UNIT III: Overview of JSP technology and Invoking Java code with JSP scripting elements & The JSP page directives

UNIT III: Overview of JSP technology and Invoking Java code with JSP scripting elements & The JSP page directives

The Need for JSP, Benefits of JSP, Installation of JSP, Basic syntax, Invoking Java code from JSP, Using JSP Expression, Using Scriptlets to make parts of the JSP page conditional, The *Import* attribute, The *contentType*and *pageEncoding* attribute, Generating Excel Spreadsheet, The *session* attribute, The *isELIgnored* attribute, The *errorPage* and *isErrorPage* attribute

JavaServer Pages (JSP) technology enables you to mix regular, static HTML with dynamically generated content. You simply write the regular HTML in the normal manner, using familiar Web-page-building tools. You then enclose the code for the dynamic parts in special tags, most of which start with <% and end with %>.

You can think of servlets as Java code with HTML inside; you can think of JSP as HTML with Java code inside. Now, neither servlets nor JSP pages are restricted to using HTML, but they usually do, and this over-simplified description is a common way to view the technologies.

Even though servlets and JSP pages are equivalent behind the scenes, they are not equally useful in all situations. Separating the static HTML from the dynamic content provides a number of benefits over servlets alone, and the approach used in JavaServer Pages offers several advantages over competing technologies.

THE NEED FOR JSP:

Servlets are indeed useful. They are all tasks related to *programming* or data *processing*. But servlets are not so good at *presentation*. Servlets have the following deficiencies when it comes to generating the output:

- **It is hard to write and maintain the HTML.** Using print statements to generate HTML? Hardly convenient: you have to use parentheses and semicolons, have to insert backslashes in front of embedded double quotes, and have to use string concatenation to put the content together. Besides, it simply does not look like HTML, so it is harder to visualize.

- **You cannot use standard HTML tools.** All those great Web-site development tools you have are of little use when you are writing Java
  code.

The HTML is inaccessible to non-Java developers. If the HTML is embedded within Java code, a Web development expert who does not know the Java programming lan

BENEFITS OF JSP:

JSP pages are translated into servlets. So, fundamentally, any task JSP pages can perform could also be accomplished by servlets. However, this underlying equivalence does not mean that servlets and JSP pages are equally appropriate in all scenarios.
The issue is not the power of the technology, it is the convenience, productivity, and maintainability of one or the other. After all, anything you can do on a particular computer platform in the Java programming language you could also do in assembly language. But it still matters which you choose.

JSP provides the following benefits over servlets alone:

• **It is easier to write and maintain the HTML.** Your static code is ordinary HTML: no extra backslashes, no double quotes, and no lurking Java syntax.

• **You can use standard Web-site development tools.**
For example, we use Macromedia Dreamweaver for most of the JSP pages in the book. Even HTML tools that know nothing about JSP can be used because they simply ignore the JSP tags.

• **You can divide up your development team.** The Java programmers can work on the dynamic code. The Web developers can concentrate on the presentation layer. On large projects, this division is very important. Depending on the size of your team and the complexity of your project, you can enforce a weaker or stronger separation between the static HTML and the dynamic content.

INSTALLATION OF JSP PAGES:

Servlets require you to set your CLASSPATH, use packages to avoid name conflicts, install the class files in servlet-specific locations, and use special-purpose URLs. Not so with JSP pages. JSP pages can be placed in the same directories as normal HTML pages, images, and style sheets; they can also be accessed through URLs of the same form as those for HTML pages, images, and style sheets.
Here are a few examples of default installation locations (i.e., locations that apply when you aren't using custom Web applications) and associated URLs.

Where we list *SomeDirectory*, you can use any directory name you like, except that you are never allowed to use WEB-INF or META-INF as directory names. When using the default Web application, you also have to avoid a directory name that matches the URL prefix of any other Web application.

JSP Directories for Tomcat

(Default Web Application)
• Main Location.
install_dir/webapps/ROOT

Corresponding URL.
http://*host*/*SomeFile*.jsp
• More Specific Location (Arbitrary Subdirectory).
install_dir/webapps/ROOT/*SomeDirectory*
• Corresponding URL.
http://*host*/*SomeDirectory*/*SomeFile*.jsp

JSP Directories for JRun
(Default Web Application)
• Main Location.
install_dir/servers/default/default-ear/default-war
• Corresponding URL.
http://*host*/*SomeFile*.jsp
• More Specific Location (Arbitrary Subdirectory).
install_dir/servers/default/default-ear/default-war/*SomeDirectory*
• Corresponding URL.
http://*host*/*SomeDirectory*/*SomeFile*.jsp

JSP Directories for Resin
(Default Web Application)
• Main Location.
install_dir/doc
• Corresponding URL.
http://*host*/*SomeFile*.jsp
• More Specific Location (Arbitrary Subdirectory).
install_dir/doc/*SomeDirectory*
• Corresponding URL.
http://*host*/*SomeDirectory*/*SomeFile*.jsp

Note that, although JSP pages *themselves* need no special installation directories,any Java classes called *from* JSP pages still need to go in the standard locations used by servlet classes (e.g., .../WEB-INF/classes/*directoryMatchingPackageName*;

BASIC SYNTAX:

Here is a quick summary of the various JSP constructs

HTML Text
• **Description:**
HTML content to be passed unchanged to the client
• **Example:**
<H1>Blah</H1>

HTML Comments
• **Description:**
HTML comment that is sent to the client but not displayed by the browser
• **Example:**
<!-- Blah -->

Template Text
• **Description:**
Text sent unchanged to the client. HTML text and HTML comments are just special cases of this.
• **Example:**
Anything other than the syntax of the following subsections

JSP Comment
• **Description:**
Developer comment that is not sent to the client
• **Example:**
<%-- Blah --%>

JSP Expression
• **Description:**
Expression that is evaluated and sent to the client each time the page is requested
• **Example:**
<%= *Java Value* %>

JSP Scriptlet
• **Description:**
Statement or statements that are executed each time the page is requested
• **Example:**
<% *Java Statement* %>

JSP Declaration
• **Description:**
Field or method that becomes part of class definition when page is translated into a servlet
• **Examples:**
<%! *Field Definition* %>
<%! *Method Definition* %>

JSP Directive
• **Description:**
High-level information about the structure of the servlet code (page), code that is included at page-translation time (include), or custom tag libraries used (taglib)
Example:
<%@ *directive att*="*val*" %>

JSP Action
• **Description:**
Action that takes place when the page is requested
• **Example:**
<jsp:*blah*>...</jsp:*blah*>

JSP Expression Language Element
• **Description:**
Shorthand JSP expression
• **Example:**
${ *EL Expression* }

Custom Tag (Custom Action)
• **Description:**
Invocation of custom tag
• **Example:**
<*prefix*:*name*>
Body
</*prefix*:*name*>

Escaped Template Text
• **Description:**
Text that would otherwise be interpreted specially. Slash is removed

and remaining text is sent to the client
• **Examples:**
<\%
%\>
// Program for simple JSP

```
<HTML>
<HEAD>
<TITLE>Order Confirmation</TITLE>
</HEAD>
<BODY>
<H2>Order Confirmation</H2>
Thanks for ordering <I><%= request.getParameter("title") %></I>!
</BODY></HTML>
```

INVOKING JAVA CODE FROM JSP

There are a number of different ways to generate dynamic content from JSP. Each of these approaches has a legitimate place; the size and complexity of the project is the most important factor in deciding which approach is appropriate.
However, be aware that people err on the side of placing too much code directly in the page much more often than they err on the opposite end of the spectrum.

Although putting small amounts of Java code directly in JSP pages works fine
for simple applications, using long and complicated blocks of Java code in JSP pages yields a result that is hard to maintain, hard to debug, hard to reuse, and hard to divide among different members of the development team.

Types of JSP Scripting Elements

JSP scripting elements let you insert Java code into the servlet that will be generated from the JSP page.
There are three forms:

1. **Expressions** of the form <%= *Java Expression* %>, which are evaluated
and inserted into the servlet's output.

2. **Scriptlets** of the form <% *Java Code* %>, which are inserted into the
servlet's _jspService method (called by service).

3. **Declarations** of the form <%! *Field/Method Declaration* %>, which are inserted into the body of the servlet class, outside any existing methods.

Limiting the Amount of Java Code in JSP Pages

You have 25 lines of Java code that you need to invoke. You have two options: (1) put all 25 lines directly in the JSP page, or (2) put the 25 lines of code in a separate Java class, put the Java class in WEB INF/classes/*directoryMatchingPackageName*, and use one or two lines of JSP-based Java code to invoke it. Which is better? The second.
The second. The second! And all the more so if you have 50, 100, 500, or 1000 lines of code. Here's why:

• **Development.** You generally write regular classes in a Java-oriented environment (e.g., an IDE like JBuilder or Eclipse or a code editor like UltraEdit or emacs). You generally write JSP in an HTMLoriented environment like Dreamweaver.
The Java-oriented environment is typically better at balancing parentheses, providing tooltips, checking the syntax, colorizing the code, and so forth.

• **Compilation.** To compile a regular Java class, you press the Build button in your IDE or invoke javac. To compile a JSP page, you have to drop it in the right directory, start the server, open a browser, and enter the appropriate URL.

• **Debugging.** We know this never happens to you, but when *we* write Java classes or JSP pages, we occasionally make syntax errors.

If there is a syntax error in a regular class definition, the compiler tells you right away and it also tells you what line of code contains the error.

If there is a syntax error in a JSP page, the server typically tells you what line *of the servlet* (i.e., the servlet into which the JSP page was translated) contains the error. For tracing output at runtime, with regular classes you can use simple System.out.println statements if your IDE provides nothing better. In JSP, you can sometimes use print statements, but where those print statements are displayed varies from server to server.

• **Division of labor.** Many large development teams are composed of some people who are experts in the Java language and others who are experts in HTML but know little or no Java.

The more Java code that is directly in the page, the harder it is for the Web developers (the HTML experts) to manipulate it.

• **Testing.** Suppose you want to make a JSP page that outputs random integers between designated 1 and some bound (inclusive). You use Math.random, multiply by the range, cast the result to an int, and add 1. Hmm, that sounds right. But are you sure? If you do this directly in the JSP page, you have to invoke the page over and over to see if you get all the numbers in the designated range but no numbers outside the range. After hitting the Reload button a few dozen times, you will get tired of testing.

• **Reuse.** You put some code in a JSP page. Later, you discover that you need to do the same thing in a different JSP page. What do you do? Cut and paste? Boo! Repeating code in this manner is a cardinal sin because if (when!) you change your approach, you have to change many different pieces of code. Solving the code reuse problem is what object-oriented programming is all about. Don't forget all your good OOP principles just because you are using JSP to simplify the generation of HTML.

USING JSP EXPRESSIONS:

A JSP expression is used to insert values directly into the output. It has the following form:

<%= *Java Expression* %>

The expression is evaluated, converted to a string, and inserted in the page. This evaluation is performed at runtime (when the page is requested) and thus has full access to information about the request. For example, the following shows the date/time that the page was requested.
Current time: <%= new java.util.Date() %>

Predefined Variables
To simplify these expressions, you can use a number of predefined variables (or "implicit objects"). There is nothing magic about these variables; the system simply tells you what names it will use for the local variables in _jspService (the method that replaces doGet in servlets that result from JSP pages).

• **request**, the HttpServletRequest.
• **response**, the HttpServletResponse.
• **session**, the HttpSession associated with the request
• **out**, the Writer (a buffered version of type JspWriter) used to
send output to the client.
• **application**, the ServletContext. This is a data structure

shared by all servlets and JSP pages in the Web application and is good for storing shared data. We discuss it further in the chapters on beans (Chapter 14) and MVC (Chapter 15).
Here is an example:
Your hostname: <%= **request**.getRemoteHost() %>

JSP/Servlet Correspondence

Now, we just stated that a JSP expression is evaluated and inserted into the page output. Although this is true, it is sometimes helpful to understand what is going on behind the scenes.
It is actually quite simple: JSP expressions basically become print (or write) statements in the servlet that results from the JSP page. Whereas regular HTML becomes print statements with double quotes around the text, JSP expressions become print statements with no double quotes. Instead of being placed in the doGet method, these print statements are placed in a new method called _jspService that is called by service for both GET and POST requests, such as reading the HTML from a static byte array are quite common. Also, we oversimplified the definition of the out variable; out in a JSP page is a JspWriter, so you have to modify the slightly simpler PrintWriter that directly results from a call to getWriter. So, don't expect the code your server generates to look *exactly* like this.

Example of JSP Expression:

```
<HTML>
<HEAD>
<TITLE>JSP Expressions</TITLE>
<META NAME="keywords"
CONTENT="JSP,expressions,JavaServer Pages,servlets">
<META NAME="description"
CONTENT="A quick example of JSP expressions.">
<LINK REL=STYLESHEET
HREF="JSP-Styles.css"
TYPE="text/css">
</HEAD>
<BODY>
<H2>JSP Expressions</H2>
<UL>
<LI>Current time: <%= new java.util.Date() %>
<LI>Server: <%= application.getServerInfo() %>
<LI>Session ID: <%= session.getId() %>
<LI>The <CODE>testParam</CODE> form parameter:
<%= request.getParameter("testParam") %>
</UL>
```

```
</BODY></HTML>
```

Sample JSP Expression: Random Number

```
<H1>A Random Number</H1>
<%= Math.random() %>
```

Representative Resulting Servlet Code: Random Number

```
public void _jspService(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{ response.setContentType("text/html");
HttpSession session = request.getSession();
JspWriter out = response.getWriter();
out.println("<H1>A Random Number</H1>");
out.println(Math.random());
...
}
```

XML Syntax for Expressions

XML authors can use the following alternative syntax for JSP expressions:

<jsp:expression>*Java Expression*</jsp:expression>

In JSP 1.2 and later, servers are required to support this syntax as long as authors don't mix the XML version and the standard JSP version (<%= ... %>) in the same page.
This means that, to use the XML version, you must use XML syntax in the *entire* page.
In JSP 1.2 (but not 2.0), this requirement means that you have to enclose the entire page in a jsp:root element.
As a result, most developers stick with the classic syntax except when they are either generating XML documents (e.g., xhtml or SOAP) or when the JSP page is itself the output of some XML process (e.g., XSLT).

Note that XML elements, unlike HTML ones, are case sensitive. So, be sure to use jsp:expression in lower case.

USING SCRIPTLETS TO MAKE PARTS OF THE JSP PAGE CONDITIONAL:

Another use of scriptlets is to conditionally output HTML or other content that is *not* within any JSP tag.
Key to this approach are the facts that (a) code inside a scriptlet gets inserted into the resultant servlet's _jspService method (called by service) *exactly* as written and (b) that any static HTML (template text) before or after a scriptlet gets converted to print statements.
This behavior means that scriptlets need not contain complete Java statements and that code blocks left open can affect the static HTML or JSP outside the scriptlets.

Example:

```
<HTML>
<HEAD>
<TITLE>Wish for the Day</TITLE>
<LINK REL=STYLESHEET
HREF="JSP-Styles.css"
TYPE="text/css">
</HEAD>
<BODY>
<% if (Math.random() < 0.5) { %>
<H1>Have a <I>nice</I> day!</H1>
<% } else { %>
<H1>Have a <I>lousy</I> day!</H1>
<% } %>
</BODY></HTML>
```

The key is that the first two scriptlets do not contain complete statements, but rather partial statements that have dangling braces. This serves to capture the subsequent HTML within the if or else clauses.

Overuse of this approach can lead to JSP code that is hard to understand and maintain. Avoid using it to conditionalize large sections of HTML, and try to keep your JSP pages as focused on presentation (HTML output) tasks as possible. Nevertheless, there are some situations in which the alternative approaches are also unappealing.

The import Attribute

A JSP directive affects the overall structure of the servlet that results from the JSP page. The following templates show the two possible forms for directives. Single quotes can be substituted for the double quotes around the attribute values, but the quotation marks cannot be omitted altogether. To obtain quotation marks within an attribute value, precede them with a backslash, using \'for 'and \"for "

.

<%@ directive attribute="value" %>

<%@ directive attribute1="value1" attribute2=" value2" attributeN="valueN"%>

In JSP, there are three main types of directives: page,include, and taglib

.

The page directive lets you control the structure of the servlet by importing classes, customizing the servlet superclass, setting the content type, and the like. A page directive can be placed anywhere within the document; its use is the topic of this chapter. The second directive, include, lets you insert a file into the JSP page at the time the JSP file is translated into a servlet. An include directive should be placed in the document at the point at which you want the file to be inserted; The third directive, taglib, defines custom markup tags;

The import attribute of the page directive lets you specify the packages that should be imported by the servlet into which the JSP page gets translated

The import attribute serves the same function as, and behaves like, the Java import statement. The value for the import option is the name of the package you want to import.

Use of the importattribute takes one of the following two forms.

<%@ page import="package.class" %>
<%@ page import="package.class1,...,package.classN" %>

To import java.sql.*, use the following page directive:

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

To import multiple packages you can specify them separated by comma as follows:

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

By default, a container automatically imports java.lang.*, javax.servlet.*, javax.servlet.jsp.*, and javax.servlet.http.*.

Note that, although the JSP pages go in the normal HTML directories of the server, the classes you write that are used by JSP pages must be placed in the special Java-code directories (e.g., .../WEB-INF/classes/directoryMatchingPackageName)

The *contentType* and *pageEncoding* attribute

The contentType attribute sets the character encoding for the JSP page and for the generated response page. The default content type is text/html, which is the standard content type for HTML pages.

Use of the contentType attribute takes one of the following two forms.

<%@ page contentType="MIME-Type" %>
<%@ page contentType="MIME-Type; charset=Character-Set" %>

If you want to write out XML from your JSP, use the following page directive:

<%@ page contentType="text/xml" %>

The following statement directs the browser to render the generated page as HTML:

<%@ page contentType="text/html" %>

The following directive sets the content type as a Microsoft Word document:

<%@ page contentType="application/msword" %>

The following directive sets the content type as a Microsoft Excel Sheet:

<%@ page contentType="application/vnd.ms-excel" %>

You can also specify the character encoding for the response. For example, if you wanted to specify that the resulting page that is returned to the browser uses ISO Latin 1, you would use the following page directive:

<%@ page contentType="text/html:charset=ISO-8859-1" %>

If you want to change both the content type and the character set, you can do the following.
<%@ page contentType="someMimeType; charset=someCharacterSet" %>

However, if you only want to change the character set, it is simpler to use the pageEncoding attribute. For example, Japanese JSP pages might use the following.

<%@ page pageEncoding="Shift_JIS" %>

Generating Excel Spreadsheet

Following example shows a JSP page that uses the contentType attribute and tab-sepa-rated data to generate Excel output.
Note that the page directive and comment are at the bottom so that the carriage returns at the ends of the lines don't show up in the Excel document.

JSP does not ignore white space—JSP usually generates HTML in which most white space is ignored by the browser, but JSP itself maintains the white space and sends it to the client.

Excel.jsp

```
First     Last Email Address
Marty Hall hall@coreservlets.com
Larry Brown brown@coreservlets.com
Steve Balmer balmer@ibm.com
Scott McNealy mcnealy@microsoft.com
<%@ page contentType="application/vnd.ms-excel" %>
<%-- There are tabs, not spaces, between columns. --%>
```

The *session* attribute

The session attribute indicates whether or not the JSP page uses HTTP sessions. A value of true means that the JSP page has access to a builtin **session** object and a value of false means that the JSP page cannot access the builtin session object.

The session attribute controls whether the page participates in HTTP sessions. Use of this attribute takes one of the following two forms.

<%@ page session="true" %> <%-- Default --%>
<%@ page session="false" %>

Following directive allows the JSP page to use any of the builtin object session methods such as session.getCreationTime() or session.getLastAccessTime():

<%@ page session="true" %>

The *isELIgnored* attribute

The isELIgnored option gives you the ability to disable the evaluation of Expression Language (EL) expressions which has been introduced in JSP 2.0

The default value of the attribute is true, meaning that expressions, ${...}, are evaluated as dictated by the JSP specification. If the attribute is set to false, then expressions are not evaluated but rather treated as static text.

Use of this attribute takes one of the following two forms.
<%@ page isELIgnored="false" %>
<%@ page isELIgnored="true" %>

JSP 2.0 introduced a concise expression language for accessing request parameters, cookies, HTTP headers, bean properties and Collection elements from within a JSP page.

Following directive set an expressions not to be evaluated:

<%@ page isELIgnored="false" %>

The *errorPage* and *isErrorPage* attribute

The errorPage attribute tells the JSP engine which page to display if there is an error while the current page runs. The value of the errorPage attribute is a relative URL.

It is used as follows:
<%@ page errorPage="Relative URL" %

The exception thrown will automatically be available to the designated error page by means of the exception variable.

The following directive displays MyErrorPage.jsp when all uncaught exceptions are thrown:

<%@ page errorPage="MyErrorPage.jsp" %>

The isErrorPage attribute indicates that the current JSP can be used as the error page for another JSP.

The value of isErrorPage is either true or false. The default value of the isErrorPage attribute is false.

The isErrorPage attribute indicates whether or not the current page can act as the error page for another JSP page.

Use of isErrorPage takes one of the following two forms:

<%@ page isErrorPage="true" %>
<%@ page isErrorPage="false" %> <%-- Default --%>

For example, the handleError.jsp sets the isErrorPage option to true because it is supposed to handle errors:

<%@ page isErrorPage="true" %>