UNIT 4 JAVA SERVER PAGES-II

Structure **Page Nos.**

4.0 INTRODUCTION

In the previous unit, we have discussed the importance of JSP. We also studied the basic concepts of JSP. Now, in this unit, we will discuss some more interesting features of JSP. As you know, JSP is mainly used for server side coding.

Therefore, database handling is the core aspect of JSP. In this unit, first you will study about database handling through JSP. In that you will learn about administratively register a database, connecting a JSP to an Access database, insert records in a database using JSP, inserting data from HTML Form in a database using JSP, delete Records from Database based on Criteria from HTML Form and retrieve data from a database using JSP – result sets.

Then you will learn about how to include files and applets in JSP documents. In this topic you mainly learn about three main capabilities for including external pieces into a JSP document, i.e., jsp:include action, include directive and jsp:plugin action.

Finally, you will learn about integration of servlet and JSP. In this topic you learn about how to forward the requests, how to include static or dynamic content and how to forward requests from JSP Pages.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- understand how to connect JSP with a database;
- understand how to select, insert and delete records in database using JSP;
- understand how to include files and applets in JSP documents;
- understand how to include the output of JSP, HTML or plain text pages at the time the client requests the page;
- understand how to include JSP files at the time the main page is translated into a servlet;
- understand how to include applets that use the Java Plug-In, and
- understand how to integrate servlet and JSP.

4.2 DATABASE HANDLING IN JSP

We have already seen how to interface an HTML Form and a JSP. Now, we have to see how that JSP can talk to a database. In this section, we will understand how to:

1. Administratively register a database.
2. Connect a JSP to an Access database.
3. Insert records in a database using JSP.
4. Insert data from HTML Form in a database using JSP.
5. Delete Records from Database based on Criteria from HTML Form.
6. Retrieve data from a database using – JSP result sets.

- **Administratively Register a Database**

Java cannot talk to a database until, it is registered as a data source to your system. The easiest way to administratively identify or registrar the database to your system so your Java Server Page program can locate and communicate with it is to do the following:

1) Use MS Access to *create* a blank database in some directory such as D. (In my case, the database was saved as testCase001.mdb.) Make sure to *close* the database after it is created or you will get an invalid path *message* during the following steps.

2) Go to: Control panel > Admin tool > *ODBC* where you will identify the database as a so-called *data source*.

3) Under the *User DSN* tab, *un*-highlight any previously selected name and then click on the *Add* button.

4) On the window that then opens up, highlight *MS Access Driver* and click *Finish*.

5) On the ODBC Setup window that then opens, fill in the data source name. This is the name that you will use to refer to the database in your Java program such as Mimi.

6) Then click Select and *navigate* to the already created database in directory D. Suppose the file name is testCase001.mdb. After *highlighting* the named file, click OKs all the way back to the original window.

This completes the registration process. You could also use the create option to create the Access database from scratch. But the create setup option *destroys* any existing database copy. So, for an existing DB follow the procedure described above.

- **Connect a JSP to an Access Database**

We will now describe the Java code required to connect to the database although at this point we will not yet query it. To connect with database, the JSP program has to do several things:

1. Identify the source files for Java to handle SQL.
2. Load a software driver program that lets Java connect to the database.
3. Execute the driver to establish the connection.

A simplified JSP syntax required to do this follows is:

```
<%@ page import= "java.sql.*"  %>
<%
        Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection  conn=null;
        conn = DriverManager.getConnection("jdbc:odbc:Mimi", "", "");
        out.println ("Database Connected");
%>
```

In this syntax, the so-called page directive at the top of the page allows the program to use methods and classes from the java.sql.* package that know how to handle SQL queries. The Class.forName method loads the driver for MS Access. Remember this is Java so the name is case-sensitive. If you misspell or misidentify the data source name, you'll get an error *message* "Data source name not found and no default driver specified". The DriverManager.getConnection method connects the program to the database identified by the data source *Mimi* allowing the program to make queries, inserts, selects, etc. Be careful of the punctuation too: those are colons (:) between each of the terms. If you use misspell or use dots, you'll get an error *message* about "No suitable driver". The Connection class has a different purpose. It contains the methods that let us work with SQL queries though this is not done in this example.

The DriverManager method creates a Connection object *conn* which will be used later when we make SQL queries. Thus,

1. In order to make queries, we'll need a Statement object.
2. In order to make a Statement object, we need a Connection object (conn).
3. In order to make a Connection object, we need to connect to the database.
4. In order to connect to the database, we need to load a driver that can make the connection.

The safer and more conventional code to do the same thing would include the database connection statements in a Java *try/catch* combination. This combination acts like a safety net. If the statements in the try section fail, then the Exceptions that they caused are caught in the catch section. The catch section can merely report the nature of the Exception or error (in the string exc shown here) or do more extensive backup processing, depending on the circumstances. The code looks like:

```
<%@ page import= "java.sql.*" %>
<%
        Connection  conn=null;

        try
        {   Class.forName ("sun.jdbc.odbc.JdbcOdbcDriver");
            conn = DriverManager.getConnection("jdbc:odbc:Mimi", "",
"");
        }
        catch (Exception exc)
        {   out.println(exc.toString() + "<br>");  }
```

```
            out.println ("Database Connected");
            conn.close ( );
            out.println ("Database closed");
        %>
```

We have also added another statement at the end that closes the connection to the data base (using the close ( ) method of the connection object conn).

- **Insert Records in Database using JSP**

So far – other than connecting to the database and then closing the connection to the database – we have not had any tangible impact on the database. This section illustrates that we have actually communicated with the database by using simple SQL insert examples. In general, SQL inserts in a JSP environment would depend on data obtained from an HTML Form. But addressing that involves additional complications, so we shall stick here to simple fixed inserts with hardwired data. To do insert or indeed to use any kind of SQL queries we have to:

1. Create a Statement object - which has methods for handling SQL.
2. Define an SQL query - such as an insert query.
3. Execute the query.

Example *testCase002.* adds the following statements to insert an entry in the database:

```
Statement   stm  =  conn.createStatement ( );
String          s  =  "INSERT INTO Managers  VALUES ( 'Vivek')";
stm.executeUpdate (s);
```

The Connection object conn that was previously created has methods that allow us to in turn create a Statement object. The names for both these objects (in this case conn and stm) are of course just user-defined and could be anything we choose. The Statement object stm only has to be created once. The insert SQL is stored in a Java String variable. The table managers we created in the database (off-line) has a single text attribute (managerName) which is not indicated in the query. The value of a text attribute must be enclosed in single quotes. Finally, the Statement object's executeUpdate method is used to execute the insert query. If you run the testCase002.jsp example and check the managers table contents afterwards, you will see that the new record has been added to the table.

Generally, these statements are executed under try/catch control. Thus, the executeUpdate (s) method is handled using:

```
try                          {    stm.executeUpdate(s);        }
catch (Exception exc)   {    out.println(exc.toString());   }
```

as in *testCase002.jsp*. If the update fails because there is an error in the query string submitted (in s), then the catch clause takes over. The error or exception is set by the system in exc. The body of the catch can output the error message as shown. It could also do whatever other processing the programmer deemed appropriate.

- **Inserting Data from an HTML Form in Database using JSP**

The JSP acquires the data to be inserted into a database from an HTML Form. This interaction involves several elements:

1. An HTML Form with named input fields.

2. JSP statements that access the data sent to the server by the form.
3. Construction of an insert query based on this data by the JSP program.
4. Execution of the query by the JSP program.

To demonstrate this process, we have to define the HTML page that will be accessed by the JSP program. We will use testCase000.html which has three input fields: mName, mAge, and mSalary. It identifies the requested JSP program as testCase003.jsp. For simplicity, initially assume the Access table has a single text attribute whose value is picked up from the Form. The example testCase003.jsp must acquire the Form data, prep it for the query, build the query, then execute it. It also sends a copy of the query to the browser so you can see the constructed query. The relevant statements are:

```
String   name    =   request.getParameter ("mName");
         name    =   "" + name + "" ;
String       s   =  "INSERT INTO Managers  VALUES ("   ;
             s +=  name  ;
             s +=  ")"   ;
stm.executeUpdate (s);
out.println ("<br>"+s+"<br>");
```

In the above code, the first statement acquires the form data, the second preps it for the database insert by attaching single quotes fore and aft, the next three statements construct the SQL insert query, the next statement executes the query, and the final statement displays it for review on the browser.

- **Delete Records from Database based on Criteria from HTML Form**

We can illustrate the application of an SQL delete query using the same HTML Form. The difference between the insert and the delete is that the delete has a where clause that determines which records are deleted from the database. Thus, to delete a record where the attribute *name* has the text value *Rahul*, the fixed SQL is:

```
String       s    =   "Delete From Managers Where name = '
Rahul ' "
```

- **Retrieve Data from Database - JSP ResultSets**

Retrieving data from a database is slightly more complicated than inserting or deleting data. The retrieved data has to be put someplace and in Java that place is called a ResultSet. A ResultSet object, which is essentially a table of the returned results as done for any SQL Select, is returned by an execute**Query** method, rather than the executeUpdate method used for inserts and deletes. The steps involved in a select retrieval are:

1. Construct a desired Select query as a Java string.

2. Execute the executeQuery method, saving the results in a ResultSet object r.

3. Process the ResultSet r using two of its methods which are used in tandem:

   a) r.next ( ) method moves a pointer to the next row of the retrieved table.

b) r.getString (attribute-name) method extracts the given attribute value from the currently pointed to row of the table.

A simple example of a Select is given in testCase04 where a fixed query is defined. The relevant code is:

```
String      s    = "SELECT * FROM Managers";
ResultSet   r    =  stm.executeQuery(s);


while ( r.next( ) )
 {
        out.print  ("<br>Name: " +  r.getString ("name") );
        out.println("     Age :    " +  r.getString ("age"  ) );
 }
```

The query definition itself is the usual SQL Select. The results are retrieved from the database using stm.executeQuery (s). The while loop (because of its repeated invocation of r.next( ) advances through the rows of the table which was returned in the ResultSet r. If this table is empty, the while test fails immediately and exits. Otherwise, it points to the row currently available. The values of the attributes in that row are then accessed using the getString method which returns the value of the attribute "name" or "age". If you refer to an attribute that is not there or misspell, you'll get an error *message* "Column not found". In this case, we have merely output the retrieved data to the HTML page being constructed by the JSP. Once, the whole table has been scanned, r.next( ) returns False and the while terminates. The entire process can be included in a try/catch combination for safety.

Check Your Progress 1

Fill in the blanks:

1) …… method connects the program to the database identified by the data source.
2) …….. method is used to execute the insert query.
3) ……. method is used to execute the select query.
4) In Java the result of select query is placed in ……..

4.3 INCLUDING FILES AND APPLETS IN JSP DOCUMENTS

Now in this section, we learn how to include external pieces into a JSP document. JSP has three main capabilities for including external pieces into a JSP document.

i) jsp:include action

This includes generated page, not JSP code. It cannot access environment of main page in included code.

ii) include directive

Include directive includes dynamic page, i.e., JSP code. It is powerful, but poses maintenance challenges.

iii) jsp:plugin action

This inserts applets that use the Java plug-in. It increases client side role in dialog.

Now, we will discuss in details about this capability of JSP.

- **Including Files at Page Translation Time**

You can include a file with JSP at the page translation time. In this case file will be included in the main JSP document at the time the document is translated into a servlet (which is typically the first time it is accessed).
To include file in this way, the syntax is:

<%@ include file= “Relative URL” %>

There are two consequences of the fact that the included file is inserted at page translation time, not at request time as with jsp: include.

First, you include the actual file itself, unlike with jsp:include , where the server runs the page and inserts its output. This approach means that the included file can contain JSP constructs (such as field or method declarations) that affect the main page as a whole.

Second, if the included file changes, all the JSP files that use it need to be updated.

- **Including Files at Request Time**

As you studied, the include directive provides the facility to include documents that contain JSP code into different pages. But this directive requires you to update the modification date of the page whenever the included file changes, which is a significant inconvenience.

Now, we will discuss about the **jsp:include** action. It includes files at the time of the client request and thus does not require you to update the main file when an included file changes. On the other hand, as the page has already been translated into a servlet at request time, thus the included files cannot contain JSP.

Although the included files cannot contain JSP, they can be the result of resources that use JSP to create the output. That is, the URL that refers to the included resource is interpreted in the normal manner by the server and thus can be a servlet or JSP page. This is precisely the behaviour of the include method of the RequestDispatcher class, which is what servlets use if they want to do this type of file inclusion.
The jsp:include element has two required attributes (as shown in the sample below), these elements are:

i) **page** : It refers to a relative URL referencing the file to be included

ii) **flush:** This must have the value true.

```
<jsp:include page="Relative URL" flush="true" />
```

Although you typically include HTML or plain text documents, there is no requirement that the included files have any particular file extension.

- **Including Applets for the Java Plug-In**

To include ordinary applets with JSP, you don't need any special syntax; you need to just use the normal HTML APPLET tag. But, these applets must use JDK 1.1 or JDK 1.02, because neither Netscape 4.x nor Internet Explorer 5.x support the Java 2 platform (i.e., JDK 1.2). This lack of support imposes several restrictions on applets these are:

- In order to use Swing, you must send the Swing files over the network. This process is time consuming and fails in Internet Explorer 3 and Netscape 3.x and 4.01-4.05 (which only support JDK 1.02), since Swing depends on JDK 1.1.
- You cannot use Java 2D.
- You cannot use the Java 2 collections package.
- Your code runs more slowly, since most compilers for the Java 2 platform are significantly improved over their 1.1 predecessors.

To solve this problem, Sun developed a browser plug-in for Netscape and Internet Explorer that lets you use the Java 2 platform for applets in a variety of browsers. It is a reasonable alternative for fast corporate intranets, especially since applets can automatically prompt browsers that lack the plug-in to download it. But, since, the plug-in is quite large (several megabytes), it is not reasonable to expect users on the WWW at large to download and install it just to run your applets. As well as, the normal APPLET tag will not work with the plug-in, since browsers are specifically designed to use only their built-in virtual machine when they see APPLET. Instead, you have to use a long and messy OBJECT tag for Internet Explorer and an equally long EMBED tag for Netscape. Furthermore, since, you typically don't know which browser type will be accessing your page, you have to either include both OBJECT and EMBED (placing the EMBED within the COMMENT section of OBJECT) or identify the browser type at the time of the request and conditionally build the right tag. This process is straightforward but tedious and time consuming.

The **jsp:plugin** element instructs the server to build a tag appropriate for applets that use the plug-in. Servers are permitted some leeway in exactly how they implement this support, but most simply include both OBJECT and EMBED.

The simplest way to use jsp:plugin is to supply four attributes: type, code, width, and height. You supply a value of applet for the type attribute and use the other three attributes in exactly the same way as with the APPLET element, with two exceptions, i.e., the attribute names are case sensitive and single or double quotes are always required around the attribute values.
For example, you could replace

```
<APPLET CODE= “MyApplet.class” WIDTH=475 HEIGHT=350>
</APPLET>
```

with

```
<jsp:plugin type="applet" code="MyApplet.class" width="475"
height="350">
</jsp:plugin>
```

The jsp:plugin element has a number of other optional attributes. A list of these attributes is:

- **type:**

For applets, this attribute should have a value of applet. However, the Java Plug-In also permits you to embed JavaBeans elements in Web pages. Use a value of bean in such a case.

- **code :**

This attribute is used identically to the CODE attribute of APPLET, specifying the top-level applet class file that extends Applet or JApplet. Just remember that the name code must be lower case with jsp:plugin (since, it follows XML syntax), whereas with APPLET, case did not matter (since, HTML attribute names are never case sensitive).

- **width :**

This attribute is used identically to the WIDTH attribute of APPLET, specifying the width in pixels to be reserved for the applet. Just remember that you must enclose the value in single or double quotes.

- **height :**

This attribute is used identically to the HEIGHT attribute of APPLET, specifying the height in pixels to be reserved for the applet. Just remember that you must enclose the value in single or double quotes.

- **codebase :**

This attribute is used identically to the CODEBASE attribute of APPLET, specifying the base directory for the applets. The code attribute is interpreted relative to this directory. As with the APPLET element, if you omit this attribute, the directory of the current page is used as the default. In the case of JSP, this default location is the directory where the original JSP file resided, not the system-specific location of the servlet that results from the JSP file.

- **align:**

This attribute is used identically to the ALIGN attribute of APPLET and IMG, specifying the alignment of the applet within the web page. Legal values are left, right, top, bottom, and middle. With jsp:plugin, don’t forget to include these values in single or double quotes, even though quotes are optional for APPLET and IMG.

- **hspace :**

This attribute is used identically to the HSPACE attribute ofAPPLET, specifying empty space in pixels reserved on the left and right of the applet. Just remember that you must enclose the value in single or double quotes.

- **vspace :**

This attribute is used identically to the VSPACE attribute of APPLET, specifying empty space in pixels reserved on the top and bottom of the applet. Just remember that you must enclose the value in single or double quotes.

- **archive :**

This attribute is used identically to the ARCHIVE attribute of APPLET, specifying a JAR file from which classes and images should be loaded.

- **name :**

This attribute is used identically to the NAME attribute of APPLET, specifying a name to use for inter-applet communication or for identifying the applet to scripting languages like JavaScript.

- **title :**

This attribute is used identically to the very rarely used TITLE attribute of APPLET (and virtually all other HTML elements in HTML 4.0), specifying a title that could be used for a tool-tip or for indexing.

- **jreversion :**

This attribute identifies the version of the Java Runtime Environment (JRE) that is required. The default is 1.1.

- **iepluginurl :**

This attribute designates a URL from which the plug-in for Internet Explorer can be downloaded. Users who don't already have the plug-in installed will be prompted to download it from this location. The default value will direct the user to the Sun site, but for intranet use you might want to direct the user to a local copy.

- **nspluginurl :**

This attribute designates a URL from which the plug-in for Netscape can be downloaded. The default value will direct the user to the Sun site, but for intranet use you might want to direct the user to a local copy.

The jsp:param and jsp:params Elements

The jsp:param element is used with jsp:plugin in a manner similar to the way that PARAM is used with APPLET, specifying a name and value that are accessed from within the applet by getParameter. There are two main differences between jsp:param and param with applet, these are :

First, since jsp:param follows XML syntax, attribute names must be lower case, attribute values must be enclosed in single or double quotes, and the element must end with />, not just >.

Second, all jsp:param entries must be enclosed within a jsp:params element.
So, for example, you would replace

```
<APPLET CODE= "MyApplet.class" WIDTH=475 HEIGHT=350>
        <PARAM NAME="PARAM1" VALUE= "VALUE1">
        <PARAM NAME= "PARAM2" VALUE= "VALUE2">
</APPLET>
```

with

```
<jsp:plugin type= "applet" code= "MyApplet.class" width= "475" height="350">
        <jsp:params>
                <jsp:param name="PARAM1" value= "VALUE1" />
                <jsp:param name= "PARAM2" value="VALUE2" />
        </jsp:params>
</jsp:plugin>
```

The jsp:fallback Element

The jsp:fallback element provides alternative text to browsers that do not support OBJECT or EMBED. You use this element in almost the same way as you would use alternative text placed within an APPLET element.
So, for example, you would replace

```
<APPLET CODE="MyApplet.class" WIDTH=475 HEIGHT=350>
        <B>Error: this example requires Java.</B>
</APPLET>
```

with

```
<jsp:plugin type="applet" code= "MyApplet.class" width="475" height="350">
        <jsp:fallback>
                <B>Error: this example requires Java.</B>
        </jsp:fallback>
</jsp:plugin>
```

4.4 INTEGRATING SERVLET AND JSP

As you have seen, servlets can manipulate HTTP status codes and headers, use cookies, track sessions, save information between requests, compress pages, access databases, generate GIF images on-the-fly, and perform many other tasks flexibly and efficiently. Therefore servlets are great when your application requires a lot of real programming to accomplish its task.

But, generating HTML with servlets can be tedious and can yield a result that is hard to modify. That's where JSP comes in; it let's you separate much of the presentation from the dynamic content. That way, you can write the HTML in the normal manner, even using HTML-specific tools and putting your web content developers to work on your JSP documents.

Now, let us discuss the limitation of JSP. As you know, the assumption behind a JSP document is that it provides a single overall presentation. But what if you want to give totally different results depending on the data that you receive? Beans and custom tags, although extremely powerful and flexible, but they don't overcome the limitation that the JSP page defines a relatively fixed top-level page appearance. The solution is to use both servlets and Java Server Pages. If you have a complicated application that may require several substantially different presentations, a servlet can handle the initial request, partially process the data, set up beans, then forward the results to one of a number of different JSP pages, depending on the circumstances.

In early JSP specifications, this approach was known as the model 2 approach to JSP. Rather than completely forwarding the request, the servlet can generate part of the output itself, then include the output of one or more JSP pages to obtain the final result.

4.4.1 Forwarding Requests

The key to letting servlets forward requests or include external content is to use a requestDispatcher. You obtain a RequestDispatcher by calling the getRequestDispatcher method of ServletContext, supplying a URL relative to the server root.

For example, to obtain a RequestDispatcher associated with http://yourhost/presentations/presentation1.jsp, you would do the following:

```
String url = “/presentations/presentation1.jsp”;
RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher(url);
```

Once, you have a RequestDispatcher, you use forward to completely transfer control to the associated URL and use include to output the associated URL’s content. In both cases, you supply the HttpServletRequest and HttpServletResponse as arguments. Both methods throw Servlet-Exception and IOException. For example, the example 4.1 shows a portion of a servlet that forwards the request to one of three different JSP pages, depending on the value of the operation parameter. To avoid repeating the getRequestDispatcher call, I use a utility method called gotoPage that takes the URL, the HttpServletRequest and the HttpServletResponse; gets a RequestDispatcher; and then calls forward on it.

Example 4.1: Request Forwarding Example

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        String operation = request.getParameter(“operation”);
        if (operation == null) {
                operation = “unknown”;
        }
        if (operation.equals(“operation1”)) {
                gotoPage(“/operations/presentation1.jsp”,request, response);
        }
        else if (operation.equals(“operation2”)) {
                gotoPage("/operations/presentation2.jsp", request, response);
        }
        else {
                gotoPage("/operations/unknownRequestHandler.jsp",request,
                response);
        }
}

private void gotoPage(String address, HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
    RequestDispatcher dispatcher
=getServletContext().getRequestDispatcher(address);
    dispatcher.forward(request, response);
}
```

Using Static Resources

In most cases, you forward requests to a JSP page or another servlet. In some cases, however, you might want to send the request to a static HTML page. In an e-commerce site, for example, requests that indicate that the user does not have a valid account name might be forwarded to an account application page that uses HTML forms togather the requisite information. With GET requests, forwarding requests to a static HTML page is perfectly legal and requires no special syntax; just supply the address of the HTML page as the argument to getRequestDispatcher. However, since, forwarded requests use the same request method as the original request, POST requests cannot be forwarded to normal HTML pages. The solution to this problem is to simply rename the HTML page to have a .jsp extension. Renaming somefile.html to somefile.jsp does not change its output for GET requests, but somefile.html cannot handle POST requests, whereas somefile.jsp gives an identical response for both GET and POST.

Supplying Information to the Destination Pages

To forward the request to a JSP page, a servlet merely needs to obtain a RequestDispatcher by calling the getRequestDispatcher method of ServletContext, then call forward on the result, supplying the Http- ServletRequest and HttpServletResponse as arguments. That's fine as far as it goes, but this approach requires the destination page to read the information it needs out of the HttpServletRequest. There are two reasons why it might not be a good idea to have the destination page look up and process all the data itself. *First*, complicated programming is easier in a servlet than in a JSP page. *Second*, multiple JSP pages may require the same data, so it would be wasteful for each JSP page to have to set up the same data. A better approach is for, the original servlet to set up the information that the destination pages need, then store it somewhere that the destination pages can easily access. There are two main places for the servlet to store the data that the JSP pages will use: in the HttpServletRequest and as a bean in the location specific to the scope attribute of jsp:useBean.

The originating servlet would store arbitrary objects in the HttpServlet-Request by using

request.setAttribute("key1", value1);

The destination page would access the value by using a JSP scripting element to call

Type1 value1 = (Type1)request.getAttribute("key1");

For complex values, an even better approach is to represent the value as a bean and store it in the location used by jsp:useBean for shared beans. For example, a scope of application means that the value is stored in the ServletContext, and ServletContext uses setAttribute to store values. Thus, to make a bean accessible to all servlets or JSP pages in the server or Web application, the originating servlet would do the following:

```
Type1 value1 = computeValueFromRequest(request);
getServletContext().setAttribute("key1", value1);
```

The destination JSP page would normally access the previously stored value by using jsp:useBean as follows:

```
<jsp:useBean id= "key1" class= "Type1" scope="application" />
```

Alternatively, the destination page could use a scripting element to explicitly call application.getAttribute("key1") and cast the result to Type1. For a servlet to make

data specific to a user session rather than globally accessible, the servlet would store the value in the HttpSession in the normal manner, as below:

```
Type1 value1 = computeValueFromRequest(request);
HttpSession session = request.getSession(true);
session.putValue("key1", value1);
```

The destination page would then access the value by means of

```
<jsp:useBean id= "key1" class= "Type1" scope= "session" />
```

The Servlet 2.2 specification adds a third way to send data to the destination page when using GET requests: simply append the query data to the URL. For example,

```
String address = "/path/resource.jsp?newParam=value";
RequestDispatcher dispatcher =getServletContext().getRequestDispatcher(address);
dispatcher.forward(request, response);
```

This technique results in an additional request parameter of newParam (with a value of value) being added to whatever request parameters already existed. The new parameter is added to the beginning of the query data so that it will replace existing values if the destination page uses getParameter (use the first occurrence of the named parameter) rather than get- ParameterValues (use all occurrences of the named parameter).

Interpreting Relative URLs in the Destination Page

Although a servlet can forward the request to arbitrary locations on the same server, the process is quite different from that of using the sendRedirect method of HttpServletResponse.

First, sendRedirect requires the client to reconnect to the new resource, whereas the forward method of RequestDispatcher is handled completely on the server.

Second, sendRedirect does not automatically preserve all of the request data; forward does.

Third, sendRedirect results in a different final URL, whereas with forward, the URL of the original servlet is maintained.
This final point means that, if the destination page uses relative URLs for images or style sheets, it needs to make them relative to the server root, not to the destination page's actual location. For example, consider the following style sheet entry:

```
<LINK REL=STYLESHEET HREF= "my-styles.css" TYPE= "text/css">
```

If the JSP page containing this entry is accessed by means of a forwarded request, my-styles.css will be interpreted relative to the URL of the originating servlet, not relative to the JSP page itself, almost certainly resulting in an error. The solution is to give the full server path to the style sheet file, as follows:
<LINK REL=STYLESHEET HREF= "/path/my-styles.css" TYPE= "text/css">
The same approach is required for addresses used in <IMG SRC=...> and <A HREF=...>.

4.4.2 Including Static or Dynamic Content

If a servlet uses the forward method of RequestDispatcher, it cannot actually send any output to the client—it must leave that entirely to the destination page. If the servlet wants to generate some of the content itself but use a JSP page or static HTML

document for other parts of the result, the servlet can use the include method of RequestDispatcher instead. The process is very similar to that for forwarding requests: Call the getRequestDispatcher method of ServletContext with an address relative to the server root, then call include with the HttpServletRequest and HttpServletResponse.

The two differences when include is used are that you can send content to the browser before making the call and that control is returned to the servlet after the include call finishes. Although the included pages (servlets, JSP pages, or even static HTML) can send output to the client, they should not try to set HTTP response headers. Here is an example:

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("...");
RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher("/path/resource");
dispatcher.include(request, response);
out.println("...");
```

The include method has many of the same features as the forward method. If the original method uses POST, so does the forwarded request. Whatever request data was associated with the original request is also associated with the auxiliary request, and you can add new parameters (in version 2.2 only) by appending them to the URL supplied to getRequestDispatcher. Version 2.2 also supports the ability to get a RequestDispatcher by name (getNamedDispatcher) or by using a relative URL (use the getRequestDispatcher method of the HttpServletRequest). However, include does one thing that forward does not: it automatically sets up attributes in the Http-ServletRequest object that describe the original request path in case, the included servlet or JSP page needs that information. These attributes, available to the included resource by calling getAttribute on the Http-ServletRequest, are listed below:

- javax.servlet.include.request_uri
- javax.servlet.include.context_path
- javax.servlet.include.servlet_path
- javax.servlet.include.path_info
- javax.servlet.include.query_string

Note that this type of file inclusion is not the same as the nonstandard servlet chaining supported as an extension by several early servlet engines. With servlet chaining, each servlet in a series of requests can see (and modify) the output of the servlet before it. With the include method of RequestDispatcher, the included resource cannot see the output generated by the original servlet. In fact, there is no standard construct in the servlet specification that reproduces the behaviour of servlet chaining.

Also note that this type of file inclusion differs from that supported by the JSP include directive. There, the actual source code of JSP files was included in the page by use of the include directive, whereas the include method of RequestDispatcher just includes the result of the specified resource. On the other hand, the jsp:include action has behavior similar to that of the include method, except that jsp:include is available only from JSP pages, not from servlets.

4.4.3 Forwarding Requests from JSP Pages

The most common request forwarding scenario is that the request first comes to a servlet and the servlet forwards the request to a JSP page. The reason is a servlet usually handles the original request is that checking request parameters and setting up beans requires a lot of programming, and it is more convenient to do this programming in a servlet than in a JSP document. The reason that the destination page is usually a JSP document is that JSP simplifies the process of creating the HTML content.

However, just because this is the usual approach doesn't mean that it is the only way of doing things. It is certainly possible for the destination page to be a servlet. Similarly, it is quite possible for a JSP page to forward requests elsewhere. For example, a request might go to a JSP page that normally presents results of a certain type and that forwards the request elsewhere only when it receives unexpected values. Sending requests to servlets instead of JSP pages requires no changes whatsoever in the use of the RequestDispatcher. However, there is special syntactic support for forwarding requests from JSP pages. In JSP, the jsp:forward action is simpler and easier to use than wrapping up Request-Dispatcher code in a scriptlet. This action takes the following form:

```
<jsp:forward page= "Relative URL" />
```

The page attribute is allowed to contain JSP expressions so that the destination can be computed at request time.

For example, the following sends about half the visitors to http://host/examples/page1.jsp and the others to http://host/examples/page2.jsp.

```
<%
    String destination;
    if (Math.random() > 0.5) {
        destination = "/examples/page1.jsp";
    }
    else {
        destination = "/examples/page2.jsp";
    }
%>
<jsp:forward page= "<%= destination %>" />
```

Check Your Progress 2

Give right choice for the following:

1) ---------- includes generated page, not JSP code.

   a) include directive
   b) jsp:include action
   c) jsp:plugin action

2) ---------- attribute of jsp:plugin designates a URL from which the plug-in for Internet Explorer can be downloaded.

   a) Codebase
   b) Archive
   c) Iepluginurl
   d) Nspluginurl

3) A RequestDispatcher can obtain by calling the ---------- method of ----------, supplying a URL relative to the server root.

   a) ServletContext, getRequestDispatcher
   b) HttpServletRequest, getRequestDispatcher
   c) getRequestDispatcher, HttpServletRequest
   d) getRequestDispatcher, ServletContext

Explain following questions in brief:

4) Write a note on jsp:fallback Element.

   ……..……..……

   ……

   ……

4.5 SUMMARY

In this unit, we first studied about database handling in JSP. In this topic we saw that to communicate with a database through JSP, first that database is needed to be registered on your system. Then JSP makes a connection with the database. For this purpose DriverManager.getConnection method is used. This method connects the program to the database identified by the data source by creating a connection object.

This object is used to make SQL queries. For making SQL queries, a statement object is used. This statement object is created by using connection object's createStatement ( ) method. Finally statement object's executeUpdate method is used to execute the insert and delete query and executeQuery method is used to execute the SQL select query.

Next, we studied about how to include files and applets in JSP documents. In this, we studied that JSP has three main capabilities for including external pieces into a JSP document. These are:

- *jsp:include action*: It is used to include generated pages, not JSP code.
- *include directive*: It includes dynamic page, i.e., JSP code.
- *jsp:plugin action*: This inserts applets that use the Java plug-in.

Finally we studied about integrating servlet and JSP. The key to let servlets forward requests or include external content is to use a requestDispatcher. This RequestDispatcher can be obtained by calling the getRequestDispatcher method of ServletContext, supplying a URL relative to the server root. But if you want to forward the request from JSP, jsp:forward action is used.

4.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1) DriverManager.getConnection
2) Statement object's executeUpdate
3) Statement object's executeQuery
4) ResultSet

Check Your Progress 2

1) b

2) c

3) d

4) The jsp:fallback element provides alternative text to browsers that do not support OBJECT or EMBED. This element can be used in almost the same way as alternative text is placed within an APPLET element.

For example,

```
<APPLET CODE=: “MyApplet.class” WIDTH=475 HEIGHT=350>
        <B>Error: this example requires Java.</B>
</APPLET>
```

can be replaced with

```
<jsp:plugin type= “applet” code= “MyApplet.class” width= “475” height= “350”>
        <jsp:fallback>
                <B>Error: this example requires Java.</B>
        </jsp:fallback>
</jsp:plugin>
```

4.7 FURTHER READINGS/REFRENCES

- Phil Hanna, *JSP: The Complete Reference*
- Hall, Marty, *Core Servlets and JavaServer Pages*
- Annunziato Jose & Fesler, Stephanie, *Teach Yourself JavaServer Pages in 24 Hours,*
- Bruce W. Perry, *Java Servlet & JSP*, Cookbook (Paperback),