BEA WebLogic Server

Programming WebLogic JSP

BEA WebLogic Server 6.0
Document Date: March 6, 2001

Programming WebLogic JSP

Part Number	Document Date	Software Version
	March 6, 2001	BEA WebLogic Server Version 6.0

Contents

About This Document

1. Overview

What Is JSP? 1-1
WebLogic Implementation of JSP 1-2
How JSP Requests Are Handled 1-2
Additional Information 1-3

2. Administering WebLogic JSP

Overview 2-1
Setting JSP Operating Parameters 2-2

3. WebLogic JSP Reference

JSP Tags 3-1
Reserved Words 3-3
Directives 3-5
- Using a Directive to Set Character Encoding 3-5
- The taglib Directive 3-6

Declarations 3-6
Scriptlets 3-6
Expressions 3-7
- Example with HTML and Embedded Java 3-8

Actions 3-9
- Using JavaBeans in JSP 3-9
  - Instantiating the JavaBean Object 3-10
  - Doing Setup Work at JavaBean Instantiation 3-10
  - Using the JavaBean Object 3-11

Defining the Scope of a JavaBean Object 3-11
Using Sessions with JSP 3-12
Deploying Applets from JSP 3-12
Using the WebLogic JSP Compiler 3-14
Syntax 3-14
JSP Compiler Options 3-15
Precompiling JSPs 3-17
.... 3-18

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

Overview 4-19
Using the WebLogic Custom Tags in a Web Application 4-20
Cache Tag 4-20
Refreshing a Cache 4-21
Flushing a Cache 4-21
Process Tag 4-25
Repeat Tag 4-26

5. Troubleshooting

Debugging Information in the Browser 5-1
Error 404—Not Found 5-2
Error 500—Internal Server Error 5-2
Error 503—Service Unavailable 5-2
Errors Using the <jsp:plugin> tag 5-2
Symptoms in the Log File 5-2
Page Compilation Failed Errors 5-3

About This Document

This document describes how to program using JavaServer Pages (JSP)..

The document is organized as follows:

- Chapter 1, "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 5, "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 or go directly to the WebLogic Server Product Documentation page at http://e-docs.bea.com/wls/docs60.

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/docs60/taglib/index.html`.
- Deploying and Configuring Web Applications at `http://e-docs.bea.com/wls/docs60/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 your 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.

CHAPTER

1 Overview

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

This section includes the following topics:

- 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 are creating 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 Two Enterprise Edition (J2EE).

JSP allows 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, JSPs may be deployed 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/docs60/taglib/index.html.)

WebLogic Server also supports the Servlet 2.2 specification (`http://java.sun.com/products/servlet/download.html#specs`) from Sun Microsystems.

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. WebLogic Server converts the JSP into a servlet class that implements `javax.servlet.jsp.JspPage` using the JSP compiler. 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 3-14. 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/docs60/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/docs60/taglib/index.html`.
- *Programming WebLogic HTTP Servlets* at `http://e-docs.bea.com/wls/docs60/servlet/index.html`.
- Deploying and Configuring Applications at `http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html`
- Writing Web Application Deployment Descriptors at `http://e-docs.bea.com/wls/docs60/programming/webappdeployment.html`

2 Administering WebLogic JSP

This section provides a brief overview of administration and configuration tasks for WebLogic JavaServer Pages (JSP). It includes the following topics:

- Overview
- Setting JSP Operating Parameters
- For a complete discussion of JSP administration and configuration see the "Deploying and Configuring Web Applications" section of the *WebLogic Server Administration Guide* at `http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html`.

Overview

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.

Web Applications are then deployed on one or more *Web Servers* running in WebLogic Server. A Web Server can use virtual hosting to respond to requests for different DNS names that are mapped to each Web Server. Each Web Server can host multiple Web Applications.

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.

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/docs60/programming/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.

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 "Writing Web Application Deployment Descriptors" at `http://e-docs.bea.com/wls/docs60/programming/webappdeployment.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/docs60/programming/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

This section provides a reference on writing JavaServer Pages (JSPs). The following topics are included:

- JSP Tags
- Reserved Words
- Directives
- Scriptlets
- Expressions
- Actions
- 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 short-hand tag has an XML equivalent. For details, see the related sections later in this document.

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 3-6.

`Directive`	`<%@ dir-type dir-attr %>`
	. . . or use the XML equivalent:
	`<jsp:directive.dir_type`
	`dir_attr />`
		Gives a *directive* to the application server. The `dir_type` determines the type of directive being given, which can accept a list of directives given as *`name="quotedValue"`* pairs separated by white space. See "Directives" on page 3-5.

`Declarations`	`<%! declaration %>`
	. . . or use XML equivalent...
	`<jsp:declaration>`
	`  declaration;`
	`</jsp:declaration>`
		Declares a variable or method that may 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 3-7.

`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 ... >`
		JSP Actions 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 3-9.

Reserved Words

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 Microsystem's JSP Homepage at `http://www.java.sun.com/products/jsp/index.html`.

Note: You may 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/docs60/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. Instead of using `page`, we recommend that you use the Java keyword `this` to reference the servlet instance and get access to initialization parameters.

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/docs60/servlet/index.html.

Directives

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. You can use either of two types of syntax: shorthand or XM:

- 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.

All JSP1.1 directives are supported by WebLogic JSP.

Using a 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 *`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 JSP Descriptor Element at http://e-docs.bea.com/wls/docs60/programming/weblogic_xml.html#jsp-descriptor.

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/docs60/taglib/index.html.

Declarations

Use declarations to define variables and methods at the class-scope level of the generated JSP servlet. Declarations made between these 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 the "Example with HTML and Embedded Java", shown later in this document, 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.
- 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

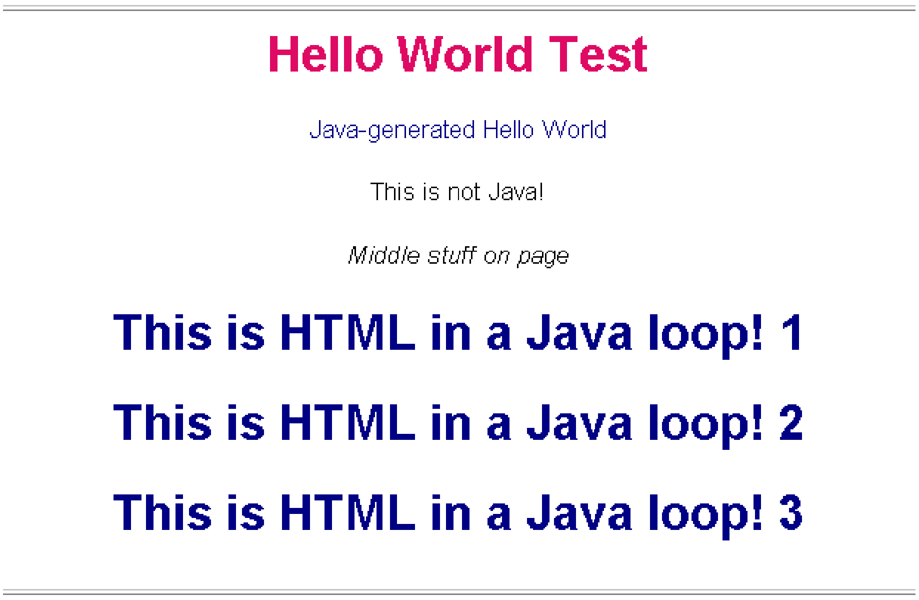
```
<% out.print( expr ); %>
```

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.

Example 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:


Hello World Test

Java-generated Hello World

This is not Java!

Middle stuff on page

This is HTML in a Java loop! 1

This is HTML in a Java loop! 2

This is HTML in a Java loop! 3

Actions

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

Using JavaBeans in JSP

The `<jsp:useBean>` 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 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`.

`applications`

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`.

Using Sessions with JSP

Sessions in WebLogic JSP perform according to the JSP 1.1 specification. The following suggestions highlight some important issues for using sessions:

- Objects stored in sessions should be small in size. For example, a session should not be used to store an EJB, but an EJB primary key instead. Large amounts of data should be stored in a database, and the session should hold only a simple string reference to the data.
- When sessions are used with dynamic reloading of servlets or JSP, the objects stored in the servlet session should 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.

Deploying Applets from JSP

JSP provides a convenient way to generate HTML that contains the appropriate client browser tag to include the Java Plug-in in a Web page. 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 Sun's 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 JSPServlet 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.

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 may 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 jspc. 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 specifices 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 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.

`-nowrite`
: Runs the compilation process without actually producing a `.class` file. If you combine this option with the `-keepgenerated` flag, the compiler creates only the intermediate `.java` file, which can be useful for debugging.

`-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 by defining the following context parameter in the `web.xml` deployment descriptor:

```
<context-param>
  <param-name>weblogic.jsp.precompile</param-name>
  <param-value>true</param-value>
</context-param>
```

For more information on the `web.xml` deployment descriptor, see Writing Web Application Deployment Descriptors at `http://e-docs.bea.com/wls/docs60/programming/webappdeployment.html`.

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

This section discusses the use of three custom JSP tags provided with the WebLogic Server distribution: the `cache` tag, the `repeat` tag, and the `process` tag. The following topics are included:

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

Overview

BEA provides three specialized JSP tags that you can use in your JSP pages. The tags are called: `cache`, `repeat`, and `process.` Thes 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 Deploying and Configuring Web Applications at `http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.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 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/docs60/programming/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 supports caching the work that is done within the body of the tag. To this end, 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.

Caches are stored using soft references to prevent the caching system from using too much system memory. Unfortunately, due to incompatibilities with the HotSpot VM and the Classic VM, soft references are not used when WebLogic Server is running within the HotSpot VM.

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 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 below) 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 sepcify an altenate unit by postfixing the value with descired 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.
`async`	`no`	`false`	If the async parameter is set to `true`, the cache will be updated asynchronously, if possible. The user that initiates the cache hit sees the old data.
`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" async="true">
<!--get the new headlines from the database every 15 minutes and
 display them-->
<!--do not let anyone see the pause while they are retrieved-->
</wl:cache>
```

Process Tag

The `<wl:process>` tag is used 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 on your page.

Table 4-2 Process Tag Attributes

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

The `<wl:repeat>` tag is used 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

Table 4-3 Repeat Tag Attributes

Tag Attribute	Required	Type	Description
count	No	Int	Iterate over first *`count`* entries in the set.
id	No	String	Variable used to store current object being iterated over.
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

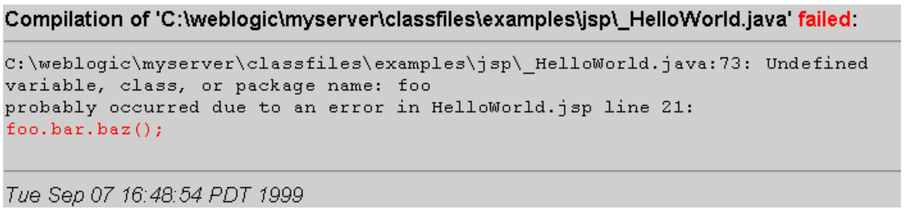
5 Troubleshooting

This section describes several techniques for debugging your JSP files. The following topics are included:

- 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:



Compilation of 'C:\weblogic\myserver\classfiles\examples\jsp_HelloWorld.java' failed:

```
C:\weblogic\myserver\classfiles\examples\jsp\_HelloWorld.java:73: Undefined
variable, class, or package name: foo
probably occurred due to an error in HelloWorld.jsp line 21:
foo.bar.baz();
```

Tue Sep 07 16:48:54 PDT 1999

To disable this mechanism, set the `verbose` attribute to `false` in the jsp-descriptor (see `http://e-docs.bea.com/wls/docs60/programming/weblogic_xml.html#jsp-descriptor`) element 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 5-3. 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 "Configuring Web Applications" at `http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html`.

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/docs60/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 "Configuring Web Applications" at `http://e-docs.bea.com/wls/docs60/adminguide/config_web_app.html`.

`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 3-14.

`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
actions 3-9
administration 2-1
applets 3-12
application 3-4
C
cache tag
    attributes 4-22
    overview 4-20
caching 4-20
character encoding 3-5
compile 3-14
compiler 5-3
compiling 5-3
config 3-4
configuration 2-2
contentType 3-5
custom tags 4-19
    and Web Applications 4-20
    cache 4-20
    configuration 4-20
    process 4-25
    repeat 4-26
customer support contact information vii
D
debugging 5-1
declaration 3-2
declarations 3-6
deployment descriptor 2-2
directive 3-2
    contentType 3-5

taglib 3-6
directives 3-5
documentation, where to find it vi
E
encoding 3-5
errors
404 5-2
500 5-2
503 5-2
jsp plugin tag 5-2
page compilation 5-3
expression 3-2
expressions 3-7
H
HTTP
requests 1-2
J
Java Plugin 3-13
JavaBeans 3-9
JSP administration 2-1, 2-2
JSP compiler
options 3-15
syntax 3-14
JSP configuration 2-2
L
log file 5-2
O
out 3-3
P
page 3-4
pageContext 3-4
parameters 2-2
plugin 3-13
printing product documentation vi
process tag
attributes 4-25
overview 4-25
R
repeat tag
overview 4-26

request 3-3, 3-11
reserved words 3-3
  application 3-4
  config 3-4
  out 3-3
  page 3-4
  pageContext 3-4
  request 3-3
  response 3-3
  session 3-4
response 3-3
S
scope 3-11
  application 3-11
  page 3-11
  session 3-11
scriptlet 3-2
scriptlets 3-6
serializable 3-12
Servlet 2.2 specification 1-2
session 3-4
sessions 3-12
setting up JSP 2-2
support
  technical vii
T
taglib 3-6, 4-20
tags 3-1, 4-19
  custom 4-19
  declaration 3-2
  directive 3-2
  scriptlet 3-2
troubleshooting
  browser 5-1
V
verbose 5-1
W
web application 2-2
web.xml 4-20
weblogic.xml 2-2