BEA WebLogic Server™

and BEA WebLogic Express™

Programming WebLogic JSP

BEA WebLogic Server Version 6.1
Document Date: December 19, 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic JSP

Part Number	Document Date	Software Version
N/A	December 19, 2001	BEA WebLogic Server Version 6.1

Contents

About This Document

1. JSP Overview

What Is JSP? 1-1
WebLogic Implementation of JSP 1-2
WebLogic Server 6.1 with J2EE 1.2 and J2EE 1.3 Functionality 1-3
WebLogic Server 6.1 with J2EE 1.2 Plus Additional J2EE 1.3 Features. 1-3
WebLogic Server 6.1 with J2EE 1.2 Certification 1-3
How JSP Requests Are Handled 1-3
Additional Information 1-4

2. Administering WebLogic JSP

Overview of WebLogic JSP Administration 2-1
Setting JSP Operating Parameters 2-2

3. WebLogic JSP Reference

JSP Tags 3-2
Reserved Words for Implicit Objects 3-3
Directives for WebLogic JSP 3-5
Using the page Directive to Set Character Encoding 3-6
Using the taglib Directive 3-6
Declarations 3-6
Scriptlets 3-7
Expressions 3-8
Example of a JSP with HTML and Embedded Java 3-9
Actions 3-10

Using JavaBeans in JSP 3-10
Instantiating the JavaBean Object 3-11
Doing Setup Work at JavaBean Instantiation 3-11
Using the JavaBean Object 3-12
Defining the Scope of a JavaBean Object 3-12
Forwarding Requests 3-13
Including Requests 3-13
Securing User-Supplied Data in JSPs 3-14
Using a WebLogic Server Utility Method 3-15
Using Sessions with JSP 3-16
Deploying Applets from JSP 3-17
Using the WebLogic JSP Compiler 3-19
Running JSPC on Windows Systems 3-19
JSP Compiler Syntax 3-19
JSP Compiler Options 3-20
Precompiling JSPs 3-23
System Properties and JSPs 3-23

4. Using Custom WebLogic JSP Tags (cache, process, repeat)

Overview of WebLogic Custom JSP Tags 4-1
Using the WebLogic Custom Tags in a Web Application 4-2
Cache Tag 4-2
Refreshing a Cache 4-3
Flushing a Cache 4-3
Process Tag 4-7
Repeat Tag 4-8

5. Using WebLogic JSP Form Validation Tags

Overview of WebLogic JSP Form Validation Tags 5-1
Validation Tag Attribute Reference 5-2
<wl:summary> 5-2
<wl:form> 5-4
<wl:validator> 5-4
Using WebLogic JSP Form Validation Tags in a JSP 5-6
Creating HTML Forms Using the <wl:form> Tag 5-8

Defining a Single Form 5-8
Defining Multiple Forms 5-8
Re-Displaying the Values in a Field When Validation Returns Errors 5-9
Re-Displaying a Value Using the <input> Tag 5-9
Re-Displaying a Value Using the Apache Jakarta <input:text> Tag 5-9
Using a Custom Validator Class 5-10
Extending the CustomizableAdapter Class 5-11
Sample User-Written Validator Class 5-11
Sample JSP with Validator Tags 5-12

6. Using the WebLogic EJB to JSP Integration Tool

Overview of the WebLogic EJB-to-JSP Integration Tool 6-2
Basic Operation 6-3
Interface Source Files 6-3
Build Options Panel 6-4
Troubleshooting 6-5
Using EJB Tags on a JSP Page 6-6
EJB Home Methods 6-6
Stateful Session and Entity Beans 6-7
Default Attributes 6-8

7. Troubleshooting

Debugging Information in the Browser 7-1
Error 404—Not Found 7-2
Error 500—Internal Server Error 7-2
Error 503—Service Unavailable 7-2
Errors Using the <jsp:plugin> tag 7-2
Symptoms in the Log File 7-3
Page Compilation Failed Errors 7-3

About This Document

This document describes how to program e-commerce applications by using JavaServer Pages (JSP) and WebLogic Server.

The document is organized as follows:

- Chapter 1, “JSP Overview,”provides an introduction and reference for the basic syntax of JSP and information about how to use JSP with WebLogic Server.
- Chapter 2, “Administering WebLogic JSP,” provides a brief overview of administration and configuration tasks for WebLogic JSP.
- Chapter 3, “WebLogic JSP Reference,” provides a reference on writing JSPs.
- Chapter 4, “Using Custom WebLogic JSP Tags (cache, process, repeat),” discusses the use of three custom JSP tags provided with the WebLogic Server distribution: the `cache` tag, the `repeat` tag, and the `process` tag.
- Chapter 7, “Troubleshooting,” describes several techniques for debugging your JSP files.

Audience

This document is written for application developers who want to build e-commerce applications using JSP and the Java 2 Platform, Enterprise Edition (J2EE) from Sun Microsystems. It is assumed that readers know Web technologies, object-oriented programming techniques, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File→Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Related Information

- JSP 1.1 Specification from Sun Microsystems, available at `http://java.sun.com/products/jsp/download.html`.
- Programming WebLogic JSP Tag Extensions at `http://e-docs.bea.com/wls/docs61/taglib/index.html`.
- Deploying and Configuring Web Applications at `http://e-docs.bea.com/wls/docs61/adminguide/config_web_app.html`.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

Convention	Usage

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`
`...`	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information

Convention	Usage
. . .	Indicates the omission of items from a code example or from a syntax line.

1 JSP Overview

This document is an introduction and reference for the basic syntax of JavaServer Pages (JSP). It provides information about how to use JSP with WebLogic Server. It is not intended as a comprehensive guide to programming with JSP.

The following sections provide an overview of JSP:

- What Is JSP?
- WebLogic Implementation of JSP
- How JSP Requests Are Handled
- Additional Information

What Is JSP?

JavaServer Pages (JSP) is a Sun Microsystems specification for combining Java with HTML to provide dynamic content for Web pages. When you create dynamic content, JSPs are more convenient to write than HTTP servlets because they allow you to embed Java code directly into your HTML pages, in contrast with HTTP servlets, in which you embed HTML inside Java code. JSP is part of the Java 2 Enterprise Edition (J2EE).

JSP enables you to separate the dynamic content of a Web page from its presentation. It caters to two different types of developers: HTML developers, who are responsible for the graphical design of the page, and Java developers, who handle the development of software to create the dynamic content.

Because JSP is part of the J2EE standard, you can deploy JSPs on a variety of platforms, including WebLogic Server. In addition, third-party vendors and application developers can provide JavaBean components and define custom JSP tags that can be referenced from a JSP page to provide dynamic content.

WebLogic Implementation of JSP

BEA WebLogic JSP supports the JSP 1.1 specification (see `http://java.sun.com/products/jsp/download.html`) from Sun Microsystems. JSP 1.1 includes support for defining custom JSP tag extensions. (See Programming JSP Extensions at http://e-docs.bea.com/wls/docs61/taglib/index.html.)

WebLogic Server also supports the Servlet 2.2 specification (`http://java.sun.com/products/servlet/download.html#specs`) from Sun Microsystems, and the proposed final draft of the Servlet 2.3 specification. For more information, see Servlet 2.3 at `http://e-docs.bea.com/wls/docs61/notes/new.html#servlet-webapp`.

Note: WebLogic Server version 6.1 supports the JSP 1.2 specification with the following exceptions:

The jsp:id mechanism has not been implemented

The following feature has not been implemented:

A JAR containing a packaged tag libraries can be dropped into the WEB-INF/lib directory to make its classes available at request time

The following DTD Elements are not supported:

- The <listener> element in the taglib.tld is not registered with the webapp
- The <example> element in the taglib.tld is not honored.

We still use the older signature of the TaglibraryValidator.validate() method which returns a String

The Servlet 2.3 and JSP 1.2 specifications are part of the J2EE 1.3 specification. To use these features, please see "WebLogic Server 6.1 with J2EE 1.2 and J2EE 1.3 Functionality" on page 3.

WebLogic Server 6.1 with J2EE 1.2 and J2EE 1.3 Functionality

BEA WebLogic Server 6.1 is the first e-commerce transaction platform to implement advanced J2EE 1.3 features. To comply with the rules governing J2EE, BEA Systems provides two separate downloads: one with J2EE 1.3 features enabled, and one that is limited to J2EE 1.2 features only. Both downloads offer the same container and differ only in the APIs that are available.

WebLogic Server 6.1 with J2EE 1.2 Plus Additional J2EE 1.3 Features

With this download, WebLogic Server defaults to running with J2EE 1.3 features enabled. These features include EJB 2.0, JSP 1.2, Servlet 2.3, and J2EE Connector Architecture 1.0. When you run WebLogic Server 6.1 with J2EE 1.3 features enabled, J2EE 1.2 applications are still fully supported. The J2EE 1.3 feature implementations use non-final versions of the appropriate API specifications. Therefore, application code developed for BEA WebLogic Server 6.1 that uses the new features of J2EE 1.3 may be incompatible with the J2EE 1.3 platform supported in future releases of BEA WebLogic Server.

WebLogic Server 6.1 with J2EE 1.2 Certification

With this download, WebLogic Server defaults to running with J2EE 1.3 features disabled and is fully compliant with the J2EE 1.2 specification and regulations.

How JSP Requests Are Handled

WebLogic Server handles JSP requests in the following sequence:

1. A browser requests a page with a `.jsp` file extension from WebLogic Server.
2. WebLogic Server reads the request.

3. Using the JSP compiler, WebLogic Server converts the JSP into a servlet class that implements the `javax.servlet.jsp.JspPage` interface. The JSP file is compiled only when the page is first requested, or when the JSP file has been changed. Otherwise, the previously compiled JSP servlet class is re-used, making subsequent responses much quicker.
4. The generated `JspPage` servlet class is invoked to handle the browser request.

It is also possible to invoke the JSP compiler directly without making a request from a browser. For details, see "Using the WebLogic JSP Compiler" on page 3-19. Because the JSP compiler creates a Java servlet as its first step, you can look at the Java files it produces, or even register the generated `JspPage` servlet class as an HTTP servlet (See `http://e-docs.bea.com/wls/docs61/servlet/index.html`).

Additional Information

- JavaServer Pages Tutorial from Sun Microsystems at `http://java.sun.com/products/jsp/docs.html`
- JSP product overview from Sun Microsystems at `http://www.java.sun.com/products/jsp/index.html`
- JSP 1.1 Specification from Sun Microsystems at `http://java.sun.com/products/jsp/download.htm`
- Programming JSP Extensions at `http://e-docs.bea.com/wls/docs61/taglib/index.html`
- Programming WebLogic HTTP Servlets at `http://e-docs.bea.com/wls/docs61/servlet/index.html`
- Assembling and Configuring Web Applications at `http://e-docs.bea.com/wls/docs61/webapp/index.html`

2 Administering WebLogic JSP

The following sections provide an overview of administration and configuration tasks required to deploy WebLogic JavaServer Pages (JSP):

- Overview of WebLogic JSP Administration
- Setting JSP Operating Parameters

For a complete discussion of JSP administration and configuration see Configuring JSP at `http://e-docs.bea.com/wls/docs61/webapp/components.html#configuring-jsp`.

Overview of WebLogic JSP Administration

In keeping with the Java 2 Enterprise Edition standard, JSPs are deployed as part of a *Web Application*. A Web Application is a grouping of application components, such as HTTP servlets, JavaServer Pages (JSP), static HTML pages, images, and other resources.

In a Web Application, the components are organized using a standard directory structure. You can deploy your application using this directory structure or you can archive the files into a single file called a Web Application Archive (`.war`) and deploy the `.war` file. You define information about the resources and operating parameters of

a Web Application using two *deployment descriptors*, which are included in the files of the Web Application. For more information, see Assembling and Configuring Web Applications at `http://e-docs.bea.com/wls/docs61/webapp/index.html`.

The first deployment descriptor, `web.xml`, is defined in the Servlet 2.2 specification from Sun Microsystems. It provides a standardized format that describes the Web Application. The second deployment descriptor, `weblogic.xml`, is a WebLogic-specific deployment descriptor that maps resources defined in the `web.xml` file to resources available in WebLogic Server, defines JSP parameters, and defines HTTP session parameters. For more information, see "Writing Web Application Deployment Descriptors" at `http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html`.

JSPs do not require specific mappings as do HTTP servlets. To deploy JSPs in a Web Application, simply place them in the root directory (or in a sub-directory of the root) of the Web Application. No additional registrations are required. You can deploy both servlets and JSPs in the same Web Application.

Setting JSP Operating Parameters

Parameters that govern the behavior of JSPs are defined in `weblogic.xml`, the WebLogic-specific deployment descriptor of your Web Application. For more information about editing this file, see "Assembling and Configuring Web Applications at `http://e-docs.bea.com/wls/docs61/webapp/index.html`

A complete description of JSP properties in the WebLogic-specific deployment descriptor, including their default values is provided in the jsp-descriptor section, available at `http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#jsp-descriptor`.

Parameters set in `weblogic.xml` include:

- `compileCommand`
- `compileFlags`
- `compilerclass`
- `encoding`

- `keepgenerated`
- `packagePrefix`
- `pageCheckSeconds`
- `verbose`
- `workingDir`

CHAPTER

3 WebLogic JSP Reference

The following sections provide reference information for writing JavaServer Pages (JSPs):

- JSP Tags
- Reserved Words for Implicit Objects
- Directives for WebLogic JSP
- Scriptlets
- Expressions
- Example of a JSP with HTML and Embedded Java
- Actions
- Securing User-Supplied Data in JSPs
- Using Sessions with JSP
- Deploying Applets from JSP
- Using the WebLogic JSP Compiler

JSP Tags

The following table describes the basic tags that you can use in a JSP page. Each shorthand tag has an XML equivalent.

Table 3-1 Basic Tags for JSP Pages

JSP Tag	Syntax	Description

`Scriptlet`	`<% java_code %>`
	. . . or use the XML equivalent:
	`<jsp:scriptlet>`
	`  java_code`
	`</jsp:scriptlet>`
		Embeds Java source code scriptlet in your HTML page. The Java code is executed and its output is inserted in sequence with the rest of the HTML in the page. For details, see "Scriptlets" on page 3-7.

`Directive`	`<%@ dir-type dir-attr %>`
	. . . or use the XML equivalent:
	`<jsp:directive.dir_type dir_attr />`
		Directives contain messages to the application server.
		A directive can also contain name/value pair attributes in the form `attr="value"`, which provides additional instructions to the application server. See "Directives for WebLogic JSP" on page 3-5.

`Declarations`	`<%! declaration %>`
	. . . or use XML equivalent...
	`<jsp:declaration>`
	`  declaration;`
	`</jsp:declaration>`
		Declares a variable or method that can be referenced by other declarations, scriptlets, or expressions in the page. See "Declarations" on page 3-6.

`Expression`	`<%= expression %>`
	. . . or use XML equivalent...
	`<jsp:expression>`
	`expression`
	`</expression>`
		Defines a Java expression that is evaluated at page request time, converted to a `String`, and sent inline to the output stream of the JSP response. See "Expressions" on page 3-8.

Table 3-1 Basic Tags for JSP Pages

JSP Tag	Syntax	Description
Actions	<jsp:useBean ... > JSP body is included if the bean is instantiated here </jsp:useBean> <jsp:setProperty ... > <jsp:getProperty ... > <jsp:include ... > <jsp:forward ... > <jsp:plugin ... >	Provide access to advanced features of JSP, and only use XML syntax. These actions are supported as defined in the JSP 1.1 specification. See "Actions" on page 3-10.

Reserved Words for Implicit Objects

JSP reserves words for implicit objects in scriptlets and expressions. These implicit objects represent Java objects that provide useful methods and information for your JSP page. WebLogic JSP implements all implicit objects defined in the JSP 1.1 specification. The JSP API is described in the Javadocs available from the Sun Microsystems JSP Home Page at `http://www.java.sun.com/products/jsp/index.html`.

Note: Use these implicit objects only within `Scriptlets` or `Expressions`. Using these keywords from a method defined in a declaration causes a translation-time compilation error because such usage causes your page to reference an undefined variable.

`request`

`request` represents the `HttpServletRequest` object. It contains information about the request from the browser and has several useful methods for getting cookie, header, and session data.

`response`

`response` represents the `HttpServletResponse` object and several useful methods for setting the response sent back to the browser from your JSP page. Examples of these responses include cookies and other header information.

Warning: You cannot use the `response.getWriter()` method from within a JSP page; if you do, a run-time exception is thrown. Use the `out` keyword to send the JSP response back to the browser from within your scriptlet code whenever possible. The WebLogic Server implementation of `javax.servlet.jsp.JspWriter` uses `javax.servlet.ServletOutputStream`, which implies that you *can* use `response.getServletOutputStream()`. Keep in mind, however, that this implementation is specific to WebLogic Server. To keep your code maintainable and portable, use the `out` keyword.

`out`

`out` is an instance of `javax.jsp.JspWriter` that has several methods you can use to send output back to the browser.

If you are using a method that requires an output stream, then `JspWriter` does not work. You can work around this limitation by supplying a buffered stream and then writing this stream to `out`. For example, the following code shows how to write an exception stack trace to `out`:

```
ByteArrayOutputStream ostr = new ByteArrayOutputStream();
exception.printStackTrace(new PrintWriter(ostr));
out.print(ostr);
```

`pageContext`

`pageContext` represents a `javax.servlet.jsp.PageContext` object. It is a convenience API for accessing various scoped namespaces and servlet-related objects, and provides wrapper methods for common servlet-related functionality.

`session`

`session` represents a `javax.servlet.http.HttpSession` object for the request. The session directive is set to true by default, so the `session` is valid by default. The JSP 1.1 specification states that if the session directive is set to `false`, then using the `session` keyword results in a fatal translation time error. For more information about using sessions with servlets, see *Programming WebLogic HTTP Servlets* at `http://e-docs.bea.com/wls/docs61/servlet/index.html`.

`application`

`application` represents a `javax.servlet.ServletContext` object. Use it to find information about the servlet engine and the servlet environment.

When forwarding or including requests, you can access the servlet `requestDispatcher` using the `ServletContext`, or you can use the JSP

`forward` directive for forwarding requests to other servlets, and the JSP `include` directive for including output from other servlets.

`config`

`config` represents a `javax.servlet.ServletConfig` object and provides access to the servlet instance initialization parameters.

`page`

`page` represents the servlet instance generated from this JSP page. It is synonymous with the Java keyword `this` when used in your scriptlet code.

To use `page`, you must cast it to the class type of the servlet that implements the JSP page, because it is defined as an instance of `java.lang.Object`. By default, the servlet class is named after the JSP filename. For convenience, we recommend that you use the Java keyword `this` to reference the servlet instance and get access to initialization parameters, instead of using `page`.

For more information on the underlying HTTP servlet framework, see the related developers guide, Programming WebLogic HTTP Servlets at http://e-docs.bea.com/wls/docs61/servlet/index.html.

Directives for WebLogic JSP

Use directives to instruct WebLogic JSP to perform certain functions or interpret the JSP page in a particular way. You can insert a directive anywhere in a JSP page. The position is generally irrelevant (except for the `include` directive), and you can use multiple directive tags. A directive consists of a directive type and one or more attributes of that type.

You can use either of two types of syntax: shorthand or XML:

- Shorthand:

```
<%@ dir_type dir_attr %>
```

- XML:

```
<jsp:directive.dir_type dir_attr />
```

Replace `dir_type` with the directive type, and `dir_attr` with a list of one or more directive attributes for that directive type.

There are three types of directives `page`, `taglib`, or `include`.

Using the page Directive to Set Character Encoding

To specify a character encoding set, use the following directive at the top of the page:

```
<%@ page contentType="text/html; charset=custom-encoding" %>
```

Replace the `custom-encoding` with a standard HTTP-style character set name (see `http://www.isi.edu/in-notes/iana/assignments/character-sets`).

The character set you specify with a `contentType` directive specifies the character set used in the JSP as well as any JSP *included* in that JSP.

You can specify a default character encoding by specifying it in the WebLogic-specific deployment descriptor for your Web Application. For more information, see the jsp-descriptor section at `http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#jsp-descriptor`.

Using the taglib Directive

Use a `taglib` directive to declare that your JSP page uses custom JSP tag extensions that are defined in a tag library. For details about writing and using custom JSP tags, see "Programming WebLogic JSP Extensions" at http://e-docs.bea.com/wls/docs61/taglib/index.html.

Declarations

Use declarations to define variables and methods at the class-scope level of the generated JSP servlet. Declarations made between JSP tags are accessible from other declarations and scriptlets in your JSP page. For example:

```
<%!
  int i=0;
```

```
  String foo= "Hello";
  private void bar() {
    // ...java code here...
  }
%>
```

Remember that class-scope objects are shared between multiple threads being executed in the same instance of a servlet. To guard against sharing violations, synchronize class scope objects. If you are not confident writing thread-safe code, you can declare your servlet as not-thread-safe by including the following directive:

```
<%@ page isThreadSafe="false" %>
```

By default, this attribute is set to `true`. When set to `false`, the generated servlet implements the `javax.servlet.SingleThreadModel` interface, which prevents multiple threads from running in the same servlet instance. Setting `isThreadSafe` to `false` consumes additional memory and can cause performance to degrade.

Scriptlets

JSP scriptlets make up the Java body of your JSP servlet's HTTP response. To include a scriptlet in your JSP page, use the shorthand or XML scriptlet tags shown here:

Shorthand:

```
<%
  // Your Java code goes here
%>
```

XML:

```
<jsp:scriptlet>
  // Your Java code goes here
</jsp:scriptlet>
```

Note the following features of scriptlets:

- You can have multiple blocks of scriptlet Java code mixed with plain HTML.

- You can switch between HTML and Java code anywhere, even within Java constructs and blocks. In "Example of a JSP with HTML and Embedded Java" on page 3-9 the example declares a Java loop, switches to HTML, and then switches back to Java to close the loop. The HTML within the loop is generated as output multiple times as the loop iterates.
- You can use the predefined variable `out` to print HTML text directly to the servlet output stream from your Java code. Call the `print()` method to add a string to the HTTP page response.

  Any time you print data that a user has previously supplied, BEA recommends that you remove any HTML special characters that a user might have entered. If you do not remove these characters, your Web site could be exploited by cross-site scripting. See "Securing User-Supplied Data in JSPs" on page 3-14.
- The Java tag is an *inline* tag; it does not force a new paragraph.

Expressions

To include an expression in your JSP file, use the following tag:

```
<%= expr %>
```

Replace *expr* with a Java expression. When the expression is evaluated, its `string` representation is placed inline in the HTML response page. It is shorthand for

```
<% out.print( expr ); %>
```

This technique enables you to make your HTML more readable in the JSP page. Note the use of the expression tag in the example in the next section.

Expressions are often used to return data that a user has previously supplied. Any time you print user-supplied data, BEA recommends that you remove any HTML special characters that a user might have entered. If you do not remove these characters, your Web site could be exploited by cross-site scripting. See "Securing User-Supplied Data in JSPs" on page 3-14.

Example of a JSP with HTML and Embedded Java

The following example shows a JSP with HTML and embedded Java:

```
<html>
  <head><title>Hello World Test</title></head>

<body bgcolor=#ffffff>
<center>
<h1> <font color=#DB1260> Hello World Test </font></h1>
<font color=navy>

<%

    out.print("Java-generated Hello World");
%>

</font>
<p> This is not Java!
<p><i>Middle stuff on page</i>
<p>
<font color=navy>

<%
     for (int i = 1; i<=3; i++) {
%>
        <h2>This is HTML in a Java loop! <%= i %> </h2>
<%
     }
%>

</font>
</center>
</body>
</html>
```

After the code shown here is compiled, the resulting page is displayed in a browser as follows:

Actions

You use JSP actions to modify, use, or create objects that are reperesented by JavaBeans. Actions use XML syntax exclusively.

Using JavaBeans in JSP

The `<jsp:useBean>` action tag allows you to instantiate Java objects that comply with the JavaBean specification, and to refer to them from your JSP pages.

To comply with the JavaBean specification, objects need:

- A public constructor that takes no arguments

- A `setVariable()` method for each `variable` field
- A `getVariable()` method for each `variable` field

Instantiating the JavaBean Object

The `<jsp:useBean>` tag attempts to retrieve an existing named Java object from a specific scope and, if the existing object is not found, may attempt to instantiate a new object and associate it with the name given by the `id` attribute. The object is stored in a location given by the `scope` attribute, which determines the availability of the object. For example, the following tag attempts to retrieve a Java object of type `examples.jsp.ShoppingCart` from the HTTP session under the name `cart`.

```
<jsp:useBean id="cart"
    class="examples.jsp.ShoppingCart" scope="session"/>
```

If such an object does not currently exist, the JSP attempts to create a new object, and stores it in the HTTP session under the name `cart`. The class should be available in the `CLASSPATH` used to start WebLogic Server, or in the `WEB-INF/classes` directory of the Web Application containing the JSP.

It is good practice to use an `errorPage` directive with the `<jsp:useBean>` tag because there are run-time exceptions that must be caught. If you do not use an `errorPage` directive, the class referenced in the JavaBean cannot be created, an `InstantiationException` is thrown, and an error message is returned to the browser.

You can use the `type` attribute to cast the JavaBean type to another object or interface, provided that it is a legal type cast operation within Java. If you use the attribute without the `class` attribute, your JavaBean object must already exist in the scope specified. If it is not legal, an `InstantiationException` is thrown.

Doing Setup Work at JavaBean Instantiation

The `<jsp:useBean>` tag syntax has another format that allows you to define a body of JSP code that is executed when the object is instantiated. The body is not executed if the named JavaBean already exists in the specified scope. This format allows you to set up certain properties when the object is first created. For example:

```
<jsp:useBean id="cart" class="examples.jsp.ShoppingCart"
 scope=session>
    Creating the shopping cart now...
    <jsp:setProperty name="cart"
```

```
     property="cartName" value="music">
</jsp:useBean>
```

Note: If you use the `type` attribute without the `class` attribute, a JavaBean object is never instantiated, and you should not attempt to use the tag format to include a body. Instead, use the single tag format. In this case, the JavaBean must exist in the specified scope, or an `InstantiationException` is thrown. Use an `errorPage` directive to catch the potential exception.

Using the JavaBean Object

After you instantiate the JavaBean object, you can refer to it by its `id` name in the JSP file as a Java object. You can use it within scriptlet tags and expression evaluator tags, and you can invoke its `setXxx()` or `getXxx()` methods using the `<jsp:setProperty>` and `<jsp:getProperty>` tags, respectively.

Defining the Scope of a JavaBean Object

Use the `scope` attribute to specify the availability and life-span of the JavaBean object. The scope can be one of the following:

`page`
: This is the default scope for a JavaBean, which stores the object in the `javax.servlet.jsp.PageContext` of the current page. It is available only from the current invocation of this JSP page. It is not available to included JSP pages, and it is discarded upon completion of this page request.

`request`
: When the `request` scope is used, the object is stored in the current `ServletRequest`, and it is available to other included JSP pages that are passed the same request object. The object is discarded when the current request is completed.

`session`
: Use the `session` scope to store the JavaBean object in the HTTP session so that it can be tracked across several HTTP pages. The reference to the JavaBean is stored in the page's `HttpSession` object. Your JSP pages must be able to participate in a session to use this scope. That is, you must not have the `page` directive `session` set to `false`.

`application`

At the `application`-scope level, your JavaBean object is stored in the Web Application. Use of this scope implies that the object is available to any other servlet or JSP page running in the same Web Application in which the object is stored.

For more information about using JavaBeans, see the JSP 1.1 specification at `http://www.java.sun.com/products/jsp/index.html`.

Forwarding Requests

If you are using any type of authentication, a forwarded request made with the `<jsp:forward>` tag, by default, does not require the user to be re-authenticated. You can change this behavior to require authentication of a forwarded request by adding the `<check-auth-on-forward/>` element to the `<container-descriptor>` element of the WebLogic-specific deployment descriptor, `weblogic.xml`. For example:

```
<container-descriptor>
   <check-auth-on-forward/>
</container-descriptor>
```

For information on editing the WebLogic-specific deployment descriptor, see Writing the WebLogic-Specific Deployment Descriptor at `http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html#weblogic-xml`.

Including Requests

You can use the `<jsp:include>` tag to include another resource in a JSP. This tag takes two attributes:

page

Use the page attribute to specify the included resource. For example:

```
<jsp:include page="somePage.jsp"/>
```

flush

Setting this boolean attribute to `true` buffers the page output and then flushes the buffer before including the resource.

Setting `flush="false"` can be useful when the `<jsp:include>` tag is located within another tag on the JSP page and you want the included resource to be processed by the tag.

Securing User-Supplied Data in JSPs

Expressions and scriptlets enable a JSP to receive data from a user and return the user supplied data. For example, the sample JSP in Listing 3-1 prompts a user to enter a string, assigns the string to a parameter named `userInput`, and then uses the `<%= request.getParameter("userInput")%>` expression to return the data to the browser.

Listing 3-1 Using Expressions to Return User-Supplied Content

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <h1>My Sample JSP</h1>
      <form method="GET" action="mysample.jsp">
          Enter string here:
          <input type="text" name="userInput" size=50>
          <input type=submit value="Submit">
       </form>
    <br>
    <hr>
    <br>
    Output from last command:
    <%= request.getParameter("userInput")%>
  </body>
</html>
```

This ability to return user-supplied data can present a security vulnerability called **cross-site scripting**, which can be exploited to steal a user's security authorization. For a detailed description of cross-site scripting, refer to "Understanding Malicious Content Mitigation for Web Developers" (a CERT security advisory) at http://www.cert.org/tech_tips/malicious_code_mitigation.html.

To remove the security vulnerability, before you return data that a user has supplied, scan the data for any of the HTML special characters in Table 3-2. If you find any special characters, replace them with their HTML entity or character reference. Replacing the characters prevents the browser from executing the user-supplied data as HTML.

Table 3-2 HTML Special Characters that Must Be Replaced

Replace this special character:	With this entity/character reference:
<	<
>	>
(	&40;
)	&41;
#	&35;
&	&38;

Using a WebLogic Server Utility Method

WebLogic Server provides the `weblogic.servlet.security.Utils.encodeXSS()` method to replace the special characters in user-supplied data. To use this method, provide the user-supplied data as input. For example,

```
<%= weblogic.servlet.security.Utils.encodeXSS(
request.getParameter("userInput"))%>
```

To secure an entire application, you must use the `encodeXSS()` method **each time** you return user-supplied data. While the example in Listing 3-1 is an obvious location in which to use the `encodeXSS()` method, Table 3-3 describes other locations to consider.

Table 3-3 Code that Returns User-Supplied Data

Page Type	User-Supplied Data	Example
Error page	Erroneous input string, invalid URL, username	An error page that says "*username* is not permitted access."
Status page	Username, summary of input from previous pages	A summary page that asks a user to confirm input from previous pages.
Database display	Data presented from a database	A page that displays a list of database entries that have been previously entered by a user.

Using Sessions with JSP

Sessions in WebLogic JSP perform according to the JSP 1.1 specification. The following suggestions pertain to using sessions:

- Store small objects in sessions. For example, a session should not be used to store an EJB, but an EJB primary key instead. Store large amounts of data in a database. The session should hold only a simple string reference to the data.
- When you use sessions with dynamic reloading of servlets or JSP, the objects stored in the servlet session must be serializable. Serialization is required because the servlet is reloaded in a new class loader, which results in an incompatibility between any classes loaded previously (from the old version of the servlet) and any classes loaded in the new class loader (for the new version of the servlet classes). This incompatibility causes the servlet to return `ClassCastException` errors.
- If session data *must* be of a user-defined type, the data class should be serializable. Furthermore, the session should store the serialized representation of the data object. Serialization should be compatible across versions of the data class.

- If you need to log out an authenticated user, see the following section in *Programming WebLogic HTTP Servlets:* Logging Out and Ending a Session at `http://e-docs.bea.com/wls/docs61/servlet/progtasks.html#sessionend`.

Deploying Applets from JSP

Using the JSP provides a convenient way to include the Java Plug-in in a Web page, by generating HTML that contains the appropriate client browser tag. The Java Plug-in allows you to use a Java Runtime Environment (JRE) supplied by Sun Microsystems instead of the JVM implemented by the client Web browser. This feature avoids incompatibility problems between your applets and specific types of Web browsers. The Java Plug-in is available from Sun at `http://java.sun.com/products/plugin/`.

Because the syntax used by Internet Explorer and Netscape is different, the servlet code generated from the `<jsp:plugin>` action dynamically senses the type of browser client and sends the appropriate `<OBJECT>` or `<EMBED>` tags in the HTML page.

The `<jsp:plugin>` tag uses many attributes similar to those of the `<APPLET>` tag, and some other attributes that allow you to configure the version of the Java Plug-in to be used. If the applet communicates with the server, the JVM running your applet code must be compatible with the JVM running WebLogic Server.

In the following example, the plug-in action is used to deploy an applet:

```
<jsp:plugin type="applet" code="examples.applets.PhoneBook1"
 codebase="/classes/" height="800" width="500"
 jreversion="1.1"
 nspluginurl=
 "http://java.sun.com/products/plugin/1.1.3/plugin-install.html"
 iepluginurl=
"http://java.sun.com/products/plugin/1.1.3/
  jinstall-113-win32.cab#Version=1,1,3,0" >

<jsp:params>
  <param name="weblogic_url" value="t3://localhost:7001">
  <param name="poolname" value="demoPool">
</jsp:params>
```

```
<jsp:fallback>
  <font color=#FF0000>Sorry, cannot run java applet!!</font>
</jsp:fallback>


</jsp:plugin>
```

The sample JSP syntax shown here instructs the browser to download the Java Plug-in version 1.3.1 (if it has not been downloaded previously), and run the applet identified by the `code` attribute from the location specified by `codebase`.

The `jreversion` attribute identifies the spec version of the Java Plug-in that the applet requires to operate. The Web browser attempts to use this version of the Java Plug-in. If the plug-in is not already installed on the browser, the `nspluginurl` and `iepluginurl` attributes specify URLs where the Java Plug-in can be downloaded from the Sun Web site. Once the plug-in is installed on the Web browser, it is not downloaded again.

Because WebLogic Server uses the Java 1.3.x VM, you must specify the Java Plug-in version 1.3.x in the `<jsp:plugin>` tag. To specify the 1.3 JVM in the previous example code, replace the corresponding attribute values with the following:

```
jreversion="1.3"
nspluginurl=
"http://java.sun.com/products/plugin/1.3/plugin-install.html"
iepluginurl=
"http://java.sun.com/products/plugin/1.3/jinstall-131-win32.cab"
```

The other attributes of the plug-in action correspond with those of the `<APPLET>` tag. You specify applet parameters within a pair of `<params>` tags, nested within the `<jsp:plugin>` and `</jsp:plugin>` tags.

The `<jsp:fallback>` tags allow you to substitute HTML for browsers that are not supported by the `<jsp:plugin>` action. The HTML nested between the `<fallback>` and `</jsp:fallback>` tags is sent instead of the plug-in syntax.

Using the WebLogic JSP Compiler

Because the JSP Servlet automatically calls the WebLogic JSP compiler to process your JSP pages, you generally do not need to use the compiler directly. However, in some situations, such as when you are debugging, accessing the compiler directly is useful. This section is a reference for the compiler.

The WebLogic JSP compiler parses your JSP file into a `.java` file, and then compiles the generated `.java` file into a Java class, using a standard Java compiler.

Running JSPC on Windows Systems

When you run the JSP compiler on Windows systems, output files names are always created with lower case names. To prevent this behavior, and preserve the case used in class names, set the system property, `weblogic.jsp.windows.caseSensitive` to `true`. You can set the property at the command line when compiling a JSP using this following command:

```
java -Dweblogic.jsp.windows.caseSensitive=true weblogic.jspc *.jsp
```

or include this command in your WebLogic Server startup scripts:

```
-Dweblogic.jsp.windows.caseSensitive=true
```

JSP Compiler Syntax

The JSP compiler works in much the same way that other WebLogic compilers work (including the RMI and EJB compilers). To start the JSP compiler, enter the following command.

```
$ java weblogic.jspc -options fileName
```

Replace *`fileName`* with the name of the JSP file that you want to compile. You can specify any *`options`* before or after the target *`fileName`*. The following example uses the `-d` option to compile `myFile.jsp` into the destination directory, `weblogic/classes`:

```
$ java weblogic.jspc -d /weblogic/classes myFile.jsp
```

Note: If you are precompiling JSPs that are part of a Web Application and that reference resources in the Web Application (such as a JSP tag library), you must use the `-webapp` flag to specify the location of the Web Application. The `-webapp` flag is described in the following listing of JSP compiler options.

JSP Compiler Options

You can use any combination of the following options:

`-classpath`
: Add a list (separated by semi-colons on Windows NT/2000 platforms or colons on UNIX platforms) of directories that make up the desired `CLASSPATH`. Include directories containing any classes required by the JSP. For example (to be entered on one line):

```
$ java weblogic.jspc
   -classpath java/classes.zip;/weblogic/classes.zip
   myFile.JSP
```

`-charsetMap`
: Specifies mapping of IANA or unofficial charset names used in JSP `contentType` directives to java charset names. For example:
`-charsetMap x-sjis=Shift_JIS,x-big5=Big5`

  The most common mappings are built into the JSP compiler. Use this option only if a desired charset mapping is not recognized.

`-commentary`
: Causes the JSP compiler to include comments from the JSP in the generated HTML page. If this option is omitted, comments do not appear in the generated HTML page.

`-compileAll`
: Recursively compiles all JSPs in the current directory, or in the directory specified with the `-webapp` flag. (See the listing for `-webapp` in this list of options.). JSPs in subdirectories are also compiled.

`-compileFlags`
: Passes one or more command-line flags to the compiler. Enclose multiple flags in quotes, separated by a space. For example:
`java weblogic.jspc -compileFlags "-g -v" myFile.jsp`

`-compiler`
: Specifies the Java compiler to be used to compile the class file from the generated Java source code. The default compiler used is `javac`. The Java compiler program should be in your `PATH` unless you specify the absolute path to the compiler explicitly.

`-compilerclass`
: Runs a Java compiler as a Java class and not as a native executable.

`-d <dir>`
: Specifies the destination of the compiled output (that is, the class file). Use this option as a shortcut for placing the compiled classes in a directory that is already in your `CLASSPATH`.

`-depend`
: If a previously generated class file for a JSP has a more recent date stamp than the JSP source file, the JSP is not recompiled.

`-debug`
: Compile with debugging on.

`-deprecation`
: Warn about the use of deprecated methods in the generated Java source file when compiling the source file into a class file.

`-docroot directory`
: See `-webapp`.

`-encoding default|named character encoding`
: Valid arguments include (a) `default` which specifies using the default character encoding of your JDK, (b) a named character encoding, such as `8859_1`. If the `-encoding` flag is not specified, an array of bytes is used.

`-g`
: Instructs the Java compiler to include debugging information in the class file.

`-help`
: Displays a list of all the available flags for the JSP compiler.

`-J`
: Takes a list of options that are passed to your compiler.

`-k`
: When compiling multiple JSPs with a single command, the compiler continues compiling even if one or more of the JSPs failed to compile.

`-keepgenerated`
: Keeps the Java source code files that are created as an intermediary step in the compilation process. Normally these files are deleted after compilation.

`-noTryBlocks`
: If a JSP file has numerous or deeply nested custom JSP tags and you receive a `java.lang.VerifyError` exception when compiling, use this flag to allow the JSPs to compile correctly.

`-nowarn`
: Turns off warning messages from the Java compiler.

`-O`
: Compiles the generated Java source file with optimization turned on. This option overrides the `-g` flag.

`-package` *`packageName`*
: Sets the package name that is prepended to the package name of the generated Java HTTP servlet. Defaults to `jsp_servlet`.

`-superclass` *`classname`*
: Sets the classname of the superclass extended by the generated servlet. The named superclass must be a derivative of `HttpServlet` or `GenericServlet`.

`-verbose`
: Passes the `verbose` flag to the Java compiler specified with the `compiler` flag. See the compiler documentation for more information. The default is off.

`-verboseJavac`
: Prints messages generated by the designated JSP compiler.

`-version`
: Prints the version of the JSP compiler.

`-webapp` *`directory`*
: Name of a directory containing a Web Application in exploded directory format. If your JSP contains references to resources in a Web Application such as a JSP tag library or other Java classes, the JSP compiler will look for those resources in this directory. If you omit this flag when compiling a JSP that requires resources from a Web Application, the compilation will fail.

Precompiling JSPs

You can configure WebLogic Server to precompile your JSPs when a Web Application is deployed or re-deployed or when WebLogic Server starts up by setting the `precompile` parameter to true in the `<jsp-descriptor>` element of the `weblogic.xml` deployment descriptor:

For an exploded webapp, precompilation only occurs on the administration server. For an archived webapp, precompilation will occur on the administration server and the managed server once during the first deployment.

For more information on the `web.xml` deployment descriptor, see Assembling and Configuring Web Applications at `http://e-docs.bea.com/wls/docs61/webapp/index.html`.

Windows NT command length limitations can be overcome using the new `compilerclass` option for WebLogic JSPs. It can be configured in the weblogic.xml file.

The in memory `compilerclass` option uses the compiler class used by Sun to internally compile Java files. This does not require creating a new process and thus is more efficient than compiling each Java file separately using a new process.

The compilerclass can be used by adding the following to weblogic.xml:

```
<jsp-descriptor>
 <jsp-param>
   <param-name>compilerclass</jsp-param>
   <param-value>com.sun.tools.javac.Main</param-value>
 </jsp-param>
</jsp-descriptor>
```

System Properties and JSPs

weblogic.jspc

,weblogic.jsp.windows.caseSensitive is NOT a JSPC option, it is a system property.

You can either call java -Dweblogic.jsp.windows.caseSensitive=true weblogic.jspc *.jsp or put -Dweblogic.jsp.windows.caseSensitive=true in the start server script.

4 Using Custom WebLogic JSP Tags (cache, process, repeat)

The following sections describe the use of three custom JSP tags—`cache`, `repeat`, and `process`—provided with the WebLogic Server distribution:

- Overview of WebLogic Custom JSP Tags
- Using the WebLogic Custom Tags in a Web Application
- Cache Tag
- Process Tag
- Repeat Tag

Overview of WebLogic Custom JSP Tags

BEA provides three specialized JSP tags that you can use in your JSP pages: `cache`, `repeat`, and `process`. These tags are packaged in a tag library jar file called `weblogic-tags.jar`. This jar file contains classes for the tags and a tag library descriptor (TLD). To use these tags, you copy this jar file to the Web Application that contains your JSPs and reference the tag library in your JSP.

Using the WebLogic Custom Tags in a Web Application

Using the WebLogic custom tags requires that you include them within a Web Application. For more information on Web Applications, see Assembling and Configuring Web Applications at `http://e-docs.bea.com/wls/docs61/webapp/index.html`.

To use these tags in your JSP:

1. Copy the `weblogic-tags.jar` file from the `ext` directory of your WebLogic Server installation to the `WEB-INF/lib` directory of the Web application containing the JSPs that will use the WebLogic Custom Tags.
2. Reference this tag library descriptor in the `<taglib>` element of the Web Application deployment descriptor, `web.xml`. For example:

   ```
   <taglib>
     <taglib-uri>weblogic-tags.tld</taglib-uri>
     <taglib-location>
        /WEB-INF/lib/weblogic-tags.jar
     </taglib-location>
   </taglib>
   ```

   For more information, see Writing Web Application Deployment Descriptors at `http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html`.
3. Reference the tag library in your JSP with the `taglib` directive. For example:

   ```
   <%@ taglib uri="weblogic-tags.tld" prefix="wl" %>
   ```

Cache Tag

The cache tag enables caching the work that is done within the body of the tag. It supports both output (transform) data and input (calculated) data. Output caching refers to the content generated by the code within the tag. Input caching refers to the values to which variables are set by the code within the tag. Output caching is useful

when the final form of the content is the important thing to cache. Input caching is important when the view of the data can vary independently of the data calculated within the tag.

If one client is already recalculating the contents of a cache and another client requests the same content it does not wait for the completion of the recalculation, instead it shows whatever information is already in the cache. This is to make sure that the web site does not come to a halt for all your users because a cache is being recalculated. Additionally, the async attribute means that no one, not even the user that initiates the cache recalculation waits.

Caches are stored using soft references to prevent the caching system from using too much system memory.

Refreshing a Cache

You can force the refresh of a cache by setting the `_cache_refresh` object to `true` in the scope that you want affected. For example, to refresh a cache at session scope, specify the following:

```
<% request.setAttribute("_cache_refresh", "true"); %>
```

If you want all caches to be refreshed, set the cache to the `application` scope. If you want all the caches for a user to be refreshed, set it in the `session` scope. If you want all the caches in the current request to be refreshed, set the `_cache_refresh` object either as a parameter or in the request.

The `<wl:cache>` tag specifies content that must be updated each time it is displayed. The statements between the `<wl:cache>` and `</wl:cache>` tags are only executed if the cache has expired or if any of the values of the key attributes (see the Cache Tag Attributes table) have changed.

Flushing a Cache

Flushing a cache forces the cached values to be erased; the next time the cache is accessed, the values are recalculated. To flush a cache, set its `flush` attribute to `true`. The cache must be named using the `name` attribute. If the cache has the `size` attribute

set, all values are flushed. If the cache sets the `key` attribute but not the `size` attribute, you can flush a specific cache by specifying its `key` along with any other attributes required to uniquely identify the cache (such as `scope` or `vars`).

For example:

1. Define the cache.

```
<wl:cache name="dbtable" key="parameter.tablename"
scope="application">
// read the table and output it to the page
</wl:cache>
```

2. Update the cached table data.

3. Flush the cache using the `flush` attribute in an empty tag (an empty tag ends with `/` and does not use a closing tag). For example

```
<wl:cache name="dbtable" key="parameter.tablename"
scope="application" flush="true"/>
```

Table 4-1 Cache Tag Attributes

Attribute	Required	Default Value	Description

`timeout`	no	`-1`	Cache timeout property. The amount of time, in seconds, after which the statements within the cache tag are refreshed. This is not proactive; the value is refreshed only if it is requested. If you prefer to use a unit of time other than seconds, you can specify an alternate unit by postfixing the value with desired unit:
			`ms = milliseconds`
			`s = seconds (default)`
			`m = minutes`
			`h = hours`
			`d = days`

`scope`	no	`application`	Specifies the scope in which the data is cached. Valid scopes include: `page, request, session, application`. Most caches will be either `session` or `application` scope.

Table 4-1 Cache Tag Attributes

Attribute	Required	Default Value	Description
`key`	no	`--`	Specifies additional values to be used when evaluating whether to cache the values contained within the tags. The list of keys is comma-separated. The value of this attribute is the name of the variable whose value you wish to use as a key into the cache. You can additionally specify a scope by prepending the name of the scope to the name. For example: `parameter.key | page.key | request.key | application.key | session.key` It defaults to searching through the scopes in the order shown in the preceding list. Each named key is available in the cache tag as a scripting variable. A list of keys is comma-separated.
`name`	no	`--`	A unique name for the cache that allows caches to be shared across multiple JSP pages. This same buffer is used to store the data for all pages using the named cache. This attribute is useful for textually included pages that need cache sharing. If this attribute is not set, a unique name is chosen for the cache. We recommended that you avoid manually calculating the name of the tag; the `key` functionality can be used equivalently in all cases. The name is calculated as `weblogic.jsp.tags.CacheTag.` plus the URI plus a generated number representing the tag in the page you are caching. If different URIs reach the same JSP page, the caches are not shared in the default case. Use named caches in this case.
`size`	no	`-1` (unlimited)	For caches that use keys, the number of entries allowed. The default is an unlimited cache of keys. With a limited number of keys the tag uses a *least-used system* to order the cache. Changing the value of the size attribute of a cache that has already been used does not change the size of that cache.

Table 4-1 Cache Tag Attributes

Attribute	Required	Default Value	Description
vars	no	--	In addition to caching the transformed output of the cache, you can also cache calculated values within the block. These variables are specified exactly the same way as the cache keys. This type of caching is called *Input caching*.
flush	no	none	When set to true, the cache is flushed. This attribute must be set in an empty tag (ends with /).

The following examples show how you can use the <wl:cache> tag.

Listing 4-1 Examples of Using the cache Tag

```
<wl:cache>
<!--the content between these tags will only be
 refreshed on server restart-->
</wl:cache>

<wl:cache key="request.ticker" timeout="1m">
<!--get stock quote for whatever is in the request parameter ticker
 and display it, only update it every minute-->
</wl:cache>

<!--incoming parameter value isbn is the number used to lookup the
 book in the database-->
<wl:cache key="parameter.isbn" timeout="1d" size="100">
<!--retrieve the book from the database and display
the information -- the tag will cache the top 100
most accessed book descriptions-->
</wl:cache>

<wl:cache timeout="15m">
<!--get the new headlines from the database every 15 minutes and
 display them-->
</wl:cache>
```

Process Tag

Use the <wl:process> tag for query parameter-based flow control. By using a combination of the tag's four attributes, you can selectively execute the statements between the <wl:process> and </wl:process> tags. The process tag may also be used to declaratively process the results of form submissions. By specifying conditions based on the values of request parameters you can include or not include JSP syntax on your page.

Table 4-2 Process Tag Attributes

Tag Attribute	Required	Description
name	no	Name of a query parameter.
notname	no	Name of a query parameter.
value	no	Value of a query parameter.
notvalue	no	Value of a query parameter.

The following examples show how you can use the <wl:process> tag:

Listing 4-2 Examples of Using the process tag:

```
<wl:process notname="update">
<wl:process notname="delete">
<!--Only show this if there is no update or delete parameter-->
<form action="<%= request.getRequestURI() %>">
  <input type="text" name="name"/>
  <input type="submit" name="update" value="Update"/>
  <input type="submit" name="delete" value="Delete"/>
</form>
</wl:process>
</wl:process>

<wl:process name="update">
<!-- do the update -->
</wl:process>
```

```
<wl:process name="delete">
<!--do the delete-->
</wl:process>

<wl:process name="lastBookRead" value="A Man in Full">
<!--this section of code will be executed if lastBookRead exists
 and the value of lastBookRead is "A Man in Full"-->
</wl:process>
```

Repeat Tag

Use the `<wl:repeat>` tag to iterate over many different types of sets, including Enumerations, Iterators, Collections, Arrays of Objects, Vectors, ResultSets, ResultSetMetaData, and the keys of a Hashtable. You can also just loop a certain number of times by using the `count` attribute. Use the set attribute to specify the type of Java objects.

Table 4-3 Repeat Tag Attributes

Tag Attribute	Required	Type	Description
`set`	No	Object	The set of objects that includes: ■ Enumerations ■ Iterators ■ Collections ■ Arrays ■ Vectors ■ Result Sets ■ Result Set MetaData ■ Hashtable keys
`count`	No	Int	Iterate over first *`count`* entries in the set.
`id`	No	String	Variable used to store current object being iterated over.

Table 4-3 Repeat Tag Attributes

Tag Attribute	Required	Type	Description
`type`	No	String	Type of object that results from iterating over the set you passed in. Defaults to `Object`. This type must be fully qualified.

The following example shows how you can use the `<wl:repeat>` tag.

Listing 4-3 Examples of Using the repeat Tag

```
<wl:repeat id="name" set="<%= new String[] { "sam", "fred", "ed" }
%>">
  <%= name %>
</wl:repeat>

<% Vector v = new Vector();%>
<!--add to the vector-->

<wl:repeat id="item" set="<%= v.elements() %>">
<!--print each element-->
</wl:repeat>
```

CHAPTER

5 Using WebLogic JSP Form Validation Tags

The following sections describe how to use WebLogic JSP form validation tags:

- Overview of WebLogic JSP Form Validation Tags
- Validation Tag Attribute Reference
- Using WebLogic JSP Form Validation Tags in a JSP
- Creating HTML Forms Using the <wl:form> Tag
- Using a Custom Validator Class
- Sample JSP with Validator Tags

Overview of WebLogic JSP Form Validation Tags

WebLogic JSP form validation tags provide a convenient way to validate the entries an end user makes to HTML form text fields generated by JSP pages. Using the WebLogic JSP form validation tags prevents unnecessary and repetitive coding of commonly used validation logic. The validation is performed by several custom JSP tags that are included with the WebLogic Server distribution. The tags can

- Verify that required fields have been filled in (`Required Field Validator` class).

- Validate the text in the field against a regular expression (`Regular Expression Validator` class).
- Compare two fields in the form (`Compare Validator` class).
- Perform custom validation by means of a Java class that you write (`Custom Validator` class).

WebLogic JSP form validation tags include:

- `<wl:summary>`
- `<wl:form>`
- `<wl:validator>`

When a validation tag determines that data in a field is not been input correctly, the page is re-displayed and the fields that need to be re-entered are flagged with text or an image to alert the end user. Once the form is correctly filled out, the end user's browser displays a new page specified by the validation tag.

Validation Tag Attribute Reference

This section describes the WebLogic form validation tags and their attributes. Note that the prefix used to reference the tag can be defined in the `taglib` directive on your JSP page. For clarity, the `wl` prefix is used to refer to the WebLogic form validation tags throughout this document.

<wl:summary>

`<wl:summary>` is the parent tag for validation. Place the opening `<wl:summary>` tag before any other element or HTML code in the JSP. Place the closing `</wl:summary>` tag anywhere *after* the closing `</wl:form>` tag(s).

`name`

(Optional) Name of a vector variable that holds all validation error messages generated by the `<wl:validator>` tags on the JSP page. If you do not define this attribute, the default value, `errorVector`, is used. The text of the error

message is defined with the `errorMessage` attribute of the `<wl:validator>` tag.

To display the values in this vector, use the `<wl:errors/>` tag. To use the `<wl:errors/>` tag, place the tag on the page where you want the output to appear. For example:

```
<wl:errors color="red"/>
```

Alternately, you can use a scriptlet. For example:

```
<% if (errorVector.size() > 0) {
   for (int i=0; i < errorVector.size(); i++) {
   out.println((String)errorVector.elementAt(i));
   out.println("<br>");
   }
} %>
```

Where *errorVector* is the name of the vector assigned using the `name` attribute of the `<wl:summary>` tag.

The `name` attribute is required when using multiple forms on a page.

`headerText`

A variable that contains text that can be displayed on the page. If you only want this text to appear when when errors occur on the page, you can use a scriptlet to test for this condition. For example:

```
<% if(summary.size() >0 ) {
   out.println(headerText);
   }
%>
```

Where *summary* is the name of the vector assigned using the `name` attribute of the `<wl:summary>` tag.

`redirectPage`

URL for the page that is displayed if the form validation does not return errors. This attribute is not required if you specify a URL in the `action` attribute of the `<wl:form>` tag.

Note: Do not set the `redirectPage` attribute to the same page containing the `<wl:summary>` tag—you will create an infinite loop causing a `StackOverFlow` exception.

<wl:form>

The <wl:form> tag is similar to the HTML <form> tag and defines an HTML form that can be validated using the the WebLogic JSP form validataion tags. You can define multiple forms on a single JSP by uniquely identifying each form using the name attribute.

`method`

Enter GET or POST. Functions exactly as the method attribute of the HTML <form> tag.

`action`

URL for the page that is displayed if the form validation does not return errors. The value of this attribute takes precedence over the value of the redirectPage attribute of the <wl:summary> tag and is useful if you have multiple forms on a single JSP page.

Note: Do not set the action attribute to the same page containing the <wl:form> tag—you will create an infinite loop causing a StackOverFlow exception.

`name`

Functions exactly as the name attribute of the HTML <form> tag. Identifies the form when multiple forms are used on the same page. The name attribute is also useful for JavaScript references to a form.

<wl:validator>

Use one or more <wl:validator> tags for each form field. If, for instance, you want to validate the input against a regular expression and also require that something be entered into the field you would use two <wl:validator> tags, one using the RequiredFieldValidator class and another using the RegExpValidator class. (You need to use both of these validators because blank values are evaluated by the Regular Expression Field Validator as valid.)

`errorMessage`

A string that is stored in the vector variable defined by the name attribute of the <wl:summary> tag.

`expression`
: When using the `RegExpValidator` class, the regular expression to be evaluated.

  If you are not using `RegExpValidator`, you can omit this attribute.

`fieldToValidate`
: Name of the form field to be validated. The name of the field is defined with the `name` attribute of the HTML `<input>` tag.

`validatorClass`
: The name of the Java class that executes the validation logic. Three classes are provided for your use. You can also create your own custom validator class. For more information, see "Using a Custom Validator Class" on page 5-10.

  The available validation classes are:

  `weblogicx.jsp.tags.validators.RequiredFieldValidator`
  : Validates that some text has been entered in the field.

  `weblogicx.jsp.tags.validators.RegExpValidator`
  : Validates the text in the field using a standard regular expression.

    Note: A blank value is evaluated as valid.

  `weblogicx.jsp.tags.validators.CompareValidator`
  : Checks to see if two fields contain the same string. When using this class, set the `fieldToValidate` attribute to the two fields you want to compare. For example:

    ```
    fieldToValidate="field_1,field_2"
    ```

    Note: If both fields are blank, the comparison is evaluated as valid.

  `myPackage.myValidatorClass`
  : Specifies a custom validator class.

Using WebLogic JSP Form Validation Tags in a JSP

To use a validation tag in a JSP:

1. Write the JSP.

   a. Enter a `taglib` directive to reference the tag library containing the WebLogic JSP Form Validation Tags. For example:

      ```
      <%@ taglib uri="tagl" prefix="wl" %>
      ```

      Note that the prefix attribute defines the prefix used to reference all tags in your JSP page. Although you may set the prefix to any value you like, the tags referred to in this document using the `wl` prefix.

   b. Enter the `<wl:summary> ... </wl:summary>` tags.

      Place the opening `<wl:summary ...>` tag *before* any HTML code, JSP tag, scriptlet, or expression on the page.

      Place the closing `</wl:summary>` tag anywhere *after* the `</wl:form>` tag(s).

   c. Define an HTML form using the `<wl:form>` JSP tag that is included with the supplied tag library. For more information, see "<wl:form>" on page 5-4 and "Creating HTML Forms Using the <wl:form> Tag" on page 5-8. Be sure to close the form block with the `</wl:form>` tag. You can create multiple forms on a page if you uniquely define the `name` attribute of the `<wl:form>` tag for each form.

   d. Create the HTML form fields using the HTML `<input>` tag.

   e. Add `<wl:validator>` tags. For the syntax of the tags, see "<wl:validator>" on page 5-4. Place `<wl:validator>` tags on the page where you want the error message or image to appear. If you use multiple forms on the same page, place the `<wl:validator>` tag inside the `<wl:form>` block containing the form fields you want to validate.

      The following example shows a validation for a required field:

```
<wl:form name="FirstForm" method="POST" action="thisJSP.jsp">

<wl:validator
 errorMessage="Field_1 is required" expression=""
 fieldToValidate="field_1"
 validatorClass=
  "weblogicx.jsp.tags.validators.RequiredFieldValidator"
>
  <img src="images/warning.gif">
  <font color=red>Field 1 is a required field</font>
</wl:validator>

<p> <input type="text" name = "field_1">  </p>
<p> <input type="text" name = "field_2">  </p>
<p> <input type="submit" value="Submit FirstForm">  </p>
</wl:form>
```

If the user fails to enter a value in `field_1`, the page is redisplayed, showing a `warning.gif` image, followed by the text (in red) "`Field 1 is a required field,`" followed by the blank field for the user to re-enter the value.

2. Copy the `weblogic-vtags.jar` file from the `ext` directory of your WebLogic Server installation into the `WEB-INF/lib` directory of your Web Application. You may need to create this directory.

3. Configure your Web Application to use the tag library by adding a `<taglib>` element to the `web.xml` deployment descriptor for the Web Application. For example:

```
<taglib>
  <taglib-uri>tagl</taglib-uri>
  <taglib-location>
    /WEB-INF/lib/weblogic-vtags.jar
  </taglib-location>
</taglib>
```

For more information on Web Application deployment descriptors, see Writing Web Application Deployment Descriptors at `http://e-docs.bea.com/wls/docs61/webapp/webappdeployment.html`.

Creating HTML Forms Using the <wl:form> Tag

This section contains information on creating HTML forms in your JSP page. You use the `<wl:form>` tag to create a single form or multiple forms on a page.

Defining a Single Form

Use the `<wl:form>` tag that is provided in the `weblogic-vtags.jar` tag library: For example:

```
<wl:form method="POST" action="nextPage.jsp">
<p> <input type="text" name ="field_1">  </p>
<p> <input type="text" name ="field_2">  </p>
<p> <input type="submit" value="Submit Form">  </p>
</wl:form>
```

For information on the syntax of this tag see "<wl:form>" on page 5-4.

Defining Multiple Forms

When using multiple forms on a page, use the `name` attribute to identify each form. For example:

```
<wl:form name="FirstForm" method="POST" action="thisJSP.jsp">
<p> <input type="text" name="field_1">  </p>
<p> <input type="text" name="field_2">  </p>
<p> <input type="submit" value="Submit FirstForm">  </p>
</wl:form>

<wl:form name="SecondForm" method="POST" action="thisJSP.jsp">
<p> <input type="text" name="field_1">  </p>
<p> <input type="text" name="field_2">  </p>
<p> <input type="submit" value="Submit SecondForm">  </p>
</wl:form>
```

Re-Displaying the Values in a Field When Validation Returns Errors

When the JSP page is re-displayed after the validator tag has found errors, it is useful to re-display the values that the user already entered, so that the user does not have to fill out the entire form again. Use the `value` attribute of the HTML `<input>` tag or use a tag library available from the Apache Jakarta Project. Both procedures are described next.

Re-Displaying a Value Using the <input> Tag

You can use the `javax.servlet.ServletRequest.getParameter()` method together with the `value` attribute of the HTML `<input>` tag to re-display the user's input when the page is re-displayed as a result of failed validation. For example:

```
<input type="text" name="field_1"
 value="<%= request.getParameter("field_1") %>" >
```

To prevent cross-site scripting security vulnerabilities, replace any HTML special characters in user-supplied data with HTML entity references. See "Securing User-Supplied Data in JSPs" on page 3-14.

Re-Displaying a Value Using the Apache Jakarta <input:text> Tag

You can also use a JSP tag library available free from the Apache Jakarta Project, which provides the `<input:text>` tag as a replacement for the HTML `<input>` tag. For example, the following HTML tag:

```
<input type="text" name="field_1">
```

could be entered using the Apache tag library as:

```
<input:text name="field_1">
```

For more information and documentation, download the Input Tag library, available at `http://jakarta.apache.org/taglibs/doc/input-doc/intro.html`.

To use the Apache tag library in your JSP:

1. Copy the `input.jar` file from the Input Tag Library distribution file into the `WEB-INF/lib` directory of your Web Application.

2. Add the following directive to your JSP:

```
<%@ taglib uri="input" prefix="input" %>
```

3. Add the following entry to the `web.xml` deployment descriptor of your Web Application:

```
<taglib>
    <taglib-uri>input</taglib-uri>
    <taglib-location>/WEB-INF/lib/input.jar</taglib-location>
</taglib>
```

Using a Custom Validator Class

To use your own validator class:

1. Write a Java class that extends the `weblogicx.jsp.tags.validators.CustomizableAdapter` abstract class. For more information, see "Extending the CustomizableAdapter Class" on page 5-11.
2. Implement the `validate()` method. In this method:
   a. Look up the value of the field you are validating from the `ServletRequest` object. For example:

      ```
      String val = req.getParameter("field_1");
      ```

   b. Return a value of `true` if the field meets the validation criteria.
3. Compile the validator class and place the compiled `.class` file in the `WEB-INF/classes` directory of your Web Application.
4. Use your validator class in a `<wl:validator>` tag by specifying the class name in the `validatorClass` attribute. For example:

   ```
   <wl:validator errorMessage="This field is required"
   fieldToValidate="field_1"
   validatorClass="mypackage.myCustomValidator">
   ```

Extending the CustomizableAdapter Class

The `CustomizableAdapter` class is an abstract class that implements the `Customizable` interface and provides the following helper methods:

`getFieldToValidate()`
: Returns the name of the field being validated (defined by the `fieldToValidate` attribute in the `<wl:validator>` tag)

`getErrorMessage()`
: Returns the text of the error message defined with the `errorrMessage` attribute in the `<wl:validator>` tag.

`getExpression()`
: Returns the text of the `expression` attribute defined in the `<wl:validator>` tag.

Instead of extending the `CustomizableAdapter` class, you can implement the `Customizable` interface. For more information, see the Javadocs for weblogicx.jsp.tags.validators.Customizable at `http://e-docs.bea.com/wls/docs61/javadocs/weblogicx/jsp/tags/validators/Customizable.html`.

Sample User-Written Validator Class

Listing 5-1 Example of a User-written Validator Class

```
import weblogicx.jsp.tags.validators.CustomizableAdapter;

public class myCustomValidator extends CustomizableAdapter{

    public myCustomValidator(){
       super();
    }

    public boolean validate(javax.servlet.ServletRequest req)
       throws Exception {
       String val = req.getParameter(getFieldToValidate());
        // perform some validation logic
        // if the validation is successful, return true,
         // otherwise return false
```

```
        if (true) {
            return true;
        }
        return false;
    }

}
```

Sample JSP with Validator Tags

This sample code shows the basic structure of a JSP that uses the WebLogic JSP form validation tags. A complete functioning code example is also available if you installed the examples with your WebLogic Server installation. Instructions for running the example are available at `samples/examples/jsp/tagext/form_validation/package.html`, in your WebLogic Server installation.

Listing 5-2 JSP with WebLogic JSP Form Validation Tags

```
<%@ taglib uri="tagl" prefix="wl" %>
<%@ taglib uri="input" prefix="input" %>


<wl:summary
name="summary"
headerText="<font color=red>Some fields have not been filled out
correctly.</font>"
redirectPage="successPage.jsp"
>

<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
</head>

<body bgcolor="#FFFFFF">
```

```
<% if(summary.size() >0 )   {
    out.println("<h3>" + headerText + "</h3>");
} %>


<% if (summary.size() > 0) {
       out.println("<H2>Error Summary:</h2>");
       for (int i=0; i < summary.size(); i++) {
       out.println((String)summary.elementAt(i));
       out.println("<br>");
       }
} %>


<wl:form method="GET" action="successPage.jsp">

 User Name: <input:text name="username"/>
 <wl:validator
   fieldToValidate="username"

validatorClass="weblogicx.jsp.tags.validators.RequiredFieldValida
tor"
   errorMessage="User name is a required field!"
 >
   <img src=images/warning.gif> This is a required field!
 </wl:validator>

<p>

 Password: <input type="password" name="password">
 <wl:validator
   fieldToValidate="password"

validatorClass="weblogicx.jsp.tags.validators.RequiredFieldValida
tor"
   errorMessage="Password is a required field!"
 >
   <img src=images/warning.gif> This is a required field!
 </wl:validator>

 <p>

 Re-enter Password: <input type="password" name="password2">
 <wl:validator
   fieldToValidate="password,password2"
   validatorClass="weblogicx.jsp.tags.validators.CompareValidator"
   errorMessage="Passwords don't match"
 >
```

```
    <img src=images/warning.gif> Passwords don't match.
  </wl:validator>

  <p>

  <input type="submit" value="Submit Form">  </p>

</wl:form>

</wl:summary>


</body>
</html>
```

CHAPTER

6 Using the WebLogic EJB to JSP Integration Tool

The following sections describe how to use the WebLogic EJB-to-JSP integration tool to create JSP tag libraries that you can use to invoke EJBs in a JavaServer Page (JSP). This document assumes at least some familiarity with both EJB and JSP.

- Overview of the WebLogic EJB-to-JSP Integration Tool
- Basic Operation
- Interface Source Files
- Build Options Panel
- Troubleshooting
- Using EJB Tags on a JSP Page
- EJB Home Methods
- Stateful Session and Entity Beans
- Default Attributes

Overview of the WebLogic EJB-to-JSP Integration Tool

Given an EJB jar file, the WebLogic EJB-to-JSP integration tool will generate a JSP tag extension library whose tags are customized for calling the EJB(s) of that jar file. From the perspective of a client, an EJB is described by its remote interface. For example:

```
public interface Trader extends javax.ejb.EJBObject {
  public TradeResult buy(String stockSymbol, int shares);
  public TradeResult sell(String stockSymbol, int shares);
}
```

For Web Applications that call EJBs, the typical model is to invoke the EJB using Java code from within a JSP scriptlet (<% ... %>). The results of the EJB call are then formatted as HTML and presented to the Web client. This approach is both tedious and error-prone. The Java code required to invoke an EJB is lengthy, even in the simplest of cases, and is typically not within the skill set of most Web designers responsible for HTML presentation.

The EJB-to-JSP tool simplifies the EJB invocation process by removing the need for java code. Instead, you invoke the EJB is invoked using a JSP tag library that is custom generated for that EJB. For example, the methods of the Trader bean above would be invoked in a JSP like this:

```
<% taglib uri="/WEB-INF/trader-tags.tld" prefix="trade" %>
<b>invoking trade: </b><br>

<trade:buy stockSymbol="BEAS" shares="100"/>

<trade:sell stockSymbol="MSFT" shares="200"/>
```

The resulting JSP page is cleaner and more intuitive. A tag is (optionally) generated for each method on the EJB. The tags take attributes that are translated into the parameters for the corresponding EJB method call. The tedious machinery of invoking the EJB is hidden, encapsulated inside the handler code of the generated tag library. The generated tag libraries support stateless and stateful session beans, and entity beans. The tag usage scenarios for each of these cases are slightly different, and are described below.

Basic Operation

You can run the WebLogic EJB-to-JSP integration tool in command-line mode using the following command:

```
java weblogic.servlet.ejb2jsp.Main
```

or graphical mode. For all but the simplest EJBs, the graphical tool is preferable.

Invoke the graphical tool as follows:

```
java weblogic.servlet.ejb2jsp.gui.Main
```

Initially, no ejb2jsp project is loaded by the Web Application. Create a new project by selecting the **File -> New** menu item, browsing in the file chooser to an EJB jar file, and selecting it. Once initialized, you can modify, save, and reload ejb2jsp projects for future modification.

The composition of the generated tag library is simple: for each method, of each EJB, in the jar file, a JSP tag is generated, with the same name as the method. Each tag expects as many attributes as the corresponding method has parameters.

Interface Source Files

When a new EJB jar is loaded, the tool also tries to find the Java source files for the home and remote interfaces of your EJB(s). The reason is that, although the tool can generate tags only by introspecting the EJB classes, it cannot assign meaningful attribute names to the tags whose corresponding EJB methods take parameters. In the **Trader** example in “Overview of the WebLogic EJB-to-JSP Integration Tool” on page 2, when the EJB jar is loaded, the tool tries to find a source file called **Trader.java**. This file is then parsed and detects that the **buy()** method takes parameters called **stockSymbol** and **shares**. The corresponding JSP tag will then have appropriately named attributes that correspond to the parameters of the **buy()** method.

When a new EJB jar is loaded, the tool operates on the premise that the source directory is the same directory where the EJB jar is located. If that is not the case, the error is not fatal. After the new project is loaded, under the **Project Build Options** panel, you can adjust the **EJB Source Path** element to reflect the correct directory. You can then select the **File -> Resolve Attributes** menu to re-run the resolve process.

When looking for java source files corresponding to an interface class, the tool searches in both the directory specified, and in a sub-directory implied by the interface's java package. For example, for **my.ejb.Trader**, if the directory given is **C:/src**, the tool will look for both **C:/src/Trader.java** and **C:/src/my/ejb/Trader.java**.

Access to the source files is not strictly necessary. You can always modify attribute names for each tag in a project by using the tool. However, parsing the source files of the EJB's public interface was developed as the quickest way to assign meaningful attribute names.

Build Options Panel

Use this panel to set all parameters related to the local file system that are needed to build the project. Specify the Java compiler, the Java package of the generated JSP tag handlers, and whether to keep the generated Java code after a project build, which can be useful for debugging.

You can also use this panel to specify the type of tag library output you want. For use in a J2EE web application, a tag library should be packaged one of two ways: as separate class files and a Tag Library Descriptor (.tld) file, or as a single taglib jar file. Either output type is chosen with the **Output Type** pull-down. For development and testing purposes, **DIRECTORY** output is recommended, because a Web Application in WebLogic Server must be re-deployed before a jar file can be overwritten.

For either **DIRECTORY** or **JAR**, the output locations must be chosen appropriately so that the tag library will be found by a web application. For example, if you wish to use the tag library in a web application rooted in directory **C:/mywebapp**, then the **DIRECTORY classes** field should be specified as:

```
C:/mywebapp/WEB-INF/classes
```

and the **DIRECTORY .tld File** field should be something like:

```
C:/mywebapp/WEB-INF/trader-ejb.tld
```

The **Source Path**, described earlier, is edited in the **Build Options** panel as well. The **Extra Classpath** field can be used if your tag library depends on other classes not in the core WebLogic Server or J2EE API. Typically, nothing will need to be added to this field.

Troubleshooting

Sometimes, a project fails to build because of errors or conflicts. This section describes the reasons for those errors, and how they may be resolved.

- **Missing build information** One of the necessary fields in the **Build Options** panel is unspecified, like the java compiler, the code package name, or a directory where the output can be saved. The missing field(s) must be filled in before the build can succeed.
- **Duplicate tag names** When an EJB jar is loaded, the tool records a tag for each method on the EJB, and the tag name is the same as the method name. If the EJB has overloaded methods (methods with the same name but different signatures), the tag names conflict. Resolve the conflict by renaming one of the tags or by disabling one of the tags. To rename a tag, navigate to the tag in question using the tree hierarchy in the left window of the tool. In the tag panel that appears in the right window, modify the **Tag Name** field. To disable a tag, navigate to the tag in question using the tree hierarchy in the left window of the tool. In the tag panel that appears in the right window, deselect the **Generate Tag** box. For EJB jars that contain multiple EJBs, you can disable tags for an entire bean may as well.
- **Meaningless attribute names arg0, arg1...** This error occurs when reasonable attribute names for a tag could not be inferred from the EJB's interface source files. To fix this error, navigate to the tag in question in the project hierarchy tree. Select each of the attribute tree leaves below the tag, in order. For each attribute, assign a reasonable name to the **Attribute Name** field, in the panel that appears on the right side of the tool.
- **Duplicate attribute names** This occurs when a single tag expecting multiple attributes has two attributes with the same name. Navigate to the attribute(s) in question, and rename attributes so that they are all unique for the tag.

Using EJB Tags on a JSP Page

Using the generated EJB tags on a JSP page is simply a matter of declaring the tag library on the page, and then invoking the tags like any other tag extension:

```
<% taglib uri="/WEB-INF/trader-ejb.tld"
 prefix="trade" %>
<trade:buy stockSymbol="XYZ" shares="100"/>
```

For EJB methods that have a non-void return type, a special, optional tag attribute "_return", is built-in. When present, the value returned from the method is made available on the page for further processing:

```
<% taglib uri="/WEB-INF/trader-ejb.tld"
  prefix="trade" %>
<trade:buy stockSymbol="XYZ"
 shares="100" _return="tr"/>
<% out.println("trade result: " + tr.getShares()); %>
```

For methods that return a primitive numeric type, the return variable is a Java object appropriate for that type (for example, "int" -> java.lang.Integer, etc).

EJB Home Methods

EJB 2.0 allows for methods on the EJB home interface that are neither **create()** or **find()** methods. Tags are generated for these home methods as well. To avoid confusion, the tool prepends **"home-"** to the tags for each method on an EJB's home, when a new project is loaded. These methods may be renamed, if desired.

Stateful Session and Entity Beans

Typical usage of a “stateful” bean is to acquire an instance of the bean from the bean's Home interface, and then to invoke multiple methods on a single bean instance. This programming model is preserved in the generated tag library as well. Method tags for stateful EJB methods are required to be inside a tag for the EJB home interface that corresponds to a **find()** or **create()** on the home. All EJB method tags contained within the find/create tag operate on the bean instance found or created by the enclosing tag. If a method tag for a stateful bean is not enclosed by a find/create tag for its home, a run-time exception occurs. For example, given the following EJB:

```
public interface AccountHome extends EJBHome {

  public Account create(String accountId, double initialBalance);
  public Account findByPrimaryKey(String accountID);
    /* find all accounts with balance above some threshold */
  public Collection findBigAccounts(double threshold);
}

public interface Account extends EJBObject {
  public String getAccountID();
  public double deposit(double amount);
  public double withdraw(double amount);
  public double balance();
}
```

Correct tag usage might be as follows:

```
<% taglib uri="/WEB-INF/account-ejb.tld" prefix="acct" %>
<acct:home-create accountId="103"
  initialBalance="450.0" _return="newAcct">
 <acct:deposit amount="20"/>
 <acct:balance _return="bal"/>
 Your new account balance is: <%= bal %>
</acct:home-create>
```

If the "_return" attribute is specified for a find/create tag, a page variable will be created that refers to the found/created EJB instance. Entity beans finder methods may also return a collection of EJB instances. Home tags that invoke methods returning a collection of beans will iterate (repeat) over their tag body, for as many beans as are returned in the collection. If "_return" is specified, it is set to the current bean in the iteration:

```
<b>Accounts above $500:</b>
<ul>
<acct:home-findBigAccounts threshold="500" _return="acct">
<li>Account <%= acct.getAccountID() %>
     has balance $<%= acct.balance() %>
</acct:home-findBigAccounts>
</ul>
```

The preceding example will display an HTML list of all Account beans whose balance is over $500.

Default Attributes

By default, the tag for each method requires that all of its attributes (method parameters) be set on each tag instance. However, the tool will also allow "default" method parameters to be specified, in case they are not given in the JSP tag. You can specify default attributes/parameters in the **Attribute** window of the EJB-to-JSP tool. The parameter default can come from an simple **EXPRESSION**, or if more complex processing is required, a default **METHOD** body may be written. For example, in the Trader example in “Overview of the WebLogic EJB-to-JSP Integration Tool” on page 2, suppose you want the “buy” tag to operate on stock symbol “XYZ” if none is specified. In the Attribute panel for the “stockSymbol” attribute of the “buy” tag, you set the “Default Attribute Value” field to **EXPRESSION**, and enter “XYZ” (quotes included!) in the **Default Expression** field. The buy tag then acts as if the stockSymbol="XYZ" attribute were present, unless some other value is specified.

Or if you want the shares attribute of the "buy" tag to be a random number between 0-100, we would set "Default Attribute Value" to **METHOD**, and in the **Default Method Body** area, you write the body of a Java method that returns int (the expected type for the "shares" attribute of the "buy" method):

```
long seed = System.currentTimeMillis();
java.util.Random rand = new java.util.Random(seed);
int ret = rand.nextInt();
/* ensure that it is positive...*/
ret = Math.abs(ret);
/* and < 100 */
return ret % 100;
```

Because your default method bodies appear within a JSP tag handler, your code has access to the **pageContext** variable. From the JSP PageContext, you can gain access to the current HttpServletRequest or HttpSession, and use session data or request parameters to generate default method parameters. For example, to pull the "shares" parameter for the "buy" method out of a ServletRequest parameter, you could write the following code:

```
HttpServletRequest req =
 (HttpServletRequest)pageContext.getRequest();
String s = req.getParameter("shares");
if (s == null) {
  /* webapp error handler will redirect to error page
   * for this exception
   */
  throw new BadTradeException("no #shares specified");
}
int ret = -1;
try {
  ret = Integer.parseInt(s);
} catch (NumberFormatException e) {
  throw new BadTradeException("bad #shares: " + s);
}
if (ret <= 0)
  throw new BadTradeException("bad #shares: " + ret);
return ret;
```

The generated default methods are assumed to throw exceptions. Any exceptions raised during processing will be handled by the JSP's errorPage, or else by the registered exception-handling pages of the Web Application.

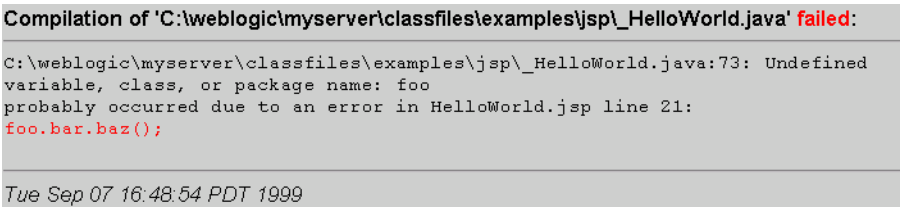
7 Troubleshooting

The following sections describe several techniques for debugging your JSP files:

- Debugging Information in the Browser
- Symptoms in the Log File

Debugging Information in the Browser

The most useful feature for debugging your JSP pages is the output that is sent to the browser by default. This output displays the location of the error in the generated HTTP servlet Java file, a description of the error, and the approximate location of the error code in the original JSP file. For example, when a compilation fails, the following message is displayed in the browser:



To disable this mechanism, set the `verbose` attribute to `false` in the jsp-descriptor element, `http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#jsp-descriptor` in the WebLogic-specific deployment descriptor of your Web Application.

Error 404—Not Found

Check that you have typed the URL of the JSP file correctly, and that it is relative to the root directory of your Web Application.

Error 500—Internal Server Error

Check the WebLogic Server log file for error messages, and see "Page Compilation Failed Errors" on page 3. This error usually indicates a `ClasssNotFound` exception has occured during JSP compilation.

Error 503—Service Unavailable

Indicates that WebLogic Server cannot find the compiler it requires to compile your JSPs. For more information about defining a JSP compiler, see "jsp-descriptor section, available at `http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#jsp-descriptor`.

Errors Using the <jsp:plugin> tag

If you use the `<jsp:plugin>` tag in your JSP and the applet fails to load, carefully check the syntax of the tag. You can check for possible syntax errors by examining the generated HTML page. If you see `<jsp:plugin ...` anywhere in the page, the syntax of the tag is not correct.

Symptoms in the Log File

This section describes JSP-related error messages in the WebLogic Server log file. As WebLogic Server runs, verbose messages are saved in a WebLogic log file. For more information about WebLogic log files, see "Using Log Messages to Manage WebLogic Servers" at `http://e-docs.bea.com/wls/docs61/adminguide/logging.html`.

Page Compilation Failed Errors

The following errors may occur if the JSP compiler fails to translate the JSP page into a Java file, or if it cannot compile the generated Java file. Check the log file for the following error messages:

`CreateProcess: ...`
: This indicates that the Java compiler cannot be found or is not a valid executable. For information about specifying a Java compiler, see jsp-descriptor section, available at `http://e-docs.bea.com/wls/docs61/webapp/weblogic_xml.html#jsp-descriptor`.

`Compiler failed:`
: The Java code generated from your JSP page cannot be compiled by the Java compiler. You can use the JSP compiler independently to inspect and debug the generated Java code in more detail. For more information see "Using the WebLogic JSP Compiler" on page 19.

`Undefined variable or classname:`
: If you are using a custom variable, make sure it is defined before you use it in a scriptlet or define it in a declaration. You may see this error if you attempt to use an implicit object from a declaration. Use of implicit objects in a declaration is not supported in JSP.

Index

A

action 5-4
actions 3-10
administration 2-1
applets 3-17
application 3-4

C

cache tag
    attributes 4-4
    overview 4-2
caching 4-2
character encoding 3-6
compile 3-19
compiler 7-3
compiling 7-3
config 3-5
configuration 2-2
contentType 3-6
custom tags 4-1
    and Web Applications 4-2
    cache 4-2
    configuration 4-2
    process 4-7
custom validator 5-10
customer support contact information ix

D

debugging 7-1
declaration 3-2
declarations 3-6
deployment descriptor 2-2
directive 3-2
    contentType 3-6
    taglib 3-6
directives 3-5
documentation, where to find it viii

E

encoding 3-6
errors
    404 7-2
    500 7-2
    503 7-2
    jsp plugin tag 7-2
    page compilation 7-3
expression 3-2, 5-5
expressions 3-8

F

fieldToValidate 5-5
form 5-4, 5-8
    action 5-4
    method 5-4
    name 5-4
form tag 5-8
form validation 5-1

G

getParameter() 5-9

H

headerText 5-3
HTML
    form tag 5-4
HTML forms 5-8
HTTP
    requests 1-3

I

input tag 5-9
    Apache Jakarta 5-9

J

Java Plugin 3-17
JavaBeans 3-10
JSP administration 2-1, 2-2
JSP compiler
    options 3-20
    syntax 3-19
JSP configuration 2-2

L

log file 7-3

M

method 5-4

N

name 5-4

O

out 3-4

P

page 3-5
pageContext 3-4
parameters 2-2
plugin 3-17
printing product documentation viii
process tag
    attributes 4-7
    overview 4-7

R

redirectPage 5-3
re-displaying value from a form 5-9
regular expression validation 5-5
request 3-3, 3-12
reserved words 3-3
    application 3-4
    config 3-5
    out 3-4
    page 3-5
    pageContext 3-4
    request 3-3
    response 3-3
    session 3-4
response 3-3

S

scope 3-12
    application 3-13
    page 3-12
    session 3-12
scriptlet 3-2
scriptlets 3-7
serializable 3-16
Servlet 2.2 specification 1-2
session 3-4
sessions 3-16
setting up JSP 2-2
summary 5-2

headerText 5-3
name attribute 5-2
redirectPage 5-3
support
technical ix

T

taglib 3-6, 4-2
tags 3-2, 4-1
custom 4-1
declaration 3-2
directive 3-2
scriptlet 3-2
troubleshooting
browser 7-1

V

validation 5-1
validation tag
form 5-4
validation tags
summary 5-2
validator 5-4
validation tags, using in a JSP 5-6
validator 5-1, 5-4
custom 5-10
errorMessage attribute 5-4
expression attribute 5-5
fieldToValidate 5-5
validatorClass 5-5
validatorClass 5-5
verbose 7-1

W

web application 2-2
web.xml 4-2
weblogic.xml 2-2
wl-form 5-4
wl-summary 5-2