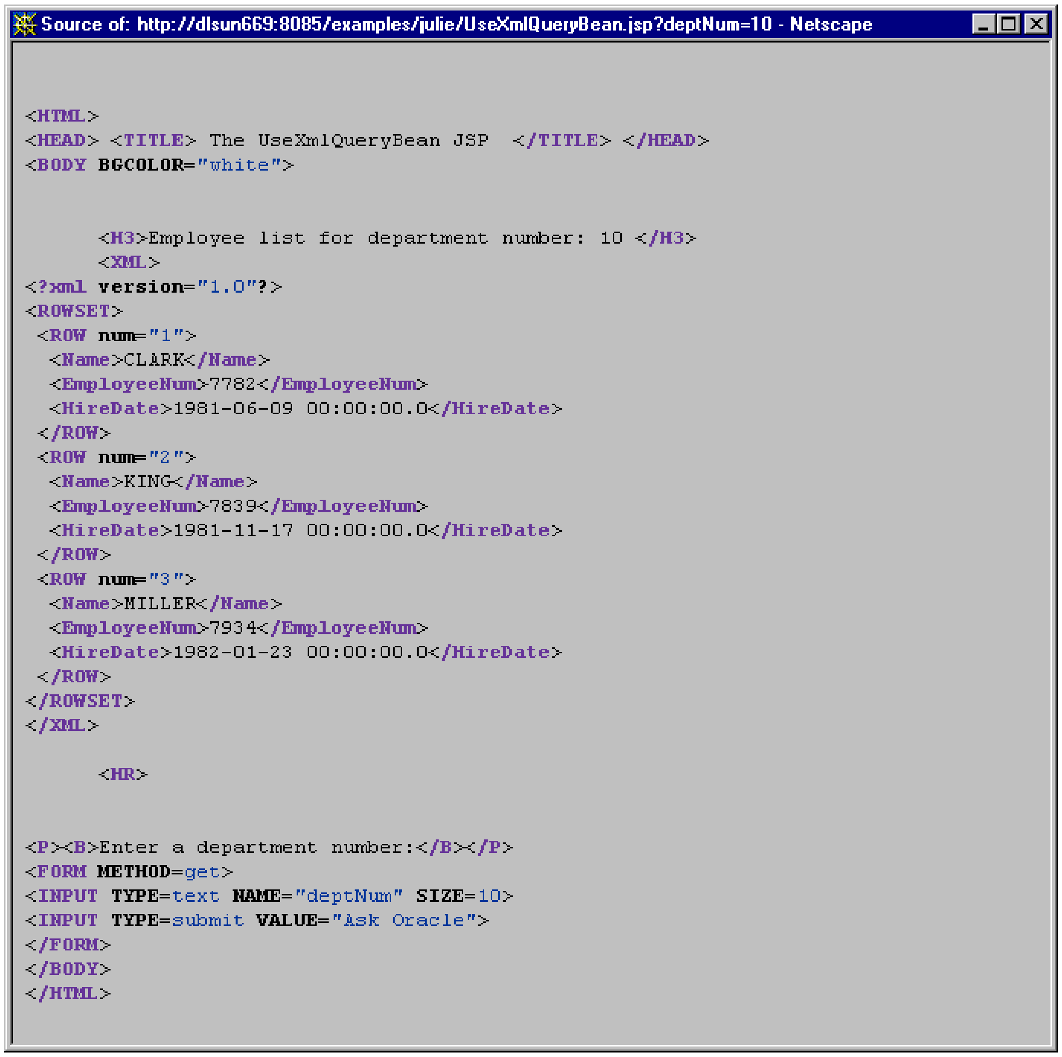
5.2 Using the Oracle XML-SQL Utility

Oracle Technology Network (*http://technet.us.oracle.com*) provides several freely downloadable utilities for XML, and XML-SQL is one of them. This is a very useful package that can convert a JDBC resultset into formatted XML output. Conversely, it can also insert input XML data into a canonical definition of SQL tables. In the example below we illustrate the XML-generation capability of this tool. As you will notice in the file XMLQueryBean.sqlj shown below, the bean executes a parameterized SQL query against the `emp` table and fetches the results into a SQLJ iterator called `emps`. The JDBC resultset underlying this iterator is extracted through the `emps.getResultSet()` method call and used to construct an instance of the class `oracle.xml.sql.query.OracleXMLQuery.` Subsequently, a call to the method `getXMLString()` of this instance generates XML output that is in-lined in the returned HTML page (Figure 15). The Internet Explorer 5.0 browser is capable of displaying such XML islands embedded in HTML pages. The output can be formatted using stylesheets before displaying to the user.

Notice that the SQL query in the `#sql` statement uses SQL column aliases to rename fetched columns. By default, these names are used by XML-SQL as tags for the generated XML elements. There are facilities to provide explicit tags and stylesheets, but exploring this functionality is outside the scope of this paper.

```
    /**********  File XMLQueryBean.sqlj  **********/
    package mybeans;
    import java.sql.*;
    import sqlj.runtime.ref.DefaultContext;
    import oracle.sqlj.runtime.Oracle;
    import oracle.xml.sql.query.*;

    public class XMLQueryBean {
      String deptNum;
      String result = null;
      public void XMLQueryBean() {  }
      public String getResult() {
        if (result != null) return result;  else return runXMLQuery();
      }
      public void setDeptNum(String deptNum) {
          result = null;
          this.deptNum = deptNum;
      }
      DefaultContext dc = null;
      private String runXMLQuery() {
        #sql iterator empIter (String Name, int EmployeeNum, Date HireDate);
        empIter emps = null;

  try {
    if (dc == null) dc = Oracle.getConnection("jdbc:oracle:oci8:@", "scott", "tiger");
```

```
      #sql [dc] emps = { SELECT ename as "Name", empno as "EmployeeNum",
                          hiredate as "HireDate"
                          FROM emp WHERE deptno = :deptNum ORDER BY ename };

      try {
        OracleXMLQuery xq = new OracleXMLQuery( dc.getConnection(), emps.getResultSet());
        return ("<XML>\n" + xq.getXMLString() + "</XML>\n");
       } catch (Exception e) {
          return ("XML conversion error: " + e.getMessage());
       }
    } catch (SQLException e) {
       return ("SQL Error: " + e.getMessage());
    } finally {
      try {
       if (emps != null) emps.close();
      } catch (SQLException ignored) {}
    }
  }
……… // methods that implement HttpSessionBindingListener interface, as in Querybean.sqlj
}
```


Source of: http://dlsun669:8085/examples/julie/UseXmlQueryBean.jsp?deptNum=10 - Netscape

```
<HTML>
<HEAD> <TITLE> The UseXmlQueryBean JSP  </TITLE> </HEAD>
<BODY BGCOLOR="white">


      <H3>Employee list for department number: 10 </H3>
      <XML>
<?xml version="1.0"?>
<ROWSET>
 <ROW num="1">
  <Name>CLARK</Name>
  <EmployeeNum>7782</EmployeeNum>
  <HireDate>1981-06-09 00:00:00.0</HireDate>
 </ROW>
 <ROW num="2">
  <Name>KING</Name>
  <EmployeeNum>7839</EmployeeNum>
  <HireDate>1981-11-17 00:00:00.0</HireDate>
 </ROW>
 <ROW num="3">
  <Name>MILLER</Name>
  <EmployeeNum>7934</EmployeeNum>
  <HireDate>1982-01-23 00:00:00.0</HireDate>
 </ROW>
</ROWSET>
</XML>

      <HR>


<P><B>Enter a department number:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="deptNum" SIZE=10>
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>
```

Figure 15. HTML page with an embedded XML island generated using SQLJ and XML-SQL

5.3 Using the XSQL Servlet

The XSQL utility lets you embed SQL queries inline within an XML page using a dynamic `<query>` tag. A nice feature is that it has both a command-line as well as a servlet interface, and the latter can be invoked from within a JSP using the standard `<jsp:include>` tag. Below is an example that illustrates how to do this. But first, consider the small sample of an XSQL page named emp.xsql as shown in Figure 16 below:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="rowcol.xsl"?>
<query connection="demo" sort="ENAME" null-indicator="yes" >
      SELECT * FROM EMP
      WHERE ENAME LIKE '{@find}%'
      ORDER BY {@sort}
</query>
```

Figure 16. An XSQL page with an embedded SQL query

An XSQL page conforms to standard XML syntax and can specify stylesheets for post-processing the generated output. It uses the special `<query>` tag to perform a SQL query. The query tag supports several attributes including, among others, the database connection, null-handling, plus search and sort parameters for the query. In our case we run the query against the `emp` table using the database connection named `demo`. The parameters for this connection are specified separately in a configuration file (not shown here) that also follows XML syntax. After the query result is formatted into XML the stylesheet is applied. For our example the stylesheet being used is rowcol.xsl, which looks as follows:

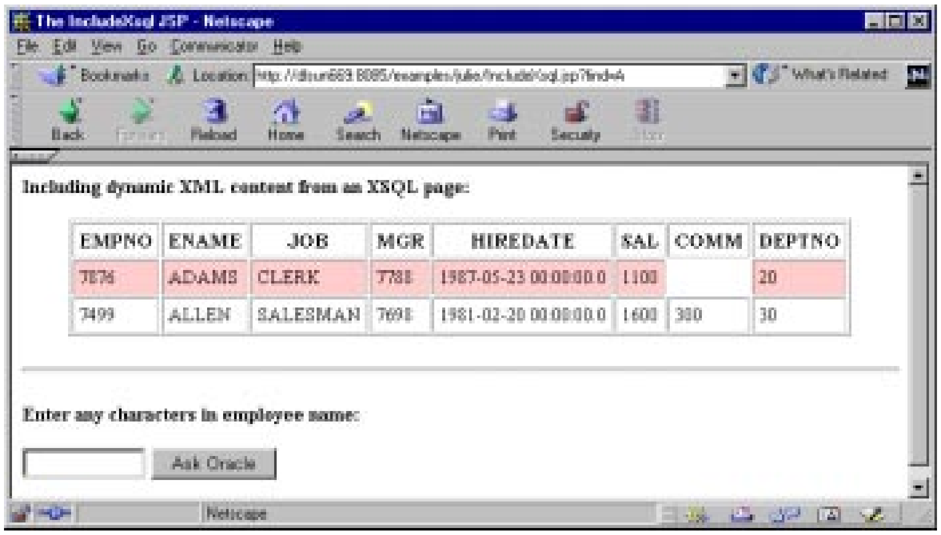
```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
                xmlns="http://www.w3.org/TR/REC-html40"  result-ns="" indent="yes">
  <xsl:template match="/">
    <html>
      <head> <style>
        .page    {font-family: Tahoma, sans-serif; background-color:white}
        .roweven {background-color: white;} .rowodd  {background-color: pink;}
      </style> </head>
      <body class="page"> <center>
      <table border="1" cellpadding="4">
      <xsl:for-each select="ROWSET/ROW[1]">
        <tr> <xsl:for-each select="*">
              <th>
                <xsl:attribute name="class">
                   <xsl:choose>
                    <xsl:when test="position() mod 2 = 1">colodd</xsl:when>
                    <xsl:when test="position() mod 2 = 0">coleven</xsl:when>
                  </xsl:choose>
                </xsl:attribute>
                 <xsl:value-of select="name(.)"/>
              </th>
        </xsl:for-each> </tr>
      </xsl:for-each>
      <xsl:for-each select="ROWSET/ROW">
        <tr> <xsl:attribute name="class">
              <xsl:choose>
                <xsl:when test="position() mod 2 = 1">rowodd</xsl:when>
                <xsl:when test="position() mod 2 = 0">roweven</xsl:when>
              </xsl:choose>
            </xsl:attribute>
            <xsl:for-each select="*">
              <td> <xsl:attribute name="class">
                   <xsl:choose>
                     <xsl:when test="position() mod 2 = 1">colodd</xsl:when>
                     <xsl:when test="position() mod 2 = 0">coleven</xsl:when>
                   </xsl:choose>
                  </xsl:attribute>
                   <xsl:value-of select="."/>
              </td>
           </xsl:for-each>
        </tr>
      </xsl:for-each>
      </table> </center> </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Employees whose names match the `find` parameter are selected by the query and ordered alphabetically by their name. An important point to note is that the `sort` parameter is specified as an attribute of the query tag but `find` is not defined inside the XSQL page. This means that the search condition will be picked up from the environment that the XSQL page is invoked from, in our case from a JSP. The dynamic contents of this page can be included in a JSP as follows:

```
<HTML>
<HEAD><TITLE> The IncludeXsql JSP </TITLE></HEAD>
<BODY BGCOLOR="white">
<%@ page buffer="5" autoFlush="false" %>
<P><B>Including dynamic XML content from an XSQL page:<B></P>
<P>
<jsp:include page="emp.xsql" flush="true" />
</P><HR>
<P><B>Enter any characters in employee name:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="find" SIZE=10>
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>
```

Figure 17. Including the dynamic contents of an XSQL page in a JSP

Notice that the JSP uses a form with an input field named `find`, which is no accident. It is intended to match the `find` parameter in our emp.xsql page which is invoked from the JSP via the standard `<jsp:include>` tag. Query execution in the emp.xsql page is done using JDBC, the resultset formatted into XML using the XML-SQL utility described in the previous section, and then the given stylesheet is applied before the result is returned to the JSP for inclusion in the HTTP response. Figure 18 below shows the output for a sample search for employee names starting with the letter A.


The IncludeXsql JSP - Netscape
File Edit View Go Communicator Help
Bookmarks Location: http://dlsun689:8085/examples/jules/includeXsql.jsp?find=A
What's Related
Back Forward Reload Home Search Netscape Print Security Stop
Ask Oracle
Netscape

Including dynamic XML content from an XSQL page:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7876	ADAMS	CLERK	7788	1987-05-23 00:00:00.0	1100		20
7499	ALLEN	SALESMAN	7698	1981-02-20 00:00:00.0	1600	300	30

Enter any characters in employee name:

Figure18. Sample output from the IncludeXsql JSP

6. JSP Product Availability and Technology Preview

Oracle has developed its own JSP runtime that supports the complete JSP 1.0 specification as well as Oracle defined extensions. These extensions include:

- Scriptable life-cycle events using ASP-style globals.jsa file in the web application
- A templating language called JSP Markup Language (JML) to augment scripting in Java
- SQLJ integration in JSP scriptlets
- The ability to transform all or part of the result of a JSP using XSL.

Oracle supports the execution of servlets and JSPs in several different server environments. These environments include Oracle Application Server Release 4.0.8.1, and upcoming releases of Oracle WebDB (3.0), Oracle Lite, and Oracle JServer. The Oracle JSP translator also runs on any standard servlet runner, most notably Apache. Development of JSP and servlet based web applications is supported in Oracle JDeveloper 3.0, including the ability to debug JSP by setting breakpoints in its source.

The Oracle JSP translator is freely downloadable off the Oracle Technology Network site (*http://technet.oracle.com*). The site also contains information about the use of the above extensions.

6.1 ORACLE'S JSP PLANS AND EXTENSIONS

The following is an outline of JSP features and functionality that are being considered for upcoming Oracle releases:

- **Supporting JSP 1.1**: The translator currently conforms to the JSP 1.0 specification that is based on the Servlet 2.1 API. In future we will support the JSP 1.1 specification built on the Servlet 2.2 specification, which are both part of the Java2 Enterprise Edition (J2EE). The major enhancement in JSP 1.1 is the definition of a portable tag extension mechanism. This will enable 3rd parties and application developers to build JSP tags (like our JML tags) and know they will work on all JSP runtimes.
- **Improved Oracle Database access**: Though we have shown in this paper a variety of ways to access Oracle data from a JSP, we are considering building a collection of data access beans that are suited for use within a JSP. In particular, these beans would be the basis for extending the JML tags to provide database access.
- **Improved XML/XSL integration**: We expect to add additional capabilities to extend and simplify the use of XML/XSL in a JSP environment.

6.2 RUNNING JSPS INSIDE THE ORACLE DATABASE

Java being a strongly-type and memory-safe language is ideal for programming server applications. The Oracle 8*i* database embeds a Java Virtual Machine to allow the storage and execution of Java code inside the database server. This Java Virtual Machine, known as the JServer, is different from the one in JDK and is specially optimized for the Oracle database. It runs in the same address space as the database process, thereby providing fast but safe access to SQL data. As we have seen in this paper, Oracle provides both JDBC and SQLJ standards for accessing object-relational data in SQL tables. Distributed programming schemes such as Enterprise Java Beans and CORBA objects are also supported by JServer. Different sessions are handled in distinct *virtual* Java execution environments, providing isolation and increased scalability. For example, individual sessions have their own garbage collector and do not affect the performance of other concurrent Java sessions as in the case of the JDK. The JServer has been demonstrated to be highly scalable, effectively handling thousands of clients without significant deterioration in performance [10].

In an upcoming release the JServer will also host an embedded web server component. This will allow HTTP-based communication with the Oracle database and support the direct execution of servlets and JSPs inside the database, closest to SQL data. The web server namespace will be modeled using the JNDI (Java Naming and Directory Interface) specification from Sun Microsystems. Oracle's JSP translator will be integrated with this web server. We plan to generate servlet code that is specially optimized for the JServer execution scenario, taking advantage of the JServer performance features such as shared read-only resources and hot-loaded classes.

7. Summary

To summarize, this paper examines database access issues for JavaServer Pages that generate dynamic content. We first provide an overview of the capabilities of the JSP framework. JSPs support Java-based scripting to allow generation of dynamic HTML content. Using special JSP tags, Java scriptlets may be interspersed with static HTML in the same source file. This file is translated into a servlet by the JSP translator, and can be executed on any standards-compliant servlet runner. The compact JSP framework also integrates support for JavaBeans, which are reusable entities of component-based programming. JavaBeans can be conveniently invoked from a JSP via the `<jsp:usebean>` tag, and their lifetimes can be scoped to be associated with the JSP page, the HTTP request, HTTP session, or the web application. In Section 2 we illustrated these JSP features using detailed code samples.

The latter part of the paper deals specifically with JSPs that perform SQL operations to generate their dynamic content. Access to Oracle is shown using both SQLJ and JDBC in JSP scriptlets. JavaBeans that modularize SQL operations are also illustrated. We discuss the important issue of database resource management, such as closing database connections and re-using JDBC statement handles. Formatting of SQL query results is a significant topic for web application designers. In this paper we demonstrated three different ways to handle the presentation of SQL query results: (1) directly processing the JDBC resultset to generate HTML tables, (2) using the Oracle XML-SQL utility to generate embedded XML islands, and finally (3) using the XSQL servlet and its special `<query>` tag to execute and format via XSL the result of a SQL query. In conclusion, we provide the product status of Oracle JSPs and a brief technology preview for upcoming Oracle releases.

8. Acknowledgments

I am very grateful to the many people at Oracle who have shared their ideas and insights. In particular, I would like to thank Mike Freedman for in-depth technical discussions on the various issues addressed in this paper. Thanks are also due to members of the Webserver team in the Java Platform Group – it is indeed a pleasure and a privilege to be working with them. I specially wish to thank Rakesh Dhoopar for submitting the paper abstract, Ekkehard Rohwedder for clarifications and discussions on SQLJ issues, Steven Muench for helping to get the XSQL servlet example working with JSPs, and Gael Stevens for her help in reviewing the paper. The code sample for XSQL and the XSQL stylesheet are adapted from the demos written by Steve Muench and debugged with expert help from Mike Freedman.

9. References

[1] *JavaServer Pages 1.0 Specification*, Sun Microsystems
[2] *Java Developer's Guide*, Oracle 8.1.5
[3] *JDBC Devloper's Guide and Reference*, Oracle 8.1.5
[4] *JAVA Servlet Programming*, Hunter and Crawford, O'Reilly
[5] *SQLJ Developer's Guide and Reference*, Oracle 8.1.5
[6] *Developing JavaBeans*, Robert Englander, O'Reilly
[7] *Oracle XML-SQL utility*, http://technet.oracle.com
[8] *Oracle XSQL servlet*, http://technet.oracle.com
[9] *XML and XSL documentation*, http://www.w3c.org
[10] *JServer Performance and Scalability*, Oracle technical white paper, http://www.oracle.com/java