JavaServer Pages™ Specification

Version 1.1

please send comments to jsp-spec-comments@eng.sun.com

THE NETWORK IS THE COMPUTER™

Java Software
A Division of Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, California 94303
415 960-1300 fax 415 969-9131

November 30, 1999

Eduardo Pelegrí-Llopart, Larry Cable

JavaServer Pages™ Specification ("Specification")
Version: 1.1
Status: Final Release
Release: 12/17/99

NOTICE.
This Specification is protected by copyright and the information described herein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of this Specification may be reproduced in any form by any means without the prior written authorization of Sun Microsystems, Inc. (“Sun”) and its licensors, if any. Any use of this Specification and the information described herein will be governed by these terms and conditions and the Export Control and General Terms as set forth in Sun's website Legal Terms. By viewing, downloading or otherwise copying this Specification, you agree that you have read, understood, and will comply with all the terms and conditions set forth herein.

Sun hereby grants you a fully-paid, non-exclusive, non-transferable, worldwide, limited license (without the right to sublicense), under Sun's intellectual property rights that are essential to practice this Specification, to internally practice this Specification solely for the purpose of creating a clean room implementation of this Specification that: (i) includes a complete implementation of the current version of this Specification, without subsetting or supersetting; (ii) implements all of the interfaces and functionality of this Specification, as defined by Sun, without subsetting or supersetting; (iii) includes a complete implementation of any optional components (as defined by Sun in this Specification) which you choose to implement, without subsetting or supersetting; (iv) implements all of the interfaces and functionality of such optional components, without subsetting or supersetting; (v) does not add any additional packages, classes or interfaces to the "java.*" or "javax.*" packages or subpackages (or other packages defined by Sun); (vi) satisfies all testing requirements available from Sun relating to the most recently published version of this Specification six (6) months prior to any release of the clean room implementation or upgrade thereto; (vii) does not derive from any Sun source code or binary code materials; and (viii) does not include any Sun source code or binary code materials without an appropriate and separate license from Sun. This Specification contains the proprietary information of Sun and may only be used in accordance with the license terms set forth herein. This license will terminate immediately without notice from Sun if you fail to comply with any provision of this license. Sun may, at its sole option, terminate this license without cause upon ten (10) days notice to you. Upon termination of this license, you must cease use of or destroy this Specification.

TRADEMARKS.
No right, title, or interest in or to any trademarks, service marks, or trade names of Sun or Sun's licensors is granted hereunder. Sun, Sun Microsystems, the Sun logo, Java, Jini, JavaServer Pages, Enterprise JavaBeans, Java Compatible, JDK, JDBC, JAVASCRIPT, JavaBeans, JavaMail, Write Once, Run Anywhere, and Java Naming and Directory Interface are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES.
THIS SPECIFICATION IS PROVIDED "AS IS". SUN MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of this Specification in any product.

THIS SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. SUN MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY.
TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE SPECIFICATION, EVEN IF SUN AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
You will indemnify, hold harmless, and defend Sun and its licensors from any claims arising or resulting from: (i) your use of the Specification; (ii) the use or distribution of your Java application, applet and/or clean room implementation; and/or (iii) any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND.
Use, duplication, or disclosure by the U.S. Government is subject to the restrictions set forth in this license and as provided in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii)(Oct 1988), FAR 12.212(a) (1995), FAR 52.227-19 (June 1987), or FAR 52.227-14(ALT III) (June 1987), as applicable.

REPORT.
You may wish to report any ambiguities, inconsistencies, or inaccuracies you may find in connection with your use of the Specification ("Feedback"). To the extent that you provide Sun with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis and (ii) grant Sun a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

Contents

Preface

This is the final version of the *JavaServer Pages™ 1.1 Specification*. This specification has been developed following the Java Community Process. Comments from Experts, Participants, and the Public have been reviewed and incorporated into the specification where applicable.

JSP 1.1 extends JSP 1.0 by:

- Using Servlet 2.2 as the foundations for its semantics.
- Enabling the delivery of translated JSP pages into JSP containers.
- Providing a portable Tag Extension mechanism.

Details on the conditions under which this document is distributed are described in the license on page 2.

Who should read this document

This document is intended for:

- Web Server and Application Server vendors that want to provide JSP containers that conform to the Tag Extensions specification.
- Web Authoring Tool vendors that want to generate JSP pages that conform to the Tag Extensions specification.
- Service providers that want to deliver functionality as tag libraries.
- Sophisticated JSP page authors that want to define new tag libraries for their use, or who are responsible for creating tag libraries for the use of a group.
- Eager JSP page authors who do not want to or cannot wait for Web Authoring Tools, or even a User's Guide.

This document is not a User's Guide, but it contains some positioning and explanatory material.

Related Documents

JSP 1.1 requires only JDK™ 1.1 but it can take advantage of the Java 2 platform.

Implementors of JSP containers and authors of JSP pages will be interested in a number of other documents, of which the following are worth mentioning explicitly.

TABLE P-1 Some Related Documents

JSP home page	**http://java.sun.com/products/jsp**
Servlet home page	**http://java.sun.com/products/servlet**
JDK 1.1	http://java.sun.com/products/jdk/1.1
Java 2 Platform, Standard Edition	http://java.sun.com/products/jdk/1.2
Java 2 Platform, Enterprise Edition	http://java.sun.com/j2ee
XML in the Java Platform home page	http://java.sun.com/xml
JavaBeans™ technology home page	http://java.sun.com/beans
XML home page at W3C	http://www.w3.org/XML
HTML home page at W3C	http://www.w3.org/MarkUp
XML.org home page	http://www.xml.org

Acknowledgments

Many people contributed to the JavaServer Pages specifications. In addition to the people who helped with the JSP 1.0 specification and reference implementation we want to thank a few individuals for their special effort on the JSP 1.1 specification:

We want to thank the following people from Sun Microsystems: Suzanne Ahmed, Janet Breuer, Abhishek Chauhan, James Davidson, Chris Ferris, Michaela Gubbels, Mark Hapner, Jim Inscore, Costin Manalache, Rajiv Mordani, Mandar Raje, Bill Shannon, James Todd, Vanitha Venkatraman, Anil Vijendran, Connie Weiss and Cara Zanoff France.

The success of the Java Platform depends on the process used to define and evolve it. This open process permits the development of high quality specifications in internet time and involves many individuals and corporations. Although it is impossible to list all the individuals who have contributed, we would like to give thanks explicitly to the following individuals: JJ Allaire, Elias Bayeh, Hans Bergsten, Vince Bonfanti, Bjorn Carlston, Shane Claussen, Mike Conner, Scott Ferguson, Bob Foster, Mike Freedman, Chris Gerken, Sanjeev Kumar, Craig McClanahan, Rod Magnuson, Stefano Mazzocchi, Rod McChesney, Dave Navas, Tom Reilly, Simeon Simeonov, and Edwin Smith. Apologies to any we may have missed.

Last, but certainly not least important, we thank the software developers, Web authors and members of the general public who have read this specification, used the reference implementation, and shared their experience. You are the reason the JavaServer Pages technology exists.

CHAPTER 1

Overview

This chapter provides an overview of the JavaServer Pages technology.

1.1 The JavaServer Pages™ Technology

JavaServer Pages™ technology is the Java™ platform technology for building applications containing dynamic Web content such as HTML, DHTML, XHTML and XML. The JavaServer Pages technology enables the authoring of Web pages that create dynamic content easily but with maximum power and flexibility.

The JavaServer Pages technology offers a number of advantages:

- *Write Once, Run Anywhere™ properties*

  The JavaServer Pages technology is platform independent, both in its dynamic Web pages, its Web servers, and its underlying server components. You can author JSP pages on any platform, run them on any Web server or Web enabled application server, and access them from any Web browser. You can also build the server components on any platform and run them on any server.

- *High quality tool support*

  The Write Once, Run Anywhere properties of JSP allows the user to choose *best-of-breed* tools. Additionally, an explicit goal of the JavaServer Pages design is to enable the creation of high quality portable tools.

- *Reuse of components and tag libraries*

  The JavaServer Pages technology emphasizes the use of reusable components such as: JavaBeans™ components, Enterprise JavaBeans™ components and tag libraries. These components can be used in interactive tools for component development and page composition. This saves considerable development time while giving the cross-platform power and flexibility of the Java programming language and other scripting languages.

- *Separation of dynamic and static content*

  The JavaServer Pages technology enables the separation of static content from dynamic content that is inserted into the static template. This greatly simplifies the creation of content. This separation is supported by beans specifically designed for the interaction with server-side objects, and, specially, by the tag extension mechanism.

- *Support for scripting and actions*

  The JavaServer Pages technology supports scripting elements as well as actions. Actions permit the *encapsulation* of useful functionality in a convenient form that can also be manipulated by tools; scripts provide a mechanism to *glue together* this functionality in a per-page manner.

- *Web access layer for N-tier enterprise application architecture(s)*

  The JavaServer Pages technology is an integral part of the Java 2 Platform Enterprise Edition (J2EE), which brings Java technology to enterprise computing. You can now develop powerful middle-tier server applications, using a Web site that uses JavaServer Pages technology as a front end to Enterprise JavaBeans components in a J2EE compliant environment.

1.2 What is a JSP Page?

A JSP page is a text-based document that describes how to process a *request* to create a *response*. The description intermixes template data with some dynamic actions and leverages on the Java Platform.

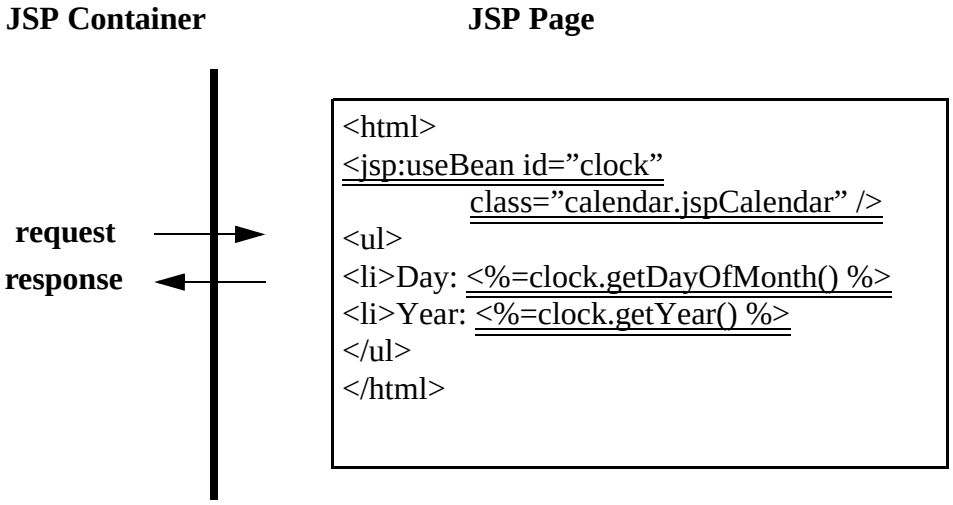
The features in the JSP technology support a number of different paradigms for authoring of dynamic content; some of them are described in Section 1.6. The next couple of examples only attempt to present the technical components of the JSP specification and are not prescribing “good” or “bad” paradigms.

An Example Using Scripting and Beans

An simple example of a JSP page is shown in FIGURE 1-1. The example shows the response page, which is intended to be a short list with the day of the month and year at the moment when the request is received. The page itself contains *fixed template text* and additional elements described by the JSP specification that are shown underlined in the figure. As the request reaches the page, the response is created based on the template text. As the first element is reached, a server-side Bean object is created with name `clock` and type

`calendar.jspCalendar.` This object can be used and modified later in the page. In particular, the next two elements access properties of the object and insert these values into the response page as strings.

FIGURE 1-1 A JSP Page using Beans and Scripting



An Example Using a Tag Library

FIGURE 1-2 is another example of a JSP page. This page uses custom actions to create the server-side object and then to produce the response data. In the example, a `taglib` directive first makes available into this page a tag library for data base queries. The directive indicates the tag library to use and provides a prefix to use locally in this page to name those actions.

Designing tag libraries is a delicate effort, analogous to that of designing a language; we are making no special effort here to define tags that are useful for any but pedagogical purposes. For the purposes of this example, we will assume that this fictitious tag library introduces four actions :

A **queryBlock** action introduces a data base connection; it can contain queryStatement actions and queryCreateRow actions. The connData attribute refers to connection-specific data, like login and password, that are to be defined elsewhere; see Appendix 5.8.3 for suggestions on where to place the information.

A **queryStatement** action must be enclosed in a queryBlock. A queryStatement's body is a SQL statement; it will use the connection data defined in the enclosing queryBlock.

A **`queryCreateRows`** action must be enclosed in a `queryBlock` action. A `queryCreateRows` action will iterate over the results of the last executed query and will generate up to as many rows as requested.

A **`queryDisplay`** action must be enclosed in a `queryCreateRows` action. A `queryDisplay` action will access the requested field from the current iteration in `queryCreateRows` and insert the value into the out object.

FIGURE 1-2 A JSP page using custom actions

```
<html>
<%@ taglib uri="http://acme.com/taglibs/simpleDB.tld" prefix="x" %>
<x:queryBlock connData="conData1">
    <x:queryStatement>
        SELECT ACCOUNT, BALANCE FROM ...
    </x:queryStatement>
    The top 10 accounts and balances are:
    <table>
        <tr><th>ACCOUNT</th><th>BALANCE</th></tr>
        <x:queryCreateRows from="1" to="10">
            <td><x:queryDisplay field="ACCOUNT"/></td>
            <td><x:queryDisplay field="BALANCE"/></td>
        </x:queryCreateRows>
    </table>
</x:queryBlock>
</html>
```

In this example:

- The x:queryCreateRows action implicitly refers to the object created by the x:queryStatement within the same x:queryBlock
- The x:queryDisplay actions refer to the current row in the query result that is being iterated over by x:queryCreateRows
- The code that locates a connection (perhaps from a connection pool), performs the JDBC™ API query, and navigates through the result of this query is hidden in the implementation of the custom actions. This encourages division of labor and isolation from changes.

Components and Containers

The JavaServer Pages technology builds on the Servlet standard extension. JavaServer Pages is a Standard Extension that is defined extending the concepts in the Servlet Standard Extension. JSP 1.1 uses the classes from Java Servlet 2.2 specification.

JSP pages and Servlet classes are collectively referred as *Web Components*. JSP pages are delivered to a *Container* that provides the services indicated in the *JSP Component Contract*.

JSP 1.1 and Servlet 2.2 rely only on features in the Java Runtime Environment 1.1, although they are compatible with, and can take advantage of, the Java 2 Runtime Environment.

1.3 Features in JSP 1.1

The JavaServer Pages specification includes:

- Standard directives
- Standard actions
- Script language declarations, scriptlets and expressions
- A portable tag extension mechanism.

Most of the integration of JSP pages within the J2EE platform is inherited from the reliance on the Servlet 2.2 specification.

1.4 Overview of JSP Page Semantics

This section provides an overview of the semantics of JSP pages

1.4.1 Translating and Executing JSP Pages

A *JSP page* is executed in a *JSP container*, which is installed on a Web server, or on a Web enabled application server. The JSP container delivers *requests* from a client to a JSP page and *responses* from the JSP page to the client. The semantic model underlying JSP pages is that of a servlet: a JSP page describes how to create a response object from a request object for a given protocol, possibly creating and/or using in the process some other objects.

All JSP containers must support HTTP as a protocol for requests and responses, but a container may also support additional request/response protocols. The default request and response objects are of type `HttpServletRequest` and `HttpServletResponse`, respectively.

A JSP page may also indicate how some events are to be handled. In JSP 1.1 only *init* and *destroy* events can be described: the first time a request is delivered to a JSP page a *jspInit()* method, if present, will be called to prepare the page. Similarly, a JSP container can reclaim the resources used by a JSP page at any time that a request is not being serviced by the JSP page by invoking first its *jspDestroy()* method; this is the same life-cycle as that of *Servlets.*

A JSP page is represented at request-time by a *JSP page implementation class* that implements the `javax.servlet.Servlet` interface. JSP pages are often implemented using a JSP page *translation phase* that is done only once, followed by some *request processing phase* that is done once per request. The translation phase creates the JSP page implementation class. If the JSP page is delivered to the JSP container in source form, the translation of a JSP source page can occur at any time between initial deployment of the JSP page into the runtime environment of a JSP container and the receipt and processing of a client request for the target JSP page.

A JSP page contains some *declarations*, some *fixed template* data, some (perhaps nested) *action instances*, and some *scripting elements.* When a request is delivered to a JSP page, all these pieces are used to create a response object that is then returned to the client. Usually, the most important part of this response object is the result stream.

1.4.2 Compiling JSP Pages

JSP pages may be *compiled* into its JSP page implementation class plus some deployment information. This enables the use of JSP page authoring tools and JSP tag libraries to author a Servlet. This has several benefits:

- Removal of the start-up lag that occurs when a JSP page delivered as source receives the first request.
- Reduction of the footprint needed to run a JSP container, as the java compiler is not needed.

If a JSP page implementation class depends on some support classes in addition to the JSP 1.1 and Servlet 2.2 classes, the support classes will have to be included in the packaged WAR so it will be portable across all JSP containers.

Appendix C contains two examples of packaging of JSP pages. One shows a JSP page that is delivered in source form (probably the most common case) within a WAR. The other shows how a JSP page is translated into a JSP page implementation class plus deployment information indicating the classes needed and the mapping between the original URL that was directed to the JSP page and the location of the Servlet.

1.4.3 Objects and Scopes

A JSP page can create and/or access some Java objects when processing a request. The JSP specification indicates that some objects are created implicitly, perhaps as a result of a directive (see Section 2.8, “Implicit Objects”); other objects are created explicitly through actions; objects can also be created directly using scripting code, although this is less common. The created objects have a *scope attribute* defining *where* there is a reference to the object and *when* that reference is removed.

The created objects may also be visible directly to the scripting elements through some scripting-level variables (see Section 1.4.7, “Objects and Variables”).

Each action and declaration defines, as part of its semantics, what objects it defines, with what scope attribute, and whether they are available to the scripting elements.

Objects are always created within some JSP page instance that is responding to some *request* object. There are several scopes:

- *page* - Objects with *page* scope are accessible only within the page where they are created. All references to such an object shall be released after the response is sent back to the client from the JSP page or the request is forwarded somewhere else. References to objects with *page* scope are stored in the `pageContext` object.
- *request* - Objects with *request* scope are accessible from pages processing the same request where they were created. All references to the object shall be released after the request is processed; in particular, if the request is forwarded to a resource in the same runtime, the object is still reachable. References to objects with *request* scope are stored in the `request` object.
- *session* - Objects with *session* scope are accessible from pages processing requests that are in the same session as the one in which they were created. It is not legal to define an object with session scope from within a page that is not session-aware (see Section 2.7.1, “The page Directive”). All references to the object shall be released after the associated session ends. References to objects with *session* scope are stored in the `session` object associated with the page activation.
- *application* - Objects with *application* scope are accessible from pages processing requests that are in the same application as they one in which they were created. All references to the object shall be released when the runtime environment reclaims the `ServletContext`. Objects with application scope can be defined (and reached) from pages that are not session-aware. References to objects with *application* scope are stored in the `application` object associated with a page activation.

A *name* should refer to a unique object at all points in the execution, i.e. all the different scopes really should behave as a single name space. A JSP container implementation may or not enforce this rule explicitly due to performance reasons.

1.4.4 Fixed Template Data

Fixed template data is used to describe those pieces that are to be used *verbatim* either in the response or as input to JSP actions. For example, if the JSP page is creating a presentation in HTML of a list of, say, books that match some search conditions, the template data may include things like the *
*, *
*, and something like *<li>The following book...*

This fixed template data is written (in lexical order) unchanged onto the output stream (referenced by the implicit *out* variable) of the response to the requesting client.

1.4.5 Directives and Actions

There may be two types of *elements* in a JSP page: *directives* or *actions*. *Directives* provide global information that is conceptually valid independent of any specific request received by the JSP page. For example, a directive can be used to indicate the scripting language to use in a JSP page. *Actions* may, and often will, depend on the details of the specific request received by the JSP page. If a JSP container uses a compiler or translator, the directives can be seen as providing information for the compilation/translation phase, while actions are information for the subsequent request processing phase.

An action may create some *objects* and may make them available to the scripting elements through some *scripting-specific variables.*

Directive elements have a syntax of the form

<%@ directive ...%>

Action elements follow the syntax of XML elements, i.e. have a start tag, a body and an end tag:

```
<mytag attr1=”attribute value” ...>
 body
</mytag>
```

or an empty tag

```
<mytab attr1=”attribute value” .../>
```

An element has an *element type* describing its tag name, its valid attributes and its semantics; we refer to the type by its tag name.

Tag Extension Mechanism

An element type abstracts some functionality by defining a specialized (sub)language that allows more natural expression of the tasks desired, can be read and written more easily by tools and also can even contribute specialized yet portable tool support to create them.

The JSP specification provides a Tag Extension mechanism (see Chapter 5) that enables the addition of new actions, thus allowing the JSP page "language" to be easily extended in a *portable* fashion. A typical example would be elements to support embedded database queries. Tag libraries can be used by JSP page authoring tools and can be distributed along with JSP pages to any JSP container like Web and Application servers.

The Tag Extension mechanism can be used from JSP pages written using any valid scripting language, although the mechanism itself only assumes a Java run time environment. Custom actions provide access to the attribute values and to their body; they can be nested and their bodies can include scripting elements.

1.4.6 Scripting Languages

Scripting elements are commonly used to manipulate objects and to perform computation that affects the content generated. There are three classes of scripting elements: *declarations*, *scriptlets* and *expressions. Declarations* are used to declare scripting language constructs that are available to all other scripting elements. *Scriptlets* are used to describe actions to be performed in response to some request. Scriplets that are program fragments can also be used to do things like iterations and conditional execution of other elements in the JSP page. *Expressions* are complete expressions in the scripting language that get evaluated at response time; commonly the result is converted into a string and then inserted into the output stream.

All JSP containers must support scripting elements based on the Java programming language. Additionally, JSP containers may also support other scripting languages. All such scripting languages must support:

- Manipulation of Java objects.
- Invocation of methods on Java objects.
- Catching of Java language exceptions.

The precise definition of the semantics for scripting done using elements based on the Java programming language is given in Chapter 4.

The semantics for other scripting languages are not precisely defined in this version of the specification, which means that portability across implementations cannot be guaranteed. Precise definitions may be given for other languages in the future.

1.4.7 Objects and Variables

An object may be made accessible to code in the scripting elements through a scripting language variable. An element can define scripting variables in two places: after its start tag and after its end tag. The variables will contain at process request-time a reference to the object defined by the element, although other references exist depending on the *scope* of the object (see Section 1.4.3, "Objects and Scopes").

An element type indicates the name and type of such variables although details on the name of the variable may depend on the Scripting Language. The scripting language may also affect how different features of the object are exposed; for example, in the JavaBeans specification, properties are exposed via *getter* and *setter* methods, while these are available directly in the JavaScript™ programming language.

The exact rules for the visibility of the variables are scripting language specific. Chapter 4 defines the rules for when the `language` attribute of the `page` directive is “java”.

1.4.8 Scripts, Actions, and Beans

Scripting elements, actions and Beans are all mechanisms that can be used to describe dynamic behavior in JSP pages. Different authors and authoring tools can use these mechanisms in different ways based on their needs and their preferences. The JSP specification does not restrict their use but this section provides some guidelines that may be useful to understand their relative strengths.

Beans are a well-known and well-supported component framework for the Java platform that can be accessed easily from the Java programming language and other JSP page scripting languages. Some JSP page authors, or their support organizations, may create or reuse Bean components to use from their JSP pages.

Actions provide an abstraction that can be used to easily encapsulate common actions. Actions typically create and / or act on (server-side) objects, often Beans.

The JSP specification provides some standard actions that can be used to interact with any Bean. If the Bean is extended so it implements the `Tag` interface, then the Bean becomes a tag handler and it can be used directly in the JSP page with improved integration into the template data.

Scripting elements are very flexible; that is their power but also their danger as they can make hard understanding and maintain a page that uses them extensively; they may also make it hard for an authoring tool. In some development contexts, JSP pages will mostly contain only actions (standard or custom) with scripting elements only used as a “gluing” mechanism that can be used to “fill-in” the actions that are described using actions (and Beans and EJB components). In other development contexts JSP pages may contain significant amounts of scripting elements.

1.4.9 JSP, HTML, and XML

The JSP specification is designed to support the dynamic creation of several types of structured documents, especially those using HTML and XML.

In general, a JSP page uses some data sent to the server in an HTTP request (for example, by a QUERY argument or a POST method) to interact with information already stored on the server, and then dynamically creates some content which is then sent back to the client. The content can be organized in some standard format (like HTML, DHTML, XHTML, XML, etc.), in some ad-hoc structured text format, or not at all.

There is another relationship between JSP and XML: a JSP page has a standard translation into a valid XML document. This translation is useful because it provides a standard mechanism to use XML tools and APIs to read, manipulate, and author JSP documents. The translation is defined in Chapter 7. JSP 1.1 processors are not required to accept JSP pages in this standard XML syntax, but this may be required in a future version of the JSP specification.

1.5 Web Applications

A prototypical Web application can be composed from:

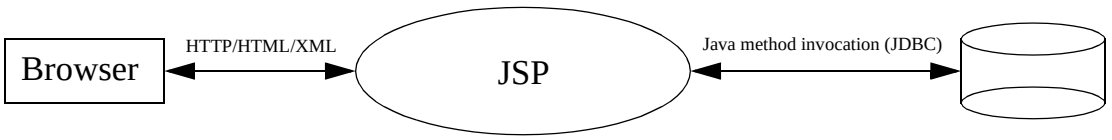
- Java Runtime Environment(s) running in the server (required)
- JSP page(s), that handle requests and generate dynamic content
- Servlet(s), that handle requests and generate dynamic content
- Server-side JavaBeans components that encapsulate behavior and state
- Static HTML, DHTML, XHTML, XML and similar pages.
- Client-side Java Applets, JavaBeans components, and arbitrary Java class files
- Java Runtime Environment(s) (downloadable via the Plugin) running in client(s)

JSP 1.1 supports portable packaging and deployment of Web Applications through the Servlet 2.2 specification. The JavaServer Pages specification inherits from the Servlet specification the concepts of Applications, ServletContexts, Sessions, Requests and Responses. See that specification for more details.

1.6 Application Model

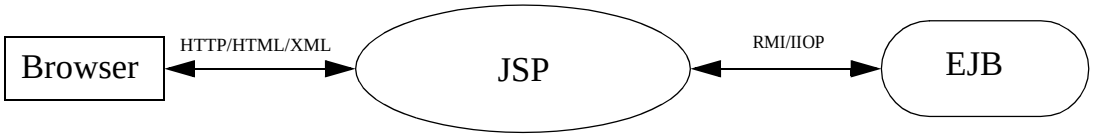
JSP pages can be used in combination with Servlets, HTTP, HTML, XML, Applets, JavaBeans components and Enterprise JavaBeans components to implement a broad variety of application architecture(s) or models.

1.6.1 Simple $2^{1/2}$-Tier Application



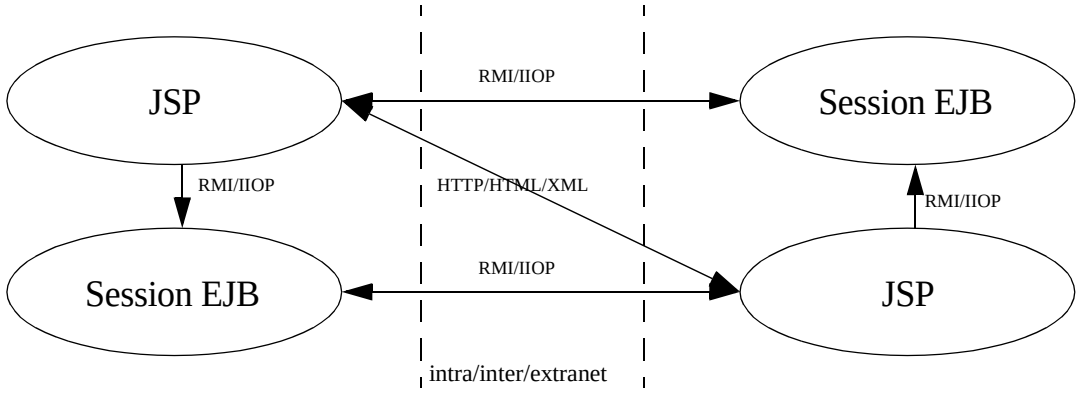
The simple 2-tier model (accessing a database in the example above) describes the cgi-bin replacement architecture that the Servlet model first enabled. This allows a JSP (or a Servlet) to directly access some external resource (such as a database or legacy application) to service a client's request. The advantage of such a scheme is that it is simple to program, and allows the page author to easily generate dynamic content based upon the request and state of the resource(s). However this architecture does not scale for a large number of simultaneous clients since each must establish/or share (ad-hoc) a (potentially scarce/expensive) connection to the resource(s) in question.

1.6.2 N-tier Application



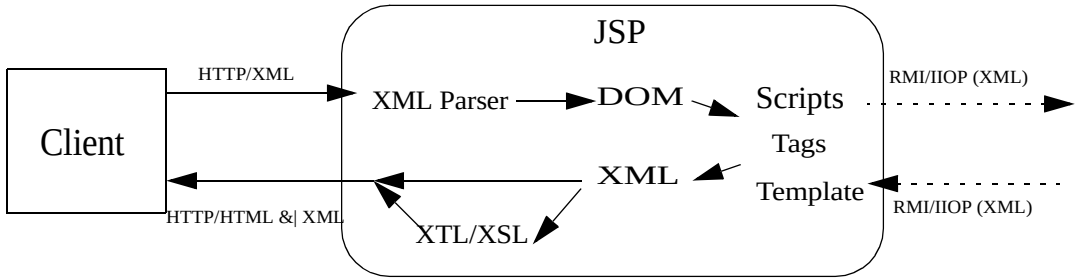
In this model the application is composed of (n>=3) tiers, where the middle tier, the JSP, interacts with the back end resources via an Enterprise JavaBeans component. The Enterprise JavaBeans server and the EJB provide managed access to resources thus addressing the performance issues. An EJB server will also support transactions and access to underlying security mechanisms to simplify programming. This is the programming model supported by the Java 2 Platform Enterprise Edition (J2EE).

1.6.3 Loosely Coupled Applications



In this model we have two loosely coupled applications (either on the same Intranet, or over an Extranet or the Internet). These applications may be peers, or act as client or server for the other. A common example of this is supply chain applications between vendor enterprises. In such situations it is important that each participant be isolated from changes in the implementation of it's dependents. In order to achieve this loose coupling the applications do not communicate using a fine grain imperative interface contract like those provided for by RMI/IIOP or Java IDL. The applications communicate with each other via HTTP, using either HTML or XML to/from a JSP page.

1.6.4 Using XML with JSP Technology

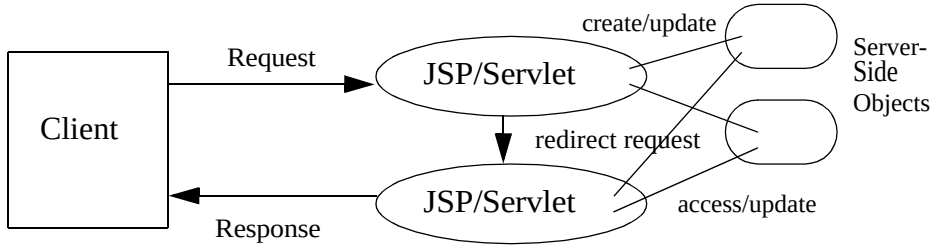


The JavaServer Pages technology is an ideal way to describe processing of XML input and output. Simple XML generation can be done by just writing the XML as static template portions within the JSP page. Dynamic generation will be done through JavaBeans components, Enterprise JavaBeans components, or via custom actions that generate XML. Similarly, input XML can be received from POST or QUERY arguments and then sent directly to JavaBeans components, Enterprise JavaBeans components, or custom actions, or manipulated via the scripting.

There are two attributes of the JSP technology that make it specially suited for describing XML processing. One is that XML fragments can be described directly in the JSP page either as templates for input into some XML-consuming component, or as templates for output to be extended with some other XML fragments. Another attribute is that the tag extension mechanism enables the creation of specific actions and directives that are targeted at useful XML manipulation operations.

Future versions of the JSP specification may include several standard actions that will support XML manipulation, including the transformation of the XML produced by the given JSP page using XTL/XSL.

1.6.5 Redirecting Requests



It is common that the data to be sent to the client varies significantly depending on properties of the client that are either directly encoded in the request object or can be discovered based on some user/client profile (e.g. stored in a login database). In this case it is very convenient to have the initial JSP page determine details about the request, perhaps create and/or update some server-side objects, and then, if necessary, redirect the request to a different JSP page.

This programming model is supported by the underlying Servlet APIs. The properties of the HTTP protocol are such that the redirect cannot be done if the response stream has started being sent back to the client; this characteristic makes the description of some common situations quite inconvenient. To address this, the JSP specification by default indicates buffering on the output stream. The JSP page can redirect the request at any point before flushing the output buffer.

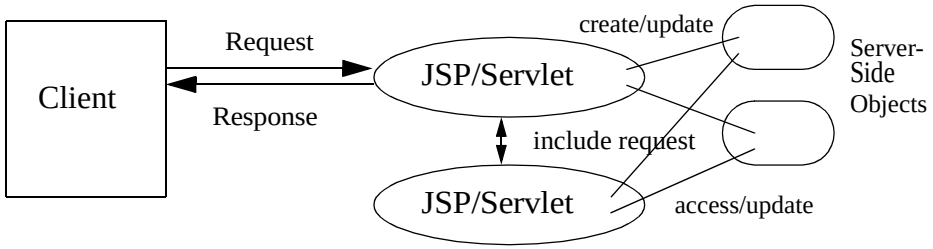
Buffering is also very convenient for error page handling, since that is done by redirecting the request.

Presentation JSP pages and Front JSP pages

In a slight variation of this model, the front component (a Servlet or a JSP) only creates and/or updates the server-side objects. In this organization, the *front* component does no presentation at all; instead all presentation is done by a *presentation* component. Although the front component could be written as a Servlet since it does no presentation, writing it as a JSP page enables the use of custom actions for the creation and update of the server-side objects. The presentation component will almost in all cases be a JSP page, and it will most likely access the server-side objects through custom actions[1].

1. Readers of the original JSP 0.92 draft will recognize the combination "*front component is servlet and presentation component is JSP*" as the model 2 mentioned in that draft.

1.6.6 Including Requests



Another useful application model involves request includes. In this model, the request reaches an initial JSP page. The page may start generating/composing some result but at some point it may want to dynamically include the contents of some other page. These contents may be static but may also be dynamically generated by some other JSP page, Servlet class, or some legacy mechanism like ASP.

Although in some cases this inclusion model is applicable to presentation-dependent contents, it is most often used in the context of a presentation-independent content, like when the data generated is actually XML (which may be converted later into some other format using, say, XSL).

CHAPTER 2

Standard Syntax and Semantics

This chapter describes the core syntax and semantics of the JavaServer Pages (JSP) 1.1 Specification, including the standard actions.

2.1 General Syntax Rules

The following general syntax rules apply to all elements in JSP pages.

2.1.1 Elements and Template Data

A JSP page has some *elements* and some *template data*. The elements are instances of some *element types* that are known to the JSP container; *template data* is everything else: i.e. anything that the JSP container does not understand.

The type of an element describes its syntax and its semantics. If the element has attributes, the type also describes the attribute names, their valid types, and their interpretation. If the element defines objects, the semantics includes what objects it defines and their types.

There are three types of elements: *directive elements*, *scripting elements*, and *action elements*; the corresponding syntax is described below. Template data is uninterpreted; it is usually passed through to the client, or to some processing component.

2.1.2 Element Syntax

Most of the JSP syntax is based on XML. Elements based on the XML syntax have either a start tag (including the element name) possibly with attributes, an optional body, and a matching end tag, or they have an empty tag possibly with attributes:

```
<mytag attr1=”attribute value” ...>
body
</mytag>
```

and

```
<mytab attr1=”attribute value” .../>
```

JSP tags are case-sensitive, as in XML and XHTML.

Scripting elements and directives are written using a syntax that is easier to author by hand. Elements using the alternative syntax are of the form <%.....%>.

All JSP pages have an equivalent valid XML document. A future JSP specification may require for JSP containers to accept JSP pages as well as their equivalent XML documents. Chapter 7 describes the XML equivalent syntax for the scripting elements and directives; these XML element types are not intended to be used within a JSP page but in the equivalent XML document.

2.1.3 Start and End Tags

Elements that have distinct start and end tags (with enclosed body) must start and end in the same file. You cannot begin a tag in one file and end it in another.

This applies also to elements in the alternate syntax. For example, a scriptlet has the syntax **<%** *scriptlet* **%>**. Both the opening **<%** characters and the closing **%>** characters must be in the same physical file.

2.1.4 Empty Elements

Following the XML specification, an element described using an empty tag is indistinguishable from one using a start tag, an empty body, and an end tag.

2.1.5 Attribute Values

Following the XML specification, attribute values always appear quoted. Both single and double quotes can be used. The entities ' and " are available to describe single and double quotes.

See also Section 2.12.1, “Request Time Attribute Values”.

2.1.6 White Space

In HTML and XML, white space is usually not significant, with some exceptions. One exception is that an XML file must start with the characters `<?xml`, with no leading whitespace characters.

This specification follows the whitespace behavior defined for XML, that is; all white space within the body text of a document is not significant, but is preserved.

For example, since directives generate no data and apply globally to the JSP page, the following input file is translated into the corresponding result file:

For this input,

	`<?xml version=”1.0” ?>`
This is the default value	`<%@ page buffer=”8kb” %>`
	`The rest of the document goes here`

The result is

	`<?xml version=”1.0” ?>`
note the empty line	
	`The rest of the document goes here`

`As another example`, for this input,

	`<% response.setContentType(“....”);`
note no white between the two elements	`whatever... %><?xml version=”1.0” ?>`
	`<%@ page buffer=”8kb” %>`
	`The rest of the document goes here`

The result is

no leading space	`<?xml version=”1.0” ?>`
note the empty line	
	`The rest of the document goes here`

2.2 Error Handling

There are two logical phases in the lifecycle/processing of a JavaServer Page source file:

- Translation (or compilation) from JSP page source into a JSP page implementation class file.
- Per client request processing by an instance of the JSP page implementation class.

Errors may occur at any point during processing of either phase. This section describes how such errors are treated by a compliant implementation.

2.2.1 Translation Time Processing Errors

The translation of a JSP page source into a corresponding JSP page implementation class using the Java technology by a JSP container can occur at any time between initial deployment of the JSP page into the runtime environment of a JSP container, and the receipt and processing of a client request for the target JSP page. If translation occurs prior to the JSP container receiving a client request for the target (untranslated) JSP page then error processing and notification is implementation dependent. Fatal translation failures shall result in subsequent client requests for the translation target to also be failed with the appropriate error; for HTTP protocols, error status code 500 (Server Error).

2.2.2 Client Request Time Processing Errors

During the processing of client requests, arbitrary runtime errors can occur in either the body of the JSP page implementation class or in some other code (Java or other implementation programming language) called from the body of the JSP page implementation class. Such errors are realized in the page implementation using the Java programming language exception mechanism to signal their occurrence to caller(s) of the offending behavior[1].

These exceptions may be caught and handled (as appropriate) in the body of the JSP page implementation class.

However, any uncaught exceptions thrown from the body of the JSP page implementation class result in the forwarding of the client request and uncaught exception to the `errorPage` URL specified by the offending JSP page (or the implementation default behavior, if none is specified).

1. Note that this is independent of scripting language; this requires that unhandled errors occurring in a scripting language environment used in a JSP container implementation to be signalled to the JSP page implementation class via the Java programming language exception mechanism.

The offending `java.lang.Throwable` describing the error that occurred is stored in the `javax.ServletRequest` instance for the client request using the `putAttribute()` method, using the name "`javax.servlet.jsp.jspException`". Names starting with the prefixes "`java`" and "`javax`" are reserved by the different specifications of the Java platform; the "`javax.servlet`" prefix is used by the Servlet and JSP specifications.

If the `errorPage` attribute of a `page` directive names a URL that refers to another JSP, and that JSP indicates that it is an error page (by setting the `page` directive's `isErrorPage` attribute to `true`) then the "`exception`" implicit scripting language variable of that page is initialized to the offending `Throwable` reference.

2.3 Comments

There are two types of comments in a JSP page: comments to the JSP page itself, documenting what the page is doing; and comments that are intended to appear in the generated document sent to the client.

Generating Comments in Output to Client

In order to generate comments that appear in the response output stream to the requesting client, the HTML and XML comment syntax is used, as follows:

```
<!-- comments ... -->
```

These comments are treated as uninterpreted template text by the JSP container. If the generated comment is to have dynamic data, this can be obtained through an expression syntax, as in:

```
<!-- comments <%= expression %> more comments ... -->
```

JSP Comments

A JSP comment is of the form

```
<%-- anything but a closing --%> ... --%>
```

The body of the content is ignored completely. Comments are useful for documentation but also to "comment out" some portions of a JSP page. Note that JSP comments do not nest.

Note that an alternative way to place a "comment" in JSP is to do so by using the comment mechanism of the scripting language. For example:

```
<% /** this is a comment ... **/ %>
```

2.4 Quoting and Escape Conventions

The following quoting conventions apply to JSP pages. Anything else is not processed.

Quoting in Scripting Elements

- A literal %> is quoted by %\>

Quoting in Template Text

- A literal <% is quoted by <\%

Quoting in Attributes

- A ‘ is quoted as \’
- A “ is quoted as \”
- A \ is quoted as \\
- A %> is quoted as %\>
- A <% is quoted as <\%

XML Representation

The quoting conventions are different to those of XML. Chapter 7 describes the details of the transformation.

2.5 Overview of Semantics

A JSP page describes how to create a *response* object from a *request* object for a given protocol, possibly creating and/or using some other objects. A JSP page is executed by a JSP container; requests sent to a JSP page are delivered by the JSP container to some JSP *page implementation* instance that is a subclass of Servlet (see Chapter 3).

2.5.1 Web Applications

A Web Application is a collection of resources that are available through some URLs. The resources include JSP pages, Java Servlet classes, static pages and other Java technology-based resources and classes to be used at the server-side as well as Java resources and classes (like Applets, JavaBeans components, and others) which are to be downloaded for use by the client. A Web Application is described in more detail in Chapter 9 of the Servlet 2.2 specification.

A Web Application contains a deployment descriptor `web.xml` that contains information about the JSP pages, Servlets, and other resources used in the Web Application. The Deployment Descriptor is described in detail in Chapter 13 of the Servlet 2.2 specification.

JSP 1.1 requires that all these resources are to be implicitly associated with and accessible through a unique `ServletContext` instance, which is available as the `application` implicit object (Section 2.8). The JSP specification inherits the notions of a Web Application from the Servlet 2.2 specification.

The application to which a JSP page belongs is reflected in the `application` object and has impact on the semantics of the following elements:

- The `include` directive (Section 2.7.6)
- The `jsp:include` action element (Section 2.13.4).
- The `jsp:forward` action (Section 2.13.5).

2.5.2 Relative URL Specifications within an Application

Elements may use *relative URL specifications*, which are called “URI paths” in the Servlet 2.1 specification. These paths are as in RFC 2396 specification; i.e. only the path part, no scheme nor authority. Some examples are:

```
“myErrorPage.jsp”
“/errorPages/SyntacticError.jsp”
“/templates/CopyrightTemplate.html”
```

When such a path starts with a “/”, it is to be interpreted by the application to which the JSP page belongs; i.e. its `ServletContext` object provides the base context URL. We call these paths “context-relative paths”.

When such a path does not start with a “/”, it is to be interpreted relative to the current JSP page: the current page is denoted by some path starting with “/” which is then modified by the new specification to produce a new path that starts with “/”; this final path is the one interpreted through the `ServletContext` object. We call these paths “page-relative paths”.

The JSP specification uniformly interprets all these paths in the context of the Web server where the JSP page is deployed; i.e. the specification goes through a map translation. The semantics applies to translation-time phase (i.e. include directives, Section 2.7.6), and to request-time phase (i.e. to include, Section 2.13.4, and forward, Section 2.13.5,actions).

If a specific tool can ascertain by some mechanism the status of the URL to resource maps at deployment time, the tool can take advantage of this information.

With the appropriate assertions, the translation phase might be performed before deploying the JSP page into the JSP container.

2.5.3 Web Containers and Web Components

A *JSP container* is a system-level entity that provides life-cycle management and runtime support for JSP pages and Servlet components. The term *Web Container* is synonymous to that a JSP container.

A Web component is either a Servlet or a JSP page. A web component may use the services of its container. The *servlet* element in a `web.xml` deployment descriptor is used to describe both types of web components; note that most JSP page components are defined implicitly in the deployment descriptor through the use of an implicit *.jsp* extension mapping.

2.5.4 JSP Pages

A JSP page implementation class defines a *_jspService()* method mapping from the *request* to the *response* object. Some details of this transformation are specific to the scripting language used; see Chapter 4. Most details are not language specific and are described in this chapter.

Most of the content of a JSP page is devoted to describing what data is written into the output stream of the response (usually sent back to the client). The description is based on a `JspWriter` object that is exposed through the implicit object *out* (see Section 2.8, “Implicit Objects”). Its value varies:

- Initially, *out* is a new `JspWriter` object. This object may be different from the stream object from *response.getWriter()*, and may be considered to be interposed on the latter in order to implement buffering (see Section 2.7.1, “The page Directive”). This is the *initial out object*. JSP page authors are prohibited from writing directly to either the `PrintWriter` or `OutputStream` associated with the `ServletResponse`.
- Within the body of some actions, *out* may be temporarily re-assigned to a different (nested) instance of `JspWriter` object. Whether this is or is not the case depends on the details of the actions semantics. Typically the content, or the results of processing the content, of these temporary streams is appended to the stream previously referred to by

out, and *out* is subsequently re-assigned to refer to that previous (nesting) stream. Such nested streams are always buffered, and require explicit flushing to a nesting stream or discarding of their contents.

- If the *initial out* `JspWriter` object is buffered, then depending upon the value of the `autoFlush` attribute of the `page` directive, the content of that buffer will either be automatically flushed out to the `ServletResponse` output stream to obviate overflow, or an exception shall be thrown to signal buffer overflow. If the *initial out* `JspWriter` is unbuffered, then content written to it will be passed directly through to the `ServletResponse` output stream.

A JSP page can also describe what should happen when some specific *events* occur. In JSP 1.1, the only events that can be described are initialization and destruction of the page; these are described using “well-known method names” in declaration elements (see page 73). Future specifications will likely define more events as well as a more structured mechanism for describing the actions to take.

2.6 Template Text Semantics

The semantics of *template (or uninterpreted) Text* is very simple: the template text is passed through to the current *out* `JspWriter` implicit object, after applying the substitutions of Section 2.4, “Quoting and Escape Conventions”.

2.7 Directives

Directives are messages to the JSP container. In JSP 1.1, directives have this syntax:

<%@ directive { attr=”value” } %>*

There may be optional white space after the “<%@” and before “%>”.

This syntax is easy to type and concise but it is not XML-compatible. Section 7 describes the mapping of directives into XML elements.

Directives do not produce any output into the current *out* stream.

The remainder of this section describes the standard directives that are available on all conforming JSP 1.1 implementations.

2.7.1 The `page` Directive

The `page` directive defines a number of page dependent attributes and communicates these to the JSP container.

A translation unit (JSP source file and any files included via the `include` directive) can contain more than one instance of the `page` directive, all the attributes will apply to the complete translation unit (i.e. page directives are position independent). However, there shall be only one occurrence of any attribute/value defined by this directive in a given translation unit with the exception of the "`import`" attribute; multiple uses of this attribute are cumulative (with ordered set union semantics). Other such multiple attribute/value (re)definitions result in a fatal translation error.

The attribute/value namespace is reserved for use by this, and subsequent, JSP specification(s).

Unrecognized attributes or values result in fatal translation errors.

Examples

The following directive provides some user-visible information on this JSP page:

```
<%@ page info="my latest JSP Example V1.1" %>
```

The following directive requests no buffering, indicates that the page is thread safe, and provides an error page.

```
<%@ page buffer="none" isThreadSafe="yes" errorPage="/oops.jsp" %>
```

The following directive indicates that the scripting language is based on Java, that the types declared in the package com.myco are directly available to the scripting code, and that a buffering of 16K should be used.

```
<%@ page language="java" import="com.myco.*" buffer="16k" %>
```

2.7.1.1 Syntax

```
<%@ page page_directive_attr_list %>

page_directive_attr_list ::= { language="scriptingLanguage" }
                             { extends="className"           }
                             { import="importList"           }
                             { session="true|false"          }
                             { buffer="none|sizekb"          }
                             { autoFlush="true|false"        }
                             { isThreadSafe="true|false"     }
                             { info="info_text"              }
                             { errorPage="error_url"         }
```

```
{ isErrorPage="true|false"      }
{ contentType="ctinfo"          }
```

The details of the attributes are as follows:

`language`

Defines the scripting language to be used in the scriptlets, expression scriptlets, and declarations within the body of the translation unit (the JSP page and any files included using the `include` directive below).

In JSP 1.1, the only defined and required scripting language value for this attribute is "`java`". This specification only describes the semantics of scripts for when the value of the language attribute is "java".

When "`java`" is the value of the scripting language, the Java Programming Language source code fragments used within the translation unit are required to conform to the Java Programming Language Specification in the way indicated in Chapter 4.

All scripting languages must provide some implicit objects that a JSP page author can use in declarations, scriptlets, and expressions. The specific objects that can be used are defined in Section 2.8, "Implicit Objects"."

All scripting languages must support the Java Runtime Environment (JRE). All scripting languages must expose the Java technology object model to the script environment, especially implicit variables, JavaBeans component properties, and public methods.

Future versions of the JSP specification may define additional values for the language attribute and all such values are reserved.

It is a fatal translation error for a directive with a non-`"java"` language attribute to appear after the first scripting element has been encountered.

`extends`

The value is a fully qualified Java programming language class name, that names the superclass of the class to which this JSP page is transformed (see Chapter 3).

This attribute should not be used without careful consideration as it restricts the ability of the JSP container to provide specialized superclasses that may improve on the quality of rendered service. See Section 5.8.1 for an alternate way to introduce objects into a JSP page that does not have this drawback.

`import`

An import attribute describes the types that are available to the scripting environment. The value is as in an import declaration in the Java programming language, i.e. a (comma separated) list of either a fully qualified Java programming language type name denoting that type, or of a package name followed by the ".*" string, denoting all the public types declared one in that package. The import list shall be imported by the translated JSP page implementation and are thus available to the scripting environment.

The default import list is `java.lang.*`, `javax.servlet.*`, `javax.servlet.jsp.*` and `javax.servlet.http.*`.

This value is currently only defined when the value of the `language` directive is "`java`".

`session`

Indicates that the page requires participation in an (http) session.

If "`true`" then the implicit script language variable named "`session`" of type `javax.servlet.http.HttpSession` references the current/new session for the page.

If "`false`" then the page does not participate in a session; the "`session`" implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and shall result in a fatal translation error.

Default is "`true`".

`buffer`

Specifies the buffering model for the initial "`out`" `JspWriter` to handle content output from the page.

If "`none`", then there is no buffering and all output is written directly through to the `ServletResponse PrintWriter`.

If a buffer size is specified (e.g `12kb`) then output is buffered with a buffer size not less than that specified.

Depending upon the value of the "`autoFlush`" attribute, the contents of this buffer is either automatically flushed, or an exception is raised, when overflow would occur.

The default is buffered with an implementation buffer size of not less than `8kb`.

`autoFlush`

Specifies whether the buffered output should be flushed automatically ("`true`" value) when the buffer is filled, or whether an exception should be raised ("`false`" value) to indicate buffer overflow.

The default is "`true`".

Note: it is illegal to set `autoFlush` to "`false`" when "`buffer=none`".

`isThreadSafe`	Indicates the level of thread safety implemented in the page. If "`false`" then the JSP container shall dispatch multiple outstanding client requests, one at a time, in the order they were received, to the page implementation for processing. If "`true`" then the JSP container may choose to dispatch multiple outstanding client requests to the page simultaneously. Page authors using "`true`" must ensure that they properly synchronize access to page shared state. Default is "`true`". Note that even if the *isThreadSafe* attribute is "`false`" the JSP page author must ensure that access to any shared objects shared in either the `ServletContext` or the `HttpSession` are properly synchronized. See Section 2.7.2
`info`	Defines an arbitrary string that is incorporated into the translated page, that can subsequently be obtained from the page's implementation of `Servlet.getServletInfo()` method.
`isErrorPage`	Indicates if the current JSP page is intended to be the URL target of another JSP page's `errorPage`. If "`true`", then the implicit script language variable "`exception`" is defined and its value is a reference to the offending `Throwable` from the source JSP page in error. If "`false`" then the "`exception`" implicit variable is unavailable, and any reference to it within the body of the JSP page is illegal and shall result in a fatal translation error. Default is "`false`"

`errorPage`	Defines a URL to a resource to which any Java programming language `Throwable` object(s) thrown but not caught by the page implementation are forwarded to for error processing. The provided URL spec is as in Section 2.5.2. The resource named has to be a JSP page in this version of the specification. If the URL names another JSP page then, when invoked that JSP page's `exception` implicit script variable shall contain a reference to the originating uncaught `Throwable`. The default URL is implementation dependent. Note the `Throwable` object is transferred by the throwing page implementation to the error page implementation by saving the object reference on the common `ServletRequest` object using the `setAttribute()` method, with a name of "`javax.servlet.jsp.jspException`". Note: if `autoFlush=true` then if the contents of the initial `JspWriter` has been flushed to the `ServletResponse` output stream then any subsequent attempt to dispatch an uncaught exception from the offending page to an `errorPage` may fail.
`contentType`	Defines the character encoding for the JSP page and for the response of the JSP page and the MIME type for the response of the JSP page. Values are either of the form "TYPE" or "TYPE; charset=CHARSET" with an optional white space after the ";". CHARSET, if present, must be the IANA value for a character encoding. TYPE is a MIME type, see the IANA registry for useful values. The default value for TYPE is "text/html"; the default value for the character encoding is ISO-8859-1. See Section 2.7.4 for complete details on character encodings.

2.7.2 Synchronization Issues

JSP Pages inherit the Servlet semantics as described in the Servlet 2.2 specification. In this section we briefly summarize the threading and distribution issues from that specification, as they apply to JSP pages.

A *Distributed Container* is one capable of distributing Web components that are tagged as *distributable* across different multiple Java Virtual Machines, perhaps running in different hosts. A *Distributable Application* is one tagged as such in its Web deployment descriptor. A *Distributable JSP Page* is one in a Distributable Application.

By default, there must be only one instance of a JSP page implementation class per JSP page definition in a container (Section 3.2 of Servlet 2.2 spec). By default, an instance of a Web Application must only be run on one Java Virtual Machine at any one time. This behavior can be overridden by declaring the Application to be Distributable (Chapter 9 of Servlet 2.2. specification).

A *single threaded JSP page* is one with a *false* value for its `isThreadSafe` attribute of a `page` directive. If `isThreadSafe="false"`, the JSP page implementation shall implement `javax.servlet.SingleThreadModel`, thus indicating that all requests dispatched to that instance shall be delivered serially to the `service()` method of the page implementation class (Section 3.3.3.1 of Servlet 2.2 spec).

However, some implementation(s) may additionally use a pool consisting of multiple page implementation class instances to do load balancing. Therefore, even when indicating that the page is not thread safe, a page author cannot assume that all requests mapped to a particular JSP page shall be delivered to the same instance of that page's implementation class. The consequence of this is that an author must assume that any mutable resources not private/unique to a particular page's instance may be accessed/updated simultaneously by 2 or more instances; thus any static field values, objects with `session` or `application` scope, or objects shared through some other (unspecified) mechanism by such instances must be accessed appropriately synchronized to avoid non-deterministic behaviors (Section 3.2.1 of Servlet 2.2 spec).

In the case of a Distributable JSP page, there is one instance of its JSP page implementation class per web component definition per Java Virtual Machine in a Distributed Container (Section 3.2 of Servlet 2.2 spec).

If multiple web component definitions in the deployment descriptor indicate the same JSP page, there will be multiple instances of the JSP page implementation class, with different initialization parameters (Section 3.3.1 of Servlet 2.2 spec).

There is only one instance of the `ServletContext` interface associated with each Web Application deployed into a Web Container. In cases where the container is distributed over many Java Virtual Machines, there is one instance per web application per Java Virtual Machine (Section 4.1 of Servlet 2.2 spec).

Context Attributes exist locally to the Java Virtual Machine in which they were created and placed. This prevents the ServletContext from being used as a distributed shared memory store. If information needs to be shared between servlets running in a distributed environment, that information should be placed in a session, a database, or in an Enterprise JavaBean (Section 4.3.1 of Servlet 2.2 spec).

Within an application that is marked as distributable, all requests that are part of a session can only be handled on a single Java Virtual Machine at any one time. In addition all objects placed into the session must implement the Serializable interface. The servlet container may throw an IllegalArgumentException if a non serializable object is placed into the session (Section 7.7.2 in Servlet 2.2 spec).

2.7.3 Specifying Content Types

A JSP page can use the `contentType` attribute of the page directive to indicate the content type of the response it provides to requests. Since this value is part of a directive, a given page will always provide the same content type. If a page determines that the response should be of a different content type, it should do so "early", determine what other JSP page or Servlet will handle this request and it should forward the request to the other JSP page or Servlet.

A registry of content types names is kept by IANA. See:

ftp://venera.isi.edu/in-notes/iana/assignments/media-types/media-types

2.7.4 Delivering Localized Content

The Java Platform support for localized content is based on a uniform representation of text internally as Unicode 2.0 (ISO010646) characters and the support for a number of character encodings to and from Unicode.

Any Java Virtual Machine (JVM) must support Unicode and Latin-1 encodings but most support many more. The character encodings supported by the JVM from Sun are described at:

http://java.sun.com/products/jdk/1.1/docs/guide/intl/encoding.doc.html

The JSP 1.1 specification assumes that JSP pages that will deliver content in a given character encoding will be written in that character encoding. In particular, the `contentType` attribute of the `page` directive describes both the character encoding of the JSP page and the character encoding of the resulting stream.

The valid names to describe the character encodings are those of IANA. They are described at:

ftp://venera.isi.edu/in-notes/iana/assignments/character-sets

The `contentType` attribute must only be used when the character encoding is organized such that ASCII characters stand for themselves, at least until the `contentType` attribute is found. The directive containing the `contentType` attribute should appear as early as possible in the JSP page.

The default character set encoding is ISO-8859-1 (also known as latin-1).

A JSP container may use some implementation-dependent heuristics and/or structure to determine what is the expected character encoding of a JSP page and then verify that `contentType` attribute is as expected.

A JSP container will raise a translation-time error if an unsupported character encoding is requested.

See Section B.1 for some implementation notes.

2.7.5 Including Data in JSP Pages

Including data is a significant part of the tasks in a JSP page. Accordingly, the JSP 1.1 specification has two include mechanisms suited to different tasks. A summary of their semantics is shown in TABLE 2-1.

TABLE 2-1 Summary of Include Mechanisms in JSP 1.1

Syntax	What	Phase	Spec	Object	Description	Section
<%@ include file=... %>	directive	translation-time	virtual	static	Content is parsed by JSP container.	2.7.6
<jsp:include page= />	action	request-time	virtual	static and dynamic	Content is not parsed; it is included in place.	2.13.4

The *Spec* column describes what type of specification is valid to appear in the given element. The JSP specification requires a relative URL spec as described in Section 2.5.2. The reference is resolved by the Web/Application server and its URL map is involved.

An include directive regards a resource like a JSP page as a static object; i.e. the bytes in the JSP page are included. An include action regards a resource like a JSP page as a dynamic object; i.e. the request is sent to that object and the result of processing it is included.

2.7.6 The `include` Directive

The `include` directive is used to substitute text and/or code at JSP page translation-time. The `<%@ include file="relativeURLspec" %>` directive inserts the text of the specified resource into the `.jsp` file. The included file is subject to the access control available to the JSP container. The `file` attribute is as in Section 2.5.2.

A JSP container can include a mechanism for being notified if an included file changes, so the container can recompile the JSP page. However, the JSP 1.1 specification does not have a way of directing the JSP container that included files have changed.

Examples

The following example requests the inclusion, at translation time, of a copyright file. The file may have elements which will be processed too.

```
<%@ include file=”copyright.html” %>
```

2.7.6.1 Syntax

```
<%@ include file="relativeURLspec" %>
```

2.7.7 The `taglib` Directive

The set of significant tags a JSP container interprets can be extended through a “tag library”.

The `taglib` directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI and associates a tag prefix that will distinguish usage of the actions in the library.

The URI identifying a tag library may be any valid URI as long as it can be used to uniquely identify the semantics of the tag library. A common mechanism is to encoding the version of a tag library into its URI.

If a JSP container implementation cannot locate (following the rules described in Section 5.3.1) a tag library description for a given URI, a fatal translation error shall result.

It is a fatal translation error for the `taglib` directive to appear after actions using the prefix introduced by the `taglib` directive.

See Chapter 5 for more specification details, and Section B.2 for an implementation note.

Examples

In the following example, a tag library is introduced and made available to this page using the `super` prefix; no other tags libraries should be introduced in this page using this prefix. In this particular case, we assume the tag library includes a `doMagic` element type, which is used within the page.

```
<%@ taglib uri="http://www.mycorp/supertags" prefix="super" />
...
<super:doMagic>
...
</super:doMagic>
```

2.7.7.1 Syntax

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

where the attributes are:

`uri`	Either an absolute URI or a relative URI specification to be interpreted as in Section 2.5.2 that uniquely identifies the tag library descriptor associated with this prefix. The URI is used to locate a description of the tag library as indicated in Chapter 5.
`tagPrefix`	Defines the *prefix* string in `<prefix>:<tagname>` that is used to distinguish a custom action, e.g <myPrefix:myTag> prefixes jsp:, jspx:, java:, javax:, servlet:, sun:, and sunw: are reserved. Empty prefixes are illegal in this version of the specification.

A fatal translation-time error will result if the JSP page translator encounters a tag with name *prefix:Name* using a prefix introduced using the taglib directive, and *Name* is not recognized by the corresponding tag library.

2.8 Implicit Objects

When you author JSP pages, you have access to certain implicit objects that are always available for use within scriptlets and expressions, without being declared first. All scripting languages are required to provide access to these objects.

Each implicit object has a class or interface type defined in a core Java technology or Java Servlet API package, as shown in TABLE 2-2.

TABLE 2-2 Implicit Objects Available in JSP Pages

Implicit Variable	Of Type	What It Represents	Scope
`request`	protocol dependent subtype of: `javax.servlet.ServletRequest` e.g: `javax.servlet.HttpServletRequest`	The request triggering the service invocation.	request
`response`	protocol dependent subtype of: `javax.servlet.ServletResponse` e.g: `javax.servlet.HttpServletResponse`	The response to the request.	page
`pageContext`	`javax.servlet.jsp.PageContext`	The page context for this JSP page.	page
`session`	`javax.servlet.http.HttpSession`	The session object created for the requesting client (if any). This variable is only valid for Http protocols.	session
`application`	`javax.servlet.ServletContext`	The servlet context obtained from the servlet configuration object (as in the call `getServletConfig().getContext()`)	application
`out`	`javax.servlet.jsp.JspWriter`	An object that writes into the output stream.	page
`config`	`javax.servlet.ServletConfig`	The `ServletConfig` for this JSP page	page
`page`	`java.lang.Object`	the instance of this page's implementation class processing the current request[1]	page

1. When the scripting language is "java" then "page" is a synonym for "this" in the body of the page.

In addition, in an error page, you can access the `exception` implicit object, described in TABLE 2-3.

TABLE 2-3 Implicit Objects Available in Error Pages

Implicit Variable	Of Type	What It Represents	scope
`exception`	`java.lang.Throwable`	The uncaught `Throwable` that resulted in the error page being invoked.	page

Object names with prefixes `jsp`, `_jsp`, `jspx` and `_jspx`, in any combination of upper and lower case, are reserved by the JSP specification.

See Section 5.8.1 for some non-normative conventions for the introduction of new implicit objects.

2.9 The pageContext Object

A `PageContext` provides an object that encapsulates implementation-dependent features and provides convenience methods. A JSP page implementation class can use a `PageContext` to run unmodified in any compliant JSP container while taking advantage of implementation-specific improvements like high performance `JspWriter`s. Generating such an implementation is not a requirement of JSP 1.1 compliant containers, although providing the `pageContext` implicit object is.

See Appendix 6 for more details.

2.10 Scripting Elements

The JSP 1.1 specification describes three scripting language elements—declarations, scriptlets, and expressions. A scripting language precisely defines the semantics for these elements but, informally, a declaration declares elements, a scriptlet is a statement fragment, and an expression is a complete language expression. The scripting language used in the current page is given by the value of the `language` directive (see Section 2.7.1, “The page Directive”). In JSP 1.1, the only value defined is `“java”`.

Each scripting element has a “<%”-based syntax as follows:

```
<%! this is a declaration %>
<% this is a scriptlet %>
<%= this is an expression %>
```

White space is optional after "<%!", "<%", and "<%=", and before "%>".

The equivalent XML elements for these scripting elements are described in Section 7.4.

2.10.1 Declarations

Declarations are used to declare variables and methods in the scripting language used in a JSP page. A declaration should be a complete declarative statement, or sequence thereof, according to the syntax of the scripting language specified.

Declarations do not produce any output into the current *out* stream.

Declarations are initialized when the JSP page is initialized and are made available to other declarations, scriptlets, and expressions.

Examples

For example, the first declaration below declares an integer, and initializes it to zero; while the second declaration declares a method.

```
<%! int i = 0; %>
<%! public String f(int i) { if (i<3) return("..."); ... } %>
```

Syntax

`<%!` *declaration(s)* `%>`

2.10.2 Scriptlets

Scriptlets can contain any code fragments that are valid for the scripting language specified in the `language` directive. Whether the code fragment is legal depends on the details of the scripting language; see Chapter 4.

Scriptlets are executed at request-processing time. Whether or not they produce any output into the *out* stream depends on the actual code in the scriptlet. Scriptlets can have side-effects, modifying the objects visible in them.

When all scriptlet fragments in a given translation unit are combined in the order they appear in the JSP page, they shall yield a valid statement or sequence thereof, in the specified scripting language.

If you want to use the %> character sequence as literal characters in a scriptlet, rather than to end the scriptlet, you can escape them by typing %\>.

Examples

Here is a simple example where the page changed dynamically depending on the time of day.

```
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) {%>
Good Morning
<% } else { %>
Good Afternoon
<% } %>
```

Syntax

<% *scriptlet* %>

2.10.3 Expressions

An expression element in a JSP page is a scripting language expression that is evaluated and the result is coerced to a `String` which is subsequently emitted into the current *out* `JspWriter` object.

If the result of the expression cannot be coerced to a `String` then either a translation time error shall occur, or, if the coercion cannot be detected during translation, a `ClassCastException` shall be raised at request time.

A scripting language may support side-effects in expressions. If so, they take effect when the expression is evaluated. Expressions are evaluated left-to-right in the JSP page. If the expressions appear in more than one run-time attribute, they are evaluated left-to-right in the tag. An expression might change the value of the *out* object, although this is not something to be done lightly.

The contents of an expression must be a complete expression in the scripting language in which they are written.

Expressions are evaluated at HTTP processing time. The value of an expression is converted to a String and inserted at the proper position in the `.jsp` file.

Examples

In the next example, the current date is inserted.

```
<%= (new java.util.Date()).toLocaleString() %>
```

Syntax

`<%=` *expression* `%>`

2.11 Actions

Actions may affect the current *out* stream and use, modify and/or create objects. Actions may, and often will, depend on the details of the specific request object received by the JSP page.

The JSP specification includes some action types that are *standard* and must be implemented by all conforming JSP containers. New action types are introduced using the `taglib` directive.

The syntax for action elements is based on XML; the only transformation needed is due to quoting conventions (see Section 7.5).

2.12 Tag Attribute Interpretation Semantics

Generally, all custom and standard action attributes and their values either remain uninterpreted by, or have well defined action-type specific semantics known to, a conforming JSP container. However there are two exceptions to this general rule: some attribute values represent request-time attribute values and are processed by a conforming JSP container, and the id and scope attributes have special interpretation.

2.12.1 Request Time Attribute Values

Action elements (both standard and custom) can define named attributes and associated values. Typically a JSP page treats these values as fixed and immutable but the JSP 1.1 provides a mechanism to describe a value that is computed at request time.

An attribute value of the form "<%= *scriptlet_expr* %>" or '<%= *scriptlet_expr* %>' denotes a *request-time attribute value*. The value denoted is that of the scriptlet expression involved. Request-time attribute values can only be used in actions. If there are more than one such attribute in a tag, the expressions are evaluated left-to-right.

Only attribute values can be denoted this way (e.g. the name of the attribute is always an explicit name), and the expression must appear by itself (e.g. multiple expressions, and mixing of expressions and string constants are not permitted; instead perform these operations within the expression).

The resulting value of the expression depends upon the expected type of the attribute's value. The type of an action element indicates the valid Java programming languag type for each attribute value; the default is `java.lang.String`.

By default, all attributes have page translation-time semantics. Attempting to specify a scriptlet expression as a value for an attribute that has page translation time semantics is illegal, and will result in a fatal translation error. The type of an action element indicates whether a given attribute will accept request-time attribute values.

Most attributes in the actions defined in the JSP 1.1 specification have page translation-time semantics.

The following attributes accept request-time attribute expressions:

- The `value` and `beanName` attributes of `jsp:setProperty` (2.13.2).
- The `page` attribute of `jsp:include` (2.13.4).
- The `page` attribute of `jsp:forward` (2.13.5).
- The `value` attribute of `jsp:param` (2.13.6).

2.12.2 The `id` Attribute

The `id="`*`name`*`"` attribute/value tuple in an element has special meaning to a JSP container, both at page translation time, and at client request processing time; in particular:

- the *`name`* must be unique within the translation unit, and identifies the particular element in which it appears to the JSP container and page.

  Duplicate `id`'s found in the same translation unit shall result in a fatal translation error.

- In addition, if the action type creates one or more object instance at client request processing time, one of these objects will usually be associated by the JSP container with the named value and can be accessed via that name in various contexts through the *`pagecontext`* object described later in this specification.

  Furthermore, the *`name`* is also used to expose a variable (name) in the page's scripting language environment. The scope of this scripting language dependent variable is dependent upon the scoping rules and capabilities of the actual scripting language used in the page. Note that this implies that the *`name`* value syntax shall comply with the variable naming syntax rules of the scripting language used in the page.

Chapter 4 provides details for the case where the language attribute is "java".

For example, the <jsp:usebean id="*name*" class="*className*" .../> action defined later herein uses this mechanism in order to, possibly instantiate, and subsequently expose the named JavaBeans component to a page at client request processing time.

For example:

```
<% { // introduce a new block %>
       ...
       <jsp:useBean id="customer" class="com.myco.Customer" />

       <%
       /*
        * the tag above creates or obtains the Customer Bean
        * reference, associates it with the name "customer" in the
        * PageContext, and declares a Java programming language
        * variable of the
        * same name initialized to the object reference in this
        * block's scope.
        */
       %>
       ...
       <%= customer.getName(); %>
       ...
<% } // close the  block %>

<%
// the variable customer is out of scope now but
// the object is still valid (and accessible via pageContext)
%>
```

2.12.3 The scope Attribute

The scope="page|request|session|application" attribute/value tuple is associated with, and modifies the behavior of the id attribute described above (it has both translation time and client request processing time semantics). In particular it describes the namespace, the implicit lifecycle of the object reference associated with the *name*, and the APIs used to access this association, as follows:

`page`	The named object is available from the `javax.servlet.jsp.PageContext` for the current page. This reference shall be discarded upon completion of the current request by the page body. It is illegal to change the instance object associated, such that its runtime type is a subset of the type of the current object previously associated.
`request`	The named object is available from the current page's `ServletRequest` object using the `getAttribute(`*`name`*`)` method. This reference shall be discarded upon completion of the current client request. It is illegal to change the value of an instance object so associated, such that its runtime type is a subset of the type(s) of the object previously so associated.
`session`	The named object is available from the current page's `HttpSession` object (which can in turn be obtained from the `ServletRequest` object) using the `getValue(`*`name`*`)` method. This reference shall be discarded upon invalidation of the current session. It is Illegal to change the value of an instance object so associated, such that its new runtime type is a subset of the type(s) of the object previously so associated. Note it is a fatal translation error to attempt to use `session` scope when the JSP page so attempting has declared, via the `<%@ page ... %>` directive (see later) that it does not participate in a `session`.
`application`	The named object is available from the current page's `ServletContext` object using the `getAttribute(`*`name`*`)` method. This reference shall be discarded upon reclamation of the `ServletContext`. It is Illegal to change the value of an instance object so associated, such that its new runtime type is a subset of the type(s) of the object previously so associated.

2.13 Standard Actions

The JSP 1.1 specification defines some standard action types that are always available, regardless of the version of the JSP container or Web server the developer uses. The standard action types are in addition to any custom types specific to a given JSP container implementation.

2.13.1 <jsp:useBean>

A `jsp:useBean` action associates an instance of a Java programming language object defined within a given `scope` available with a given `id` via a newly declared scripting variable of the same `id`.

The `jsp:useBean` action is quite flexible; its exact semantics depends on the attributes given. The basic semantic tries to find an existing object using `id` and `scope`; if it is not found it will attempt to create the object using the other attributes. It is also possible to use this action only to give a local name to an object define elsewhere, as in another JSP page or in a Servlet; this can be done by using the `type` attribute and not providing neither `class` nor `beanName` attributes.

At least one of `type` and `class` must be present, and it is not valid to provide both `class` and `beanName`. If `type` and `class` are present, `class` must be assignable (in the Java platform sense) to `type;` failure to do so is a translation-time error.

The attribute `beanName` is the name of a Bean, as specified in the JavaBeans specification for an argument to the instantiate() method in java.beans.Beans. I.e. it is of the form “a.b.c”, which may be either a class, or the name of a resource of the form “a/b/c.ser” that will be resolved in the current ClassLoader. If this is not true, a request-time exception, as indicated in the semantics of instantiate() will be raised. The value of this attribute can be a request-time attribute expression.

The actions performed are:

1. Attempt to locate an object based on the attribute values (`id, scope`). The inspection is done appropriately synchronized per scope namespace to avoid non-deterministic behavior.

2. Define a scripting language variable with the given `id` in the current lexical scope of the scripting language of the specified `type` (if given) or `class` (if type is not given).

3. If the object is found, the variable’s value is initialized with a reference to the located object, cast to the specified `type`. If the cast fails, a `java.lang.ClassCastException` shall occur. This completes the processing of this `useBean` action.

If the jsp:useBean element had a non-empty body it is ignored. This completes the processing of this useBean action.

4. If the object is not found in the specified scope and neither class nor beanName are given, a java.lang.InstantiationException shall occur. This completes the processing of this useBean action.

5. If the object is not found in the specified scope; and the class specified names a non-abstract class that defines a public no-args constructor, then that class is instantiated, and the new object reference is associated the with the scripting variable and with the specified name in the specified scope using the appropriate scope dependent association mechanism (see PageContext). After this, step 7 is performed.

   If the object is not found, and the class is either abstract, an interface, or no public no-args constructor is defined therein, then a java.lang.InstantiationException shall occur. This completes the processing of this useBean action.

6. If the object is not found in the specified scope; and beanName is given, then the method instantiate() of java.beans.Beans will be invoked with the ClassLoader of the Servlet object and the beanName as arguments. If the method succeeds, the new object reference is associated the with the scripting variable and with the specified name in the specified scope using the appropriate scope dependent association mechanism (see PageContext). After this, step 7 is performed.

7. If the jsp:useBean element has a non-empty body, the body is processed. The variable is initialized and available within the scope of the body. The text of the body is treated as elsewhere; if there is template text it will be passed through to the out stream; scriptlets and action tags will be evaluated.

   A common use of a non-empty body is to complete initializing the created instance; in that case the body will likely contain jsp:setProperty actions and scriptlets. This completes the processing of this useBean action.

Examples

In the following example, a Bean with name “connection” of type “com.myco.myapp.Connection” is available after this element; either because it was already created or because it is newly created.

```
<jsp:useBean id=″connection″ class=″com.myco.myapp.Connection″ />
```

In this next example, the timeout property is set to 33 if the Bean was instantiated.

```
<jsp:useBean id=″connection″ class=″com.myco.myapp.Connection″>
  <jsp:setProperty name=″connection″ property=″timeout″ value=″33″>
</jsp:useBean>
```

In our final example, the object should have been present in the session. If so, it is given the local name wombat with `WombatType`. A `ClassCastException` may be raised if the object is of the wrong class, and an `InstantiationException` may be raised if the object is not defined.

```
<jsp:useBean id="wombat" type="my.WombatType" scope="session"/>
```

2.13.1.1 Syntax

This action may or not have a body. If the action has no body, it is of the form:

```
<jsp:useBean id="name" scope="page|request|session|application"
typeSpec />


typeSpec ::=class="className" |
       class="className" type="typeName" |
       type="typeName" class="className" |
       beanName="beanName" type="typeName" |
       type="typeName" beanName="beanName" |
       type="typeName"
```

If the action has a body, it is of the form:

```
<jsp:useBean id="name" scope="page|request|session|application"
typeSpec >
       body
</jsp:useBean>
```

In this case, the body will be invoked if the Bean denoted by the action is created. Typically, the *body* will contain either scriptlets or `jsp:setProperty` tags that will be used to modify the newly created object, but the contents of the body is not restricted.

The `<jsp:useBean>` tag has the following attributes:

`id`	The name used to identify the object instance in the specified scope's namespace, ***and*** also the scripting variable name declared and initialized with that object reference. The name specified is case sensitive and shall conform to the current scripting language variable-naming conventions.
`scope`	The scope within which the reference is available. The default value is `page`. See the description of the `scope` attribute defined earlier herein

`class`	The fully qualified name of the class that defines the implementation of the object. The class name is case sensitive. If the class and beanName attributes are not specified the object must be present in the given scope.
`beanName`	The name of a Bean, as expected by the instantiate() method of the `java.beans.Beans` class. This attribute can accept a request-time attribute expression as a value.
`type`	If specified, it defines the type of the scripting variable defined. This allows the type of the scripting variable to be distinct from, but related to, that of the implementation class specified. The type is required to be either the class itself, a superclass of the class, or an interface implemented by the class specified. The object referenced is required to be of this type, otherwise a `java.lang.ClassCastException` shall occur at request time when the assignment of the object referenced to the scripting variable is attempted. If unspecified, the value is the same as the value of the `class` attribute.

2.13.2 <jsp:setProperty>

The `jsp:setProperty` action sets the value of properties in a Bean. The `name` attribute denotes an object that must be defined before this action appears.

There are two variants of the jsp:setProperty action. Both variants set the values of one or more properties in the Bean based on the type of the properties. The usual Bean introspection is done to discover what properties are present, and, for each, its name, whether they are simple or indexed, their type, and setter and getter methods.

Properties in a Bean can be set from one or more parameters in the request object, from a String constant, or from a computed request-time expression. Simple and indexed properties can be set using setProperty. The only types of properties that can be assigned to from String constants and request parameter values are those listed in TABLE 2-4; the conversion applied is that shown in the table. Request-time expressions can be assigned to properties of any type; no automatic conversions will be performed.

When assigning values to indexed properties the value must be an array; the rules described in the previous paragraph apply to the elements.

A conversion failure leads to an error; the error may be at translation or at request-time.

TABLE 2-4 Valid assignments in jsp:setProperty

Property Type	Conversion on String Value
boolean or Boolean	As indicated in java.lang.Boolean.valueOf(String)
byte or Byte	As indicated in java.lang.Byte.valueOf(String)
char or Character	As indicated in java.lang.Character.valueOf(String)[1]
double or Double	As indicated in java.lang.Double.valueOf(String)
int or Integer	As indicated in java.lang.Integer.valueOf(String)
float or Float	As indicated in java.lang.Float.valueOf(String)
long or Long	As indicated in java.lang.Long.valueOf(String)

1. This needs to be clarified before final, since the conversion is dependent on a character encoding.

Examples

The following two elements set a value from the request parameter values.

```
<jsp:setProperty name=”request” property=”*” />
<jsp:setProperty name=”user” property=”user” param=”username” />
```

The following element sets a property from a value

```
<jsp:setProperty name=”results” property=”row” value=”<%= i+1 %>” />
```

2.13.2.1 Syntax

```
<jsp:setProperty name="beanName" prop_expr />

prop_expr ::=    property="*"   |
                 property=”propertyName”|
                 property=”propertyName” param="parameterName"|
                 property=”propertyName” value=”propertyValue”

propertyValue ::= string
```

The value *propertyValue* can also be a request-time attribute value, as described in Section 2.12.1, “Request Time Attribute Values”.

```
propertyValue ::= expr_scriptlet[1]
```

1. See syntax for expression scriptlet “<%= ... %>”

The <jsp:setProperty> element has the following attributes:

name	The name of a Bean instance defined by a <jsp:useBean> element or some other element. The Bean instance must contain the property you want to set. The defining element (in JSP 1.1 only a <jsp:useBean> element) must appear before the <jsp:setProperty> element in the same file.
property	The name of the Bean property whose value you want to set If you set *propertyName* to * then the tag will iterate over the current ServletRequest parameters, matching parameter names and value type(s) to property names and setter method type(s), setting each matched property to the value of the matching parameter. If a parameter has a value of "", the corresponding property is not modified.
param	The name of the request parameter whose value you want to give to a Bean property. The name of the request parameter usually comes from a Web form If you omit param, the request parameter name is assumed to be the same as the Bean property name If the param is not set in the Request object, or if it has the value of "", the jsp:setProperty element has no effect (a noop). An action may not have both param and value attributes.
value	The value to assign to the given property. This attribute can accept a request-time attribute expression as a value. An action may not have both param and value attributes.

2.13.3 <jsp:getProperty>

An <jsp:getProperty> action places the value of a Bean instance property, converted to a String, into the implicit out object, from which you can display the value as output. The Bean instance must be defined as indicated in the *name* attribute before this point in the page (usually via a useBean action).

The conversion to String is done as in the println() methods, i.e. the toString() method of the object is used for Object instances, and the primitive types are converted directly.

If the object is not found, a request-time exception is raised.

Examples

```
<jsp:getProperty name="user" property="name" />
```

2.13.3.1 Syntax

```
<jsp:getProperty name="name" property="propertyName" />
```

The attributes are:

`name`	The name of the object instance from which the property is obtained.
`property`	Names the property to get.

2.13.4 <jsp:include>

A `<jsp:include .../>` element provides for the inclusion of static and dynamic resources in the same context as the current page. See TABLE 2-1 for a summary of include facilities.

The resource is specified using a relativeURLspec that is interpreted in the context of the Web server (i.e. it is mapped). See Section 2.5.2.

An included page only has access to the `JspWriter` object and it cannot set headers. This precludes invoking methods like `setCookie`(). A request-time Exception will be raised if this constraint is not satisfied. The constraint is equivalent to the one imposed on the `include`() method of the `RequestDispatcher` class.

A `jsp:include` action may have `jsp:param` subelements that can provide values for some parameters in the request to be used for the inclusion.

Request processing resumes in the calling JSP page, once the inclusion is completed.

If the page output is buffered then the buffer is flushed prior to the inclusion. See Section B.4 for an implementation note. See Section 5.4.5 for limitations on flushing when `out` is not the top-level `JspWriter`.

Examples

<jsp:include page="/templates/copyright.html"/>

2.13.4.1 Syntax

```
<jsp:include page="urlSpec" flush="true"/>
```

and

```
<jsp:include page="urlSpec" flush="true">
       { <jsp:param .... /> }*
</jsp:include>
```

The first syntax just does a request-time inclusion. In the second case, the values in the param subelements are used to augment the request for the purposes of the inclusion.

The valid attributes are:

`page`	The URL is a relative *urlSpec* is as in Section 2.5.2. Accepts a request-time attribute value (which must evaluate to a String that is a relative URL specification).
`flush`	Mandatory boolean attribute. If the value is "true", the buffer is flushed. A "false" value is not valid in JSP 1.1.

2.13.5 <jsp:forward>

A `<jsp:forward page="urlSpec" />` element allows the runtime dispatch of the current request to a static resource, a JSP pages or a Java Servlet class in the same context as the current page. A jsp:forward effectively terminates the execution of the current page. The relative *urlSpec* is as in Section 2.5.2.

The request object will be adjusted according to the value of the `page` attribute.

A `jsp:forward` action may have `jsp:param` subelements that can provide values for some parameters in the request to be used for the forwarding.

If the page output is buffered then the buffer is cleared prior to forwarding.

If the page output was unbuffered and anything has been written to it, an attempt to forward the request will result in an `IllegalStateException`.

Examples

The following element might be used to forward to a static page based on some dynamic condition.

```
<% String whereTo = "/templates/"+someValue; %>
<jsp:forward page='<%= whereTo %>' />
```

2.13.5.1 Syntax

```
<jsp:forward page="relativeURLspec" />
```

and

```
<jsp:forward page="urlSpec">
      { <jsp:param .... /> }*
</jsp:forward>
```

This tag allows the page author to cause the current request processing to be effected by the specified attributes as follows:

`page`	The URL is a relative *urlSpec* is as in Section 2.5.2. Accepts a request-time attribute value (which must evaluate to a String that is a relative URL specification).

2.13.6 <jsp:param>

The `jsp:param` element is used to provide key/value information. This element is used in the `jsp:include`, `jsp:forward` and `jsp:plugin` elements.

When doing `jsp:include` or `jsp:forward`, the included page or forwarded page will see the original request object, with the original parameters augmented with the new parameters, with new values taking precedence over existing values when applicable. The scope of the new parameters is the jsp:include or jsp:forward call; i.e. in the case of an `jsp:include` the new parameters (and values) will not apply after the include. This is the same behavior as in the `ServletRequest` `include` and `forward` methods (see Section 8.1.1 in the Servlet 2.2 specification).

For example, if the request has a parameter A=foo and a parameter A=bar is specified for forward, the forwarded request shall have A=bar,foo. Note that the new param has precedence.

2.13.6.1 Syntax

```
<jsp:param name="name" value="value" />
```

This action has two mandatory attributes: name and value. Name indicates the name of the parameter, value, which may be a request-time expression, indicates its value.

2.13.7 <jsp:plugin>

The plugin action enables a JSP page author to generate HTML that contains the appropriate client browser dependent constructs (OBJECT or EMBED) that will result in the download of the Java Plugin software (if required) and subsequent execution of the Applet or JavaBeans component specified therein.

The `<jsp:plugin>` tag is replaced by either an `<object>` or `<embed>` tag, as appropriate for the requesting user agent, and emitted into the output stream of the response. The attributes of the `<jsp:plugin>` tag provide configuration data for the presentation of the element, as indicated in the table below.

The `<jsp:param>` elements indicate the parameters to the Applet or JavaBeans component.

The `<jsp:fallback>` element indicates the content to be used by the client browser if the plugin cannot be started (either because OBJECT or EMBED is not supported by the client browser or due to some other problem). If the plugin can start but the Applet or JavaBeans component cannot be found or started, a plugin specific message will be presented to the user, most likely a popup window reporting a ClassNotFoundException

Examples

```
<jsp:plugin type=applet code=”Molecule.class” codebase=”/html” >
       <jsp:params>
              <jsp:param
                  name=”molecule”
                  value=”molecules/benzene.mol”/>
       </jsp:params>
       <jsp:fallback>
              <p> unable to start plugin </p>
       </jsp:fallback>
</jsp:plugin>
```

2.13.7.1 Syntax

```
<jsp:plugintype="bean|applet"
       code="objectCode"
       codebase="objectCodebase"
       { align="alignment"          }
       { archive="archiveList"      }
       { height="height"         }
       { hspace="hspace"         }
       { jreversion="jreversion" }
       { name="componentName"       }
       { vspace="vspace"            }
```

```
      { width="width"           }
      { nspluginurl="url"       }
      { iepluginurl="url"       } >
      { <jsp:params>
       { <jsp:param name="paramName" value="paramValue" /> }+
       </jsp:params> }
      { <jsp:fallback> arbitrary_text </jsp:fallback> }
</jsp:plugin>
```

`type`	Identifies the type of the component; a Bean, or an Applet.
`code`	As defined by HTML spec
`codebase`	As defined by HTML spec
`align`	As defined by HTML spec
`archive`	As defined by HTML spec
`height`	As defined by HTML spec
`hspace`	As defined by HTML spec
`jreversion`	Identifies the spec version number of the JRE the component requires in order to operate; the default is: “1.1”
`name`	As defined by HTML spec
`vspace`	As defined by HTML spec
`title`	As defined by the HTML spec
`width`	As defined by HTML spec
`nspluginurl`	URL where JRE plugin can be downloaded for Netscape Navigator, default is implementation defined.
`iepluginurl`	URL where JRE plugin can be downloaded for IE, default is implementation defined.

CHAPTER 3

The JSP Container

This chapter provides details on the contracts between a JSP container and a JSP page.

This chapter is independent on the Scripting Language used in the JSP page. Chapter 4 provides the details specific to when the `language` directive has “java” as its value.

This chapter also presents the precompilation protocol (see Section 3.4).

JSP page implementation classes should use the JspFactory and PageContext classes so they will take advantage of platform-specific implementations.

3.1 The JSP Page Model

As indicated in Section 1.4, “Overview of JSP Page Semantics”, a JSP page is executed by a JSP container, which is installed on a Web Server or Web enabled Application Server. The JSP container delivers requests from a client to a JSP page and responses from the JSP page to the client. The semantic model underlying JSP pages is that of a Servlet: a JSP page describes how to create a *response* object from a *request* object for a given protocol, possibly creating and/or using in the process some other objects. A JSP page may also indicate how some events (in JSP 1.1 only *init* and *destroy* events) are to be handled.

The Protocol Seen by the Web Server

The entity that processes *request* objects creating *response* objects should behave as if it were a Java technology-based class; in this specification we will simply assume it is so. This class must implement the Servlet protocol. It is the role of the JSP container to first locate the appropriate instance of such a class and then to deliver requests to it according to the Servlet protocol. As indicated elsewhere, a JSP container may need to create such a class dynamically from the JSP page source before delivering a request and response objects to it.

Thus, Servlet defines the contract between the JSP container and the JSP page implementation class. When the HTTP protocol is used, the contract is described by the HttpServlet class. Most pages use the HTTP protocol, but other protocols are allowed by this specification.

The Protocol Seen by the JSP Page Author

The JSP specification also defines the contract between the JSP container and the JSP page author. This is, what assumptions can an author make for the actions described in the JSP page.

The main portion of this contract is the *_jspService()* method that is generated automatically by the JSP container from the JSP page. The details of this contract is provided in Chapter 4.

The contract also describes how a JSP author can indicate that some actions must be taken when the *init()* and *destroy()* methods of the page implementation occur. In JSP 1.1 this is done by defining methods with name *jspInit()* and *jspDestroy()* in a declaration scripting element in the JSP page. Before the first time a request is delivered to a JSP page a *jspInit()* method, if present, will be called to prepare the page. Similarly, a JSP container can reclaim the resources used by a JSP page at any time that a request is not being serviced by the JSP page by invoking first its *jspDestroy()* method, if present.

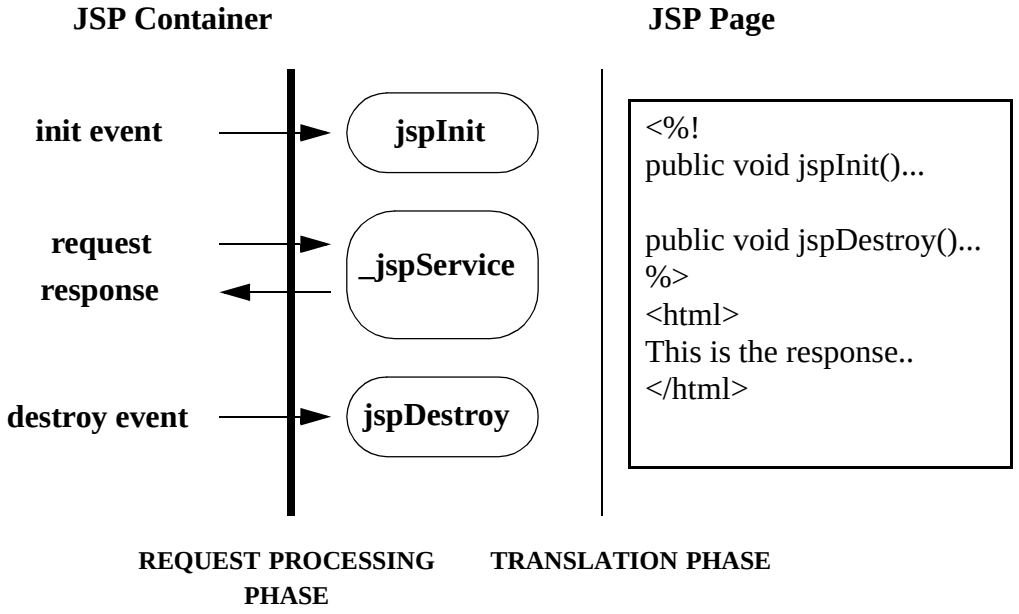
A JSP page author may **not** (re)define any of the Servlet methods through a declaration scripting element.

The JSP specification reserves the semantics of methods and variables starting with jsp, _jsp, jspx and _jspx, in any combination of upper and lower case.

The HttpJspPage Interface

The enforcement of the contract between the JSP container and the JSP page author is aided by requiring that the Servlet class corresponding to the JSP page must implement the `HttpJspPage` interface (or the `JspPage` interface if the protocol is not HTTP).

FIGURE 3-1 Contracts between a JSP Page and a JSP Container.



The involved contracts are shown in FIGURE 3-1. We now revisit this whole process in more detail.

3.2 JSP Page Implementation Class

The JSP container creates a JSP page implementation class for each JSP page. The name of the JSP page implementation class is implementation dependent.

The creation of the implementation class for a JSP page may be done solely by the JSP container, or it may involve a superclass provided by the JSP page author through the use of the *extends* attribute in the *jsp* directive. The extends mechanism is available for sophisticated users and it should be used with extreme care as it restricts what some of the decisions that a JSP container can take, e.g. to improve performance.

The JSP page implementation class will implement `Servlet` and the `Servlet` protocol will be used to deliver requests to the class.

A JSP page implementation class may depend on some support classes; if it does, and the JSP page implementation class is packaged into a WAR, those classes will have to be included in the packaged WAR so it will be portable across all JSP containers.

A JSP page author writes a JSP page expecting that the client and the server will communicate using a certain protocol. The JSP container must then guarantee that requests from and responses to the page use that protocol. Most JSP pages use HTTP, and their implementation classes must implement the `HttpJspPage` interface, which extends `JspPage`. If the protocol is not HTTP, then the class will implement an interface that extends `JspPage`.

3.2.1 API Contracts

The contract between the JSP container and a Java class implementing a JSP page corresponds to the `Servlet` interface; refer to the Servlet specification for details.

The contract between the JSP container and the JSP page author is described in TABLE 3-1. The responsibility for adhering to this contract rests only on the JSP container implementation if the JSP page does not use the *extends* attribute of the *jsp* directive; otherwise, the JSP page author guarantees that the superclass given in the extends attribute supports this contract.

TABLE 3-1 How the JSP Container Processes JSP Pages

Comments	Methods the JSP Container Invokes
Method is optionally defined in JSP page. Method is invoked when the JSP page is initialized. When method is called all the methods in servlet, including getServletConfig() are available	`void jspInit()`
Method is optionally defined in JSP page. Method is invoked before destroying the page.	`void jspDestroy()`
Method may **not** be defined in JSP page. The JSP container automatically generates this method, based on the contents of the JSP page. Method invoked at each client request.	`void _jspService(<ServletRequestSubtype>, <ServletResponseSubtype>) throws IOException, ServletException`

3.2.2 Request and Response Parameters

As shown in TABLE 3-1, the methods in the contract between the JSP container and the JSP page require request and response parameters.

The formal type of the request parameter (which this specification calls `<ServletRequestSubtype>`) is an interface that extends `javax.servlet.ServletRequest`. The interface must define a protocol-dependent request contract between the JSP container and the class that implements the JSP page.

Likewise, the formal type of the response parameter (which this specification calls `<ServletResponseSubtype>`) is an interface that extends `javax.servlet.ServletResponse`. The interface must define a protocol-dependent response contract between the JSP container and the class that implements the JSP page.

The request and response interfaces together describe a protocol-dependent contract between the JSP container and the class that implements the JSP page. The contract for HTTP is defined by the `javax.servlet.http.HttpServletRequest` and `javax.servlet.http.HttpServletResponse` interfaces.

The `JspPage` interface refers to these methods, but cannot describe syntactically the methods involving the `Servlet(Request,Response)` subtypes. However, interfaces for specific protocols that extend `JspPage` can, just as `HttpJspPage` describes them for the HTTP protocol.

Note – JSP containers that conform to this specification (in both JSP page implementation classes and JSP container runtime) must implement the `request` and `response` interfaces for the HTTP protocol as described in this section.

3.2.3 Omitting the `extends` Attribute

If the `extends` attribute of the `language` directive (see Section 2.7.1, “The page Directive”) in a JSP page is not used, the JSP container can generate any class that satisfies the contract described in TABLE 3-1 when it transforms the JSP page.

In the following code examples, CODE EXAMPLE 3-1 illustrates a generic HTTP superclass named `ExampleHttpSuper`. CODE EXAMPLE 3-2 shows a subclass named `_jsp1344` that extends `ExampleHttpSuper` and is the class generated from the JSP page. By using separate `_jsp1344` and `ExampleHttpSuper` classes, the JSP page translator needs not discover if the JSP page includes a declaration with `jspInit()` or `jspDestroy();` this simplifies very significantly the implementation.

CODE EXAMPLE 3-1 A Generic HTTP Superclass

```
imports javax.servlet.*;
imports javax.servlet.http.*;
imports javax.servlet.jsp.*;

/**
 * An example of a superclass for an HTTP JSP class
 */

abstract class ExampleHttpSuper implements HttpJspPage {
    private ServletConfig config;

final public void init(ServletConfig config) throws ServletException {
    this.config = config;
    jspInit();
}
final public ServletConfig getServletConfig() {
    return config;
}
// This one is not final so it can be overridden by a more precise method
public String getServletInfo() {
     return “A Superclass for an HTTP JSP”; // maybe better?
}
final public void destroy() {
     jspDestroy();
}

/**
* The entry point into service.
*/

final public void service(ServletRequest req, ServletResponse res)
     throws ServletException, IOException {

     // casting exceptions will be raised if an internal error.
     HttpServletRequest request = (HttpServletRequest) req;
     HttpServletResponse response = (HttpServletResponse) res;

     _jspService(request, resonse);


/**
* abstract method to be provided by the JSP processor in the subclass
* Must be defined in subclass.
*/

abstract public void _jspService(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException;
}
```

CODE EXAMPLE 3-2 The Java Class Generated From a JSP Page

```
imports javax.servlet.*;
imports javax.servlet.http.*;
imports javax.servlet.jsp.*;

/**
 * An example of a class generated for a JSP.
 *
 * The name of the class is unpredictable.
 * We are assuming that this is an HTTP JSP page (like almost all are)
 */

class _jsp1344 extends ExampleHttpSuper {

// Next code inserted directly via declarations.
// Any of the following pieces may or not be present
// if they are not defined here the superclass methods
// will be used.

    public void jspInit() {....}
    public void jspDestroy() {....}

    // The next method is generated automatically by the
    // JSP processor.
    // body of JSP page

    public void _jspService(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

        // initialization of the implicit variables

        HttpSession session = request.getSession();
        ServletContext context =
            getServletConfig().getServletContext();


        // for this example, we assume a buffered directive

        JSPBufferedWriter out = new
            JSPBufferedWriter(response.getWriter());

        // next is code from scriptlets, expressions, and static text.

    }

}
```

3.2.4 Using the `extends` Attribute

If the JSP page author uses `extends`, the generated class is identical to the one shown in CODE EXAMPLE 3-2, except that the class name is the one specified in the `extends` attribute.

The contract on the JSP page implementation class does not change. The JSP container should check (usually through reflection) that the provided superclass:

- Implements HttpJspPage if the protocol is HTTP, or JspPage otherwise.
- All of the methods in the `Servlet` interface are declared final.

Additionally, it is the responsibility of the JSP page author that the provided superclass satisfies:

- The `service()` method of the Servlet API invokes the `_jspService`() method.
- The `init(ServletConfig)` method stores the configuration, makes it available as `getServletConfig`, then invokes `jspInit`.
- The `destroy` method invokes `jspDestroy`.

A JSP container may give a fatal translation error if it detects that the provided superclass does not satisfy these requirements, but most JSP containers will not check them.

3.3 Buffering

The JSP container buffers data (if the jsp directive specifies it using the buffer attribute) as it is sent from the server to the client. Headers are not sent to the client until the first `flush` method is invoked. Therefore, none of the operations that rely on headers, such as the `setContentType`, `redirect`, or `error` methods are valid until the `flush` method is executed and the headers are sent.

JSP 1.1 includes a class that buffers and sends output, `javax.servlet.jsp.JspWriter`. The `JspWriter` class is used in the `_jspPageService` method as in the following example:

```
import javax.servlet.jsp.JspWriter;

static JspFactory _jspFactory = JspFactory.getDefaultFactory();

_jspService(<SRequest> request, <SResponse> response) {

    // initialization of implicit variables...
    PageContext pageContext = _jspFactory.createPageContext(
                              this,
                              request,
```

```
                                response,
                                false,
                                PageContext.DEFAULT_BUFFER,
                                false
                            );
    JSPWriter out = pageContext.getOut();
    // ....
    // .... the body goes here using "out"
    // ....
    out.flush();
}
```

You can find the complete listing of `javax.servlet.jsp.JspWriter` in Chapter 6.

With buffering turned on, you can still use a redirect method in a scriptlet in a .`jsp` file, by invoking `response.redirect(someURL)` directly.

3.4 Precompilation

A JSP page that is using the HTTP protocol will receive HTTP requests. JSP 1.1 compliant containers must support a simple *precompilation protocol*, as well as some basic *reserved parameter names*. Note that the precompilation protocol should not be confused with the notion of compiling a JSP page into a Servlet class (Appendix C).

3.4.1 Request Parameter Names

All request parameter names that start with the prefix "jsp" are reserved by the JSP specification and should not be used by any user or implementation except as indicated by the specification.

All JSPs pages should ignore (not depend on) any parameter that starts with "jsp_"

3.4.2 Precompilation Protocol

A request to a JSP page that has a request parameter with name "jsp_precompile" is a *precompilation request.* The "jsp_precompile" parameter may have no value, or may have values "true" or "false". In all cases, the request should not be delivered to the JSP page.

The intention of the precompilation request is that of a hint to the JSP container to *precompile* the JSP page into its JSP page implementation class. The hint is conveyed by given the parameter the value "true" or no value, but note that the request can be just ignored in all cases.

For example:

1. ?jsp_precompile

2. ?jsp_precompile="true"

3. ?jsp_precompile="false"

4. ?foobar="foobaz"&jsp_precompile="true"

5. ?foobar="foobaz"&jsp_precompile="false"

1, 2 and 4 are legal; the request will not be delivered to the page. 3 and 5 are legal; the request will be delivered to the page with no changes.

6. ?jsp_precompile="foo"

This is illegal and will generate an HTTP error; 500 (Server error).

CHAPTER 4

Scripting

This chapter describes the details of the Scripting Elements when the language directive value is “java”. The scripting language is based on the Java programming language (as specified by “The Java Language Specification”), but note that there is no valid JSP page, or a subset of a page, that is a valid Java program.

The details of the relationship between the scripting declarations, scriptlets, and scripting expressions and the Java programming language is explained in detail in the following sections. The description is in terms of the structure of the JSP page implementation class; recall that a JSP container need not necessarily generate the JSP page implementation class but it must behave as if one existed.

4.1 Overall Structure

Some details of what makes a JSP page legal are very specific to the scripting language used in the page. This is especially complex since scriptlets are just language fragments, not complete language statements.

Valid JSP Page

A JSP page is valid for a Java Platform if and only if the JSP page implementation class defined by TABLE 4-1 (after applying all include directives), together with any other classes defined by the JSP container, is a valid program for the given Java Platform.

Sun Microsystems reserves all names of the form `{_}jsp_*` and `{_}jspx_*`, in any combination of upper and lower case, for the JSP specification. Names of this form that are not defined in this specification are reserved by Sun for future expansion.

Implementation Flexibility

The transformations described in this Chapter need not be performed literally; an implementation may want to implement things differently to provide better performance, lower memory footprint, or other implementation attributes.

TABLE 4-1 Structure of the JavaProgramming Language Class

Optional imports clause as indicated via jsp directive	`import name1`
SuperClass is either selected by the JSP container or by the JSP author via jsp directive.	`class _jspXXX extends SuperClass`
Name of class (_jspXXX) is implementation dependent.	
Start of body of JSP page implementation class	`{`
(1) Declaration Section	`// declarations ...`

signature for generated method	`public void _jspService(<ServletRequestSubtype> request,`
	`  <ServletResponseSubtype> response)`
	`  throws ServletException, IOException {`

(2) Implicit Objects Section	`// code that defines and initializes request, response, page, pageContext etc.`
(3) Main Section	`// code that defines request/response mapping`
close of _jspService method	`}`
close of _jspXXX	`}`

4.2 Declarations Section

The declarations section correspond to the declaration elements.

The contents of this section is determined by concatenating all the declarations in the page in the order in which they appear.

4.3 Initialization Section

This section defines and initializes the implicit objects available to the JSP page. See Section 2.8, “Implicit Objects”.

4.4 Main Section

This section provides the main mapping between a request and a response object.

The contents of code segment 2 is determined from scriptlets, expressions, and the text body of the JSP page. These elements are processed sequentially; a translation for each one is determined as indicated below, and its translation is inserted into this section. The translation depends on the element type:

1. *Template data* is transformed into code that will place the template data into the stream currently named by the implicit variable `out`. All white space is preserved.

   Ignoring quotation issues and performance issues, this corresponds to a statement of the form:

   out.print(*template*);

2. A *scriptlet* is transformed into its Java statement fragment.

3. An *expression* is transformed into a Java statement to insert the value of the expression, converted to `java.lang.String` if needed, into the stream currently named by the implicit variable `out`. No additional newlines or space is included.

   Ignoring quotation and performance issues, this corresponds to a statement of the form:

   out.print(*expression*);

4. An action defining one or more objects is transformed into one or more variable declarations for these objects, together with code that initializes these variables. The visibility of these variables is affected by other constructs, like the scriptlets.

   The semantics of the action type determines the name of the variables (usually that of the `id` attribute, if present) and their type. The only standard action in the JSP 1.1 specification that defines objects is the jsp:usebean action; the name of the variable introduced is that of the `id` attribute, its type is that of the `class` attribute.

   Note that the value of the `scope` attribute does not affect the visibility of the variables within the generated program, it only affects where (and thus for how long) there will be additional references to the object denoted by the variable.

CHAPTER 5

Tag Extensions

This chapter describes the mechanisms for defining new actions in portable way and for introducing new actions into a JSP page.

5.1 Introduction

A Tag Library abstracts some functionality by defining a specialized (sub)language that enables a more natural use of that functionality within JSP pages. The actions introduced by the Tag Library can be used by the JSP page author in JSP pages explicitly, when authoring the page manually, or implicitly, when using an authoring tool. Tag Libraries are particularly useful to authoring tools because they make intent explicit and the parameters expressed in the action instance provide information to the tool.

Actions that are delivered as tag libraries are imported into a JSP page using the `taglib` directive, and can then be used in the page using the prefix given by the directive. An action can create new objects that can then be passed to other actions or can be manipulated programmatically through an scripting element in the JSP page.

Tag libraries are portable: they can be used in any legal JSP page regardless of the scripting language used in that page.

The tag extension mechanism includes information to:

- Execute a JSP page that uses the tag library.
- Author and modify a JSP page.
- Present the JSP page to the end user.

The JSP 1.1 specification mostly includes the first kind of information, plus basic information of the other two kinds. Later releases of the JSP specification may provide additional information; in the meanwhile, vendors may use vendor-specific information to address their needs.

A Tab Library is described via a Tag Library Descriptor, an XML document that is described further below.

No custom directives can be described using the JSP 1.1 specification.

5.1.1 Goals

The tag extension mechanism described in this chapter addresses the following goals:

Portable - An action described in a tag library must be usable in any JSP container.

Simple - Unsophisticated users must be able to understand and use this mechanism. We would like to make it very easy for vendors of functionality to expose it through actions.

Expressive - We want to enable a wide range of actions to be described in this mechanism, including:

- Nested actions.
- Scripting elements inside the body.
- Creation, use and updating of scripting variables.

Usable from different scripting languages - Although the JSP specification currently only defines the semantics for scripting based on the Java programming language, we want to leave open other scripting languages.

Building upon existing concepts and machinery- We do not want to reinvent machinery that exists elsewhere. Also, we want to avoid future conflicts whenever we can predict them.

5.1.2 Overview

The basic mechanism for defining the semantics of an action is that of a *tag handler.* A tag handler is a Java class that implements the `Tag` or `BodyTag` interfaces and that is the run-time representation of a custom action.

The JSP page implementation class instantiates (or reuses) a tag handler object for each action in the JSP page. This handler object is a Java object that implements the `javax.servlet.jsp.tagext.Tag` interface. The handler object is responsible for the interaction between the JSP page and additional server-side objects.

There are two main interfaces: `Tag` and `BodyTag`.

- `Tag` defines the basic methods that are needed in all tag handlers. These methods include setter methods to initialize a tag handler with context data and with the attribute values of the corresponding action, and the two methods: `doStartTag()` and `doEndTag()`.
- `BodyTag` provides two additional methods for when the tag handler wants to manipulate its body. The two new methods are `doInitBody()` and `doAfterBody()`.

The use of interfaces simplifies taking an existing Java object and making it a tag handler. There are also two support classes that can be used as base classes: `TagSupport` and `BodyTagSupport`.

Simple Actions

In many cases, the tag handler only needs to use the tag handler's method `doStartTag()` which is invoked when the start tag is encountered. This method needs to access the attributes of the tag and may also want to access information on the state of the JSP page; this information is passed to the Tag object before the call to `doStartTag()` through several setter method calls.

The `doEndTag()` is similar to `doStartTag()`, except that it is invoked when the end tag of the action is encountered. The result of the `doEndTag` invocation indicates whether the remaining of the page is to be evaluated or not.

A particularly simple and frequent action is one that has an empty body (no text between the start and the end tag). The Tag Library Descriptor can be used to indicate that the tag is always intended to be empty; this leads to better error checking at translation time and to better code quality in the JSP page implementation class.

Actions with Body

Recall that in general, the body of an action may contain other custom and core actions and scripting elements, as well as uninterpreted template text.

In some cases, an action is only interested in "passing through" the content of the body. This can be done using the simple Tag interface by using a special return value in `doStartTag()`.

If an action element can have a non-empty body and is interested in the content of that body, the methods `doInitBody()` and `doAfterBody()`, defined in the BodyTag interface are involved.

The control of the evaluation is actually done based on the result of method invocations as follows. The `doStartTag()` method is always invoked first and returns an `int` value that indicates if the body of the action should be evaluated or not. If so (EVAL_BODY_TAG return), a nested stream of type `BodyContent` is created and it is passed to the Body`Tag` object through `setBodyContent`. Then `doInitBody` is invoked. Next the body is evaluated, with the result going into the newly created `BodyContent` object. Finally the `doAfterBody()` method of the tag handler object is invoked.

If the invocation to `doStartTag()` returned SKIP_BODY, the body is not evaluated at all.

The `doBody()` methods may use the `BodyContent` object as it sees fit. For example, it may convert it into a String and use it as an argument. Or it may do some filter action to it before passing it through to the out stream. Or something else.

A `doAfterBody()` invocation returns an indication of whether further reevaluations of the body text should be done by the JSP page; as in the case of `doStartTag()`, if EVAL_BODY_TAG is returned, the body is reevaluated, while a return value of SKIP_BODY will stop reevaluations. Note that, since server-side objects (accessible via pageContext, or through nested handlers) may have changed, each evaluation may produce very different content to be added to the `BodyContent` object.

Cooperating Actions

Often the best way to describe some functionality is through several cooperating actions. For example, an action may be used to describe information that leads to the creation of some server-side object, while another action may use that object elsewhere in the page. One way for these actions to cooperate is explicitly, via using scripting variables: one action creates an object and gives it a name, the other refers to it through this name. Scripting variables are discussed briefly below.

Two actions can also cooperate implicitly using different conventions. For example, perhaps the last action applies, or perhaps there is only one action of a given type per JSP page. A more flexible and very convenient mechanism for action cooperation is using the nesting structure to describe scoping. Each tag handler is told of its parent tag handler (if any) using a setter method; the `findAncestorWithClass` static method in `TagSupport` can then be used to locate a tag handler with some given properties.

Actions Defining Scripting Variables

A custom action may create some server-side objects and make them available to the scripting elements by creating or updating some scripting variables. The specific variables thus effected may vary with the action instance. The details of this are described through subclasses of `javax.servlet.jsp.tagext.TagExtraInfo` which are used at JSP page translation time. The `TagExtraInfo` class provides methods that will indicate what are the names and types of the scripting variables that will be assigned objects (at request time) by the action. These methods are passed a TagData instance that describes the attributes of a given action. The responsibility of the tag library author is to faithfully indicate this information in the TagExtraInfo class; the corresponding Tag object must add the objects to the pageContext object. It is the responsibility of the JSP page translator to automatically supply all the required code to do the “synchronization” between the pageObject values and the scripting variables.

5.1.3 Examples

This section outlines some simple, and common, uses of the tag extension mechanism. See Appendix A for more details on these examples and for additional examples of custom actions.

5.1.3.1 Call Functionality, no Body

This is probably the simplest example: just collect attributes and call into some action. The only action involved is `foo`, and in this case it should have no body. I.e something like:

```
<x:foo att1="..." att2="..." att3="..." />
```

In this case we would define a `FooTag` tag handler that extends `TagSupport` only redefining `doStartTag`. The `doStartTag` method would take the attribute information, perhaps interact with the `PageContext` data and invoke into the appropriate functionality.

The entry for this tag in the Tag Library Descriptor should indicate that the action must have no body; no `TagExtraInfo` class is needed.

5.1.3.2 Call Functionality, No Body, Define Object

In a simple variation of the previous example the action defines an object.

```
<x:bar id="mybar" att1="..." att2="..." att3="..." />
```

After this, an object with name `mybar` is available to the scripting language.

The semantics of `doStartTag()` invocation is as before except that additionally it should insert the appropriate object for the “mybar” entry into the pageContext.

The Tag Library Descriptor entry for this action needs to mention a `TagExtraInfo` class that will be used to determine the scripting variables that will be created by the action; in this case “mybar” (note that id must be a translation-time attribute).

5.1.3.3 Call Functionality, Define Object by Scope

In some cases, the previous example can also be described without using a TagExtraInfo by having the bar action enclose the actions that would use the created object. In this case, the defining action needs not indicate any id attribute but it must have a body:

```
<x:bar att1="..." att2="..." att3="...">
  BODY
</x:bar>
```

The nesting actions will invoke `findAncestorWithClass` to locate the bar handler object.

5.1.3.4 Template Mechanisms

There are a number of “template” mechanisms in server-side frameworks. The simplest of these mechanisms will take a “token” and replace it by some fixed replacement text (that can be changed easily); more sophisticated mechanisms compute the replacement text, and can

pass arguments for that computation. These mechanisms can be subsumed directly into an empty-bodied action invocation, perhaps using attributes to describe the template and/or the arguments for the computation.

5.1.3.5 An HTML quoting action

The final example is an action that takes its body and performs HTML quoting. In this case, the `doStartTag()` method will save away the value of the out implicit object and request the evaluation of the body. The `doAfterBody()` method will take the nested stream, perform the quoting, and send it down to the saved out stream.

5.1.3.6 A useBean as in the JSP 0.92 specification

The 0.92 public draft of the JSP specification included a version of a `useBean` action with a variation: if the Bean created included a `processRequest(ServletRequest)` method then the method would be invoked. Observant readers will notice that a `processRequest()` is a special case of a `doStartTag()` as the request object is one of the objects available in `pageContext`.

5.2 Tag Library

A *Tag Library* is a collection of actions that encapsulate some functionality to be used from within a JSP page. A Tag library is made available to a JSP page via a `taglib` directive that identifies the Tag Library via a URI (Universal Resource Identifier).

The URI identifying a tag library may be any valid URI as long as it can be used to uniquely identify the semantics of the tag library. A common mechanism is to encoding the version of a tag library into its URI.

The URI identifying the tag library is associated with a *Tag Library Description* (TLD) file and with *tag handler* classes as indicated in Section 5.3 below.

5.2.1 Packaged Tag Libraries

JSP page authoring tools are required to accept a Tag Library that is packaged as a JAR file. When packaged sot he JAR file must have a tag library descriptor file named META-INF/taglib.tld.

5.2.2 Location of Java Classes

The request-time Tag handler classes and the translation-time TagExtraInfo classes are just Java classes. In a Web Application they must reside in the standard locations for Java classes: either in a JAR file in the WEB-INF/lib directory or in a directory in the WEB-INF/classes directory.

The previous rule indicates that a JAR containing a packaged tag libraries can be dropped into the WEB-INF/lib directory to make its classes available at request time (and also at translation time, see Section 5.3.2). The mapping between the URI and the TLD is explained further below.

5.2.3 Tag Library directive

The `taglib` directive in a JSP page declares that the page uses a tag library, uniquely identifies the tag library using a URI and associates a tag prefix that will distinguish usage of the actions in the library.

If a JSP container implementation cannot locate (following the rules described in Section 5.3.1) a tag library description for a given URI, a fatal translation error shall result.

It is a fatal translation error for the `taglib` directive to appear after actions using the prefix introduced by the `taglib` directive.

5.3 Tag Library Descriptor

The Tag Library Descriptor (TLD) is an XML document that describes a tag library. The TLD for a tag library is used by a JSP container to interpret pages that include `taglib` directives referring to that tag library. The TLD is also used by JSP page authoring tools that will generate JSP pages that use a library, and by authors who do the same manually.

The TLD includes documentation on the library as a whole and on its individual tags, version information on the JSP container and on the tag library, and information on each of the actions defined in the tag library.

Each action in the library is described by giving its name, the class for its tag handler, optional information on a `TagExtraInfo` class, and information on all the attributes of the action. Each valid attribute is mentioned explicitly, with indication on whether it is mandatory or not, whether it can accept request-time expressions, and additional information.

A TLD file is useful as a descriptive mechanism for providing information on a Tag Library. It has the advantage that it can be read by tools without having to instantiate objects or load classes. The approach we follow conforms to the conventions used in other J2EE technologies.

The DTD to the tag library descriptor is organized so that interesting elements have an optional ID attribute. This attribute can be used by other documents, like vendor-specific documents, to provide annotations of the TLD information. An alternative approach, based on XML name spaces have some interesting properties but it was not pursued in part for consistency with the rest of the J2EE descriptors.

The official DTD is described at "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"

5.3.1 Locating a Tag Library Descriptor

The URI describing a Tag Library is mapped to a Tag Library Descriptor file though two mechanisms: a map in `web.xml` described using the `taglib` element, and a default mapping.

5.3.1.1 Taglib map in web.xml

The map in `web.xml` is described using the `taglib` element of the Web Application Deployment descriptor in `WEB-INF/web.xml`, as described in the Servlet 2.2 spec and in “http://java.sun.com/j2ee/dtds/web-app_2_2.dtd”.

A `taglib` element has two subelements: `taglib-uri` and `taglib-location`.

taglib

A taglib is a subelement of web-app:

```
<!ELEMENT web-app .... taglib* .... >
```

The taglib element provides information on a tag library that is used by a JSP page within the Web Application.

A taglib element has two subelements and one attribute:

```
<!ELEMENT taglib ( taglib-uri, taglib-location ) >
<!ATTLIST taglib id ID #IMPLIED>
```

taglib-uri

A `taglib-uri` element describes a URI identifying a Tag Library used in the Web Application.

```
<!ELEMENT taglib-uri (#PCDATA) >
PCDATA ::= a URI spec. It may be either an absolute URI
specification, or a relative URI as in Section 2.5.2.
```

taglib-location

A `taglib-location` contains the location (as a resource) where to find the Tag Library Description File for this Tag Library.

```
<!ELEMENT taglib-location (#PCDATA) >
PCDATA ::= a resource location, as indicated in Section 2.5.2,
where to find the Tag Library Descriptor file.
```

Example

The use of relative URI specifications enables very short names in the `taglib` directive. For example:

```
<%@ taglib uri="/myPRlibrary" prefix="x" %>
```

and then

```
<taglib>
  <taglib-uri>/myPRlibrary</taglib-uri>
  <taglib-location>/WEB-INF/tlds/PRlibrary_1_4.tld</taglib-uri>
</taglib>
```

5.3.1.2 Default location

If there is no `taglib-uri` subelement that matches the URI used in a `taglib` directive, the tag library descriptor will be searched in the location indicated by the URI itself.

This rule only applies to URIs that are relative URI specifications (as in Section 2.5.2).

Example

This rule allows a `taglib` directive to refer directly to the TLD. This arrangement is very convenient for quick development at the expense of less flexibility and accountability. For example in the case above, it enables:

```
<%@ taglib uri="/tlds/PRlibrary_1_4.tld" prefix="x" %>
```

5.3.2 Translation-Time Class Loader

The set of classes available at translation time is the same as available at runtime: the classes in the underlying Java platform, those in the JSP container, and those in the class files in `WEB-INF/classes`, in the JAR files in `WEB-INF/lib`, and, indirectly through the use of the `class-path` attribute in the `META-INF/MANIFEST` file of these JAR files.

5.3.3 Assembling a Web Application

As part of the process of assembling a Web Application together, the Application Assembler will create a `WEB-INF/` directory, with appropriate `lib/` and `classes/` subdirectories, place JSP pages, Servlet classes, auxiliary classes, and tag libraries in the proper places and then create a `WEB-INF/web.xml` that ties everything together.

Tag libraries that have been delivered in the standard format can be dropped directly into `WEB-INF/lib`. The assembler may create `taglib` entries in `web.xml` for each of the libraries that are to be used.

Part of the assembly (and later the deployment) may create and/or change information that customizes a tag library; see Section 5.8.3.

5.3.4 Well-Known URIs

A JSP container may "know of" some specific URIs and may provide alternate implementations for the tag libraries described by these URIs, but the user must see the same behavior as that described by the, required, portable tag library description described by the URI.

A JSP container must always use the mapping specified for a URI in the web.xml deployment descriptor if present. If the deployer wants to use the platform-specific implementation of the well-known URI, the mapping for that URI should be removed at deployment time.

If there is no mapping for a given URI and the URI is not well-known to the JSP container, a translation-time error will occur.

There is no guarantee that this “well-known URI” mechanism will be preserved in later releases of the JSP specification. As experience accumulates on how to use tag extensions, the JSP specification may incorporate new functionality that will make the “well-known URI” mechanism unnecessary; at that point it may be removed.

5.3.5 The Tag Library Descriptor Format

This section describes the DTD for the Tag Library Descriptor. This is the same DTD as "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd", except for some formatting changes to extract comments and make them more readable.

Notation

```
<!NOTATION WEB-JSPTAGLIB.1_1 PUBLIC “-//Sun Microsystems, Inc.//DTD
JSP Tag Library 1.1//EN”>
```

taglib

The **taglib** element is the document root. A taglib has two attributes.

```
<!ATTLIST taglib
     id
         ID
         #IMPLIED
     xmlns
         CDATA
         #FIXED
         "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"
>
```

A taglib element also has several subelements that define:

tlibversion the version of the tag library implementation

jspversion the version of JSP specification the tag library depends upon

shortname a simple default short name that could be used by a JSP page authoring tool to create names with a mnemonic value; for example, the it may be used as the preferred prefix value in taglib directives.

uri a uri uniquely identifying this taglib.

info a string describing the “use” of this taglib

```
<!ELEMENT taglib
       (tlibversion, jspversion?,
        shortname, uri?, info?,
        tag+) >
```

tlibversion

Describes this version (number) of the taglibrary.

The syntax is:

```
<!ELEMENT tlibversion (#PCDATA) >
#PCDATA ::= [0-9]*{ “.”[0-9] }0..3
```

jspversion

Describes the JSP specification version (number) this taglibrary requires in order to function. The default is 1.1

The syntax is:

```
<!ELEMENT jspversion  (#PCDATA) >
#PCDATA ::= [0-9]*{ “.”[0-9] }0..3.
```

shortname

Defines a simple default short name that could be used by a JSP page authoring tool to create names with a mnemonic value; for example, the it may be used as the preferred prefix value in taglib directives and/or to create prefixes for IDs . Do not use white space, and do not start with digits or underscore.

The syntax is

```
<!ELEMENT shortname      (#PCDATA) >
#PCDATA ::= NMTOKEN
```

uri

Defines a public URI that uniquely identifies this version of the tag library. It is recommended that the URI identifying a tag library is actually a URL to the tag library descriptor for this specific version of the tag library.

```
<!ELEMENT uri         (#PCDATA) >
```

info

Defines an arbitrary text string describing the tag library.

```
<!ELEMENT info        (#PCDATA) >
```

tag

The **tag** defines an action in this tag library. It has one attribute:

```
<!ATTLIST tag id ID #IMPLIED >
```

The **tag** may have several subelements defining:

name	the unique action name
tagclass	the tag handler class implementing `javax.servlet.jsp.tagext.Tag`
teiclass	an optional subclass of `javax.servlet.jsp.tagext.TagExtraInfo`
bodycontent	the body content type
info	optional tag-specific information
attribute	all attributes of this action

The element syntax is as follows:

```
<!ELEMENT tag
       (name, tagclass, teiclass?,
       bodycontent?, info?, attribute*) >
```

tagclass

Defines the tag handler class implementing the `javax.serlvet.jsp.tagext.Tag` interface. This element is required.

The syntax is:

```
<!ELEMENT tagclass (#PCDATA) >
#PCDATA ::= fully qualified Java class name.
```

teiclass

Defines the subclass of `javax.servlet.jsp.tagext.TagExtraInfo` for this tag. This element is optional.

The syntax is:

```
<!ELEMENT teiclass (#PCDATA) >
 #PCDATA ::= fully qualified Java class name
```

bodycontent

Provides a hint as to the content of the body of this action. Primarily intended for use by page composition tools.

There are currently three values specified:

tagdependent The body of the action is interpreted by the tag handler itself, and is most likely in a different “language”, e.g. embedded SQL statements. The body of the action may be empty.

JSP The body of the action contains elements using the JSP syntax. The body of the action may be empty.

empty The body must be empty

The default value is “JSP”.

The syntax is:

```
<!ELEMENT bodycontent (#PCDATA) >
#PCDATA ::=  tagdependent | JSP | empty.
```

attribute

Provides information on an attribute of this action. Attribute defines an id attribute for external linkage.

```
<!ATTLIST attribute id     ID#IMPLIED>
```

The subelements of attribute are of the form:

name the attributes name (required)

required if the attribute is required or optional (optional)

rtexprvalue if the attributes value may be dynamically calculated at runtime by a scriptlet expression (optional)

The syntax is:

```
<!ELEMENT attribute
       (name, required?,
       rtexprvalue?) >
```

name

Defines the canonical name of a tag or attribute being defined

The syntax is:

```
<!ELEMENT name        (#PCDATA) >
#PCDATA ::= NMTOKEN
```

required

Defines if the nesting attribute is required or optional.

The syntax is:

```
<!ELEMENT required    (#PCDATA) >
#PCDATA ::= true | false | yes | no
```

If not present then the default is “false”, i.e the attribute is optional.

rtexprvalue

Defines if the nesting attribute can have scriptlet expressions as a value, i.e the value of the attribute may be dynamically calculated at request time, as opposed to a static value determined at translation time.

The syntax is:

```
<!ELEMENT rtexprvalue (#PCDATA) >
#PCDATA ::= true | false | yes | no
```

If not present then the default is “false”, i.e the attribute has a static value

5.4 Tag Handlers

A *tag handler* is a run-time server-side object that is created to help evaluate actions during the execution of a JSP page. A tag handler supports a run-time protocol that facilitates passing information from the JSP page to the handler.

A tag handler is a server-side invisible JavaBeans component, but it implements an additional interface to indicate that it has a richer run-time protocol. There are two interfaces that describe a tag handler: `Tag` is used for simple tag handlers that are not interested in manipulating their body content (if any); `BodyTag` is an extension of `Tag` and gives a tag handler access to its body. The `TagSupport` and `BodyTagSupport` classes can be used as base classes when creating new tag handlers.

5.4.1 Properties

A tag handler has some *properties* that are set by the JSP container (usually through the JSP page implementation class) using setter methods:

- The **pageContext** object for the JSP page where the tag is located; this object can be used to access defined objects.
- The **parent** tag handler for the enclosing action.

A tag handler may have additional properties, as any other JavaBean component. These properties will have setters and getter methods, as described in the JavaBeans component specification, and used throughout the Java platform.

All attributes of a custom action must be JavaBeans component properties, although some properties may not be exposed as attributes in the Tag Library Descriptor.

Additional translation time information (`TagExtraInfo`) associated with the action indicates the name of the variables it introduces, their types and their scope. At specific moments (after processing the start tag; after processing the end tag), the JSP container can automatically synchronize the PageContext information with variables in the scripting language so they can be made available directly through the scripting elements.

5.4.2 Basic Protocol: Tag Interface

This section describes the `Tag` interface which defines the basic contract for all tag handlers. See Section 5.4.7 for a summary of the life-cycle issues.

A tag handler has some *properties* that must be initialized before it can be used. It is the responsibility of the JSP container to invoke the appropriate setter methods to initialize these properties. Once properly set, these properties are expected to be persistent, so that if the JSP container ascertains that a property has already been set on a given tag handler instance, it needs not set it again. These properties include the properties in the Tag interface as well as other properties

Once initialized, the `doStartTag` and `doEndTag` methods can be invoked on the tag handler. Between these invocations, the tag handler is assumed to hold a state that must be preserved. After the `doEndTag` invocation, the tag handler is available for further invocations (and it is expected to have retained its properties). Once all invocations on the tag handler are completed, the `release` method is invoked on it. Once a `release` method is invoked all properties are assumed to have been reset to an unspecified value.

Properties

All tag handlers must have the following properties: pageContext, and parent. When setting properties, the order is always *pageContext and parent.* The `Tag` interface specifies the setter methods for all properties, and the getter method for parent.

setPageContext(PageContext) Sets the pageContext property of a tag handler.

setParent(Tag) Sets the parent Tag for a tag handler.

Tag getParent() Get the parent Tag for a tag handler. This method is used by the `findAncestorWithClass` static method in `TagSupport`.

Methods

There are two main action methods and one method for releasing all resources owned by a tag handler.

doStartTag() Process the start tag of this action. The `doStartTag` method assumes that the properties *pageContext* and *parent* have been set. It also assumes that any properties exposed as attributes have been set too. When this method is invoked, the body has not yet been evaluated.

At the end of this method invocation some scripting variables may be assigned from the `pageContext` object, as indicated by the optional `TagExtraInfo` class (at translation time).

This method returns an indication of whether the body of the action should be evaluated (EVAL_BODY_INCLUDE or EVAL_BODY_TAG) or not (SKIP_BODY). See below for more details.

EVAL_BODY_INCLUDE is not valid if the tag handler implements `BodyTag`; EVAL_BODY_TAG is not valid if the tag handler implements `Tag` and does not implement `BodyTag`.

This method may throw a JspException.

doEndTag() Process the end tag of this action. This method will be called after returning from `doStartTag`. The body of the action may or not have been evaluated, depending on the return value of `doStartTag`.

At the end of this method invocation some scripting variables may be assigned from the `pageContext` object, as indicated by the optional `TagExtraInfo` class (at translation time).

If this method returns EVAL_PAGE, the rest of the page continues to be evaluated. If this method returns SKIP_PAGE, the rest of the page is not evaluated and the request is completed. If this request was forwarded or included from another page (or Servlet), only the current page evaluation is completed.

This method may throw a JspException.

release() Release a tag instance.

Simple Actions with non-empty Bodies

If a tag library descriptor maps an action with a non-empty body to a tag handler that implements the `Tag` interface, the tag handler cannot *manipulate* this body because there is no mechanism for the tag handler to access that body. To make the situation more explicit, the return value of `doStartTag` is either SKIP_BODY, EVAL_BODY_INCLUDE or EVAL_BODY_TAG. The meanings are as follows:

- SKIP_BODY means do not process the body of the action (if it exists).
- EVAL_BODY_INCLUDE means process the body of the action but do not create a new BodyContent (see below) nor change the value of the out implicit object.
- EVAL_BODY_TAG means create a new `BodyContent`, change the value of the out implicit object and process the body of the action.

A typical use for EVAL_BODY_INCLUDE could be a conditional inclusion action tag. Since the body is to be passed through directly, there is no need for the tag handler to manipulate it, and thus the tag handler needs not implement `BodyTag`.

To help in catching errors, EVAL_BODY_INCLUDE is not valid in a tag handler that implements `BodyTag`, while EVAL_BODY_TAG is not valid in a tag handler that implements `Tag` but does not implement BodyTag.

5.4.3 The TagSupport Base Class

The `TagSupport` class is a utility class intended to be used as the base class for new tag handlers. The TagSupport class implements the `Tag` interface and adds additional convenience methods including getter methods for the properties in `Tag`. `TagSupport` has one static method that is included to facilitate coordination among cooperating tags.

Tag findAncestorWithClass(Tag, class) Find the instance of a given class type that is closest to a given instance. This method uses the `getParent` method from the `Tag` interface.

The return value of the `doStartTag()` method is `SKIP_BODY`. The return value of the `doEndTag()` method is `EVAL_PAGE`.

5.4.4 Body Protocol: BodyTag Interface

The `BodyTag` interface extends `Tag` with methods to manipulate the body of an action. These methods act on the *bodyContent* property of a `BodyTag` instance.

It is the responsibility of the tag handler to manipulate the body content. For example the tag handler may take the body content, convert it into a `String` using the `bodyContent.getString` method and then use it. Or the tag handler may take the body content and write it out into its enclosing JspWriter using the `bodyContent.writeOut` method.

A tag handler that implements `BodyTag` is treated as one that implements `Tag`, except that the `doStartTag` method can either return SKIP_BODY or EVAL_BODY_TAG, not EVAL_BODY_INCLUDE. If EVAL_BODY_TAG is returned, then a BodyContent object will be created to capture the body evaluation. This object is obtained by calling the `pushBody` method of the current `pageContext`, which additionally has the effect of saving the previous `out` value. The object is returned through a call to the `popBody` method of the `PageContext` class; the call also restores the value of `out`.

Properties

There is only one additional property: *bodyContent*

setBodyContent(BodyContent) Set the *bodyContent* property. It will be invoked at most once per action invocation. It will be invoked before `doInitBody` and it will not be invoked if there is no body evaluation.

When `setBodyContent` is invoked, the value of the implicit object out has already been changed in the `pageContext` object. The body passed will have not data on it.

Methods

There are two action methods:

doInitBody() Invoked before the *first* time the body is to be evaluated. Not invoked in empty tags or in tags returning SKIP_BODY in `doStartTag()`.

Depending on TagExtraInfo values, the JSP container will resynchronize some variable values after the `doInitBody` invocation.

This method may throw a JspException.

doAfterBody() Invoked after *every* body evaluation. Not invoked in empty tags or in tags returning SKIP_BODY in `doStartTag.` If `doAfterBody` returns EVAL_BODY_TAG, a new evaluation of the body will happen (followed by another invocation of

doAfterBody). If doAfterBody returns SKIP_BODY no more body evaluations will occur, the value of out will be restored using the popBody method in pageContext, and then doEndTag will be invoked.

The method re-invocations may be lead to different actions because there might have been some changes to shared state, or because of external computation..

Depending on TagExtraInfo values, the JSP container will resynchronize some variable values after every doAfterBody invocation (so a reevaluation of the body will return a different value).

This method may throw a JspException.

5.4.5 The BodyContent Class

The BodyContent is a subclass of JspWriter that can be used to process body evaluations so they can retrieved later on. The class has methods to convert its contents into a String, to read its contents, and to clear the contents.

The buffer size of a BodyContent object is "unbounded". A BodyContent object cannot be in autoFlush mode. It is not possible to invoke flush on a BodyContent object, as there is no *backing* stream. This means that it is not legal to do a jsp:include when out is not bound to the top-level JspWriter.

Instances of this class are created by invoking the pushBody and popBody methods of the PageContext class. A BodyContent is enclosed within another JspWriter (maybe another BodyContent object) following the structure of their associated actions.

The BodyContent type contains four main methods:

clearBody() This is a version of the clear() method from JspWriter that is guaranteed not to throw exceptions.

getReader() Get a reader into the contents of this instance.

getString() As getReader() but returns a String.

writeOut(Writer) Write out the content of this instance into the provided Writer.

getEnclosingWriter() Get the JspWriter enclosing this BodyContent.

5.4.6 The BodyTagSupport Base Class

The BodyTagSupport class is a utility class intended to be used as the base class for new tag handlers implementing BodyTag. The BodyTagSupport class implements the BodyTag interface and adds additional convenience methods including getter methods for the *bodyContent* property and methods to get at the previous out JspWriter.

The return value of the `doStartTag()` method is `EVAL_BODY_TAG`. The return value of the `doEndTag()` method is `EVAL_PAGE`. The return value of the `doAfterBody()` method is `SKIP_BODY`.

5.4.7 Life-Cycle Considerations

At execution time the implementation of a JSP page will use an available Tag instance with the appropriate prefix and name that is not being used, initialize it, and then follow the protocol described below. Afterwards, it will release the instance and make it available for further use. This approach reduces the number of instances that are needed at a time.

Initialization is done by setting the properties *pageContext* and *parent*, in that order, while release is done by invoking release().

An Execution Trace

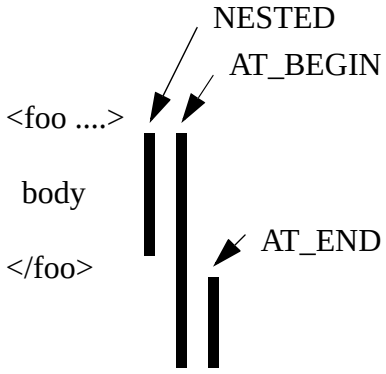
The following figure shows the run-time trace for two actions supported by a tag handler implementing `BodyTag`; setters are in italics, while actions are not. The inner boxes highlight the portion of the protocol used to interact with the body of the tag. In this example, we are assuming that the second action has the same parent but one different attribute values.

5.5 Scripting Variables

The JSP specification supports scripting variables that can be declared within a scriptlet and can be used in another. The actions in a JSP page also can be used to define scripting variables so they can used in scripting elements, or in other actions; for example, the jsp:useBean standard action may define an object which can later be used through a scripting variable.

Since the logic that decides whether an action instance will define a scripting variable may be quite complex, this information is not encoded in the Tag Library Descriptor directly; rather, the name of a `TagExtraInfo` class is given in the TLD and the `getVariableInfo` method is used at translation time to obtain information on each variable that will be created at request time when this action is executed. The method is passed a `TagData` instance that contains the translation-time attribute values.

The result of the invocation on `getVariableInfo` is an array of `VariableInfo` objects. Each such object describes a scripting variable by providing its name, its type, whether the variable is new or not, and what its scope is. Scope is best described through a picture:.



The defined values for scope are:

- **NESTED**, if the scripting variable is available between the start tag and the end tag of the action that defines it.
- **AT_BEGIN**, if the scripting variable is available from the start tag of the action that defines it until the end of the page.
- **AT_END**, if the scripting variable is available after the end tag of the action that defines it until the end of the page.

The scope value for a variable implies what methods *may* affect its value and thus, in lack of additional information, where synchronization is needed:

- for **NESTED**, after `doInitBody` and `doAfterBody` for a tag handler implementing `BodyTag`, and after `doStartTag` otherwise.
- for **AT_BEGIN**, after `doInitBody, doAfterBody,` and `doEndTag` for a tag handler implementing `BodyTag`, and `doStartTag` and `doEndTag` otherwise.
- for **AT_END**, after `doEndTag` method.

5.6 Cooperating Actions

Often two actions in a JSP page want to cooperate, perhaps by one action creating some server-side object that is used by the other. There are two basic mechanisms in the JSP specification to achieve this.

5.6.1 Ids and PageContext.

One mechanism for supporting cooperation among actions is by giving the object a name within the JSP page; the first action creates and names the object while the second action uses the name to retrieve the object.

For example, in the following JSP page fragment the `foo` action creates a server-side object and give it the name “myObject”. Then the `bar` action accesses that server-side object and takes some action.

```
<x:foo id=″myObject″ />
<x:bar ref=″myObjet″ />
```

In a JSP container implementation, the mapping between the name and the value is kept by the implicit object pageContext. This object is passed around through the tag handler instances so it can be used to communicate information: all it is needed is to know the name under which the information is stored into the pageContext.

5.6.2 Run-Time Stack

An alternative to explicit communication of information through a named object is implicit coordination based on syntactic scoping.

For example, in the following JSP page fragment the foo action might create a server-side object; later the nested bar action might access that server-side object. The object is not named within the pageContext: it is found because the specific foo element is the closest enclosing instance of a known element type.

```
<foo>
  <bar/>
</foo>
```

This functionality is supported through the `findAncestorWithClass(Tag, Class)` static method of the Tag class which uses a reference to parent tag kept by each Tag instance, which effectively provides a run-time execution stack.

5.7 Validation

Frequently there are constraints on how actions are to be used and when these constraints are not kept an error should be reported to the user. There are several mechanisms in the JSP 1.1 specification to describe syntactic and semantic constraints among actions; future specifications may add additional mechanisms.

5.7.1 Syntactic Information on the TLD

The Tag Library Descriptor contains some basic syntactic information. In particular, the attributes are described including their name, whether they are optional or mandatory, and whether they accept request-time expressions. Additionally the `bodycontent` attribute can be used to indicate that an action must be empty.

All constraints described in the TLD must be enforced. A tag library author can assume that the tag handler instance corresponds to an action that satisfies all constraints indicated in the TLD.

5.7.2 Syntactic Information in a TagExtraInfo Class

Additional translation-time validation can be done using the `isValid` method in the `TagExtraInfo` class. The `isValid` method is invoked at translation-time and is passed a `TagData` instance as its argument.

5.7.3 Raising an Error at Action Time

In some cases, additional request-time validation will be done dynamically within the methods in the tag handler. If an error is discovered, an instance of `JspError` can be thrown. If uncaught, this object will invoke the errorpage mechanism of the JSP specification.

5.8 Conventions and Other Issues

This section is not normative, although it reflects good design practices.

5.8.1 How to Define New Implicit Objects

We advocate the following style for the introduction of implicit objects:

- Define a tag library.
- Add an action called defineObjects; this action will define the desired objects.

Then the JSP page can make these objects available as follows:

```
<%@ tablig prefix="me" uri="......" %>
<me:defineObjects />
.... start using the objects....
```

This approach has the advantage of requiring no new machinery and of making very explicit the dependency.

In some cases there may be some implementation dependency in making these objects available; for example, they may be providing access to some functionality that only exists in some implementation. This can be done by having the tag extension class test at run-time for the existence of some implementation dependent feature and raise a run-time error (this, of course, makes the page not J2EE compliant, but that is a different discussion).

This mechanism, together with the access to metadata information allows for vendors to innovate within the standard.

Note: if a feature is added to a JSP specification, and a vendor also provides that feature through its vendor-specific mechanism, the standard mechanism, as indicated in the JSP specification will “win”. This means that vendor-specific mechanisms can slowly migrate into the specification as they prove their usefulness.

5.8.2 Access to Vendor-Specific information

If a vendor wants to associate with some tag library some information that is not described in the current version of the TLD, it can do so by inserting the information in a document it controls, inserting the document in the WEB-INF portion of the JAR file where the Tab Library resides, and using the standard Servlet 2.2 mechanisms to access that information.

The vendor can now use the ID machinery to refer to the element within the TLD.

5.8.3 Customizing a Tag Library

A tag library can be customized at assembly and deployment time. For example, a tag library that provides access to databases may be customized with login and password information.

There is no convenient place in `web.xml` in the Servlet 2.2 spec for customization information A standardized mechanism is probably going to be part of a forthcoming JSP specification, but in the meantime the suggestion is that a tag library author place this information in a well-known location at some resource in the `WEB-INF/` portion of the Web Application and access it via the `getResource()` call on the `ServletContext`.

CHAPTER 6

JSP Technology Classes

This chapter describes the packages that are part of the JSP 1.1 specification. The packages may be used in a number of situations, including within scripting elements, by base classes, and in implementations of Tag Extensions.

There are two packages

- `javax.servlet.jsp`
- `javax.servlet.jsp.tagext`.

The javadoc documents that accompany this specification[1] provide detailed description of the packages. This appendix provides an overview, context, and usage guidelines.

6.1 Package javax.servlet.jsp

The javax.servlet.jsp package contains a number of classes and interfaces that describe and define the contracts between a JSP page implementation class and the runtime environment provided for an instance of such a class by a conforming JSP container.

6.1.1 JspPage and HttpJspPage

Two interfaces describe the interaction between a class implementing a JSP page and the JSP container: *HttpJspPage* and *JspPage*. Chapter 3 describes the role of these two interfaces in detail. The `JspPage` contract is not further described here, see the javadoc documentation for details.

The large majority of the JSP pages use the HTTP protocol and thus are based on the `HttpJspPage` contract. This interface has three methods, two of which can be redefined by the JSP author using a declaration scripting element:

1. All JSP-related specifications are available from http://java.sun.com/products/jsp.

jspInit() The `jspInit()` method is invoked when the JSP page is initialized. It is the responsibility of the JSP implementation (and of the class mentioned by the `extends` attribute, if present) that at this point invocations to the `getServletConfig`() method will return the desired value.

A JSP page can override this method by including a definition for it in a declaration element.

The signature of this method is *void jspInit().*

jspDestroy() The `jspDestroy`() method is invoked when the JSP page is about to be destroyed.

A JSP page can override this method by including a definition for it in a declaration element.

The signature of this method is *void jspDestroy().*

_jspService() The `_jspService`() method corresponds to the body of the JSP page. This method is defined automatically by the JSP container and should never be defined by the JSP page author.

If a superclass is specified using the `extends` attribute, that superclass may choose to perform some actions in its service() method before or after calling the _jspService() method. See Section 3.2.4.

The signature of this method is *public void _jspService(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException.*

6.1.2 JspWriter

The actions and template data in a JSP page is written using the `JspWriter` object that is referenced by the implicit variable `out`. This variable is initialized by code generated automatically by the JSP container (see the `PageContext` object in Section 6.1.4).

The initial `JspWriter` object is associated with the `PrintWriter` object of the `ServletResponse` in a way that depends on whether the page is or not buffered. If the page is not buffered, output written to this `JspWriter` object will be written through to the `PrintWriter` directly, which will be created if necessary by invoking the `getWriter()` method on the `response` object. But if the page is buffered, the `PrintWriter` object will not be created until when the buffer is flushed, and operations like `setContentType()` are legal. Since this flexibility simplifies programming substantially, buffering is the default for JSP pages.

Buffering raises the issue of what to do when the buffer is exceeded. Two approaches can be taken:

- Exceeding the buffer is not a fatal error; when the buffer is exceeded, just flush the output.
- Exceeding the buffer is a fatal error; when the buffer is exceeded, raise an exception.

Both approaches are valid, and thus both are supported in the JSP technology. The behavior of a page is controlled by the `autoFlush` attribute, which defaults to true. In general, JSP pages that need to be sure that correct and complete data has been sent to their client may want to set `autoFlush` to `false`, with a typical case being that where the client is an application itself. On the other hand, JSP pages that send data that is meaningful even when partially constructed may want to set `autoFlush` to `true`; a case may be when the data is sent for immediate display through a browser. Each application will need to consider their specific needs.

An alternative considered was to make the buffer size unbounded, but this has the disadvantage that runaway computations may consume an unbounded amount of resources.

The `JspWriter` interface includes behaviors from `java.io.BufferedWriter` and `java.io.PrintWriter` API's but incorporates buffering and does not consume IOExceptions as PrintWriter does. If a particular use requires a PrintWriter, as in the case of desiring to insert an exception stack trace, one can be obtained by wrapping the JspWriter with a PrintWriter.

The usual methods found in PrintWriter are available with the only modification being that the JspWriter methods do not consume IOExceptions. The additional methods are:

clear() This method is used to clear the buffer of data. It is illegal to invoke it if the JspWriter is not buffered. And exception will be raised if the buffer has been autoFlushed and clear() is invoked. Also see clearBuffer().

The method signature is *void clear()*

clearBuffer() This method is like clear() except that no exception will be raised if the buffer has been autoFlushed().

The method signature is *void clearBuffer()*.

flush() This method is used to flush the buffer of data. The method may be invoked indirectly if the buffer size is exceeded. The underlying PrintWriter object is guaranteed to be created exactly the first time data is flushed to it.

The method signature is *void flush()*

close() This method can be used to close the stream. It needs not be invoked explicitly for the initial JspWriter as the code generated by the JSP container will automatically include a call to close().

The method signatures is *void close()*.

getBufferSize() This method returns the size of the buffer used by the JspWriter.

The method signatures is *int getBufferSize()*

getRemaining() This method returns the number of unused bytes in the buffer.

The method signature is *int getRemaining()*

isAutoFlush() This method indicates whether the JspWriter is autoFlushing.

The method signature is *boolean isAutoFlush()*

6.1.3 JspException and JspError

JspException is a generic exception class. It currently has one subclass: JspError. A JspError has a message and it reflects an unexpected error that has been found during the execution of a JSP page. If uncaught, the error will result in an invocation of the errorpage machinery.

6.1.4 PageContext

A `PageContext` provides an object that encapsulates implementation-dependent features and provides convenience methods. A JSP page implementation class that uses a `PageContext` as shown in FIGURE F-1 will run unmodified in any compliant JSP container taking advantage of implementation-specific improvements like high performance `JspWriter`s. JSP 1.1 compliant containers must generate JSP page implementation classes that use this `PageContext` object to provide insulation from platform implementation details.

A `PageContext` object is initialized by a JSP container implementation early on in the processing of the `_jspService()` method. The `PageContext` implementation itself is implementation dependent, and is obtained via a creation method on the `JspFactory`.

The `PageContext` provides a number of facilities, including:

- a single API that manages the operations available over various scope namespaces (`page, request, session, application`) such as `setAttribute()`, `getAttribute()` and `removeAttribute()`, etc.
- a mechanism for obtaining a platform dependent implementation of the `JspWriter` that is assigned to the "`out`" implicit scripting variable.
- a number of simple convenience *getter* API's for obtaining references to various request-time objects.
- mechanisms to *forward* or *include* the current request being processed to other components in the application

6.1.4.1 Creation

The PageContext class is an abstract class, designed to be extended by conforming JSP container runtime environments to provide implementation dependent implementations.

An instance of a PageContext is created by a JSP container-specific implementation class at the beginning of it's `_jspService()` method via an implementation-specific default `JspFactory`, as shown in FIGURE F-1:

FIGURE F-1 Using PageContext to Provide Implementation-Independence

```
public void _jspService(HttpServletRequest request,
                 HttpServletResponse response)
          throws IOException, ServletException {

       JSPFactory  factory     =
JSPFactory.getDefaultFactory();
       // Get the page context object for this page
       PageContext pageContext = factory.getPageContext(
                              this,
                              request,
                              response,
                              null, // e.g. no errorPageURL,
                              false, // e.g. no session
                              JspWriter.DEFAULT_BUFFER,
                              true // autoflush=true
                              );
       // Initialize implicit variables for scripting
       HttpSession session = pageContext.getSession();
       JspWriter   out     = pageContext.getOut();
       Object      page    = this;
       try {
         // body of JSP here ...
       } catch (Exception e) {
         out.clear();
         pageContext.handlePageException(e);
       } finally {
         out.close();
         factory.releasePageContext(pageContext);
       }
}
```

6.1.4.2 Usage

The PageContext object provides access to multiple functionality

Uniform Access to Multiple Scopes

These methods provide uniform access to the diverse scopes objects. The implementation must use the underlying Servlet machinery corresponding to that scope, so information can be passed back and forth between Servlets and JSP pages.

getAttribute() Access an attribute in the page scope, or null if not found.

getAttribute() Overload of previous method to access an attribute in a given scope or null if not found.

findAttribute() Searches for the named attribute in page, request, session (if valid) and application scopes in order and returns the value associated or null.

getAttributeNamesInScope() Enumerate all the attributes in a given scope

getAttributesScope() Get the scope where a given attribute is defined

removeAttribute() Remove the object reference associated with the given name, look in all scopes in the scope order.

removeAttribute() Overload of previous method to remove the object reference associated with the given name in the given scope.

setAttribute() Register the given name and object in the page scope.

setAttribute() Overload of previous method to register the given name and object in the given scope.

Access to Implicit Objects

These methods provide convenience access to implicit objects and other objects that can be obtained by different ways.

getOut() The current value of the `out` object (a JspWriter).

getException() The current value of the `exception` object (an Exception).

getPage() The current value of the `page` object (a Servlet).

getRequest() The current value of the `request` object (a ServletRequest).

getResponse() The current value of the `response` object (a ServletResponse).

getSession() The current value of the `session` object (an HttpSession).

getServletConfig() The ServletConfig instance.

getServletContext() The ServletContext instance.

Management of Nested Scopes

These methods enable the management of nested JspWriter streams to implement Tag Extensions. Note that `pushBody()` returns a `BodyContent`, while `popBody()` returns a `JspWriter`, which will need to be casted into a BodyContent in all but the top level.

pushBody() Return a new `BodyContent` object, save the current "out" `JspWriter`, and update the value of the "out" attribute in the page scope attribute namespace of the `PageContext`

popBody() Return the previous `JspWriter` "out" saved by the matching `pushBody()`, and update the value of the "out" attribute in the page scope attribute namespace of the `PageContext`.

Management of PageContext Object

The following two methods provide management of the PageContext object itself. These methods are not intended to be used by the JSP page author.

initialize() Initialize a PageContext with the given data.

release() Release a PageContext object.

Forward and Includes

These methods encapsulate forwarding and inclusion.

forward() This method is used to forward the current ServletRequest and ServletResponse to another component in the application.

The signature of this method is *void forward(String relativeUrlPath) throws ServletException, IOException.*

include() This method causes the resource specified to be processed as part of the current ServletRequest and ServletResponse being processed by the calling Thread.

The signature of this method is *void include(String relativeUrlPath) throws ServletException, IOException.*

handlePageException() Process an unhandled page level exception by performing a redirect.

The signature of this method is *void handlePageException(Exception e) throws ServletException, IOException.*

6.1.5 JspEngineInfo

The `JspEngineInfo` class provides information on the JSP container implementation. The only method currently available is:

getSpecificationVersion() Returns a String in Dewey decimal describing the highest version of the JSP specification implemented by the JSP container. See the `java.lang.Package` class in the Java 2 platform for other methods that *may* appear in this class in future specifications.

6.1.6 JspFactory

The `JspFactory` provides a mechanism to instantiate platform dependent objects in a platform independent manner. The `PageContext` class and the JspEngineInfo class are the only implementation dependent classes that can be created from the factory.

Typically at initialization time, a JSP container will call the static `setDefaultFactory()` method in order to register it's own factory implementation. JSP page implementation classes will use the `getDefaultFactory()` method in order to obtain this factory and use it to construct `PageContext` instances in order to process client requests.

JspFactory objects are not intended to be used by JSP page authors.

6.2 Package javax.servlet.jsp.tagext

The classes in the javax.servlet.jsp.tagext package are related to the Tag Extension mechanism. They are described in detail in Chapter 5. In this section we will just briefly review and group them.

A brief description of the classes follows. Classes whose name ends in Info are translation-time classes. Instances that are initialized from the Tag Library Descriptor are in bold italics:

BodyContent Encapsulates the evaluation of a tag body.

Tag The interface of a tag handler for an action that does not want to manipulate its body.

BodyTag The interface of a tag handler for an action that wants to manipulate its body.

TagSupport A base class for defining new tag handlers implementing Tag.

BodyTagSupport A base class for defining new tag handlers implementing BodyTag.

***TagAttributeInfo*:** Information on the attributes of a tag.

TagData: The attribute/value information for a tag instance. Used at translation-time only.

TagExtraInfo Tag Author-provided class to describe additional translation-time information.

TagInfo: Information needed by the JSP container at page compilation time.

***TagLibraryInfo*:** Information on a tag library.

VariableInfo Information on scripting variables.

CHAPTER 7

JSP Pages as XML Documents

This chapter defines a standard XML document for each JSP page.

The JSP page to XML document mapping is not visible to JSP 1.1 containers; it will receive substantial emphasis in the next releases of the JSP specification. Since the mapping has not received great usage, we particularly encourage feedback in this area.

7.1 Why an XML Representation

There are a number of reasons why it would be impractical to define JSP pages as XML documents when the JSP page is to be authored manually:

- An XML document must have a single top element; a JSP page is conveniently organized as a sequence of template text and elements.
- In an XML document all tags are “significant”; to “pass through” a tag, it needs to be escaped using a mechanism like CDATA. In a JSP page, tags that are undefined by the JSP specification are passed through automatically.
- Some very common programming tokens, like “<“ are significant to XML; the JSP specification provides a mechanism (the <% syntax) to “pass through” these tokens.

On the other hand, the JSP specification is not gratuitously inconsistent with XML: all features have been made XML-compliant as much as possible.

The hand-authoring friendliness of JSP pages is very important for the initial adoption of the JSP technology; this is also likely to remain important in later time-frames, but tool manipulation of JSP pages will take a stronger role then. In that context, there is an ever growing collection of tools and APIs that support manipulation of XML documents.

The JSP 1.1 specification addresses both requirements by providing a friendly syntax and also defining a standard XML document for a JSP page. A JSP 1.1-compliant tool needs not do anything special with this document.

7.2 Document Type

7.2.1 The jsp:root Element

An XML document representing a JSP page has `jsp:root` as its root element type. The root is also the place where taglibs will insert their namespace attributes. The top element has an `xmlns` attribute that enables the use of the standard elements defined in the JSP 1.1 specification.

```
<jsp:root
     xmlns:jsp="http://java.sun.com/products/jsp/dtd/jsp_1_0.dtd">
       remainder of transformed JSP page
</jsp:root>
```

7.2.2 Public ID

The proposed Document Type Declaration is:

```
<! DOCTYPE root
   PUBLIC"-//Sun Microsystems Inc.//DTD JavaServer Pages Version 1.1//EN"
      "http://java.sun.com/products/jsp/dtd/jspcore_1_0.dtd">
```

7.3 Directives

A directive in a JSP page is of the form

<%@ *directive { attr="value" }** %>

Most directives get translated into an element of the form:

<jsp:directive.*directive { attr="value" }** />

7.3.1 The page directive

In the XML document corresponding to JSP pages, the `page` directive is represented using the syntax:

```
<jsp:directive.page page_directive_attr_list />
```

See Section 2.7.1 for description of *page_directive_attr_list*.

Example

The directive:

```
<%@ page info="my latest JSP Example V1.1" %>
```

corresponds to the XML element:

```
<jsp:directive.page info="my latest JSP Example V1.1" />
```

7.3.2 The `include` Directive

In the XML document corresponding to JSP pages, the include directive is represented using the syntax:

```
<jsp:directive.include file="relativeURLspec" flush="true|false" />
```

Examples

Below are two examples, one in JSP syntax, the other using XML syntax:

```
<%@ include file="copyright.hmtl" %>
<jsp:directive.include file="htmldocs/logo.html" />
```

7.3.3 The `taglib` Directive

In the XML document corresponding to JSP pages, the taglib directive is represented as an xmlns: attribute within the root element of the JSP page document.

7.4 Scripting Elements

The JSP 1.1 specification has three scripting language elements—declarations, scriptlets, and expressions. The scripting elements have a "<%"-based syntax as follows:

```
<%! this is a declaration %>
<% this is a scriptlet %>
<%= this is an expression %>
```

7.4.1 Declarations

In the XML document corresponding to JSP pages, declarations are represented using the syntax:

```
<jsp:declaration> declaration goes here </jsp:declaration>
```

For example, the second example from Section 2.10.1:

```
<%! public String f(int i) { if (i<3) return(“...”); ... } %>
```

is translated using a CDATA statement to avoid having to quote the “<“ inside the jsp:declaration.

```
<jsp:declaration> <![CDATA[ public String f(int i) { if (i<3)
return(“...”); } ]]> </jsp:declaration>
```

DTD Fragment

```
<!ELEMENT jsp:declaration (#PCDATA) >
```

7.4.2 Scriptlets

In the XML document corresponding to JSP pages, directives are represented using the syntax:

```
<jsp:scriptlet> code fragment goes here </jsp:scriptlet>
```

DTD Fragment

```
<!ELEMENT jsp:scriptlet (#PCDATA) >
```

7.4.3 Expressions

In the XML document corresponding to JSP pages, directives are represented using the syntax:

```
<jsp:expression> expression goes here </jsp:expression>
```

DTD Fragment

```
<!ELEMENT jsp:expression (#PCDATA) >
```

7.5 Actions

The syntax for action elements is based on XML; the only transformations needed are due to quoting conventions and the syntax of request-time attribute expressions.

7.6 Transforming a JSP Page into an XML Document

The standard XML document for a JSP page is defined by transformation of the JSP page.

- Add a `<jsp:root>` element as the root. Enable a “jsp” namespace prefix for the standard tags within this root.
- Convert all the <% elements into valid XML elements as described in Section 7.4.1 and following sections.
- Convert the quotation mechanisms appropriately.
- Convert the `taglib` directive into namespace attributes of the `<jsp:root>` element.
- Create CDATA elements for all segments of the JSP page that do not correspond to JSP elements.

A quick summary of the transformation is shown in TABLE 7-1:

TABLE 7-1 XML standard tags for directives and scripting elements

JSP page element	XML equivalent
<%@ page ... %>	<jsp:directive.page ... />
<%@ taglib ... %>	jsp:root element is annotated with namespace information.
<%@ include ... %>	<jsp:directive.include .../>
<%! ... %>	<jsp:declaration> .... </jsp:declaration>
<% ... %>	<jsp:scriptlet> .... </jsp:scriptlet>
<%= .... %>	<jsp:expression> .... </jsp:expression>

7.6.1 Quoting Conventions

The quoting rules for the JSP 1.1 specification are designed to be friendly for hand authoring, they are not valid XML conventions.

Quoting conventions are converted in the generation of the XML document from the JSP page. This is not yet described in this version of the specification.

7.6.2 Request-Time Attribute Expressions

Request-time attribute expressions are of the form "<%= expression %>". Although this syntax is consistent with the syntax used elsewhere in a JSP page, it is not a legal XML syntax. The XML mapping for these expressions is into values of the form "%= expression' %", where the JSP specification quoting convention has been converted to the XML quoting convention.

7.7 DTD for the XML document

The following is a DTD for the current XML mapping:

FIGURE 7-1 DTD for the XML document

```
<!ENTITY % jsp.body "
(#PCDATA
|jsp:directive.page
|jsp:directive.include
|jsp:scriptlet
|jsp:declaration
|jsp:expression
|jsp:include
|jsp:forward
|jsp:useBean
|jsp:setProperty
|jsp:getProperty
|jsp:plugin
|jsp:fallback
|jsp:params
|jsp:param)*
">
```

```
<!ELEMENT jsp:useBean %jsp.body;>
<!ATTLIST jsp:useBean
id     ID   #REQUIRED
class  CDATA#REQUIRED
scope  (page|session|request|application) “page”>

<!ELEMENT jsp:setProperty EMPTY>
<!ATTLIST jsp:setProperty
name   IDREF#REQUIRED
propertyCDATA#REQUIRED
value  CDATA#IMPLIED
param  CDATA#IMPLIED>

<!ELEMENT jsp:getProperty EMPTY>
<!ATTLIST jsp:getProperty
name   IREF #REQUIRED
propertyCDATA#REQUIRED>

<!ELEMENTjsp:includeEMPTY>
<!ATTLISTjsp:include
flush  (true|false)"false"
page   CDATA#REQUIRED>

<!ELEMENT jsp:forward EMPTY>
<!ATTLISTjsp:forward
page   CDATA#REQUIRED>

<!ELEMENT jsp:scriptlet (#PCDATA)>

<!ELEMENT jsp:declaration (#PCDATA)>

<!ELEMENT jsp:expression (#PCDATA)>

<!ELEMENT jsp:directive.page EMPTY>
<!ATTLIST jsp:directive.page
languageCDATA“java”
extendsCDATA#IMPLIED
contentTypeCDATA“text/html; ISO-8859-1”
import CDATA#IMPLIED
session(true|false)“true”
buffer CDATA“8kb”
autoFlush(true|false)“true”
isThreadSafe(true|false)“true”
info   CDATA#IMPLIED
errorPageCDATA#IMPLIED
isErrorPage(true|false)“false”>

<!ELEMENT jsp:directive.include EMPTY>
<!ATTLIST jsp:directive.include
file CDATA #REQUIRED>
```

```
<!ELEMENT jsp:root %jsp.body;>
<!ATTLIST jsp:root
xmlns:jspCDATA#FIXED “http://java.sun.com/products/jsp/dtd/
jsp_1_0.dtd”>
```

APPENDIX **A**

Examples

This appendix describes some examples of custom actions defined using the Tag Extension mechanism. Refer to the JSP technology web site (http://java.sun.com/products/jsp) to retrieve a copy of the examples.

Each example is described briefly and the methods that are defined are explained and justified.

A.1 Simple Examples

Most tags are likely to be simple encapsulations of some functionality. The first set of examples are of this type and were already introduced in Section 5.1.3; here they are described in some more detail.

A.1.1 Call Functionality, no Body

The example of Section 5.1.3.1 is the simplest example:

```
<x:foo att1="..." att2="..." att3="..." />
```

In this case:

Tag Library Descriptor Indicates there are 3 mandatory attributes that are only translation-time, and that FooTag is the handler for tag "foo".

FooTag FooTag needs only provide a method for doStartTag(). The method `doStartTag` performs the desired actions, possibly interact with the PageContext data.

The attribute values are exposed as attributes and their values are set automatically by the JSP container.

A.1.2 Call Functionality, No Body, Define Object

The example of Section 5.1.3.2 is a simple variation of the previous example:

```
<x:bar id="mybar" att1="..." att2="..." att3="..." />
```

In this case:.

Tag Library Descriptor Indicates there are 3 mandatory attributes that are only translation-time, and that BarTag is the handler for tag "bar". For the example, assume that id is optional, in which case the TLD also indicates that as being the case.

The TLD also needs to indicate that BarExtraInfo is the name of the class that will provide information on the scripting variables introduced; see below.

BarTag `BarTag` needs only provide a method for `doStartTag()`. The method will interact with the `PageContext` data to register the created object with a name that is the value of the id attribute.

The attribute values are exposed as attributes and their values are set automatically by the JSP container.

BarExtraInfo This class, to be instantiated at translation time, needs only define a `getVariableInfo()` method. This method will look at the `TagData` object it is passed and will return either null or an array of `VariableInfo` objects of size 1, with the value corresponding to the scripting variable with name given by the it.

A.1.3 Template Mechanisms

Section 5.1.3.4 refers to a family of template mechanisms that have been used in the past. All of these mechanisms take some information and replace a token in the template page by either some fixed expansion or the result of some computation.

These mechanisms can be implemented through an empty-bodied action that is mapped to Tag handler that uses the desired information to locate the wanted resource and pushes the information into the JspWriter.

A.1.4 A 0.92-like useBean

The useBean of 0.92 will be described here. Note that the implementation will not be as efficient as ideal due to the need to do some computation at request evaluation time. Some discussion of the issues will be included.

A.2 A Set of SQL Tags

The following is a possible set of SQL tags. Note that this specific syntax is only used for pedagogical reasons, no endorsement is implied.

A.2.1 Connection, UserId, and Password

The connection tag creates a connection using some userid and password information. To show tag communication, userid and password are actually subelements of connection.

```
<x:connection id="con01"
        ref="connection.xml">
  <x:userid><%=session.getUserid()%></x:userid>
  <x:password><%=session.getPassword()%><x:password>
</x:connection>
```

In this example the “con01” object is available after the element.

This example uses the run-time stack so userid and password can locate their enclosing connection tag and can update userid and password data in there. This example also uses PageContext to register the SQL connection object with the pagecontext using "con01" so it can be retrieed later.

Tag Library Descriptor Indicates the names of the tags and their attributes. It associates Tag handlers with the tags. It also associates the ConnectionExtraInfo as the TagExtraInfo for connection.

UserIdTag UserIdTag needs access to its body; this it can do by defining a doAfterBody() method. This method will take the BodyContent and convert it into a String. Then it will use the findAncestorWithClass() static method on Tag to locate the enclosing connection tag instance and will set the desired userid information on that object.

PasswordTag This Tag handler is equivalent to UserIdTag.

ConnectionTag This Tag handler provides methods to setUserId() and to setPassword() that will be used by the enclosed actions; it also provides a getConnection() method that on-demand computes the desired SQL connection. This tag handler needs not be concerned with the body computation, but it will need to register the SQL connection object with the pageContext object if an ID is provided.

ConnectionExtraInfo This class is identical to BarExtraInfo from a previous example.

A.2.2 Query

The connection can now be used to make a query. The query element takes the body of the tag and make a query on it. The result gets inserted in place

```
<x:query id="balances" connection="con01">
       SELECT account, balance FROM acct_table
       where customer_number = <%= request.getCustno()%>
</x:query>
```

The implementation highlights are:

Tag Library Descriptor Query has two mandatory attributes (in our example), and they are described as so in the TLD. The TLD also associates QueryTag as the Tag handler class, and QueryExtraInfo as the TagExtraInfo for the query tag.

QueryTag QueryTag needs access to its body; this it can do by defining a doAfterBody() method. This method will take the BodyContent and convert it into a String. Then it will use the PageContext object to locate an SQL connection that was registered using the id that is the value to the connection attribute. The result of the query will be registered in the PageContext with the value of the id attribute as its name.

QueryExtraInfo This class is identical to BarExtraInfo from a previous example.

A.2.3 Iteration

Finally the query result can later be used to present the data by dynamically creating a series of <li> elements.

```
<ul>
<x:foreach iterate="row" in="balances">
<li>The balance for
account <%= row.getAccount()%> is <%= row.getBalance()%>
</x:foreach>
</ul>
```

Unlike query and connection, the foreach element does not define a new object for later use but it defines (and redefines) a "row" object that is accessible within its start and end tags.

The implementation of this tag requires the repeated evaluation of the body of the tag.

Tag Library Descriptor Foreach has two mandatory attributes (in our example), and they are described as so in the TLD. The TLD also associates ForEachTag as the Tag handler class, and ForEachExtraInfo as the TagExtraInfo for the foreach tag.

ForEachTag.doStartTag() ForEachTag needs to define a doStartTag() method to extract the value of the in and iterate attributes from the attribute values. The value of in ("balances" in this example) is used to get at the result data. The value of iterate ("row" in this example) is used as the key on which to store the iteration value.

The current value of the "out" variable is stored away so it can be used in doBody(). This method returns EVAL_BODY so as to force the evaluation of the body.

ForEachTag.doAfterBody() The BodyContent (obtained from getBodyContent()) contains the evaluation of the body of the element where the evaluation has been done in a context where the variable "row" is assigned the different rows of the query. This method now inserts this content into the surrounding out stream (obtained from getPreviousOut()).

This method now updates the binding of "row" and will return EVAL_BODY or SKIP_BODY depending on whether there were any more rows in the result set.

ForEachTag.doEndTag() Just clean up.

ForEachExtraInfo() The translation-time information indicates that this action defines a new scripting variable, with scope NESTED and name corresponding to the value of the "row" attribute.

APPENDIX **B**

Implementation Notes

This appendix provides implementation notes on the JSP technology. The notes are not normative and should only reinforce information described elsewhere. In particular, smarter but valid implementations are always welcome!

B.1 Delivering Localized Content

The definition in Section 2.7.4 enables but does not mandate the following implementation strategy:

- Translate the JSP page into a Servlet class source using the character encoding of the JSP page
- Compile the Servlet source using the -encoding switch. This produces a Servlet class file that has a number of (probably static) Unicode strings.
- Run the Servlet with the JspWriter configured to generate output in the encoding of the JSP page.

B.2 Processing TagLib directives

A strict 1-pass implementation would make custom actions visible only after their corresponding taglib directive appears. But this semantics can lead to the situation where a JSP page author is staring at a JSP page fragment with the assumption that a taglib directive appears before, when it really is included after. The semantics of Section 2.7.7 are designed to support an efficient implementation while minimizing JSP page author mistakes.

An implementation can still work in a 1-pass manner; it only needs to remember all the prefixes it has found and make the assumption that taglib directives appear before their use. But, if it later discovers that a taglib directive is defining a prefix that was used previously then it can cause a translation error.

B.3 Processing Tag Libraries

We describe some details of how to compile Tag Libraries and show an sketch of some code implementing a JSP page.

B.3.1 Processing a Tag Library Descriptor

The tag library descriptor is read and information is extracted from it. Some of the actions to be performed include:

- Record the mapping from tag to tag handler class
- Record the tag as a known to the JSP container so a taglib directive will introduce new actions.
- Record information on what tags must have an empty body, to be checked on individual pages later on.
- Record what are the valid attributes, and which ones can have request-time values.
- Record the TagExtraInfo classes, if any, associated with given tags,.
- Can be used to perform reflection on the tag handler classes to determine if the class implements Tag or BodyTag
- Can be used to perform introspection on the tag handler classes to determine their properties and their setter methods.

B.3.2 Processing a JSP page

When the JSP container processes a JSP page, it will perform analysis, validation, and generation of code. Actions include:

- Validate that actions who must have an empty body do.
- Validate that the only attributes that appear are those indicated in the TLD.
- Validate that the only attributes with request-time values are those indicated in the TLD.

- If there is a `TagExtraInfo` class associated in the TLD, a `TagData` object will be created with the appropriate attribute/value entries, and will be passed to the `isValid` method to determine if the attributes are valid.
- If there is a `TagExtraInfo` class associated in the TLD, a `TagData` object will be created with the appropriate attribute/value entries, and will be passed to the `getVariableInfo` method to determine if any scripting variables will be updated by this action at request time.

B.3.3 Generating the JSP Page Implementation Class

The JSP page implementation class generated by the JSP container includes code that:

- Generate the appropriate setter method invocations to set values for attributes
- Reuse tag handlers that are not being used to reduce the number of tag handler creations.
- Assume that a tag handler object retains its set properties to reduce the number of method invocations.
- Attempt to do some reorganization of setter method invocation so statically determined properties are not reset on a tag handler unnecessarily.

B.3.4 An Example

We now describe a simple example.

B.3.4.1 JSP Page Example

We will use a JSP page fragment as follows, where "chunk" is some uninterpreted template text

.

FIGURE 2-1 A JSP page fragment

```
chunk1
<x:foo id="myFoo" ...>
         chunk2
         <x:bar id="myBar" ...>
                 chunk3
         </x:bar>
         chunk4
</x:foo>
chunk5
<x:baz ref="myFoo" .../>
```

For the example, we will assume the TLD and TagExtraInfo provides the information in TABLE 2-1.

TABLE 2-1 TagInfo for the example

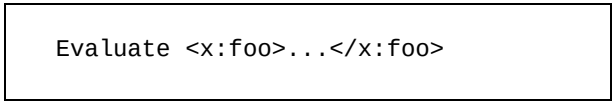
Tag	Handler	VariableInfo (name, type, scope)
foo	FooTag	myFoo, FooResult, AT_END
bar	BarTag	myBar, BarResult, AT_END
baz	BazTag	none

B.3.4.2 Implementation Code Fragment

The following code fragment can be used to implement the page fragment of FIGURE 2-1.

```
static {JspFactory _fact = JspFactory.getDefaultFactory();
}
_jspService(HttpServetRequest req, HttpServletResponse res) {

   PageContext pc = _fact.getPageContext(...); // once
   Object tempObject = null;
   int tempReturn;

   // just as an example, let's initialize all the Tag handlers
   FooTag footag = new FooTag();
   BarTag bartag = new BarTag()
   BazTag baztag = new BazTag();

   JspWriter out = pageContext.getOut(); // the initial out
```

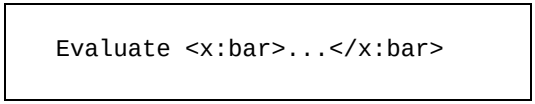
```
   // -- ditto for all other implicit objects

   EVAL chunk1;
```




```
   EVAL chunk5;

   baztag.setPageContext(pc);
   baztag.setParent(null);
   baztag.setRef(“myFoo”);
   try {
      (void)baztag.doStartTag();
      tempReturn = baztag.doEndTag();
   } finally {
      baztag.release();
   }
   if (tempReturn == SKIP_PAGE) {
      goto endOfPage; // pseudo-code
   };
endOfPage:
}
```

Where the evaluation of `<foo>...</foo>` is:

```
footag.setPageContext(pc);
footag.setParent(null);
footag.setId(“myFoo”);
try {
   if (footag.doStartTag() == EVAL_BODY_TAG) {
      try {
         out = pc.pushBody();
         foobag.setBodyContent(out);
         footag.doInitBody();

      repeat2:

         EVAL chunk2;
```




```
         EVAL chunk4;
         if (footag.doAfterBody() == EVAL_BODY_TAG) {
            goto repeat2; // pseudo-code
         }
      } finally {
```

```
            out = pc.popBody();
         }
      }
      tempResult = footag.doEndTag();
      tempObject = pc.getAttribute("myFoo");
   } finally {
      footag.release();
   }
   FooResult myFoo = (FooResult) tempObject;
   if (tempResult == SKIP_PAGE) {
      goto endOfPage; // pseudo-code
   }
```

and the evaluation of `<bar>...</bar>` is essentially the same:

```
bartag.setPageContext(pc);
bartag.setParent(footag);
bartag.setId(“myBar”);

try {
   if (bartag.doStartTag() == EVAL_BODY_TAG) {
      try {
         out = pc.pushBody();
         bartag.setBodyContent(out);
         bartag.doInitBody();

      repeat3:

         EVAL chunk3;

         if (bartag.doAfterBody() == EVAL_BODY_TAG) {
            goto repeat3; // pseudo-code
         }
      } finally {
         out = pc.popBody();
      }
   }
   tempResult = bartag.doEndTag();
   tempObject = pc.getAttribute("myBar");
} finally {
   bartag.release();
}
BarResult myBar = (BarResult) tempObject;
if (tempResult == SKIP_PAGE) {
   goto endOfPage; // pseudo-code
}
```

B.4 Implementing Buffering

Although the Servlet 2.2 specification provides for buffering, its semantics are `autoflush=true`. Buffering could be done without using the Servlet buffered stream, but this implementation does not allow for forwarding into a page that is not a JSP page. This is problematic for the implementation of `jsp:include` actions (see Section 2.13.4) since the goal is for `jsp:include` to be totally transparent to how the data is computed dynamically. Due to this, the only semantics we can use at this point still remains "flush on include" as it was in the JSP 1.0 specification.

APPENDIX **C**

Packaging JSP Pages

This appendix shows two simple examples of packaging a JSP page into a WAR for delivery into a Web container. In the first example, the JSP page is delivered in source form. This is likely to be the most common example. In the second example the JSP page is compiled into a Servlet that uses only Servlet 2.2 and JSP 1.1 API calls; the Servlet is then packaged into a WAR with a deployment descriptor such that it looks as the original JSP page to any client.

This appendix is non normative. Actually, strictly speaking, the appendix relates more to the Servlet 2.2 capabilities to the JSP 1.1 capabilities. The appendix is included here as this is a feature that JSP page authors and JSP page authoring tools are interested in.

C.1 A very simple JSP page

We start with a very simple JSP page `HelloWorld.jsp`.

```
<%@ page info="Example JSP pre-compiled" %>
<p>
Hello World
</p>
```

C.2 The JSP page packaged as source in a WAR file

The JSP page can be packaged into a WAR file by just placing it at location "`/HelloWorld.jsp`" the default JSP page extension mapping will pick it up. The `web.xml` is trivial:

```
<!DOCTYPE webapp
      SYSTEM "http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<webapp>
      <session-config>
         <session-timeout> 1 </session-timeout>
      </session-config>
</webapp>
```

C.3 The Servlet for the compiled JSP page

As an alternative, we will show how one can compile the JSP page into a Servlet class to run in a JSP container.

The JSP page is compiled into a Servlet with some implementation dependent name `_jsp_HelloWorld_XXX_Impl`. The Servlet code only depends on the JSP 1.1 and Servlet 2.2 APIs, as follows:

```
imports javax.servlet.*;
imports javax.servlet.http.*;
imports javax.servlet.jsp.*;

class _jsp_HelloWorld_XXX_Impl
extends_PlatformDependent_Jsp_Super_Impl {
      public void _jspInit() {
         // ...
      }

      public void jspDestroy() {
         // ...
      }
      static JspFactory _factory = JspFactory.getDefaultFactory();

      public void _jspService(   HttpServletRequest  request,
                        HttpServletResponse response)
            throws IOException, ServletException
      {
         Object         page     = this;
         HttpSession    session  = request.getSession();
         ServletConfig  config   = getServletConfig();
         ServletContext application = config.getServletContext();


         PageContext    pageContext
            = _factory.getPageContext( this,
```

```
                                        request,
                                        response,
                                        (String)NULL,
                                        true,
                                        JspWriter.DEFAULT_BUFFER,
                                        true
                                        );
            JspWriter      out      = pageContext.getOut();
               // page context creates initial JspWriter "out"

            try {
               out.println("<p>");
               out.println("Hello World");
               out.println("</p>");
            } catch (Exception e) {
               pageContext.handlePageException(e);
            } finally {
               _factory.releasePageContext(pageContext);
            }
         }
}
```

C.4 The Web Application Descriptor

The Servlet is made to look as a JSP page with the following `web.xml`:

```
<!DOCTYPE webapp
      SYSTEM "http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<webapp>
      <servlet>
         <servlet-name> HelloWorld </servlet-name>
         <servlet-class> HelloWorld.class </servlet-class>
      </servlet>

      <servlet-mapping>
         <servlet-name> HelloWorld </servlet-name>
         <url-pattern> /HelloWorld.jsp </url-pattern>
      </servlet-mapping>

      <session-config>
         <session-timeout> 1 </session-timeout>
      </session-config>
</webapp>
```

C.5 The WAR for the compiled JSP page

Finally everything is packaged together into a WAR:

```
/WEB-INF/web.xml
```

```
/WEB-INF/classes/HelloWorld.class
```

Note that if the Servlet class generated for the JSP page had dependent on some support classes, they would have to be included in the WAR.

APPENDIX **D**

Future

This appendix provides some information on future directions of the JSP technology.

D.1 Meta-Tag Information

A tag extension mechanism can include information:

- To execute a JSP page that uses the tag library.
- To edit a JSP page.
- To present the JSP page to the end user.

The JSP 1.1 specification concentrates on the first type of information providing some small amount of the other type of information; future specifications may address the other pieces.

D.2 Standard Tags

The tag extension mechanism enables the creation of tag libraries; some application domains have widespread applicability and there is substantial interest in defining standard tags for these domains.

D.3 Additional Application Support

We are investigating the benefits and costs of adding additional support for applications into the JSP specification.

D.4 JSP, XML and XSL Technologies

The JSP 1.0 and JSP 1.1 specification started in the direction of XML representations of JSP pages. At the time of the design of the JSP 1.0 and JSP 1.1 specifications the Java platform was lacking on APIs for the manipulation of XML documents; this deficiency is being corrected and we expect to exploit the interaction of XML and JSP technologies more fully in future specifications.

APPENDIX **E**

Changes

This appendix lists the changes in the JavaServer Pages specification.

E.1 Changes between 1.1 PR2 and 1.1 final

E.1.1 Changes

- Updated the license.
- Consistent use of the JSP page, JSP container, and similar terms. Some other minor editorial changes.
- Clarified the return values of the methods in the `TagSupport` and `BodyTagSupport` classes.
- Normalized the throws clause in the methods in `Tag`, `BodyTag`, `TagSupport` and `BodyTagSupport`. These methods now throw `JspException`.
- Added a missing `throws IOException` to the `writeOut()` method in `BodyContent`.
- Renamed `JspError` to `JspTagException`.
- The `getTagId()` and `setTagId()` methods in `TagSupport` were renamed to `getId()` and `setId()` to be consistent with the switch to properties.
- Spelled out some consequences of (the unchanged) specification that `flush()` on `BodyContent` raises an exception.
- Clarified the interpretation of the `uri` attribute in the `taglib` directive.

E.2 Changes between 1.1 PR1 and PR2

E.2.1 Additions

- Added a Glossary as Appendix F.
- Normalized use of Container and Component terminology; including changing the name of Chapter 3.
- Described the relationship of this specification to the Servlets and J2EE specifications.
- Expanded on some of the organizational models in Chapter 1 so as to cover 0.92's "model 1" and "model 2".
- Expanded Section 2.7.2 to summarize the implications of threading and distribution from the Servlet spec and to define the notion of a distributable JSP page.
- Added a description of how to package a JSP page within a WAR; changed the title of Appendix C to reflect the new material.

E.2.2 Changes

- A tag handler is now a JavaBean component; attributes are properties that have been explicitly marked as attributes in the TLD.
- The type subelement of attribute in the TLD is now defined by the type of the corresponding JavaBean component property, and has been removed from the TLD.
- Clarified implicit import list in Section 2.7.1. Clarified details on Section 2.12.1, Section 2.13.6 and Section 3.4.2.
- The names of the DTDs have changed to reflect that the JSP and Servlet specifications have a separate release vehicle to J2EE. The new names are web-jsptaglibrary_1_1.dtd and web-app_2_2.dtd.
- Decomposed the Tag abstract class into two interfaces and two support classes.
- Adjusted the semantics of the uri attribute in taglib, and the mechanism by which a tag library descriptor is located.
- Normalized the terminology on the Tag Extension mechanism.
- Indicated that a "compiled" JSP page should be packaged with any support classes it may use.
- `BodyJspWriter` is now `BodyContent` to clarify its meaning; this is similar to the PD name. The name `BodyJspWriter` was confusing some readers.
- Corrected implementation examples to show how a JSP page implementation class invokes `getDefaultFactory` only statically.

- Reorganized the material in Section B.3 for accuracy and presentation.

E.3 Changes between 1.1 PD1 and PR1

E.3.1 Additions

- Added a Tag Library Descriptor (TLD) file
- Added parameters to `jsp:include` and `jsp:forward`..
- Added `JspException` and `JspError` classes.
- Added a parent field to the `Tag` class to provide a runtime stack.
- Added pushBody() and popBody() to PageContext.
- Added appendix with an example of compiling a simple JSP page into a Servlet that is delivered within a WAR
- Upgraded the javadoc documentation
- Upgraded all the examples.
- Added a precompilation protocol.
- Reserved all request parameters starting with "jsp".

E.3.2 Changes

- Most Info classes are not to be subclassed; instead their information is now derived completely from the TLD file; TagExtraInfo is the exception.
- BodyEvaluation is now called BodyJspWriter and it is a subclass of JspWriter.
- Tag is now an abstract class; TagSupport has been removed. NodeData is now called TagData.
- Split `doBody()` into `doBeforeBody()` and `doAfterBody()` to simplify programming.
- The semantics of the nested JspWriter have changed: now there is only at most one BodyJspWriter per invocation of the action, regardless of how many times the body is evaluated.
- Return type of `doStartTag()` is now an int for better documentation and ease of extensibility.
- Added `initialize()` and `release()` methods to Tag class; clarified life-cycle requirements.

- Substantial cleanup of presentation; revisions to many classes.

E.3.3 Deletions

- Removed the ChildrenMap mechanism.
- Removed the flush="false" option in `jsp:include` as it cannot be implemented on Servlet 2.2.
- Removed the proposal for a standard Servlet tag for now. Will probably be available in a "utils" tag library.

E.4 Changes between 1.0 and 1.1 PD1

The JSP 1.1 specification builds on the JSP 1.0 specification.

E.4.1 Additions

- Enabled the compilation of JSP pages into Servlet classes that can be transported from one JSP container to another.
- Added a portable tag extension mechanism.
- Flush is now an optional attribute of jsp:include, and a false value is valid and the default.

E.4.2 Changes

- Use Servlet 2.2 instead of Servlet 2.1 (as clarified in Appendix B); Servlet 2.2 is still being finalized, but the specification is intended to be upward compatible.
- `jsp:plugin` no longer can be implemented by just sending the contents of `jsp:fallback` to the client.

E.4.3 Removals

- None so far.

APPENDIX **F**

Glossary

This appendix is a glossary of the main concepts mentioned in this specification.

action An element in a JSP page that can act on implicit objects and other server-side objects or can define new scripting variables. Actions follow the XML syntax for elements with a start tag, a body and an end tag; if the body is empty it can also use the empty tag syntax. The tag must use a prefix.

action, standard An action that is defined in the JSP specification and is always available to a JSP file without being imported.

action, custom An action described in a portable manner by a tag library descriptor and a collection of Java classes and imported into a JSP page by a taglib directive.

Application Assembler A person that combines JSP pages, servlet classes, HTML content, tag libraries, and other Web content into a deployable Web application.

component contract The contract between a component and its container, including life cycle management of the component and the APIs and protocols that the container must support.

Component Provider A vendor that provides a component either as Java classes or as JSP page source.

distributed container A JSP container that can run a Web application that is tagged as distributable and is spread across multiple Java virtual machines that might be running on different hosts.

declaration An scripting element that declares methods, variables, or both in a JSP page. Syntactically it is delimited by the <%! and %> characters.

directive An element in a JSP page that gives an instruction to the JSP container and is interpreted at translation time. Syntactically it is delimited by the <%@ and %> characters.

element A portion of a JSP page that is recognized by the JSP translator. An element can be a directive, an action, or a scripting element.

expression A scripting element that contains a valid scripting language expression that is evaluated, converted to a `String`, and placed into the implicit `out` object. Syntactically it is delimited by the <%= and %> characters

fixed template data Any portions of a JSP file that are not described in the JSP specification, such as HTML tags, XML tags, and text. The template data is returned to the client in the response or is processed by a component.

implicit object A server-side object that is defined by the JSP container and is always available in a JSP file without being declared. The implicit objects are `request`, `response`, `pageContext`, `session`, `application`, `out`, `config`, `page`, and `exception`.

JavaServer Pages technology An extensible Web technology that uses template data, custom elements, scripting languages, and server-side Java objects to return dynamic content to a client. Typically the template data is HTML or XML elements, and in many cases the client is a Web browser.

JSP container A system-level entity that provides life cycle management and runtime support for JSP and Servlet components.

JSP file A text file that contains a JSP page. In the current version of the specification, the JSP file must have a *.jsp* extension.

JSP page A text-based document that uses fixed template data and JSP elements and describes how to process a *request* to create a *response*. The semantics of a JSP page are realized at runtime by a JSP page implementation class.

JSP page, front A JSP page that receives an HTTP request directly from the client. It creates, updates, and/or accesses some server-side data and then forwards the request to a presentation JSP page.

JSP page, presentation A JSP page that is intended for presentation purposes only. It accesses and/or updates some server-side data and incorporates fixed template data to create content that is sent to the client.

JSP page implementation class The Java programming language class, a Servlet, that is the runtime representation of a JSP page and which receives the *request* object and updates the *response* object. The page implementation class can use the services provided by the JSP container, including both the Servlet and the JSP APIs.

JSP page implementation object The instance of the JSP page implementation class that receives the *request* object and updates the *response* object.

scripting element A declaration, scriptlet, or expression, whose tag syntax is defined by the JSP specification, and whose content is written according to the scripting language used in the JSP page. The JSP specification describes the syntax and semantics for the case where the language page attribute is "java".

scriptlet An scripting element containing any code fragment that is valid in the scripting language used in the JSP page. The JSP specification describes what is a valid scriptlet for the case where the language page attribute is "java". Syntactically a scriptlet is delimited by the <% and %> characters.

tag A piece of text between a left angle bracket and a right angle bracket that has a name, can have attributes, and is part of an element in a JSP page. Tag names are known to the JSP translator, either because the name is part of the JSP specification (in the case of a standard action), or because it has been introduced using a Tag Library (in the case of custom action).

tag handler A JavaBean component that implements the `Tag` or `BodyTag` interfaces and is the run-time representation of a custom action.

tag library A collection of custom actions described by a tag library descriptor and Java classes.

tag library descriptor An XML document describing a tag library.

Tag Library Provider A vendor that provides a tag library. Typical examples may be a JSP container vendor, a development group within a corporation, a component vendor, or a service vendor that wants to provide easier use of their services.

Web application An application built for the Internet, an intranet, or an extranet.

Web application, distributable A Web application that is written so that it can be deployed in a Web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the `distributable` element.

Web Application Deployer A person who deploys a Web application in a Web container, specifying at least the root prefix for the Web application, and in a J2EE environment, the security and resource mappings.

Web component A servlet class or JSP page that runs in a JSP container and provides services in response to requests.

Web Container Provider A vendor that provides a servlet and JSP container that support the corresponding component contracts.