# Creating a powerful database client with JSP

Abstract
This article shows a unique way of connecting to any RDBMS through JSP script. The goal is accomplished by only one JSP file that does not depend on any external bean, servlet or JSP, hence making this a zero install solution. This article does not talk about how to query a database but is focused on the implementation problems that were introduced by imposing a requirement of having only one JSP file. It discusses the following issues. a) How to write user defined methods within a JSP script. b) How to mix HTML and Java code within these methods. c) How to monitor session events within JSP a script.

Java Server Pages (JSP) is a powerful technology that is usually used to build the presentation logic in a multi-tiered system. Although this practice fits most of the design models, this article discusses a scenario where this rule is bent and the outcome is very useful – a small web application that can be used to submit SQL queries to any JDBC compliant database that requires no installation.

Before I talk about the actual JSP script, let me give a short background that inspired me to sit down and to write this script. I was working on the typical J2EE web based application when I discovered a need to connect to my database, Oracle in this case, to run some manual queries from a remote location. This remote location happened to be my home and the only way to submit SQL queries was either through HTTP or HTTPS since the firewall at work did not allow direct communication to my Oracle database.

To solve this problem I decided to write a very small JSP script that would accept a SQL string, send it to my database and finally send the result back to the client browser. I chose JSP to accomplish this goal for two reasons. First, deployment is extremely easy – I do not need to modify any deployment descriptor or change any server configuration. Second, I do not need to compile the script after modifications; the container takes care of that part. And finally, it allows rapid development.

My main objective in this exercise was to keep deployment simple so that this solution can be copied to a container without any modifications to the system. For this reason, I decided to write this script in only one self-contained file which does not depend on any external bean, servlet or EJB. Writing the entire script in one file introduced some interesting problems and I discuss their solutions in this article. Before addressing these problems, let's discuss the flow of events that takes place. This is illustrated in Figure 1.
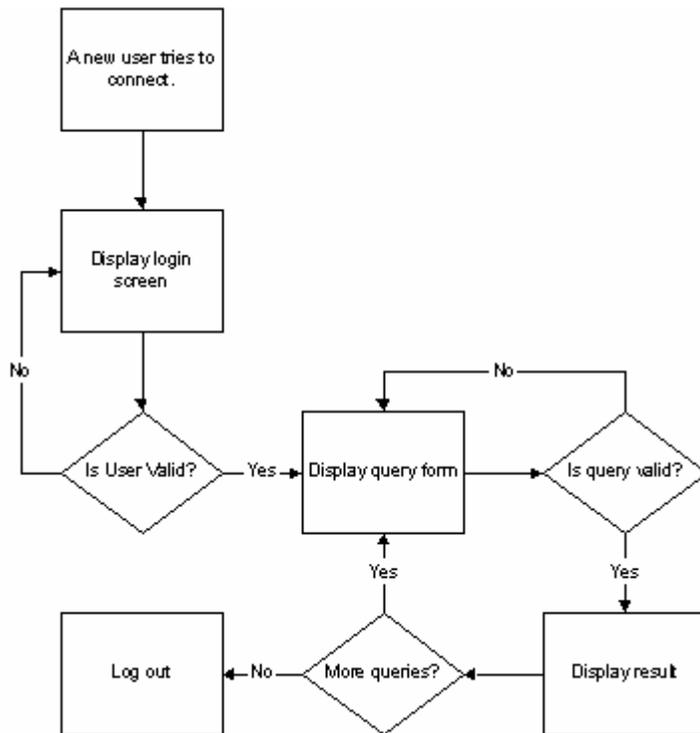
Figure 1

The flow is pretty trivial to understand. When a user first tries to connect, an HTML form is displayed where he/she can enter necessary database parameters. Upon form submission, the same JSP file receives the request and tries to establish connection with the database. This login screen is displayed again if any of the parameter are incorrect. Once a connection is established, this object is stored in the session for subsequent calls and another form is displayed where the user can type SQL scripts. The same JSP script captures this SQL query and sends it the database. In case of an error, the query form is displayed again with an appropriate error message. If the query is successful the result is displayed in HTML.

Operations

Since I submit the request to the same JSP file, there must be a mechanism to separate the logic of one request from another. This can be easily done by having a parameter in the request that distinguishes different operations performed by the script. For obvious reasons, I call this parameter an "operation". If this is a new session the login screen is displayed. The submission of the login form changes the value of the operation parameter to "1", which routes the request to the session of the code that establishes database connection. Similar logic is used on all subsequent calls where the appropriate value of the operation parameter steers the flow to different parts of the file. Each operation is implemented as an "if" statement and a pseudo code is given below.

```
String operation = request.getParameter("operation");
```

```
if(operation.equals("1")){

      if( isConnectionValid )
            displayQueryScreen
      else
            displayLoginForm


}else if(operation.equals("2")){

      getSQLString
      runQuery

}else if(operation.equals("3")){

      doSomethingElse
}
```

The table below lists all the operations that are implemented so far. If you like, you can always add additional operations to suit your requirements.

Operation	Description
NULL	Display login screen
1	Establish database connection and show the query form upon success
2	Run query
3	Log out and close database connection
4	New query
5	Display catalog menu
6	Show list of all tables
7	Show list of all views
8	Show list of all procedures
9	Show details for a table or view

My first hurdle

Although JSP scripts are written in Java, it imposes certain restrictions that are usually solved by Java beans. However, I did not have an option of using beans since they would have to be in a separate file and require compilation.

Notice that in our flow some of the HTML forms, particularly the query input screen, get displayed as a result of several operations. This means that the code to create the HTML script for the form must appear in multiple locations within the JSP script. This was the first time I felt a need to write regular Java methods within JSP scripts. I wish I could do something like:

```
<%
```

```
void displayQueryForm(){
%>
      <form method="get">
            Input Query <input type="text" name="txtInput"><br>
            <input type="submit">
      </form>
<%
}

...

if(operation.equals("1"))
      displayQueryForm();      //Call a method that is previous defined
else if(operation.equals("4")){
      displayQueryForm();      //Call the same method again
}
%>
```

I did not like the idea of copying and pasting the same code in multiple places and had to scratch my head to find a solution. Suddenly two words came into my mind: Inner Classes. Java allows an inner class within the context of a method and sure enough this method provided me a way to write methods that I can call from different locations as many times as I wanted at the cost of an extra object creation. I call this Worker class. Now my code looks like:

```
<%

class Worker(){
    void displayQueryForm(){
    %>
      <form method="get">
            Input Query <input type="text" name="txtInput"><br>
            <input type="submit">
      </form>
    <%
    }
}

...

Worker worker = new Worker();

if(operation.equals("1"))
      worker.displayQueryForm();//Call a method in Worker
else if(operation.equals("4")){
      worker.displayQueryForm();//Call the same method again
}
%>
```

A reminder: you cannot create static methods in an inner class. Therefore, you cannot avoid creating an extra Worker object.

My second hurdle

After I figured out to use Inner classes, I thought now my code would be much cleaner. However, after I saved my JSP file and called it through my browser, it generated a whole bunch of compilation errors. It turns out that I could not use embedded HTML code within the methods of my inner class since it did not have access to the "out" variable which is defined in the _jspService method of the generated Servlet code, unless "out" was declared as final; and obviously I did not have the option of classifying it as final variable since it is declared by the container.

So I changed my code to look like:

```
<%

class Worker(){
    String displayQueryForm(){

      StringBuffer buffer = new StringBuffer();

      buffer.append("<form method=\"get\">").append("\n");
      buffer.append("<Input Query <input type=\"text\"
name=\"txtInput\"><br>").append("\n");
      buffer.append("   <input type="submit">").append("\n");
      buffer.append("</form>").append("\n");


    }
}

...

Worker worker = new Worker();

if(operation.equals("1"))
      out.println(worker.displayQueryForm());
else if(operation.equals("4")){
      out.println(worker.displayQueryForm());
}
%>
```

Here I return the required HTML back to the caller, which then writes it to the output. After this change I got the JSP file to compile and run the way I wanted, however, I did not like the idea of loosing the ability to embed HTML code within my methods. After all that is what makes JSP script so powerful and maintainable.

To find a solution to this problem I had to open up the Servlet code that is generated for every JSP script by the container. I noticed that all the embedded HTML code was replaced either by a "out.write(…)" or an "out.print(…)" statement with the HTML code as its parameter. This meant that if I had a local parameter called "out" within my methods, the container would not know that this was an inner class and generated code would compile without any problem.

So I re-factor my code to look like:

```
<%

class Worker(){
    void displayQueryForm(JspWriter out) throws IOException{
    %>
      <form method="get">
            Input Query <input type="text" name="txtInput"><br>
            <input type="submit">
      </form>
    <%
    }
}

...

Worker worker = new Worker();

if(operation.equals("1"))
      worker.displayQueryForm(out);//Call a method in Worker
else if(operation.equals("4")){
      worker.displayQueryForm(out);//Call the same method again
}
%>
```

Note that all I did was to pass the "out" variable from the main class to the method in my inner class and kept the name intact. Now "out" becomes a local variable within the method and the compiler happily accepts it.

My third hurdle

Creating database connections is an expensive operation and in most applications connection objects are pooled where a pool manager creates and closes connections as necessary. The use of a connection pool requires either a custom pool manager or a driver that supports JDBC 2.0 extensions. A custom pool manager would require additional files requiring compilation that would forfeit the feature of zero deployment and I did not know what kind of JDBC drivers would be used at runtime. Therefore, a new database connection is created when a user connects the first time and it is saved in the HTTPSession object for subsequent requests and is finally closed when the user logs out. This poses a question: what happens if the user never logs out? This would leave the connection open until a) the database server kills it or b) the J2EE container is stopped.

Unfortunately, both of these options are not good enough and I had to come up with another solution.

A little research revealed that the container sends messages to any object when it is bound or unbound to the session if that object implements HttpSessionBindingListener interface. That meant that all I needed was to wrap the connection object in a class that implemented this interface and that is exactly what the ConnectionHolder class does. Refer to the code below.

```
class ConnectionHolder implements HttpSessionBindingListener{
      private Connection dbConn = null;

      ConnectionHolder(Connection dbConn){
           this.dbConn = dbConn;
      }

      public void valueBound(HttpSessionBindingEvent event){


      }

      public void valueUnbound(HttpSessionBindingEvent event){
           try{
                 if(dbConn != null){
                      dbConn.close(); //Close connection when
                                     //session becomes invalid
                 }
           }catch(SQLException se){
                 se.printStackTrace();
           }
      }
      Connection getDbConn(){
           return dbConn;
      }

}
```

Now all I had to do was to create an object of ConnectionHolder class with a new connection, store this object in the session and let the container tell us when the session becomes invalid, which could happen when a user gracefully logs out or when session timeout occurs.

Security

My topmost concern in using this file over the Internet was Security. Using plain HTTP over the Internet is never secure even if you change your database passwords religiously every week. There is always a potential of a hacker watching behind your shoulders and capturing all the necessary connection parameters; and at a later time he/she can come in

and have access to your entire database and possibly other resources in your system. If somehow I could force my users to use HTTPS instead, I could rule out the possibility of someone tampering or watching my keystrokes. Thanks to the "isSecure" method in the ServletRequest class I was able to solve this problem. By calling this method I can find out if the client is using a secure mechanism like SSL to connect and can deny any request that is not secure. Although there is a price to pay that is related to performance, it provides a secure mechanism that will get the blessing of your network administrators and DBAs.

Summary

This article discusses about some limitations that are faced by JSP programmer and provides work-around for them. Although most of systems do not use a JSP script to write business logic, there is no harm in bending this rule if your application is very small – small enough to fit in one file. It is extremely easy to develop, deploy, and maintain a JSP file and therefore, this technology is a perfect candidate for small applications.

About the author

Imran Hussain is a Java Architect working for Sun Microsystems. He holds bachelors degree in Computer Engineering and has over nine years of industry experience. He is a veteran C++ programmer and started Java in early 1997 and has been actively involved in J2EE architecture development for the pass two years.