Tutorial about JSPs

Topics in this tutorial:

This tutorial is available at: http://www.jsptut.com and http://mama.indstate.edu/users/ashok/docs/jsp/index.html

1- Getting familiar with your JSP server

If you do not have a JSP capable Web-server (sometimes known as *application servers* for configuration reasons), the first step is to download one. There are many such servers available, most of which can be downloaded for free evaluation and/or development. Some of them are:
- Blazix from Desiderata Software (*1.5 Megabytes, JSP, Servlets and EJBs*)
- ServletExec from New Atlanta/Unify (*3.8 Megabytes, JSP and Servlets*)
- JRun from Allaire (*11 Megabytes, JSP, Servlets and EJBs*)
- WebLogic from BEA Systems (*44 Megabytes, JSP, Servlets and EJBs*)
- WebSphere from IBM (*105 Megabytes, JSP, Servlets and EJBs*)

If you do not already have a server, it is recommended that you download Blazix (from http://www.blazix.com) because it includes a tag library that is used later in this tutorial in the tag library chapter. Blazix is also very small and can be easily downloaded.
Once you have a Web-server, you need to know the following information about your Web-server:

- Where to place the files
- How to access the files from your browser (with an `http:` prefix, not as `file:`)

You should be able to create a simple file, such as

```
<HTML>
<BODY>
Hello, world
</BODY>
</HTML>
```

know where to place this file and how to see it in your browser with an `http://` prefix.
Since this step is different for each Web-server, you would need to see the Web-server documentation to find out how this is done. Once you have completed this step, proceed to the next section.

2- Your first JSP

JSP simply puts Java inside HTML pages. You can take any existing HTML page and change its extension to ".jsp" instead of ".html". In fact, this is the perfect exercise for your first JSP.

Take the HTML file you used in the previous exercise. Change its extension from ".html" to ".jsp". Now load the new file, with the ".jsp" extension, in your browser. You will see the same output, but it will take longer! But only the first time. If you reload it again, it will load normally.

What is happening behind the scenes is that your JSP is being turned into a Java file, compiled and loaded. This compilation only happens once, so after the first load, the file doesn't take long to load anymore. (But everytime you change the JSP file, it will be re-compiled again.)

3- Adding dynamic content via expressions

As we saw in the previous section, any HTML file can be turned into a JSP file by changing its extension to `.jsp`. Of course, what makes JSP useful is the ability to embed Java. Put the following text in a file with .jsp extension (let us call it **`hello.jsp`**), place it in your JSP directory, and view it in a browser.

```
<HTML>
<BODY>
Hello!  The time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

Notice that each time you reload the page in the browser, it comes up with the current time.
The character sequences **`<%=`** and **`%>`** enclose Java expressions, which are evaluated at run time.
This is what makes it possible to use JSP to generate dynamic HTML pages that change in response to user actions or vary from user to user.

Exercise: Write a JSP to output the values returned by `System.getProperty` for various system properties such as `java.version`, `java.home`, `os.name`, `user.name`, `user.home`, `user.dir` etc.

4- Scriptlets

We have already seen how to embed Java expressions in JSP pages by putting them between the **`<%=`** and **`%>`** character sequences.

But it is difficult to do much programming just by putting Java expressions inside HTML.

JSP also allows you to write blocks of Java code inside the JSP. You do this by placing your Java code between **`<%`** and **`%>`** characters (just like expressions, but without the **=** sign at the start of the sequence.)

This block of code is known as a "scriptlet". By itself, a scriptlet doesn't contribute any HTML (though it can, as we will see down below.) A scriptlet contains Java code that is executed every time the JSP is invoked. Here is a modified version of our JSP from previous section, adding in a scriptlet.

```
<HTML>
<BODY>
<%
    // This is a scriptlet.  Notice that the "date"
    // variable we declare here is available in the
    // embedded expression later on.
    System.out.println( "Evaluating date now" );
    java.util.Date date = new java.util.Date();
%>
Hello!  The time is now <%= date %>
</BODY>
</HTML>
```

If you run the above example, you will notice the output from the "`System.out.println`" on the **server log**. This is a convenient way to do simple debugging (some servers also have techniques of debugging the JSP in the IDE. See your server's documentation to see if it offers such a technique.)

By itself a scriptlet does not generate HTML. If a scriptlet wants to generate HTML, it can use a variable called "**out**". This variable does not need to be declared. It is already predefined for scriptlets, along with some other variables. The following example shows how the scriptlet can generate HTML output.

```
<HTML>
<BODY>
<%
    // This scriptlet declares and initializes "date"
    System.out.println( "Evaluating date now" );
    java.util.Date date = new java.util.Date();
%>
Hello!  The time is now
<%
    // This scriptlet generates HTML output
    out.println( String.valueOf( date ));
%>
</BODY>
</HTML>
```

Here, instead of using an expression, we are generating the HTML directly by printing to the "`out`" variable. The "`out`" variable is of type: `javax.servlet.jsp.JspWriter`.

Another very useful pre-defined variable is "`request`". It is of type:
`javax.servlet.http.HttpServletRequest`

A "**request**" in server-side processing refers to the transaction between a browser and the server. When someone clicks or enters a URL, the browser sends a "request" to the server for that URL, and shows the data returned. As a part of this "request", various data is available, including the file the browser wants from the server, and if the request is coming from pressing a SUBMIT button, the information the user has entered in the form fields.

The JSP "**`request`**" variable is used to obtain information from the request as sent by the browser. For instance, you can find out the name of the client's host (if available, otherwise the IP address will be returned.) Let us modify the code as shown:

```
<HTML>
<BODY>
<%
    // This scriptlet declares and initializes "date"
    System.out.println( "Evaluating date now" );
    java.util.Date date = new java.util.Date();
%>
Hello!  The time is now
<%
    out.println( date );
    out.println( "<BR>Your machine's address is " );
    out.println( request.getRemoteHost());
%>
</BODY>
</HTML>
```

A similar variable is "**response**". This can be used to affect the response being sent to the browser. For instance, you can call `response.sendRedirect(anotherUrl);` to send a response to the browser that it should load a different URL. This response will actualy go all the way to the browser. The browser

will then send a different request, to "`anotherUrl`". This is a little different from some other JSP mechanisms we will come across, for including another page or forwarding the browser to another page.

Exercise: Write a JSP to output the entire line, "Hello! The time is now ..." but use a scriptlet for the complete string, including the HTML tags.

5-Mixing Scriptlets and HTML

We have already seen how to use the "`out`" variable to generate HTML output from within a scriptlet. For more complicated HTML, using the out variable all the time loses some of the advantages of JSP programming. It is simpler to mix scriptlets and HTML.

Suppose you have to generate a table in HTML. This is a common operation, and you may want to generate a table from a SQL table, or from the lines of a file. But to keep our example simple, we will generate a table containing the numbers from 1 to N. Not very useful, but it will show you the technique. Here is the JSP fragment to do it:

```
<TABLE BORDER=2>
<%
    for ( int i = 0; i < n; i++ ) {
        %>
        <TR>
        <TD>Number</TD>
        <TD><%= i+1 %></TD>
        </TR>
        <%
    }
%>
</TABLE>
```

You would have to supply an `int` variable "n" before it will work, and then it will output a simple table with "n" rows. The important things to notice are how the `%>` and `<%` characters appear in the middle of the "`for`" loop, to let you drop back into HTML and then to come back to the scriptlet. The concepts are simple here -- as you can see, you can drop out of the scriptlets, write normal HTML, and get back into the scriptlet. Any control expressions such as a "while" or a "for" loop or an "if" expression will control the HTML also. If the HTML is inside a loop, it will be emitted once for each iteration of the loop.

Another example of mixing scriptlets and HTML is shown below -- here it is assumed that there is a boolean variable named "`hello`" available. If you set it to true, you will see one output, if you set it to false, you will see another output.

```
<%
    if ( hello ) {
        %>
        <P>Hello, world
        <%
    } else {
        %>
        <P>Goodbye, world
        <%
    }
%>
```

It is a little difficult to keep track of all open braces and scriptlet start and ends, but with a little practice and some good formatting discipline, you will acquire competence in doing it.

Exercise: Make the above examples work. Write a JSP to output all the values returned by `System.getProperties` with "
" embedded after each property name and value. Do not output the "
" using the "out" variable.

6-Directives

We have been fully qualifying the `java.util.Date` in the examples in the previous sections. Perhaps you wondered why we don't just import java.util.*;

It is possible to use "`import`" statements in JSPs, but the syntax is a little different from normal Java. Try the following example:

```
<%@ page import="java.util.*" %>
<HTML>
<BODY>
<%
    System.out.println( "Evaluating date now" );
    Date date = new Date();
%>
Hello!  The time is now <%= date %>
</BODY>
</HTML>
```

The first line in the above example is called a "**directive**". A JSP "**directive**" starts with **<%@** characters. This one is a "page directive". The page directive can contain the list of all imported packages. To import more than one item, separate the package names by commas, e.g.

```
<%@ page import="java.util.*,java.text.*" %>
```

There are a number of JSP directives, besides the page directive. Besides the page directives, the other most useful directives are **include** and **taglib**. We will be covering taglib separately. The include directive is used to physically include the contents of another file. The included file can be HTML or JSP or anything else -- the result is as if the original JSP file actually contained the included text. To see this directive in action, create a new JSP

```
<HTML>
<BODY>
Going to include hello.jsp...<BR>
<%@ include file="hello.jsp" %>
</BODY>
</HTML>
```

View this JSP in your browser, and you will see your original `hello.jsp` get included in the new JSP.

Exercise: Modify all your earlier exercises to import the java.util packages.

7- Declarations

The JSP you write turns into a class definition. All the scriptlets you write are placed inside a single method of this class.

You can also add variable and method declarations to this class. You can then use these variables and methods from your scriptlets and expressions. To add a declaration, you must use the **<%!** and **%>** sequences to enclose your declarations, as shown below.

```
<%@ page import="java.util.*" %>
<HTML>
<BODY>
<%!
    Date theDate = new Date();
    Date getDate()
    {
        System.out.println( "In getDate() method" );
        return theDate;
    }
%>
Hello!  The time is now <%= getDate() %>
</BODY>
</HTML>
```

The example has been created a little contrived, to show variable and method declarations. Here we are declaring a Date variable `theDate`, and the method `getDate`. Both of these are available now in our scriptlets and expressions.

But this example no longer works! The date will be the same, no matter how often you reload the page. This is because these are declarations, and will only be evaluated once when the page is loaded! (Just as if you were creating a class and had variable initialization declared in it.)

Exercise: Modify the above example to add another function `computeDate` which re-initializes `theDate`. Add a scriptlet that calls `computeDate` each time.

8-Tags

Another important syntax element of JSP are **tags**. JSP tags do not use **<%**, but just the **<** character. A JSP tag is somewhat like an HTML tag. JSP tags can have a "start tag", a "tag body" and an "end tag". The start and end tag both use the tag name, enclosed in < and > characters. The end starts with a / character after the < character. The tag names have an embedded colon character **:** in them, the part before the colon describes the type of the tag. For instance:

```
<some:tag>
body
</some:tag>
```

If the tag does not require a body, the start and end can be conveniently merged together, as
`<some:tag/>`

Here by closing the start tag with a /> instead of > character, we are ending the tag immediately, and without a body. (This syntax convention is the same as XML.)

Tags can be of two types: **loaded from an external tag library**, or **predefined tags**. **Predefined tags** start with **`jsp:`** characters. For instance, `jsp:include` is a predefined tag that is used to include other pages.

We have already seen the `include` directive. jsp:include is similar. But instead of loading the text of the included file in the original file, it actually calls the included target at run-time (the way a browser would call the included target. In practice, this is actually a simulated request rather than a full round-trip between the browser and the server). Following is an example of `jsp:include` usage

```
<HTML>
<BODY>
Going to include hello.jsp...<BR>
<jsp:include page="hello.jsp"/>
</BODY>
</HTML>
```

Try it and see what you get. Now change the "`jsp:include`" to "`jsp:forward`" and see what is the difference. These two predefined tags are frequently very useful.

Exercise: Write a JSP to do either a `forward` or an `include`, depending upon a boolean variable (hint: The concepts of mixing HTML and scriptlets work with JSP tags also!)

9-Sessions

On a typical Web site, a visitor might visit several pages and perform several interactions. If you are programming the site, it is very helpful to be able to associate some data with each visitor. For this purpose, "**`session`**"s can be used in JSP.

A **session** is an object associated with a visitor. Data can be put in the session and retrieved from it, much like a Hashtable. A different set of data is kept for each visitor to the site.
Here is a set of pages that put a user's name in the session, and display it elsewhere. Try out installing and using these. First we have a form, let us call it `GetName.html.`

```
<HTML>
<BODY>
<FORM METHOD=POST ACTION="SaveName.jsp">
What's your name? <INPUT TYPE=TEXT NAME=username SIZE=20>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

The target of the form is "`SaveName.jsp`", which saves the user's name in the session. Note the variable "**`session`**". This is another variable that is normally made available in JSPs, just like `out` and `request` variables. (In the @page directive, you can indicate that you do not need sessions, in which case the "session" variable will not be made available.)

```
<%
   String name = request.getParameter( "username" );
   session.setAttribute( "theName", name );
%>
<HTML>
<BODY>
<A HREF="NextPage.jsp">Continue</A>
</BODY>
</HTML>
```

The `SaveName.jsp` saves the user's name in the session, and puts a link to another page, `NextPage.jsp`. `NextPage.jsp` shows how to retrieve the saved name.

```
<HTML>
<BODY>
Hello, <%= session.getAttribute( "theName" ) %>
</BODY>
</HTML>
```

If you bring up two different browsers (not different windows of the same browser), or run two browsers from two different machines, you can put one name in one browser and another name in another browser, and both names will be kept track of.

The session is kept around until a timeout period. Then it is assumed the user is no longer visiting the site, and the session is discarded.

Exercise: Add another attribute "age" to the above example.

10-Beans and Forms Processing

Forms are a very common method of interactions in Web sites. JSP makes forms processing particularly easy. The standard way of handling forms in JSP is to define a "bean". This is not a full Java bean. You just need to define a class that has a field corresponding to each field in the form. The class fields must have "setters" that match the names of the form fields. For instance, let us modify our `GetName.html` to also collect email address and age. The new version of `GetName.html` is

```
<HTML>
<BODY>
<FORM METHOD=POST ACTION="SaveName.jsp">
What's your name? <INPUT TYPE=TEXT NAME=username SIZE=20><BR>
What's your e-mail address? <INPUT TYPE=TEXT NAME=email SIZE=20><BR>
What's your age? <INPUT TYPE=TEXT NAME=age SIZE=4>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

To collect this data, we define a Java class with fields "`username`", "`email`" and "`age`" and we provide setter methods "`setUsername`", "`setEmail`" and "`setAge`", as shown. A "setter" method is just a method that starts with "`set`" followed by the name of the field. The first character of the field name is upper-cased. So if the field is "`email`", its "`setter`" method will be "`setEmail`". Getter methods are defined similarly, with "get" instead of "set". Note that the setters (and getters) must be public.

```
public class UserData {
    String username;
    String email;
    int age;

    public void setUsername( String value )
    {
        username = value;
    }

    public void setEmail( String value )
    {
        email = value;
    }

    public void setAge( int value )
    {
        age = value;
    }

    public String getUsername() { return username; }

    public String getEmail() { return email; }

    public int getAge() { return age; }
}
```

The method names must be exactly as shown. Once you have defined the class, compile it and make sure it is available in the Web-server's `classpath`. The server may also define special folders where you can place bean classes, e.g. with Blazix you can place them in the "`classes`" folder.

Now let us change "`SaveName.jsp`" to use a bean to collect the data.

```
<jsp:useBean id="user" class="UserData" scope="session"/>
<jsp:setProperty name="user" property="*"/>
<HTML>
<BODY>
<A HREF="NextPage.jsp">Continue</A>
</BODY>
</HTML>
```

All we need to do now is to add the `jsp:useBean` tag and the `jsp:setProperty` tag.

The **`useBean`** tag will look for an instance of the "`UserData`" in the session. If the instance is already there, it will update the old instance. Otherwise, it will create a new instance of `UserData` (the instance of the `UserData` is called a bean), and put it in the session.

The **`setProperty`** tag will automatically collect the input data, match names against the bean method names, and place the data in the bean.

Let us modify `NextPage.jsp` to retrieve the data from bean.

```
<jsp:useBean id="user" class="UserData" scope="session"/>
<HTML>
<BODY>
You entered<BR>
Name: <%= user.getUsername() %><BR>
Email: <%= user.getEmail() %><BR>
Age: <%= user.getAge() %><BR>
</BODY>
</HTML>
```

Notice that the same useBean tag is repeated. The bean is available as the variable named "`user`" of class "`UserData`". The data entered by the user is all collected in the bean.

We do not actually need the "`SaveName.jsp`", the target of `GetName.html` could have been `NextPage.jsp`, and the data would still be available the same way as long as we added a `jsp:setProperty` tag. But in the next section, we will actually use `SaveName.jsp` as an error handler that automatically forwards the request to `NextPage.jsp`, or asks the user to correct the erroneous data.

Exercise:
1) Write a JSP/HTML set that allows a user to enter the name of a system property, and then displays the value returned by `System.getProperty` for that property name (handle errors appropriately.)
2) Go back to the exercises where you manually modified boolean variables. Instead of a boolean variable, make these come from a HIDDEN form field that can be set to true or false.

11-Tag Libraries

JSP 1.1 introduces a method of extending JSP tags, called "tag libraries". These libraries allow addition of tags similar to `jsp:include` or `jsp:forward`, but with different prefixes other than `jsp:` and with additional features.

To introduce you to tag libraries, in this tutorial we use the Blazix tag library as an example. This tag library comes bundled with the Blazix server, which you can download free for learning and evaluation.

Each tag-library will have its own tag-library specific documentation. In order to use the tag library, you use the "**`taglib`**" directive to specify where your tag library's "description" resides. For the Blazix tag library the (recommended) directive is as follows:

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
```

The "uri" specifies where to find the tag library description. The "prefix" is unique for the tag library. This directive is saying that we will be using the tags in this library by starting them with **`blx:`**

The Blazix tag library provides a `blx:getProperty` tag. This tag can be used to allow the user to edit form data. In our `GetName.jsp` file, we will now add a `jsp:useBean` and place the form inside blx:getProperty. The new `GetName.jsp` is:

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
```

```
<jsp:useBean id="user" class="UserData" scope="session"/>
<HTML>
<BODY>
<blx:getProperty name="user" property="*">
<FORM METHOD=POST ACTION="SaveName.jsp">
What's your name? <INPUT TYPE=TEXT NAME=username SIZE=20><BR>
What's your e-mail address? <INPUT TYPE=TEXT NAME=email SIZE=20><BR>
What's your age? <INPUT TYPE=TEXT NAME=age SIZE=4>
<P><INPUT TYPE=SUBMIT>
</FORM>
</blx:getProperty>
</BODY>
</HTML>
```

Note that the `blx:getProperty` doesn't end with `/>` but is instead terminated by a separate `</blx:getProperty>` line. This puts all the form input fields inside the `blx:getProperty` so they can be appropriately modified by the tag library.

Try putting a link to `GetName.jsp` from the `NextPage.jsp`, and you will see that the bean's data shows up automatically in the input fields. The user can now edit the data.

We still have a couple of problems. The user cannot clear out the name field. Moreover, if the user enters a bad item in the "age" field, something which is not a valid integer, a Java exception occurs.

We will use another tag from the Blazix tag library to take care of this. Blazix offers a `blx:setProperty` tag that can be used to take care of these problems. `blx:setProperty` allows us to define an exception handler method. If an exception occurs, we can collect an error message for the user and continue processing.
Following is a version of `SaveName.jsp` that processes any errors, and either shows the user `GetName.jsp` again to user can enter the data correctly, or automatically forwards to `NextPage.jsp`.

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
<%!
    boolean haveError;
    StringBuffer errors;

    public void errorHandler( String field,
                              String value,
                              Exception ex )
    {
        haveError = true;
        if ( errors == null )
            errors = new StringBuffer();
        else
            errors.append( "<P>" );
        errors.append( "<P>Value for field \"" +
                       field + "\" is invalid." );
        if ( ex instanceof java.lang.NumberFormatException )
            errors.append( " The value must be a number." );
    }
%>
<%
    // Variables must be initialized outside declaration!
    haveError = false;
    errors = null;
```

```
%>
<HTML>
<BODY>
<jsp:useBean id="user" class="UserData" scope="session"/>
<blx:setProperty name="user"
     property="*"
     onError="errorHandler"/>
<%
    if ( haveError ) {
        out.println( errors.toString());
        pageContext.include( "GetName.jsp" );
    } else
        pageContext.forward( "NextPage.jsp" );
%>
</BODY>
</HTML>
```

Note that `haveError` and `errors` must be re-initialized each time, therefore they are being initialized outside of the declaration.

[Also notice the use of `pageContext.include` and `pageContext.forward`. These are like `jsp:include` and `jsp:forward`, but are more convenient to use from within Java blocks. `pageContext` is another pre-defined variable that makes it easy to do certain operations from within Java blocks.]

Here, if an error occurs during the processing of `blx:setProperty`, we display the error and then include the `GetName.jsp` again so user can correct the error. If no errors occur, we automatically forward the user to `NextPage.jsp`.

There is still a problem with the forms, the "age" shows up as zero initially rather than being empty. This can be fixed by adding "`emptyInt=0`" to both the `blx:getProperty` and `blx:setProperty` tags (bean fields should be initialized to 0.) It happens that "0" is not a valid value for age, so we can use "0" to mark empty strings. If "0" were a valid value for age, we could have added "`emptyInt=-1`" (and made sure to initialize the bean fields to -1.)

Another small problem is that the "<HTML>" tag gets doubled if there is an error and we end up including "`GetName.jsp`". A more elegant solution is to remove the out.println, and pass back the error as shown:

```
<%
    if ( haveError ) {
        request.setAttribute( "errors",
                 errors.toString());
        pageContext.forward( "GetName.jsp" );
    } else
        pageContext.forward( "NextPage.jsp" );
%>
```

We can then do a "`request.getAttribute`" in the `GetName.jsp`, and if the returned value is non-null, display the error. This is left as an exercise.

Exercise: Read the documentation on Blazix or another tag library, and use some tags from this library.

12-Form Editing

A tag library such as the one that comes with the Blazix server, may not be available in your environment. How can you allow similar features without using a tag library?

It is a little tedious, but it can be done. Basically, you must edit each HTML tag yourself, and put in a default value. The following examples shows how we modify `GetName.jsp` to provide features similar to `blx:getProperty` but with manual HTML tag editing:

```
<jsp:useBean id="user" class="UserData" scope="session"/>
<HTML>
<BODY>
<FORM METHOD=POST ACTION="SaveName.jsp">
What's your name? <INPUT TYPE=TEXT NAME=username
         SIZE=20 VALUE="<%= user.getUsername() %>"><BR>
What's your e-mail address? <INPUT TYPE=TEXT
         NAME=email SIZE=20
         VALUE="<%= user.getEmail() %>"><BR>
What's your age? <INPUT TYPE=TEXT NAME=age
         SIZE=4 VALUE=<%= user.getAge() %>>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

As you can see, this simply involves adding a "VALUE" field in the INPUT tags, and initializing the field with an expression. To handle exceptions during input processing, a simple approach is to use "`String`" fields in the bean, and do the conversion to the target datatype yourself. This will allow you to handle exceptions.

Exercise: Modify the earlier example to do everything without the Blazix tag library (you can restrict this to only one field.)

13-Log-in pages

Some sites require that all users log-in using a username and password, before being able to visit any page. This can be done using JSP sessions or servlets, and in fact this was a common technique for a while. But starting with a new release of Servlets specifications (2.2) from Sun, this feature is now very simple to implement.

It is no longer necessary to use JSP techniques to provide login/password protection, but it is still a very common requirement of Web-sites, therefore a brief overview is provided here.

To password-protect your site, you just need to design a login page. This page can be as simple or complicated as you need it to be. It must contain a `<FORM>` tag, with the `METHOD` set to `POST` and the `ACTION` set to "`j_security_check`".

```
<FORM METHOD=POST ACTION=j_security_check>
```

The target `j_security_check` is provided by the application server, and does not need to be coded.
The form must contain two `<INPUT>` fields, named `j_username` and `j_password` respectively for the username and password. Typically, the username field will be a `TEXT` input field, and the password field will be a `PASSWORD` input field.
After this, you must tell your application server to password protect your pages using the login page you have provided. The details will vary from server to server, but a good implementation will provide you hooks that you can use, for example, to match usernames and passwords against a database. (E.g., in

Blazix you can supply an implementation of the interface `desisoft.deploy.AuthCheck` to check usernames and passwords against a database or other sources.)

Exercise: Read your application server's documentation and add login/password protection to some of your JSPs.

14-Database Access in JSP

Database access is very common in JSPs. Most database access these days is done using SQL. Therefore, if you do not know SQL, the first step is to learn SQL. Teaching SQL is outside the scope of this tutorial, but there are many excellent references available on the web. (See the further reading page if you need some pointers.)

Once you know how to write SQL queries, all you then need is to be able to execute SQL query strings from Java programs or JSP pages, and to be able to examine and manipulate any returned values.

In Java, SQL access is provided via JDBC (the `java.sql.*` package.) One approach to database access in JSP is simply to use JDBC, by putting JDBC calls in Java scriptlets.

Because of tag libraries, in JSP it is typically a little easier to use SQL. Therefore it is not necessary to do the full JDBC setup. In this page, we will see how to use the Blazix tag library for SQL access. (The sample file is only for Windows computers, users of other systems would need to create test databases on their systems with advice from someone familiar with doing this on their system.)

The first step is to download the provided sample database file jspsqlsample.mdb, and configure an ODBC connection to it named "`jspsql`". If you do not know how to configure ODBC connections, visit the setting up ODBC connections page.

Once you have your ODBC connection configured, add the following lines to your web configuration (`web.ini`) file:

```
dataSource.name: myDataSource
dataSource.myDataSource.odbc:  jspsql
```

This tells the server to use the ODBC connection named "jspsql".

The sample database file contains a table *SingleItem* which contains a single row of data. (If you have *Microsoft Access* available on your machine, you should be able to open and look at the database file.)

The following query will retrieve a single item from this table.

SELECT Goal FROM SingleItem

Write and try out the following JSP.

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
<HTML>
<BODY>
<P>
<blx:sqlConnection jndiName="myDataSource">
<P>The goal is <blx:sqlGet query="SELECT Goal FROM SingleItem"/>
</blx:sqlConnection>
</BODY>
</HTML>
```

Here, the `blx:sqlConnection` tag is specifying the "`myDataSource`" datasource, so the tag library will know that we want to access our `jspsqlsample.mdb` file. The `blx:sqlGet` is retrieving the result of our query.

Often queries will return multiple rows, and will contain multiple items in each row. For such queries, the tag `blx:sqlQuery` can be utilized. Try the following JSP.

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
<HTML>
<BODY>
<P>
<blx:sqlConnection jndiName="myDataSource">

<blx:sqlQuery id="sampleQuery">
SELECT DayNumber,TaskLearned FROM Jsptut
</blx:sqlQuery>

<TABLE>
<TR><TD>Day Number</TD><TD>Task Learned</TD></TR>
<blx:sqlExecuteQuery resultSet="rs" queryRef="sampleQuery">
<TR>
<TD><%= rs.getInt("DayNumber") %></TD>
<TD><%= rs.getString("TaskLearned") %></TD>
</TR>
</blx:sqlExecuteQuery>
</TABLE>
</blx:sqlConnection>
</BODY>
</HTML>
```

The `blx:sqlQuery` tag is being used here to write out the query itself. Then the `blx:sqlExecuteQuery` is being used to retrieve the rows of the query. Everything between `<blx:sqlExecuteQuery>` and `</blx:sqlExecuteQuery>` will be repeatedly executed, once for each row of the query. Therefore there will be many rows in the table, once for each row in the database. Within the body of the `blx:sqlExecuteQuery`, we have access to the Java variable "rs", which is of type `java.sql.resulSet`. We can retrieve the items from the row using either the column number or the name. We could also have used the following to retrieve the items:

```
<TD><%= rs.getInt(1) %></TD>
<TD><%= rs.getString(2) %></TD>
```

To execute queries that do not return any values (such as INSERT, DELETE and UPDATE statements,) use the `blx:executeUpdate` tag.

Exercise: 1) Write a page to execute and display the result of the following query thar returns a single item:

SELECT DayNumber FROM Jsptut WHERE TaskLearned='Scriptlets'

2) The Jsptut table has a third column named Url. Modify the table sample above to display this column as well. Make the URLs come out as hyperlinks.
3) Write a set of JSP pages that lets a user update the day number for any given task, and then displays the updated table. (Hint: You will need a WHERE clause to compare the task name, like in exercise 1 above. You will also need the form processing skills you learned in earlier lessons.)
4) Modify your JSP pages so users can add rows to the Jsptut table.
5) Add a facility to delete rows also.

Setting up an ODBC Connection on a Windows machine

Open up the Control Panel and look for an icon named "ODBC Data Sources" or "ODBC-32" or just "Data Sources".. If you are on a Windows/2000 machine, this icon will be inside "Administrative Services" folder in the Control Panel.

Now select the "System DSN" tab, and click "Add". Select the Microsoft Access driver and click the "`Finish`" button. You will now be taken to the Microsoft Access specific page. Specify the data source name as "jspsql" and click the "Select.." button to specify the `jspsqlsample.mdb` file you have downloaded. Add in a description if you prefer, and click OK.

More About SQL Access in JSP

This JSP script is about as simple as they come. The <jsp:usebean . . . /> instantiates the bean, and the scriptlet directs that bean to do all the work. As you will see, the method runSql( ) handles the database connectivity, builds and executes the SQL, and presents all of the output in a single string variable. In this example, the bean builds the HTML for presentation. It might be better if we moved the HTML generation code to its own JavaBean method or to a JSP Tag Library. For now, however, we'll do it all within the same JavaBean method to make this example as simple as possible. First, lets look at the JSP, and then we'll build the bean.

The JSP

```
<html>
<head><title>JSP The Short Course - Lesson 13 </title></head>
<body>

  <jsp:useBean id="myBean" scope="page"
class="rayexamples.Lesson13" />
  <% out.print ( myBean.runSql() ) ; %>

</body>
</html>
```

Build a Basic Bean

Nothing fancy here, just a basic bean. We won't even have getter and setter exposed methods. We don't need them, but we do have one exposed method called runSql (). That's where all the work is done. (The complete listing of the JSP and JavaBean are available at the end of the lesson.) Let's start writing our JavaBean.

Build a Basic Bean

```
package rayexamples;
import java.sql.* ;
import java.io.Serializable;

public class Lesson13 implements java.io.Serializable
{
  public Lesson13() {}
```

Add Database Connectivity

In order to connect to a database, we need to use Java-compliant connectivity software, and our choices are to use either the JDBC/ODBC bridge, or the more straightforward JDBC. We're going to use JDBC because it's a proven technology, easier to use, and free of the bugs that continue to plague JDBC/ODBC.

Do you have JDBC (Java Database Connectivity) installed on your computer? Probably, because it is loaded along with the JDK. However, in addition to JDBC you need a JDBC driver to connect to your database. Your database should come with a JDBC driver. If it doesn't contact the database manufacturer to learn more about where you can download a JDBC driver. Sun also maintains a list of available JDBC drivers.

Important Note: When you get your JDBC driver it will have to be installed onto your system. Follow the instructions that come with the JDBC driver. In addition, you will have to install the JDBC driver so that your JSP container will have access to the JDBC driver's class files. In the case of Tomcat, this can be accomplished by copying your JDBC driver's JAR files to the Tomcat Lib directory. Once this is done you will have to restart Tomcat.

Let's take a look at the connection process. We can't assume in this tutorial which database management system you will be using (Oracle, DB2, Sybase, MySQL, etc.), so you'll have to follow your vendor's instructions for creating the the tables for this tutorial. All of our examples use the Adaptive Server Anywhere from Sybase, and we will display the table formats as they appear in the Sybase product so you can create your own version locally. This lesson deals with only one table, the Employee table, which has no key references to address.

```
emp_id            integer
manager_id        integer
emp_fname         char(20)
emp_lname         char(20)
dept_id           integer
street            char(40)
city              char(20)
state             char(4)
zip_code          char(9)
```

```
phone             char(10)
status            char(1)
ss_number         char(11)
salary            numeric(20,3)
start_date        date
termination_date  date
birth_date        date
bene_health_ins   char(1)
bene_life_ins     char(1)
bene_day_care     char(1)
sex               char(1)
```

Once you've established your Employee table and populated it with some data, you need to do two things: Tell your Java code where the driver is so it can get loaded into memory and then connect to the database.

Class.forName("com.sybase.jdbc2.jdbc.SybDriver") ; The parameter for this strange-looking string will be provided either by your database administrator or found in your database installation documentation. This single statement is all it takes to load a driver to memory.

To connect to the database using the driver, you need the following two lines.

String localDatabase = "jdbc:sybase:Tds:127.0.0.1:2638" ;

jdbc:	establishes that we are using the JDBC driver.
sybase:Tds:	is the name of the jdbc driver for my database, and it was provided by the Sybase documentation.
127.0.0.1:	is the address of your database server, and is dictated by your database installation. This database (Sybase Adaptive Server) exists on my computer, and 127.0.0.1 is its address.
2638	is the port where your database listens for data requests. In this case, port 2638 is the standard port that is used by Sybase Adaptive Server database. You can find this port number from the documentation of your database or from your database administrator.

sqlca = DriverManager.getConnection(localDatabase, "dba","sql") ; makes the connection. The DriverManager class has many methods available, but the only one you'll need is getConnection(). The first argument in the getConnection() method is the String localDatabase described above.

The last two arguments in the getConnection() method are the user-id and password. The defaults used in the SQL Anywhere database (and most of the other databases) are "dba" and "sql", but these are changed once the database is installed and running. Your installation may take the issue of security a bit more seriously than I have with my single-machine environment, so you'll will need to use your own values here.

To summarize, here's the connectivity code for the JavaBean.

Connectivity Code

```
public String runSql ()
```

```
  {
    String             browserOutput  = "";
    Connection sqlca  = null;
    Statement  sqlStatement  = null;
    ResultSet  myResultSet  = null;

    /* Connect to database, run SQL and produce output */
    try
    {
      /* Connection to the database */
      Class.forName("com.sybase.jdbc2.jdbc.SybDriver");
      String theDatabase = "jdbc:sybase:Tds:127.0.0.1:2638";
      sqlca = DriverManager.getConnection(theDatabase,
"dba","sql");
```

Set Up and Execute the SQL

JDBC requires you place your SQL statement within a statement object. You retrieve the results of executing your SQL statement into a ResultSet.

The executeQuery() method is used for all SQL queries (select statements). For insert, update and delete statements, you will use the executeUpdate() method.

Here's the SQL and Execution code for the JavaBean.

SQL & Execution Code

```
      /* Construct and execute the sql, positioning before the
first row in the result
         set.  Note that the row selection is limited to those
whose emp_id is less than 250.
         This is to keep the result set small for display
purposes. */
       sqlStatement   = sqlca.createStatement();
       myResultSet  =  sqlStatement.executeQuery
        ("select emp_id, " +
                      "emp_lname, " +
                      "city, " +
                      "state," +
                      "zip_code, " +
                      "emp_fname " +
                      "from employee " +
                      "where emp_id < 250  " +
                      "order by emp_lname, emp_fname");
```

Set Up the Output

These three lines establish the HTML to set up a table for our output.

The first line establishes the table, sets the border to zero thickness (no lines show), centers the table on the user's screen, and sets the width to 75% of the width of the user's browser window.

The second line places a caption above the table. This caption is italicized and bold.

The third line adds the column headings from left to right. Each column heading is a <th> column heading </th> group, but note that the first contains align=left. This causes the current and all subsequent column headings to be left-justified. You can change this when you need a column centered or right-justified, but remember to set it back for following columns.

Establish a Table to Display the Dynamic Output

```
/* Construct the heading for the table */
browserOutput =
  "<table border=0 align=center width=75%>" +
  "<caption><i><b>" +
  "Employee Listing</b></i></caption>" ;

/* Construct the column headings for the table */
browserOutput += "<th align=left>Emp_id</th>" +
  "<th>First Name</th>" +
  "<th>Last Name</th>" +
  "<th>City</th>" +
  "<th>State</th>" +
  "<th>Zip</th>" ;
```

Loop, Reading Until You're Finished

This code loops through the result set, getting each column for the current row, making sure it's a String, and putting it into the presentation table. Please note:

- You must use the **next( )** method, because JDBC always positions the row pointer BEFORE the first row in the result set, expecting the programmer to move it through the result set as required.
- Use the HTML tags <tr> to begin a new table row, and </tr> to end the row.
- Use the HTML tags <td> for table data within a cell, and </td> to end each piece of data as you move across the row.

Using the "while" loop with the next( ) method moves you through the result set until you reach the end. It then falls out of the loop and you can close the statement. That's all there is to it.

Create the HTML Output

```
      /* Move to next row and & its contents to the html output
*/
      while(myResultSet.next())
      {
        browserOutput += "&ltTR>&ltTD>" +
          myResultSet.getObject("emp_id").toString() +
"</TD>&ltTD>" +

myResultSet.getObject("emp_fname").toString()+"</TD>&ltTD>" +
          myResultSet.getObject("emp_lname").toString() +
```

```
"</TD>&ltTD>" +
          myResultSet.getObject("city").toString() +
"</TD>&ltTD>" +
          myResultSet.getObject("state").toString() +
"</TD>&ltTD>" +
          myResultSet.getObject("zip_code").toString() +
"</TD></TR>" ;
      }
      sqlStatement.close();
      }
```

Handle Exceptions

JDBC exception handling is really straightforward, and the following code is really just boiler plate that will take care of everything.

Handle Exceptions

```
      catch (SQLException e)
      {
        browserOutput = " Error: SQL error of: " +
e.getMessage();
      }
      catch (Exception e)
      {
        browserOutput = " Error: JDBC Class Creation: " +
e.getMessage();
      }
      finally
      {
      try
      {
        sqlca.close();
      }
      catch(SQLException e)
      {
        browserOutput = " Error: Closing connection: " +
e.getMessage();
      }
    }
```

Return the Output to the Browser

Recall that we are developing one long string of HTML to present to the browser, so it's logical that we need to wrap up the HTML in an orderly manner. That's what we're doing here by adding the </table> tag. After that, the string is complete and we have only to return it to the JSP.

Return the Output

```
    /* Complete the html and return it to the Java Server Page
*/
```

```
    browserOutput += "</table>" ;
    return browserOutput;
  }
}
```

That's it! You've got all the information you need to add dynamically-generated content from a database. Now you can take a look at the Lesson13 JavaBean in its final form.

The Bean

The JavaBean

```
package rayexamples;
import java.sql.* ;
import java.io.Serializable;

public class Lesson13 implements java.io.Serializable
{
  public Lesson13() {}

  public String runSql ()
  {
    String      browserOutput  = "";
    Connection    sqlca  = null;
    Statement    sqlStatement  = null;
    ResultSet    myResultSet  = null;

    /* Connect to database, run SQL and produce output */
    try
    {
      /* Connection to the database */
      Class.forName("com.sybase.jdbc2.jdbc.SybDriver");
      String theDatabase = "jdbc:sybase:Tds:127.0.0.1:2638";
      sqlca = DriverManager.getConnection(theDatabase,
"dba","sql");

      /* Construct and execute the sql, positioning before the
        first row in the result set */
      sqlStatement   = sqlca.createStatement();
      myResultSet  =  sqlStatement.executeQuery
        ("select emp_id, " +
        "emp_lname, " +
        "city, " +
        "state," +
        "zip_code, " +
        "emp_fname " +
        "from employee " +
        "where emp_id < 250  " +
        "order by emp_lname, emp_fname");

      /* Construct the heading for the table */
      browserOutput =
        "&lttable border=0 align=center width=75%>" +
        "&ltcaption>&lti>&ltb>" +
        "Employee Listing</b></i></caption>" ;
```

```
      /* Construct the column headings for the table */
      browserOutput += "&ltth align=left&gtEmp_id</th>" +
        "&ltth&gtFirst Name</th>" +
        "&ltth&gtLast Name</th>" +
        "&ltth&gtCity</th>" +
        "&ltth&gtState</th>" +
        "&ltth&gtZip</th>" ;

      /* Move to next row and & its contents to the html output
*/
      while(myResultSet.next())
      {
        browserOutput += "&ltTR>&ltTD>" +
          myResultSet.getObject("emp_id").toString() +
"</TD>&ltTD>" +

myResultSet.getObject("emp_fname").toString()+"</TD>&ltTD>" +
          myResultSet.getObject("emp_lname").toString() +
"</TD>&ltTD>" +
          myResultSet.getObject("city").toString() +
"</TD>&ltTD>" +
          myResultSet.getObject("state").toString() +
"</TD>&ltTD>" +
          myResultSet.getObject("zip_code").toString() +
"</TD></TR>" ;
      }
      sqlStatement.close();
      }
      catch (SQLException e)
      {
        browserOutput = " Error: SQL error of: " +
e.getMessage();
      }
      catch (Exception e)
      {
        browserOutput = " Error: JDBC Class creation: " +
e.getMessage();
      }
      finally
      {
      try
      {
        sqlca.close();
      }
      catch(SQLException e)
      {
        browserOutput = " Error: Closing connection: " +
e.getMessage();
      }
    }
    /* Complete the html and return it to the Java Server Page
*/
    browserOutput += "</table>" ;
    return browserOutput;
  }
```

```
}
```

The Output

How the Output Looks

Employee Listing

Emp_id	First Name	Last Name	City	State	Zip
191	Jeannette	Bertrand	Acton	MA	01720
160	Robert	Breault	Milton	MA	02186
129	Philip	Chin	Atlanta	GA	30339
105	Matthew	Cobb	Waltham	MA	02154
195	Marc	Dill	Milton	MA	02186
247	Kurt	Driscoll	Waltham	MA	02154
184	Melissa	Espinoza	Stow	MA	01775
207	Jane	Francis	Concord	MA	01742
249	Rodrigo	Guevara	Framingham	MA	01701
148	Julie	Jordan	Winchester	MA	01890
243	Natasha	Shishov	Waltham	MA	02154
102	Fran	Whitney	Needham	MA	02192

15-Sending Email

To be able to send e-mail, you need access to an "SMTP server". If your email address is "`you@yourhost.com`", then there is a good chance your SMTP server is either "`yourhost.com`" or something like "`mail.yourhost.com`" or "`smtp.yourhost.com`". You need to find out exactly what it is. Your email program should have a "Settings" page which shows you the name of your SMTP server (perhaps shown as "mail server" or "outgoing mail server".)

Once you have the SMTP server information, you are ready to send email out from your JSP pages. Following is a small sample that uses the Blazix Tag library to send an email with an attachment.
First of all, let us write an HTML page to start things off.

```
<HTML>
<BODY>
<FORM METHOD=POST ACTION="SendMail.jsp">
Please enter name: <INPUT TYPE=TEXT NAME=username SIZE=20><BR>
Please enter email address: <INPUT TYPE=TEXT NAME=email SIZE=20><BR>
<P><INPUT TYPE=SUBMIT>
</FORM>
</BODY>
</HTML>
```

Now let us write the target JSP, `SendMail.jsp`. Replace "`yoursmtphost.com`" by your SMTP server, and "`you@youremail.com`" by your email address. Before testing it, also create a file "`C:\contents.txt`" which will be sent as an attachment.

```
<%@ taglib prefix="blx" uri="/blx.tld" %>
<HTML>
<BODY>
<%
  // Get username.
  String email = request.getParameter( "email" );
%>
<% if ( email == null || email.equals( "" )) { %>
Please enter an email address.
<% } else { %>
<blx:email host="yoursmtphost.com" from="you@youremail.com">
<blx:emailTo><%= email %></blx:emailTo>
Thank you for registering with us.  You registered the
following name: <%= request.getParameter( "username" ) %>
Your registration was received at <%= new java.util.Date() %>
Attached, please find a contents file.
<blx:emailAttach file="C:\\contents.txt"
      contentType="text/plain" name="contents.txt/>
</blx:email>
<!--  Also write out some HTML -->
Thank you.  A confirmation email has been sent to <%= email %>
<% } %>
</BODY>
</HTML>
```

Exercise:
1) Send an HTML file as an attachment, changing the contentType to "text/html" instead of "text/plain".
2) Send the HTML file as an attachment, without using any other text. In the "blx:email" tag, also set "noText=true".

Annex- JSP Quick Reference Card