BEA WebLogic Server™

Programming WebLogic XML

BEA WebLogic Server Version 6.1
Document Date: June 24, 2002

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming WebLogic XML

Part Number	Document Date	Software Version
N/A	June 24, 2002	BEA WebLogic Server Version 6.1

Contents

2. Developing XML Applications with WebLogic Server

3. XML Programming Techniques

4. Administering WebLogic Server XML

5. XML Reference

Index

About This Document

This document explains how to use the BEA WebLogic Server™ XML software. It defines concepts associated with using the XML software and describes the development process for XML applications. In addition, the document includes descriptions of the application programming interfaces (APIs), administrative tasks, and XML tools.

The document is organized as follows:

- Chapter 1, “XML Overview,” provides a basic description of the XML software and its components.
- Chapter 2, “Developing XML Applications with WebLogic Server,” describes how to develop XML applications using WebLogic Server and XML tools.
- Chapter 3, “XML Programming Techniques,” describes specific programming techniques for tasks such as using message-driven beans and JMS queues with XML documents, and so on.
- Chapter 4, “Administering WebLogic Server XML,” describes the Administration Console XML Registry and how to perform XML configuration tasks.
- Chapter 5, “XML Reference,” provides pointers to specifications and application programming interfaces supported by the XML software.

Audience

This document is written for system administrators and programmers who design, develop, configure, and manage XML applications. It is assumed that readers know Web technologies, XML, XSLT, the Java programming language, and the Servlet and JSP APIs of the J2EE specification.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the WebLogic Server Product Documentation page at http://e-docs.bea.com/wls/docs61.

How to Print the Document

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

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

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

Related Information

For related information about XML, see “Learning About XML” on page 1-15.

Contact Us!

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

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`

Convention	Usage

|	Separates mutually exclusive choices in a syntax line. *Example*:
	`java weblogic.deploy [list|deploy|undeploy|update]`
	`password {application} {source}`

. . .	Indicates one of the following in a command line: ■ An argument can be repeated several times in the command line. ■ The statement omits additional optional arguments. ■ You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

1 XML Overview

The following sections provide an overview of XML technology and the WebLogic Server XML subsystem:

- What Is XML?
- How Do You Describe an XML Document?
- Why Use XML?
- What Are XSL and XSLT?
- What Are DOM and SAX?
- What Is JAXP?
- Common Uses of XML and XSLT
- WebLogic Server XML Features
- Learning About XML

What Is XML?

Extensible Markup Language (XML) is a markup language used to describe the content and structure of data in a document. It is a simplified version of Standard Generalized Markup Language (SGML). XML is an industry standard for delivering content on the Internet. Because it provides a facility to define new tags, XML is also extensible.

Like HTML, XML uses tags to describe content. However, rather than focusing on the presentation of content, the tags in XML describe the meaning and hierarchical structure of data. This functionality allows for the sophisticated data types that are required for efficient data interchange between different programs and systems. Further, because XML enables separation of content and presentation, the content, or data, is portable across heterogeneous systems.

The XML syntax uses matching start and end tags (such as `<name>` and `</name>`) to mark up information. Information delimited by tags is called an element. Every XML document has a single root element, which is the top-level element that contains all the other elements. Elements that are contained by other elements are often referred to as sub-elements. An element can optionally have attributes, structured as name-value pairs, that are part of the element and are used to further define it.

The following sample XML file describes the contents of an address book:

```
<?xml version="1.0"?>

<address_book>
  <person gender="f">
    <name>Jane Doe</name>
    <address>
      <street>123 Main St.</street>
      <city>San Francisco</city>
      <state>CA</state>
      <zip>94117</zip>
    </address>
    <phone area_code=415>555-1212</phone>
  </person>
  <person gender="m">
    <name>John Smith</name>
    <phone area_code=510>555-1234</phone>
    <email>johnsmith@somewhere.com</email>
  </person>
</address_book>
```

The root element of the XML file is the `address_book`. The address book currently contains two entries in the form of `person` elements: Jane Doe and John Smith. Jane Doe's entry includes her address and phone number; John Smith's includes his phone and email address. Note that the structure of the XML document defines the `phone` element as storing the area code using the `area_code` attribute rather than a sub-element in the body of the element. Also note that not all sub-elements are required for the `person` element.

How Do You Describe an XML Document?

There are two ways to describe an XML document: DTDs and Schemas.

Document Type Definitions (DTDs) define the basic requirements on the structure of an XML document. A DTD describes the elements and attributes that are valid in an XML document, and the contexts in which they are valid. In other words, a DTD specifies which tags are allowed within certain other tags, and which tags and attributes are optional.

The following example shows a DTD that describes the preceding address book sample XML document:

```
<!DOCTYPE address_book [
<!ELEMENT person (name, address?, phone?, email?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (street, city, state, zip)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>

<!ATTLIST person gender CDATA #REQUIRED>
<!ATTLIST phone area_code CDATA #REQUIRED>
]>
```

Schemas are a recent development in XML specifications and are intended to supersede DTDs. They describe XML documents with more flexibility and detail than DTDs do, and are XML documents themselves, which DTDs are not. The schema specification, currently under development, is a product of the World Wide Web Consortium (W3C) and is intended to address many limitations of DTDs. For detailed information on XML schemas, see http://www.w3.org/TR/xmlschema-0/.

The following example shows a schema that describes the preceding address book sample XML document:

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<xsd:element     name="address_book" type="bookType"/>

<xsd:complexType name="bookType">
  <xsd:element name=name="person"     type="personType"/>
</xsd:complexType>
```

```
<xsd:complexType name="personType">
  <xsd:element   name="name"     type="xsd:string"/>
  <xsd:element   name="address"  type="addressType"/>
  <xsd:element   name="phone"    type="phoneType"/>
  <xsd:element   name="email"    type="xsd:string"/>
  <xsd:attribute name="gender"   type="xsd:string"/>
</xsd:complexType>

<xsd:complexType name="addressType">

  <xsd:element   name="street"  type="xsd:string"/>
  <xsd:element   name="city"    type="xsd:string"/>
  <xsd:element   name="state"   type="xsd:string"/>
  <xsd:element   name="zip"     type="xsd:string"/>
</xsd:complexType>

<xsd:simpleType name="phoneType">
  <xsd:restriction base="xsd:string"/>
  <xsd:attribute name="area_code" type="xsd:string"/>
</xsd:simpleType>

</xsd:schema>
```

An XML document can include a DTD or Schema as part of the document itself, reference an external DTD or Schema using the DOCTYPE declaration, or not include or reference a DTD or Schema at all. The following excerpt from an XML document shows how to reference an external DTD called `address.dtd`:

```
<?xml version=1.0?>
<!DOCTYPE address_book SYSTEM "address.dtd">
<address_book>
...
```

XML documents only need to be accompanied by a DTD or Schema if they need to be validated by a parser or if they contain complex types. An XML document is considered *valid* if 1) it has an associated DTD or Schema, and 2) it complies with the constraints expressed in the associated DTD or Schema. If, however, an XML document only needs to be *well-formed*, then the document does not have to be accompanied by a DTD or Schema. A document is considered well-formed if it follows all the rules in the W3C Recommendation for XML 1.0. For the full XML 1.0 specification, see http://www.w3.org/XML/.

Why Use XML?

An industry typically uses data exchange methods that are meaningful and specific to that industry. With the advent of e-commerce, businesses conduct an increasing number of relationships with a variety of industries and, therefore, must develop expert knowledge of the various protocols used by those industries for electronic communication.

The extensibility of XML makes it a very effective tool for standardizing the format of data interchange among various industries. For example, when message brokers and workflow engines must coordinate transactions among multiple industries or departments within an enterprise, they can use XML to combine data from disparate sources into a format that is understandable by all parties.

What Are XSL and XSLT?

The Extensible Stylesheet Language (XSL) is a W3C standard for describing presentation rules that apply to XML documents. XSL includes both a transformation language, (XSLT), and a formatting language. These two languages function independently of each other. XSLT is an XML-based language and W3C specification that describes how to transform an XML document into another XML document, or into HTML, PDF, or some other document format.

An XSLT transformer accepts as input an XML document and an XSLT document. The template rules contained in an XSLT document include patterns that specify the XML tree to which the rule applies. The XSLT transformer scans the XML document for patterns that match the rule, and then it applies the template to the appropriate section of the original XML document.

What Are DOM and SAX?

DOM and SAX are two Java application programming interfaces (APIs) for parsing XML data. The two APIs differ in their approach to parsing, with each API having its strengths and weaknesses.

SAX

SAX stands for the *Simple API for XML*. It is a standard interface for event-based XML parsing. SAX defines events that can occur as a parser is reading through an XML document, such as the start or the end of an element. Programmers provide handlers to deal with different events as the document is parsed.

Programmers that use the SAX API to parse XML documents have full control over what happens when these events occur and can, as a result, customize the parsing process extensively. For example, a programmer might decide to stop parsing an XML document as soon as the parser encounters an error that indicates that the document is invalid, rather than waiting until the entire document is parsed, thus improving performance.

The WebLogic Server built-in parser (Apache Xerces) supports SAX Version 2.0. Programmers who have created programs that use Version 1.0 of SAX to parse XML documents should read about the changes between the two versions and update their programs accordingly.

DOM

DOM stands for the *Document Object Model*. DOM reads an XML document into memory and represents it as a tree; each node of the tree represents a particular piece of data from the original XML document. Because the tree structure is a standard programming mechanism for representing data, traversing and manipulating the tree using Java is relatively easy, fast, and efficient. The main drawback, however, is that the entire XML document has to be read into memory for DOM to create the tree, which might decrease the performance of an application as the XML documents get larger.

The WebLogic Server built-in parser (Apache Xerces) supports DOM Level 2.0 Core. Programmers who have created programs that use Level 1.0 of DOM to parse XML documents should read about the changes between the two versions and update their programs accordingly. For detailed information about the differences, refer to http://www.w3.org/DOM/DOMTR.

What Is JAXP?

The previous section discusses two APIs, SAX and DOM, that programmers can use to parse XML data. The *Java API for XML Processing* (JAXP) provides a means to get to these parsers. JAXP also defines a Plugability layer that allows programmers to use any compliant parser or transformer.

To facilitate XML application development and the work required to move XML applications built on WebLogic Server to other Web application servers, WebLogic Server implements the Java API for XML Processing (JAXP). JAXP was developed by Sun Microsystems to make XML applications portable; it provides basic support for parsing and transforming XML documents through a standardized set of Java platform APIs. JAXP 1.1, included in the WebLogic Server distribution, is configured to use the built-in parser. Therefore, by default, XML applications built using WebLogic Server use JAXP.

The WebLogic Server distribution contains the interfaces and classes needed for JAXP 1.1. JAXP 1.1 contains explicit support for SAX Version 2 and DOM Level 2. The Javadoc for JAXP is included with the WebLogic Server online reference documentation.

JAXP Packages

JAXP contains the following two packages:

- `javax.xml.parsers`
- `javax.xml.transform`

The javax.xml.parsers package contains the classes to parse XML data in SAX Version 2.0 and DOM Level 2.0 mode. To parse an XML document in SAX mode, a programmer first instantiates a new SaxParserFactory object with the newInstance() method. This method looks up the specific implementation of the parser to load based on a well-defined list of locations. The programmer then obtains a SaxParser instance from the SaxParserFactory and executes its parse() method, passing it the XML document to be parsed. Parsing an XML document in DOM mode is similar, except that the programmer uses the DocumentBuilder and DocumentBuilderFactory classes instead.

For detailed information on using JAXP to parse XML documents, see "Parsing XML Documents" on page 2-2.

The javax.xml.transform package contains classes to transform XML data, such as an XML document, a DOM tree, or SAX events, into a different format. The transformer classes work similarly to the parser classes. To transform an XML document, a programmer first instantiates a TransformerFactory object with the newInstance() method. This method looks up the specific implementation of the XSLT transformer to load based on a well-defined list of locations. The programmer then instantiates a new Transformer object based on a specific XSLT style sheet and executes its transform() method, passing it the XML object to transform. The XML object might be an XML file, a DOM tree, and so on.

For detailed information on using JAXP to transform XML objects, see "Using JAXP to Transform XML Data" on page 2-13.

Common Uses of XML and XSLT

How you use XML and XSLT depends on your particular business needs.

Using XML and XSLT to Separate Content from Presentation

XML and XSLT are often used in applications that support multiple client types. For example, suppose you have a Web-based application that supports both browser-based clients and Wireless Application Protocol (WAP) clients. These clients understand different markup languages, HTML and Wireless Markup Language (WML), respectively, but your application must deliver content that is appropriate for both.

To accomplish this goal, you can write your application to first produce an XML document that represents the data it is sending to the client. Then the application can transform the XML document that represents the data into HTML or WML, depending on the client's browser type. Your application can determine the client browser type by examining the `User-Agent` request header of an HTTP request. Once the application knows the client browser type, it uses the appropriate XSLT style sheet to transform the document into the correct markup language. See the SnoopServlet example included in the `examples/servlets` directory of your WebLogic Server distribution for an example of how to access this type of header information.

This method of rendering the same XML document using different markup languages in respective client types helps concentrate the effort required to support multiple client types into the development of the appropriate XSLT style sheets. Additionally, it allows your application to adapt to other clients types easily, if necessary.

For additional information about XSLT, see "Additional Resources" on page 5-4.

XML as a Message Format for Business-to-Business Communication

In a business-to-business (B2B) environment, Company A and Company B want to exchange information about e-commerce transactions in which both are involved. Company A is a major e-commerce site. Company B is a small affiliate that sells Company A's products to a niche group of customers. When Company B sends customers to Company A, Company B is compensated in two ways: it receives, from Company A, both money and information about other customers that make the same

sort of purchases as those made by the customers referred by Company B. To exchange information, Company A and Company B must agree on a data format for information that is machine readable and that operates with systems from both companies easily.

XML is the logical data format to use in this scenario, but selecting this format is only the first step. The companies must then agree on the format of the XML messages to be exchanged. Because Company A has a one-to-many relationship with its affiliates, Company A must define the format of the XML messages that will be exchanged.

To define the format of XML messages, or XML documents, Company A creates two document type definitions (DTDs): one that describes the information that A will provide about customers and one that describes the information that A wants to receive about a newly affiliated company. Company B must also create two DTDs: one to process the XML documents received from Company A and one to prepare an XML document in a format that can be processed by Company A.

WebLogic Server XML Features

WebLogic Server consolidates XML technologies applicable to WebLogic Server and XML applications based on WebLogic Server. The WebLogic Server XML subsystem allows customers to use standard parsers, a the WebLogic FastParser, XSLT transformers, and DTDs and XML Schemas to process and convert XML files.

The WebLogic Server XML subsystem includes the following features:

- XML Document Parsers
- XML Document Transformer
- JAXP Plugability Layer Implementation
- WebLogic Servlet Attributes
- WebLogic XSLT JSP Tag Library
- XML Registry For Configuring Parsers and Transformers
- XML Registry for Configuring External Entity Resolution
- Code Examples

XML Document Parsers

WebLogic Server includes the following two parsers:

Parser	Description
Built-in	The built-in parser is based on the Apache Xerces parser version 1.3.1. You can use the built-in parser in either Simple API For XML (SAX) mode or Document Object Model (DOM) mode.
WebLogic FastParser	A high-performance XML parser specifically designed for processing small to medium size documents, such as SOAP and WSDL files associated with WebLogic Web services. Configure WebLogic Server to use FastParser if your application mostly handles small to medium size (up to 10,000 elements) XML documents. **Note:** Previous versions of WebLogic Server included the ability to create custom parsers. Because WebLogic FastParser can be used for the types of XML documents that customized parsers were meant for, WebLogic FastParser effectively replaces the customized parser feature, and the ability to generate a customized parser has been deprecated. For detailed information on using WebLogic FastParser, refer to "Using the WebLogic FastParser" on page 2-9.

You can also use any other XML parser of your choice by configuring it in the XML Registry using the Administration Console. You can configure a single instance of WebLogic Server to use one parser for a particular application and use another parser for a different application.

XML Document Transformer

WebLogic Server includes a built-in XSLT transformer that is based on the Apache Xalan XSLT transformer version 2.0.1. You can use this built-in XSLT transformer or other XSLT transformers in your XML application to transform XML documents. For more information about transforming XML documents, see "Using JAXP to Transform XML Data" on page 2-13.

JAXP Plugability Layer Implementation

Java API for XML Processing (JAXP) 1.1 is a Java-standard, parser-independent API for XML. For more information on JAXP, see “What Is JAXP?” on page 1-7.

Note: WebLogic Server uses the XML Registry, accessed through the Administration console, to plug-in parsers and transformers rather than using system properties, as defined by the JAXP 1.1 specification.

WebLogic Servlet Attributes

WebLogic Server supports the following special Servlet attributes:

- `org.xml.sax.HandlerBase`
- `org.xml.sax.helpers.DefaultHandler`
- `org.w3c.dom.Document`

Calling the `setAttribute` (for SAX parsing) and `getAttribute` (for DOM parsing) methods on a `ServletRequest` object with the preceding attributes will parse any given XML document.

The following code sections show an example of how to use these methods:

```
request.setAttribute("org.xml.sax.helpers.DefaultHandler", new DefHandler());

org.w3c.dom.Document = (Document)request.getAttribute("org.w3c.dom.Document");
```

Note: The `setAttribute` and `getAttribute` methods are provided for convenience only; they are not required to parse XML from a Servlet.

WebLogic XSLT JSP Tag Library

The JSP tag library provides a simple tag that enables access to the built-in XSLT transformer from within a Java Server Page (JSP) running on WebLogic Server. Currently, this tag supports the built-in XSLT transformer only; you cannot use the tag to transform an XML document from within a JSP using a different transformer.

The JSP tag library is included in `xmlx-tags.jar`, which is installed when you install your WebLogic Server distribution.

Note: The JSP tag library is provided for convenience only; it is not required to access XSLT transformers from within a JSP.

XML Registry For Configuring Parsers and Transformers

The XML Registry simplifies administration and configuration tasks by separating these tasks from the XML application. Use the Administration Console (a graphical user interface, or GUI, for WebLogic Server administration) to configure the parsers and transformers for an instance of WebLogic Server.

Note: Each WebLogic Server domain can include any number of registries; each WebLogic Server in a domain can be assigned zero or one registry.

By using the XML Registry, you:

- Can specify the parser or transformer at deployment time, not only at build time.
- Do not need to include any parser- or transformer- dependent code in your applications.
- Can support multiple parsers and transformers in a single server more conveniently.

You can use the XML Registry to perform the following tasks:

- Configure an alternative XML parser instead of the built-in parser shipped in this version of WebLogic Server.
- Configure an alternative XSLT transformer instead of the built-in transformer shipped in this version of WebLogic Server.
- Configure an XML parser that should be used to process a particular document type.

All the preceding capabilities are available if your application uses the standard Java API for XML Processing (JAXP), which is included in this version of WebLogic Server. These capabilities are for use on the server side only.

XML Registry for Configuring External Entity Resolution

WebLogic XML supports external entity resolution through the XML Registry. An example of an external entity is a DTD file that is used to validate an XML document. To use this feature, open the Administration Console and use the XML Registry to enter the `Public ID` or `System ID` associated with the external entity.

In addition to storing external entities locally, you can configure WebLogic Server to retrieve and cache external entities from external repositories that support an HTTP interface, such as a URL. You can configure WebLogic Server to cache the external entity in memory or on the disk and specify how long the entity should remain cached before it is considered out of date.

For more information about using the XML Registry for external entity resolution, see "External Entity Configuration Tasks" on page 4-11.

Code Examples

WebLogic Server includes examples of parsing and transforming XML documents.

The examples are located in the ***BEA_HOME*/samples/examples/xml** directory, where ***BEA_HOME*** refers to the main WebLogic server installation directory.

For detailed instructions on how to build and run the examples, invoke the Web page *BEA_HOME*`/samples/examples/xml/package-summary.html` in your browser.

Editing XML Files

To edit XML files, use the BEA XML Editor, an entirely Java-based XML stand-alone editor. It is a simple, user-friendly tool for creating and editing XML files. It displays XML file contents both as a hierarchical XML tree structure and as raw XML code. This dual presentation of the document provides you with the following two methods of editing the XML document:

- The hierarchical tree view allows structured, limited constrained editing, providing you with a set of allowable functions at each point in the hierarchical

XML tree structure. The allowable functions are syntactically dictated and in accordance with the XML document's DTD or schema, if one is specified.

- The raw XML code view allows free-form editing of the data.

BEA XML Editor can validate XML code according to a specified DTD or XML schema.

For detailed information about using the BEA XML Editor, see its on-line help.

You can download BEA XML Editor from the BEA dev2dev at http://dev2dev.bea.com/resourcelibrary/utilitiestools/xml.jsp?highlight=utilitiestools.

Learning About XML

To learn about XML, see the following online courses and tutorials:

- A Technical Introduction to XML at "http://www.xml.com/pub/a/98/10/guide0.html"
- XML Authoring Tutorial at "http://www.xml.com/pub/r/32."
- Working with XML and Java at "http://java.sun.com/xml/tutorial_intro.html."
- Tutorials for using the Java 2 platform and XML technology at "http://developerlife.com/."
- XML, Java, and the Future of the Web at "http://www.xml.com/pub/a/w3j/s3.bosak.html."
- Chapter 14 of The XML Bible: XSL Transformations at "http://metalab.unc.edu/xml/books/bible/updates/14.html."
- XSL Tutorial by Miloslav Nic at http://zvon.vscht.cz/HTMLonly/XSLTutorial/Books/Book1/index.html.
- SAX 2.0: The Simple API for XML at "http://www.saxproject.org/"
- Document Object Model (DOM) at "http://www.w3.org/DOM/"

CHAPTER

2 Developing XML Applications with WebLogic Server

The following sections describe how to use the Java programming language and WebLogic Server to develop XML applications. It is assumed that you know how to use Java Servlets and Java Server Pages (JSPs) to write Java applications. For information about how to write servlet and JSP applications, see *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs61/servlet/index.html and *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs61/jsp/index.html.

- Developing XML Applications: Main Steps
- Parsing XML Documents
- Generating XML Documents
- Using JAXP to Transform XML Data
- Using the JSP Tag to Transform XML Data
- Using Transformers Other Than the Built-In Transformer

Developing XML Applications: Main Steps

Programmers using the WebLogic Server XML subsystem typically perform some or all of the following programming tasks when developing XML applications:

1. Parse an XML document.

   The XML document can originate from a number of sources. For example, a programmer might develop a Servlet to receive an XML document from a client, write an EJB to receive an XML document from a Servlet or another EJB, and so on. In each instance, the XML document may have to be parsed so that its data can be manipulated.

   For more information on this task, refer to "Parsing XML Documents" on page 2-2.

2. Generate a new XML document.

   After a Servlet or EJB has received and parsed an XML document and possibly manipulated the data in some way, the Servlet or EJB might need to generate a new XML document to send back to the client or to pass on to another EJB.

   For more information on this task, refer to "Generating XML Documents" on page 2-10.

3. Transform XML data into another format.

   After parsing an XML document or generating a new one, the Servlet or EJB may need to transform it into another format, such as HTML, WML, or plain text.

   For more information on this task, refer to "Using JAXP to Transform XML Data" on page 2-13.

Parsing XML Documents

This section describes how to parse XML documents.

As mentioned previously, you use the Administration Console XML Registry to configure the following:

- Per-doctype parsers, which supersede the built-in parser for the specified doctype
- External entity resolution, or the process that an XML parser goes through when requested to find an external file in the course of parsing an XML document

For detailed information on how to use the Administration Console for these tasks, refer to Chapter 4, "Administering WebLogic Server XML."

For a complete example of parsing an XML document in SAX mode, see the *BEA_HOME*/samples/examples/xml/sax directory, where *BEA_HOME* refers to the main WebLogic server installation directory.

Parsing XML Documents Using JAXP in SAX Mode

The following code example shows how to configure a SAX parser factory to create a validating parser. The example also shows how to register the `MyHandler` class with the parser. The `MyHandler` class can override any method of the `DefaultHandler` class to provide custom behavior for SAX parsing events or errors.

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

...
MyHandler handler = new MyHandler();
// MyHandler extends org.xml.sax.helpers.DefaultHandler.

  //Obtain an instance of SAXParserFactory.
  SAXParserFactory spf = SAXParserFactory.newInstance();
  //Specify a validating parser.
  spf.setValidating(true); // Requires loading the DTD.
  //Obtain an instance of a SAX parser from the factory.
  SAXParser sp = spf.newSAXParser();
  //Parse the documnt.
  sp.parse("http://server/file.xml", handler);
...
```

Note: If you want to use a parser other than the built-in parser, you must use the WebLogic Server Administration Console to specify the parser in the XML Registry; otherwise the `SaxParserFactory.newInstance` method returns the built-in parser. For instructions about configuring WebLogic Server to use a parser other than the built-in parser, see "Configuring a Parser or Transformer Other Than the Built-In" on page 4-4.

For a complete example of parsing an XML document in SAX mode, see the *BEA_HOME*/samples/examples/xml/sax directory, where *BEA_HOME* refers to the main WebLogic server installation directory.

Parsing XML Documents Using JAXP in DOM Mode

The following code example shows how to parse an XML document and create an `org.w3c.dom.Document` tree from a `DocumentBuilder` object:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

...
//Obtain an instance of DocumentBuilderFactory.
DocumentBuilderFactory dbf =
                DocumentBuilderFactory.newInstance();
//Specify a validating parser.
dbf.setValidating(true); // Requires loading the DTD.
//Obtain an instance of a DocumentBuilder from the factory.
DocumentBuilder db = dbf.newDocumentBuilder();
//Parse the document.
Document doc = db.parse(inputFile);
...
```

Note: If you want to use a parser other than the built-in parser, you must use the WebLogic Server Administration Console to specify it; otherwise the `DocumentBuilderFactory.newInstance` method returns the built-in parser. For instructions about configuring WebLogic Server to use a parser other than the built-in parser, see "Configuring a Parser or Transformer Other Than the Built-In" on page 4-4.

For a complete example of parsing an XML document in DOM mode, see the *`BEA_HOME`*`/samples/examples/xml/dom` directory, where *`BEA_HOME`* refers to the main WebLogic server installation directory.

Parsing XML Documents in a Servlet

Support for the `setAttribute` and `getAttribute` methods was added to version 2.2 of the Java Servlet Specification. Attributes are objects associated with a request. The request object encapsulates all information from the client request. In the HTTP protocol, this information is transmitted from the client to the server by the HTTP headers and message body of the request.

With WebLogic Server, you can use these methods to parse XML documents. The `setAttribute` method is used for SAX mode parsing; the `getAttribute` method, for DOM mode parsing.

For a complete example of parsing an XML document in a Servlet, see the *BEA_HOME*`/samples/examples/xml/attributes` directory, where *BEA_HOME* refers to the main WebLogic server installation directory.

Using the org.xml.sax.DefaultHandler Attribute to Parse a Document

The following code example shows how to use the `setAttribute` method:

```
import weblogic.servlet.XMLProcessingException;
import org.xml.sax.helpers.DefaultHandler;
...
public void doPost(HttpServletRequest request,
                           HttpServletResponse response)
   throws ServletException, IOException  {
   try {
     request.setAttribute("org.xml.sax.helpers.DefaultHandler",
                          new DefaultHandler());
    } catch(XMLProcessingException xpe) {
      System.out.println("Error in processing XML");
      xpe.printStackTrace();
      return;
    }
...
```

You can also use the `org.xml.sax.HandlerBase` attribute to parse an XML document, although it is deprecated:

```
request.setAttribute("org.xml.sax.HandlerBase",
                     new HandlerBase());
```

Note: This code example shows a simple way to parse a document using SAX and the `setAttribute` method. This method of parsing a document is a WebLogic Server convenience feature, and it is not supported by other servlet vendors. Therefore, if you plan to run your application on other servlet platforms, do not use this feature.

Using the org.w3c.dom.Document Attribute to Parse a Document

The following code example shows how to use the `getAttribute` method.

```
import org.w3c.dom.Document;
import weblogic.servlet.XMLProcessingException;

...

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

try {
   Document doc = request.getAttribute("org.w3c.dom.Document");
 } catch(XMLProcessingException xpe) {
   System.out.println("Error in processing XML");
   xpe.printStackTrace();
   return;
 }
...
```

Note: This code example shows a simple way to parse a document using DOM and the `getAttribute` method. This method of parsing a document is a WebLogic Server convenience feature, and it is not supported by other servlet vendors. Therefore, if you plan to run your application on other servlet platforms, do not use this feature.

Validating and Non-Validating Parsers

As previously discussed, a *well-formed* document is one that is syntactically correct according to the rules outlined in the W3C Recommendation for XML 1.0. A *valid* document is one that follows the constraints specified by its DTD or schema.

A non-validating parser verifies that a document is well-formed, but does not verify that it is valid. The WebLogic FastParser, described in "Using the WebLogic FastParser" on page 2-9, is non-validating by default.

To turn on validation while parsing a document (assuming you are using a validating parser), you must:

- Set the `SAXParserFactory.setValidating()` method to true, as shown in the following example:

  ```
  SAXParserFactory factory = SAXParserFactory.newInstance();
  factory.setValidating(true);
  ```

- Ensure that the XML document you are parsing includes (either in-line or by reference) a DTD or a schema.

Handling Entity Resolution While Parsing an XML Document

This section provides general information about external entities; how they are identified and resolved by an XML parser; and the features provided by WebLogic Server to improve the performance of external entity resolution in your XML applications.

For a complete example of resolving an external entity while parsing an XML document, see the *BEA_HOME*/samples/examples/xml/entityresolution directory, where *BEA_HOME* refers to the main WebLogic server installation directory.

General Information About External Entities

External entities are chunks of text that are not literally part of an XML document, but are referenced inside the XML document. The actual text might reside anywhere - in another file on the same computer or even somewhere on the Web. While parsing a document, if the parser encounters an external entity reference, it fetches the referenced chunk of text, places the text into the XML document, then continues parsing. An example of an external entity is a DTD; rather than including the full text of the DTD in the XML document, the XML document has a reference to the DTD that is stored in a separate file.

There are two ways to identify an external entity: a system identifier and a public identifier. System identifiers use URIs to reference an external entity based on its location. Public identifiers use a publicly declared name to refer the information.

The following example shows how a public identifier is used to reference the DTD for the application.xml file that describes a J2EE application archive (*.ear file):

```
<!DOCTYPE application PUBLIC "-//Sun Microsystems,
Inc.//DTD J2EE Application 1.2//EN">
```

The following example shows a reference to an external DTD by a system identifier only:

```
<!DOCTYPE application SYSTEM
"http://java.sun.com/j2ee/dtds/application_1_2.dtd">
```

Here is a reference that uses both the public and system identifier; note that the keyword SYSTEM is omitted:

```
<!DOCTYPE application
PUBLIC "-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/application_1_2.dtd">
```

Using the WebLogic Server Entity Resolution Features

Use the following WebLogic Server features to improve the performance of external entity resolution in your XML applications:

- Permanently store a copy of an external entity on the computer that hosts the WebLogic Administration Server.
- Specify that WebLogic server automatically retrieve and cache an external entity that resides in an external repository that supports an HTTP interface, such as a URL. You can specify that WebLogic Server cache the entity either in memory or on disk and specify when the cached entry becomes stale, at which point WebLogic Server automatically updates the cached entry.

  Using this feature, you do not have to actually copy the external entity to the local computer. The XML application refers to the actual external entity only at specified time intervals, rather than each time the document is parsed, thus potentially greatly improving the performance of your application while also keeping as up to date with the latest external entity as desired.

You use the XML Registry to create entity resolution entries to identify where the external entry is located (locally or at a URL) and what the caching options are for entities on the Web. You identify the external entity entry using a system or public identifier. Then, in your XML document, when you reference this external entity, WebLogic Server fetches the local copy or the cached copy (whichever you have configured) when parsing the document.

For detailed information on creating external entity registries with the XML Registry, refer to “External Entity Configuration Tasks” on page 4-11.

Using Parsers Other Than the Built-In Parser

If you use JAXP to parse your XML documents, the WebLogic Server XML Registry (which is configured through the Administration Console) offers the following options:

- Accept the built-in parser as the server-wide parser.
- Configure the WebLogic FastParser as the server-wide parser.
- Configure a parser of your choice as the server-wide parser.
- Configure a parser for a particular DTD based on its system or public identifier, or its root tag.

For instructions on how to use the XML Registry to configure parsing options, see "XML Parser and Transformer Configuration Tasks" on page 4-4.

Using the WebLogic FastParser

WebLogic Server includes a high-performance non-validating XML parser (called WebLogic FastParser) specifically designed to parse small to medium (up to 10,000 elements) XML documents. For larger documents, the performance of this parser is comparable to that of other standard parsers, such as Apache Xerces.

Note: WebLogic FastParser supports only SAX-style parsing.

You can specify that WebLogic FastParser be used as the WebLogic Server-wide parser, or just for a particular DOCTYPE by using the XML Registry as described in "XML Parser and Transformer Configuration Tasks" on page 4-4. Set the `SAXParserFactory` field to `weblogic.xml.babel.jaxp.SAXParserFactoryImpl`.

Note: Previous versions of WebLogic Server included the ability to create custom parsers. Because you can use the WebLogic FastParser for the types of XML documents that customized parsers were meant for, FastParser effectively replaces the customized parser feature, and the ability to generate a customized parser has been deprecated.

Generating XML Documents

This section describes how to generate XML documents from a DOM document tree and by using JSP.

Generating XML from a DOM Document Tree

This section describes two ways to create an XML document from a DOM document tree:

- Using the Apache `serialize` classes
- Using the JAXP `Transformer` classes

Using the Apache Serialize Class

To generate an XML document from a DOM document tree, you can use the class `weblogic.apache.xml.serialize` to convert a DOM document tree to XML text. For a description of this class, see Javadoc for `weblogic.apache.xml.serialize`.

The following code example shows how to use this class.

Note: The following example does not use JAXP but rather the Apache proprietary API directly.

```
package test;

import java.io.OutputStreamWriter;
import java.util.Date;
import java.text.DateFormat;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import weblogic.apache.xerces.dom.DocumentImpl;
import weblogic.apache.xml.serialize.DOMSerializer;
import weblogic.apache.xml.serialize.XMLSerializer;

public class WriteXML {

  public static void main(String[] args) throws Exception {
```

```
    // Create a DOM tree.
    Document doc= new DocumentImpl();
    Element message = doc.createElement("message");
    doc.appendChild(message);
    Element text = doc.createElement("text");
    text.appendChild(doc.createTextNode("Hello world."));
    message.appendChild(text);
    Element timestamp = doc.createElement("timestamp");
    timestamp.appendChild(
      doc.createTextNode(
               DateFormat.getDateInstance().format(new Date()))
    );
    message.appendChild(timestamp);

    // Serialize the DOM to XML text and output to stdout.
    DOMSerializer xmlSer =
      new XMLSerializer(new OutputStreamWriter(System.out),null);
    xmlSer.serialize(doc);
  }
}
```

Using the JAXP Transformer Class

You can use the `javax.xml.transform.Transformer` class to serialize a DOM object into an XML stream, as shown in the following example segment:

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

import org.w3c.dom.Document;

import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import java.io.*;

...

TransformerFactory trans_factory =
TransformerFactory.newInstance();
Transformer xml_out = trans_factory.newTransformer();
Properties props = new Properties();
props.put(“method”, “xml”);
xml_out.setOutputProperties(props);
xml_out.transform(new DOMSource(doc), new
StreamResult(System.out));
```

In the example, the `Transformer.transform()` does the work of converting a DOM object into an XML stream. The `transform()` method takes as input a `javax.xml.transform.dom.DOMSource` object, created from the DOM tree stored in the `doc` variable, and converts it into a `javax.xml.transform.stream.StreamResult` object and writes the resulting XML document to the standard output.

Generating XML Documents in a JSP

You typically use JSPs to generate HTML, but you can also use a JSP to generate an XML document.

Using JSPs to generate XML requires that you set the content type of the JSP page as follows:

```
<%@ page contentType="text/xml"%>
    ... XML document
```

The following code shows an example of how to use JSP to generate an XML document:

```
<?xml version="1.0">

<%@ page contentType="text/xml"
import="java.text.DateFormat,java.util.Date" %>

<message>
  <text>
    Hello World.
  </text>
  <timestamp>
<%
out.print(DateFormat.getDateInstance().format(new Date()));
%>
  </timestamp>
</message>
```

For more information about using JSP to generate XML, see `http://java.sun.com/products/jsp/html/JSPXML.html`.

Using JAXP to Transform XML Data

Transformation refers to converting an XML document (the *source* of the transformation) into another format, typically a different XML document, HTML, Wireless Markup Language (WML) (the *result* of the transformation.) Version 1.1 of JAXP provides plugable transformation, which means that you can use any JAXP-compliant transformer engine.

JAXP provides the following interfaces to transform XML data into a variety of formats:

- `javax.xml.transform`: This package contains the generic APIs for transforming documents. This package does not have any dependencies on SAX or DOM and makes the fewest possible assumptions about the format of the source and result.
- `javax.xml.transform.stream`: This package implements stream- and URI-specific transformation APIs. In particular, it defines the `StreamSource` and `StreamResult` classes that enable you to specify `InputStreams` and URLs as the source of a transformation and `OutputStreams` and URLs as the results, respectively.
- `javax.xml.transform.dom`: This package implements DOM-specific transformation APIs. In particular, it defines the `DOMSource` and `DOMResult` classes that enable you to specify a DOM tree as either the source or result, or both, of a transformation.
- `javax.xml.transform.sax`: This package implements SAX-specific transformation APIs. In particular, it defines the `SAXSource` and `SAXResult` classes that enable you to specify `org.xml.sax.ContentHandler` events as either the source or result, or both, of a transformation.

Transformation encompasses many possible combinations of inputs and outputs.

For a complete example of transforming an XML document, see the *`BEA_HOME`*`/samples/examples/xml/xslt` directory, where *`BEA_HOME`* refers to the main WebLogic server installation directory.

Example of Transforming an XML Document Using JAXP

The following example snippet shows how to use JAXP to transform `myXMLdoc.xml` into a different XML document using the `mystylesheet.xsl` stylesheet:

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

Transformer trans;
TransformerFactory factory = TransformerFactory.newInstance();
String stylesheet = “file://stylesheets/mystylesheet.xsl”;
String xml_doc = “file://xml_docs/myXMLdoc.xml”;

trans = factory.newTransformer(new StreamSource(stylesheet));
trans.transform(new StreamSource(xml_doc),
                new StreamResult(System.out));
```

For an example of how to transform a DOM document into an XML stream, see “Using the JAXP Transformer Class” on page 2-11.

Converting From the Xalan API to JAXP 1.1 API

If your application contain Xalan-specific code, BEA recommends that you modify it to use JAXP instead.

This section briefly describes the changes you must make to your XML application in order to convert from the Xalan API to JAXP. The section compares two code segments that perform a similar transformation task: one code segment uses the Xalan API directly and the other uses JAXP.

The following Java code segment uses JAXP:

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;

...

Transformer trans;
TransformerFactory factory = TransformerFactory.newInstance();
```

```
String stylesheet = "file://stylesheets/mystylesheet.xsl";
String xml_doc = "file://xml_docs/myXMLdoc.xml";

trans = factory.newTransformer(new StreamSource(stylesheet));
trans.transform(new StreamSource(xml_doc),
                new StreamResult(System.out));
```

The following Java code segment uses the Xalan API directly:

```
 /*
 * This code example was taken from code examples provided by the
 * Apache Software Foundation. It consists of voluntary
 * contributions made by many individuals on behalf of the Apache
 * Software Foundation and was originally based on software
 * copyright (c) 1999, Lotus Development Corporation.,
 * http://www.lotus.com. For more information on the Apache
 * Software Foundation, please see <http://www.apache.org/>.
 */

import org.apache.xalan.xslt.XSLTProcessorFactory;
import org.apache.xalan.xslt.XSLTInputSource;
import org.apache.xalan.xslt.XSLTResultTarget;
import org.apache.xalan.xslt.XSLTProcessor;

...

XSLTProcessor processor = XSLTProcessorFactory.getProcessor();

String stylesheet = "file://stylesheets/mystylesheet.xsl";
String xml_doc = "file://xml_docs/myXMLdoc.xml";

processor.process(new XSLTInputSource(xml_doc),
                      new XSLTInputSource(stylesheet),
                      new XSLTResultTarget(System.out));
```

The following table summarizes the names of the Xalan and JAXPI interfaces and methods used in the preceding examples to transform XML documents; use this table as a first step toward converting your existing Xalan application to a full JAXP application.

Note: This table does not include an entire mapping between Xalan and JAXP, but rather covers only the main classes and methods used in the preceding examples. Refer to the Apache and Sun Web sites at http://www.apache.org and http://java.sun.com/xml/index.html for more detailed information on each API.

Description of Class or Interface	Xalan 1.X	JAXP 1.1
Main class used to transform XML documents	`XSLTProcessor`	`Transformer`
Factory class used to create the transformer objects	`XSLTProcessorFactory`	`TransformerFactory`
Method used to create a new instance of the factory	n/a	`TransformerFactory.newInstance()`
Method used to create a new transformer object	`XSLTProcessorFactory.getProcessor()`	`TransformerFactory.newTransformer()`
Class that holds the source of the transformation, such as the XML document or an XSL stylesheet	`XSLTInputSource`	`StreamSource`
Class that holds the result of the transformation	`XSLTResultTarget`	`StreamResult`
Method that performs the transformation	`XSLTProcessor.process()`	`Transformer.transform()`

Using the JSP Tag to Transform XML Data

WebLogic Server provides a small JSP tag library for convenient access to an XSLT transformer from within a JSP. You can use this tag to transform XML documents into HTML, WML, and so on, but it is not required.

The JSP tag library consists of one main tag, `x:xslt`, and two subtags you can use within the `x:xslt` tag: `x:stylesheet` and `x:xml`.

XSLT JSP Tag Syntax

The XSLT JSP tag syntax is based on XML. A JSP tag consists of a start tag, an optional body, and a matching end tag. The start tag includes the element name and optional attributes.

The following syntax describes how to use the three XSLT JSP tags provided by WebLogic Server in a JSP. The attributes are optional, as are the subtags `x:stylesheet` and `x:xml`. The tables following the syntax describe the attributes of the `x:xslt` and `x:stylesheet` tags; the `x:xml` tag does not have any attributes.

```
<x:xslt  [xml="uri of XML file"]
         [media="media type to determine stylesheet"]
         [stylesheet="uri of stylesheet"]
    <x:xml>In-line XML goes here
    </x:xml>
    <x:stylesheet [media="media type to determine stylesheet"]
                  [uri="uri of stylesheet"]
    </x:stylesheet>
</x:xslt>
```

The following table describes the attributes of the `x:xslt` tag.

x:xslt Tag Attribute	Required	Data Type	Description
xml	No	String	Specifies the location of the XML file that you want to transform. The location is relative to the document root of the Web application in which the tag is used.
media	No	String	Defines the document output type, such as HTML or WML, that determines which stylesheet to use when transforming the XML document. This attribute can be used in conjunction with the `media` attribute of any enclosed `x:stylesheet` tags within the body of the `x:xslt` tag. The value of the `media` attribute of the `x:xslt` tag is compared to the value of the `media` attribute of any enclosed `x:stylesheet` tags. If the values are equal, then the stylesheet specified by the `uri` attribute of the `x:stylesheet` tag is applied to the XML document. **NOTE**: It is an error to set both the `media` and `stylesheet` attributes within the same `x:xslt` tag.

x:xslt Tag Attribute	Required	Data Type	Description
stylesheet	No	String	Specifies the location of the stylesheet to use to transform the XML document. The location is relative to the document root of the Web application in which the tag is used. **NOTE**: It is an error to set both the `media` and `stylesheet` attributes within the same `x:xslt` tag.

The following table describes the attributes of the `x:stylesheet` tag.

x:stylesheet Tag Attribute	Required	Data Type	Description
media	No	String	Defines the document output type, such as HTML or WML, that determines which stylesheet to use when transforming the XML document. Use this attribute in conjunction with the `media` attribute of enveloping x:xslt tag. The value of the `media` attribute of the `x:xslt` tag is compared to the value of the `media` attribute of the enclosed `x:stylesheet` tags. If the values are equal, then the stylesheet specified by the `uri` attribute of the `x:stylesheet` tag is applied to the XML document.
uri	No	String	Specifies the location of the stylesheet to use when the value of the `media` attribute matches the value of the `media` attribute of the enveloping `x:xslt` tag. The location is relative to the document root of the Web application in which the tag is used.

XSLT JSP Tag Usage

The `x:xslt` tag can be used with or without a body, and its attributes are optional. This section describes the rules that dictate how the tag behaves depending on whether you specify a body or one or more attributes.

If the `x:xslt` JSP tag is an empty tag (no body), the following statements apply:

- If no attributes are set, the XML document is processed using the servlet path and the default media stylesheet. You specify the default media stylesheet in

your XML file with the `<?xml-stylesheet>` processing instruction; the default stylesheet is the one that does not have a `media` attribute.

This type of processing allows you to register the JSP page that contains the tag extension as a file servlet that performs XSLT processing.

- If only the `media` attribute is set, the XML document is processed using the servlet path and the specified media type. The value of the `media` type attribute of the `x:xslt` tag is compared to the value of the `media` attribute of any `<?xml-stylesheet>` processing instructions in your XML document; if any match then the corresponding stylesheet is applied. If none match then the default media stylesheet is used. The media type attribute is used to define the document output type (for example, XML, HTML, postscript, or WML). This feature enables you to organize stylesheets by document output type.
- If only the `xml` attribute is set, the specified XML document is processed using the default media stylesheet.
- If the `media` and `xml` attributes are set, the specified XML document is processed using the specified media type.
- If the `stylesheet` attribute is defined, the XML document is processed using the specified stylesheet.

Caution: It is an error to set both the `media` and `stylesheet` attributes within the same `x:xslt` tag.

An XSLT JSP tag that has a body may contain `<x:xml>` tags and/or `<x:stylesheet>` tags. The following statements apply:

- The `<x:xml>` tag allows you specify an XML document for inline processing. This tag has no attributes.
- The `<x:stylesheet>` tag, when used without any attributes, allows you specify the default stylesheet inline.
- Use the `uri` attribute of the `<x:stylesheet>` tag to specify the location of the default stylesheet.
- If you want to specify different stylesheets for different media types, you can use multiple `<x:stylesheet>` tags with different values for the `media`

attribute. You can specify a stylesheet for each media type in the body of the tag, or specify the location of the stylesheet with the `uri` attribute.

Transforming XML Documents Using an XSLT JSP Tag

To use an XSLT JSP tag to transform XML documents, perform the following steps:

1. Open the `xmlx.zip` file in the *`BEA Home`*`/wlserver6.0/ext` directory; extract the `xmlx-tags.jar` file; and put it in the `/lib` directory of your Web application, where *`BEA Home`* is the top-level directory in which you installed the WebLogic Server distribution.

2. Add a `<taglib>` entry to the `web.xml` file. For example:

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

3. To use the tags, add the following line to your JSP page:

   ```
   <%@ taglib uri="xmlx.tld" prefix="x"%>
   ```

4. Configure the transformer. The following procedure shows a generic way to configure the transformer:

   a. Enter the following code line to create an `xslt.jsp` file:

   ```
   <%@ taglib uri="xmlx.tld" prefix="x"%><x:xslt/>
   ```

   b. Register the `xslt.jsp` file in your `web.xml` file, as follows:

   ```
   <servlet>
      <servlet-name>myxsltinterceptor</servlet-name>
      <jsp-file>xslt.jsp</jsp-file>
   </servlet>
   <servlet-mapping>
      <servlet-name>myxsltinterceptor</servlet-name>
      <url-pattern>/xslt/*</url-pattern>
   </servlet-mapping>
   ```

   c. Put your XML/DTD/XSL documents or servlets in your Web application.

d. Add an `xslt` prefix to the pathname for the XML document (for example, change `docs/fred.xml` to `xslt/docs/fred.xml`) and then access the document. Because of the `<url-pattern>` entry in the `web.xml` file, WebLogic Server automatically runs the XSLT transformer on the XML document and sets the default stylesheet in the document.

e. To define media type, add code to the JSP to determine the media type for the XML document and the content type for the output.

f. Pass the media type into the `xslt` tag and then set the content type of the response object.

Note: The other forms of the XSLT JSP tag are used when stylesheets are not specified in the XML document or your XML stylesheet can be generated inline.

Example of Using the XSLT JSP Tag in a JSP

The following snippet of code from a JSP shows how to use the XSLT JSP tag to transform XML into HTML or WML, depending on the type of client that is requesting the JSP. If the client is a browser, the JSP returns HTML; if the client is a wireless device, the JSP returns WML.

First the JSP uses the `getHeader()` method of the `HttpServletRequest` object to determine the type of client that is requesting the JSP and sets the *`myMedia`* variable to `wml` or `html` appropriately. If the JSP set the *`myMedia`* variable to `html`, then it applies the `html.xsl` stylesheet to the XML document contained in the *`content`* variable. Similarly, if the JSP set the *`myMedia`* variable to `wml`, then it applies the `wml.xsl` stylesheet.

```
<%
   String clientType = request.getHeader("User-Agent");
   // default to WML client
   String myMedia = "wml";

   // if client is an HTML browser

   if (clientType.indexOf("Mozilla") != -1) {
      myMedia = "http"
   }
%>

<x:xslt media="<%=myMedia%>">
```

```
  <x:xml><%=content%></x:xml>
  <x:stylesheet media="html" uri="html.xsl"/>
  <x:stylesheet media="wml"  uri="wml.xsl"/>
</x:xslt>
```

Using Transformers Other Than the Built-In Transformer

The WebLogic Server XML Registry (which you configure using the Administration Console) offers the following options:

- Accept the built-in transformer as the server-wide transformer.
- Configure a transformer other than the built-in transformer as the server-wide transformer. The transformer must be JAXP-compliant.

For instructions on how to use the XML Registry to configure transforming options, see “XML Parser and Transformer Configuration Tasks” on page 4-4.

3 XML Programming Techniques

The following sections provide information about specific XML programming techniques for developing a J2EE application that processes XML data:

- Sending and Receiving XML To and From Servlets and JSPs
- Handling XML Documents in a JMS Application
- Accessing External Entities That Do Not Have an HTTP Interface
- XML Document Header Information

Sending and Receiving XML To and From Servlets and JSPs

In a typical J2EE application, a client application sends XML data to a Servlet or a JSP that processes the XML data. The Servlet or JSP then either sends the data on to another J2EE component, such as a JMS destination or an EJB, or sends the processed XML data back to the client in the form of another XML document.

To send and receive XML data from a Java client to a WebLogic Server-hosted Servlet or JSP and back, use the `java.net.URLConnection` class. This class represents the communication link between an application and an URL, which in this case is the URL that invokes the Servlet or JSP. Instances of the `URLConnection` class send the XML document using the HTTP POST method.

The following Java client program from the WebLogic XML examples shows how to send and receive XML data to and from a JSP:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class Client {

  public static void main(String[] args) throws Exception {
    if (args.length < 2) {
      System.out.println("Usage:  java examples.xml.Client URL Filename");
    }
    else {
      try {
        URL url = new URL(args[0]);
        String document = args[1];
        FileReader fr = new FileReader(document);
        char[] buffer = new char[1024*10];
        int bytes_read = 0;
        if ((bytes_read = fr.read(buffer)) != -1)
          {
            URLConnection urlc = url.openConnection();
            urlc.setRequestProperty("Content-Type","text/xml");
            urlc.setDoOutput(true);
            urlc.setDoInput(true);
            PrintWriter pw = new PrintWriter(urlc.getOutputStream());

            // send xml to jsp
            pw.write(buffer, 0, bytes_read);
            pw.close();

            BufferedReader in = new BufferedReader(new
InputStreamReader(urlc.getInputStream()));
            String inputLine;
            while ((inputLine = in.readLine()) != null)
                      System.out.println(inputLine);

            in.close();
            }
          }
          catch (Exception e) {
          e.printStackTrace();
          }
       }
    }
}
```

The example first shows how to open a URL connection to the JSP using a URL from the argument list, obtain the output stream from the connection, and print the XML document provided in the argument list to the output stream, thus sending the XML data to the JSP. The example then shows how to use the `getInputStream()` method of the `URLConnection` class to read the XML data that the JSAP returns to the client application.

The following code segments from a sample JSP shows how the JSP receives XML data from the client application, parses the XML document, and sends XML data back:

```
BufferedReader br = new BufferedReader(request.getReader());
DocumentBuilderFactory fact = DocumentBuilderFactory.newInstance();
DocumentBuilder db = fact.newDocumentBuilder();
Document doc = db.parse(new InputSource(br));

...

PrintWriter responseWriter = response.getWriter();
responseWriter.println("<?xml version='1.0'?>");
```

...
For detailed information on programming WebLogic Servlets and JSPs, see *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs61/servlet/index.html and *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs61/jsp/index.html

Handling XML Documents in a JMS Application

WebLogic Server provides the following extensions to some Java Message Service (JMS) classes to specifically handle XML documents in an JMS application:

- `weblogic.jms.extensions.WLSession`, which extends the JMS class `javax.jms.Session`
- `weblogic.jms.extensions.WLQueueSession`, which extends the JMS class `javax.jms.QueueSession`
- `weblogic.jms.extensions.WLTopicSession`, which extends the JMS class `javax.jms.TopicSession`

- `weblogic.jms.extensions.XMLMessage`, which extends the JMS class `javax.jms.TextMessage`

If you use the `XMLMessage` class to send and receive XML documents in a JMS application, rather than the more generic `TextMessage` class, you can use XML-specific message selectors to filter unwanted messages. In particular, you can use the method `JMS_BEA_SELECT` to specify an XPath query to search for an XML fragment in the XML document. Based on the results of the query, a message consumer might decide not to receive the message, thus possibly reducing network traffic and improving performance of the JMS application.

To use the `XMLMessage` class to contain XML messages in a JMS application, you must create either a `WLQueueSession` or `WLTopicSession` object, depending on whether you want to use JMS queues or topics, rather than the generic `QueueSession` or `TopicSession` objects, after you have created a JMS `Connection`. Then use the `createXMLMessage()` method of the `WLSession` interface to create an `XMLMessage` object.

For detailed information on using `XMLMessage` objects in your JMS application, see *Programming WebLogic JMS* at http://e-docs.bea.com/wls/docs61/jms/index.html.

Accessing External Entities That Do Not Have an HTTP Interface

WebLogic Server can retrieve and cache external entities that reside in external repositories, as long as they have an HTTP interface, such as an URL, that returns the entity. See “External Entity Configuration Tasks” on page 4-11 for detailed information on using the XML Registry to configure external entities.

If you want to access an external entity that is stored in a repository that does *not* have an HTTP interface, you must create one. For example, assume you store the DTDs for your XML documents in a database table, with columns for the system id, public id, and text of the DTD. To access the DTD as an external entity from a WebLogic XML application, you could create a Servlet that uses JDBC to access the DTDs in the database.

Because you invoke Servlets with URLs, you now have an HTTP interface to the external entity. When you create the entity registry entry in the XML Registry, you specify the URL that invokes the Servlet as the location of the external entity. When WebLogic Server is parsing an XML document that contains a reference to this external entity, it invokes the Servlet, passing it the public and system id, which the Servlet can internally use to query the database.

XML Document Header Information

Sometimes you might want to only get information about an XML document, such as the root element, system ID, or public ID, instead of getting all the actual data within the document. In this case, fully parsing the document is unnecessary, and indeed might decrease the performance of your application if the XML document is very large.

Instead of parsing the XML document, you can get header information about the XML document by using the `weblogic.xml.sax.XMLInputSource` class, which is Weblogic Server's extension to the `org.xml.sax.InputSource` class. The following example segment shows how to use this class:

```
import weblogic.xml.sax.XMLInputSource;
...

    String inputXML = "file://xml_docs/myXMLdoc.xml";
    XMLInputSource xis = new XMLInputSource(inputXML);
    String docType = xis.getRootTag();
    String publicID = xis.getPublicId();
    String systemID = xis.getSystemId();
    String namespaceURI = xis.getNamespaceURI();
```

See the *WebLogic Server API Reference* for more information on the `weblogic.xml.sax.XMLInputSource` class.

4 Administering WebLogic Server XML

The following sections describe XML administration with WebLogic Server:

- Overview of Administering WebLogic Server XML
- XML Parser and Transformer Configuration Tasks
- External Entity Configuration Tasks

Overview of Administering WebLogic Server XML

You access the XML Registry through the Administration Console and use it to configure WebLogic Server for XML applications.

To invoke the Administration Console in your browser, enter the following URL:

```
http://host:port/console
```

where

- *host* refers to the computer on which WebLogic Administration server is running.

- *port* refers to the port number where WebLogic Administration server is listening for connection requests. The default port number for WebLogic Administration server is 7001.

XML Administration Tasks

You create, configure, and use the XML Registry through the Administration Console. The benefits of using the Administration Console XML Registry are as follows:

- Configuration of XML Registry changes take effect automatically at run time, provided you use JAXP in your XML applications.
- When you make changes to the XML Registry, it is not necessary to change your XML application code.
- Entity resolution is done locally. You can use the XML Registry either to define a local copy of an entity or to specify that WebLogic Server cache an entity from the Web for a specified duration and use the cached copy rather than the one out on the Web.

You can use the XML Registry to specify:

- An alternative server-wide XML parser instead of the built-in parser.
- An XML parser per document.
- An alternative server-wide transformer instead of the built-in transformer.
- External entities that are to be resolved by using local copies of the entities. Once you specify these entities, the Administration Server stores local copies of them in the file system and automatically distributes them to the server's parser at parse time. This feature eliminates the need to construct and set SAX EntityResolvers.
- External entities to be cached by WebLogic Server after retrieval from the Web. You specify how long these external entities should be cached before WebLogic Server re-retrieves them and when WebLogic should first retrieve the entities, either at application run time or when WebLogic Server starts.

These capabilities are for use on the server side only.

How the XML Registry Works

You can create as many XML Registries as you like; however, you can associate only one XML Registry with a particular instance of WebLogic Server.

If an instance of WebLogic Server does not have an XML Registry associated with it, then the built-in parser and transformer are used when parsing or transforming documents. In addition, you cannot configure external entity resolution to increase the performance of your XML applications.

Once you associate an XML Registry with an instance of WebLogic Server, all XML configuration options are available for XML applications that use that server.

You can configure the following two types of entries for a given XML registry:

- to configure parsers and transformers.
- to configure external entity resolution.

Note: The XML Registry is case sensitive. For example, if you are configuring a parser for an XML document type whose root element is `<CAR>`, you must enter `CAR` in the Root Element Tag field and not `car` or `Car`.

Parser Selection Within the XML Registry

The XML Registry is automatically consulted whenever you use JAXP to write your XML applications. WebLogic Server follows an ordered lookup when determining which parser class to load:

1. Use the parser defined for a particular document type.
2. Use the alternative server-wide parser defined in the XML Registry associated with the WebLogic Server instance.
3. Use the built-in Xerces parser.

The process is also true for transformers, except for the first step, because you cannot define a transformer for a particular document type.

Additionally, when WebLogic Server starts, a SAX entity resolver is automatically set so that it can resolve entities that are declared in the registry. As a result, users are not required to modify their XML application code to control the parsers used, or to set the location of local copies of external entities. The parser being used and the location of the external entity is controlled by the XML Registry.

Note: If you elect to use an API provided by a parser instead of JAXP, the XML Registry has no effect on the processing of XML documents. For this reason, it is highly recommended that you always use JAXP in your XML applications.

XML Parser and Transformer Configuration Tasks

By default, WebLogic Server is configured to use the built-in parser and transformer to parse and transform XML documents. In release 6.1, the built-in XML parser is Apache Xerces and the built-in transformer is Apache Xalan. As long as you use the default, you do not have to perform any configuration tasks for your XML applications. If you want to use a parser or transformer other than the built-in, you must use the XML Registry to configure them, as described in the following sections.

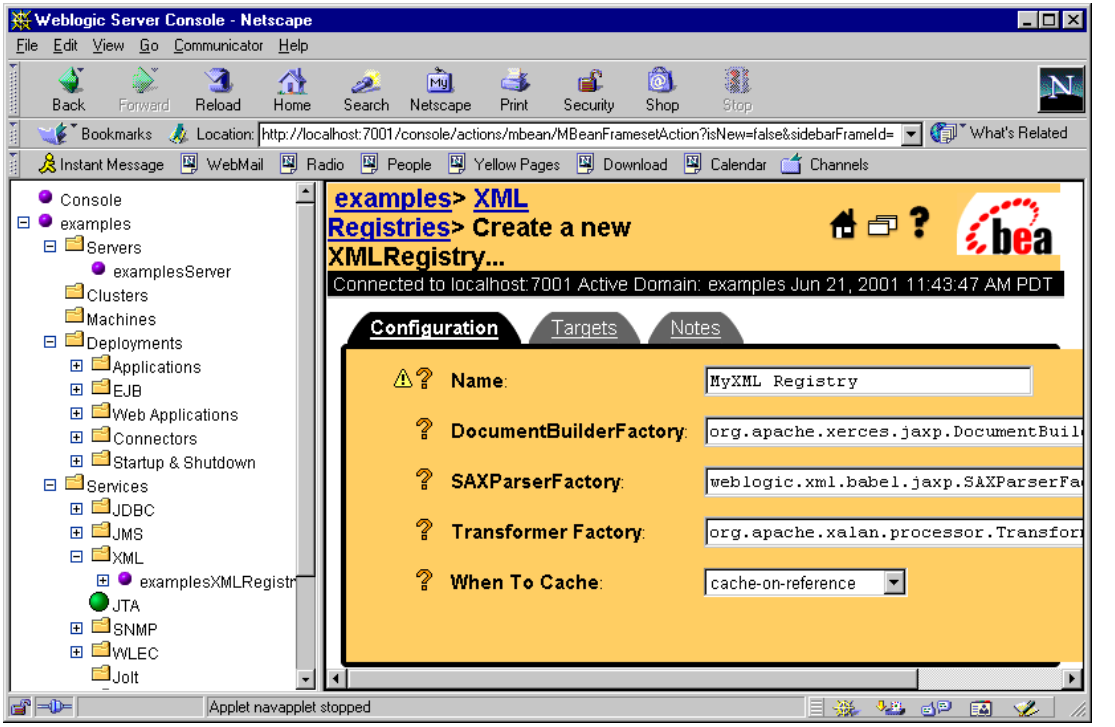
Configuring a Parser or Transformer Other Than the Built-In

The following procedure first describes how to create an XML registry that defines SAX and DOM parsers and transformers. It then describes how to associate the new XML Registry with an instance of WebLogic Server so that the server starts to use the new parsers and transformer.

1. Start the WebLogic Administration server and invoke the Administration Console in your browser. See “Overview of Administering WebLogic Server XML” on page 4-1 for information on invoking the Administration Console.

2. In the left pane, right-click the XML node under the Services node and select Configure a new XML Registry from the drop-down menu. The window to create a new XML registry is displayed, as shown in the following figure:

Figure 4-1 Main XML Registry Window in Administration Console



3. Enter a unique registry name in the Name field and set the DocumentBuilderFactory, SaxParserFactory, and TransformerFactory fields to the appropriate Factory parser and transformer classes.

   For example, to use WebLogic FastParser, enter the following information:

```
Name: WebLogic FastParser
DocumentBuilderFactory:
SAXParserFactory: weblogic.xml.babel.jaxp.SAXParserFactoryImpl
TransformerFactory:
```

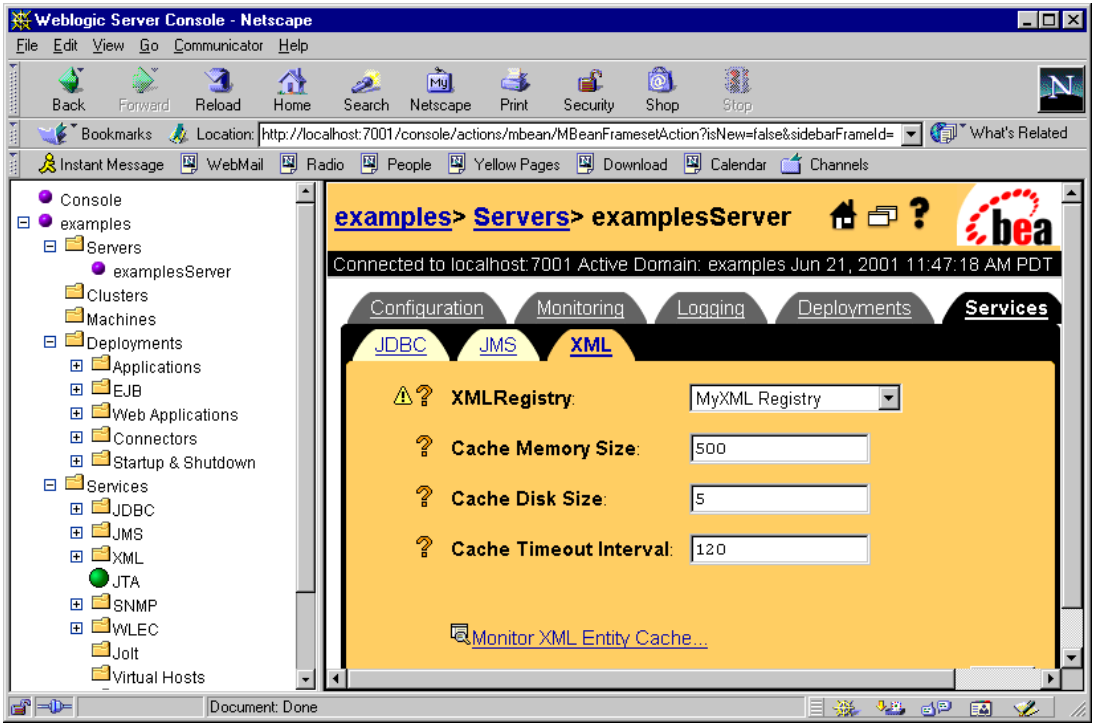
Note that in the preceding example, DocumentBuilderFactory and TransformerFactory have been left blank. This means that for DOM parsing and transformation, the built-in parser and transformer are used, respectively. The WebLogic FastParser will only be used for SAX parsing.

If you want to directly specify the Apache Xerces parser and Xalan transformer, enter the following information:

```
Name: Apache Xerces/Xalan Registry
DocumentBuilderFactory: org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
SAXParserFactory: org.apache.xerces.jaxp.SAXParserFactoryImpl
TransformerFactory: org.apache.xalan.processor.TransformerFactoryImpl
```

4. Click the Create button. The XML Registry is created and listed under the XML node in the left pane.
5. In the left pane under the Servers node, click the name of the server with which you want to associate the new XML registry.
6. In the right pane, select the Services tab.
7. Select the XML tab. The window to configure XML properties of WebLogic Server appears in the right pane, as shown in the following figure:

Figure 4-2 Window to Configure XML Properties in Administration Console



8. Select the XML registry name that you want to associate with this server in the XML Registry field and click the Apply button.
9. Restart your server so the new settings to take effect.

Configuring a Parser for a Particular Document Type

When you configure a parser for a particular document type, you can use the document's system id, public id, or root element tag to identify the document type.

Note: The following procedure assumes that you are going to create a new XML registry, add the necessary parser registry entries, and associate it with a server. If you have already associated an existing XML registry with your server, skip to step 5.

To configure a parser for a particular document type, follow these steps:

1. Start the WebLogic Administration server and invoke the Administration Console in your browser.

   See "Overview of Administering WebLogic Server XML" on page 4-1 for information on invoking the Administration Console.

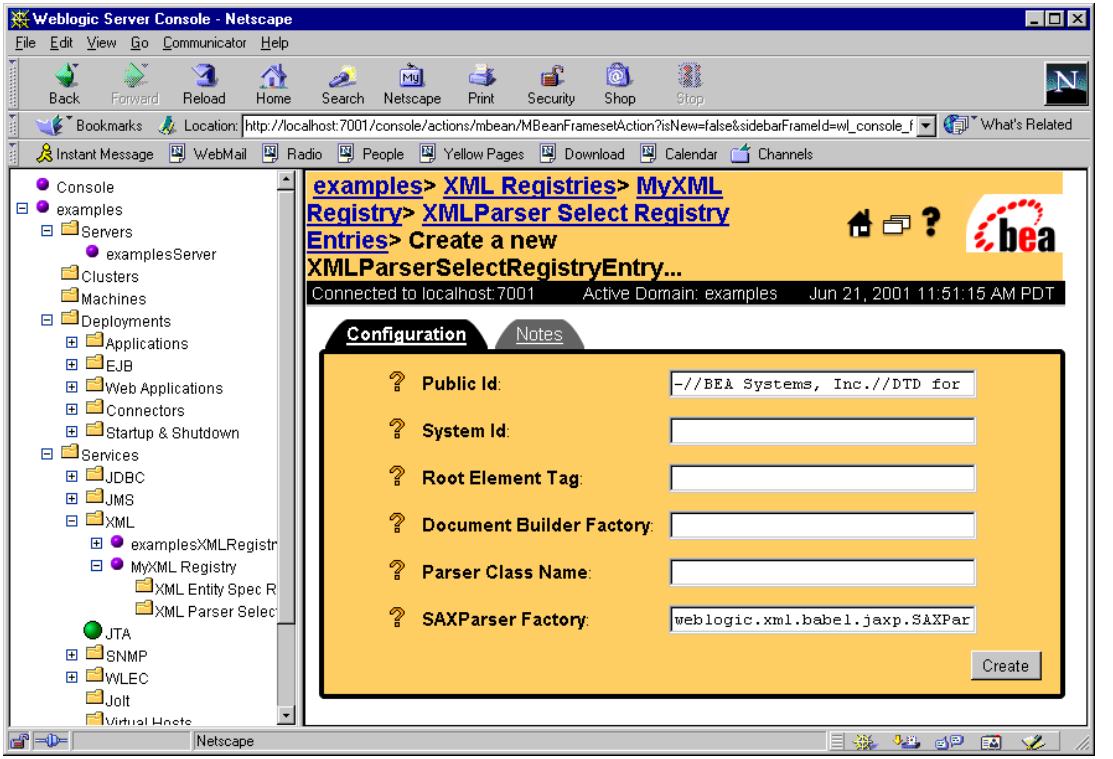
2. In the left pane, right-click the XML node under the Services node and select Configure a new XML Registry from the drop-down menu. The window to create a new XML registry is displayed, as shown in Figure 4-1.

3. Enter a unique registry name in the Name field. If you want to configure default parsers and transformer for your server, enter the factory class names in the DocumentBuilderFactory, SaxParserFactory, and TransformerFactory fields. Otherwise, leave these fields blank.

4. Click the Create button. The XML Registry is created and listed under the XML node in the left pane.

5. Under the XML node in the left pane, right-click the XML Parser Select Registry Entry node under your XML registry. Select Configure a New XMLParserSelectRegistryEntry from the drop-down menu. A blank window for entering document type information appears in the right pane, as shown in the following figure:

Figure 4-3 Configuring an XML Parser Using the Administration Console



6. Enter the document type information in one of the following ways:

   a. Use either the Public Id or the System Id field to specify the doctype. For example, for the `car.xml` (see Listing 4-1), enter `-//BEA Systems, Inc.//DTD for cars//EN` in the `Public Id` field.

   b. Specify the Root Element Tag name of the document. For the `car.xml` example, enter `CAR` in the Root Element Tag field.

      If your XML document defines a namespace, be sure to enter the fully qualified root element tag, such as `VEHICLES:CAR`.

Listing 4-1 car.xml File

```
<?xml version="1.0"?>
<!-- This XML document describes a car -->
<!DOCTYPE CAR PUBLIC "-//BEA Systems, Inc.//DTD for cars//EN"
"http://www.bea.com/dtds/car.dtd">
<CAR>
<MAKE>Toyota</MAKE>
<MODEL>Corrolla</MODEL>
<YEAR>1998</YEAR>
<ENGINE>1.5L</ENGINE>
<HP>149</HP>
</CAR>
```

7. Set the DocumentBuilderFactory or SaxParserFactory fields to the appropriate Factory parser classes.

   For example, enter `weblogic.xml.babel.jaxp.SAXParserFactoryImpl` in the SaxParserFactory field to specify that this document type be parsed by WebLogic FastParser.

   Note: Do not enter any information in the Parser Class Name field; this field is for backward compatibility with previous versions of WebLogic Server only.

8. Click the Create button. The XMLParserSelect registry entry is created.
9. In the left pane under the Servers node, click the name of the server with which you want to associate the new XML registry.
10. In the right pane, select the Services tab.
11. Select the XML tab. The window to configure XML properties of WebLogic Server appears in the right pane, as shown in Figure 4-2.
12. In the XML Registry field, select the XML registry name that you want to associate with this server, and click the Apply button.
13. Restart your server so the new settings to take effect.

External Entity Configuration Tasks

You can use the XML Registry to configure external entity resolution and to configure and monitor the external entity cache.

Configuring External Entity Resolution

You can configure external entity resolution with WebLogic Server in the following two ways:

- Physically copy the entity files to a directory accessible by WebLogic Administration Server and specify that the Administration Server use the local copy whenever the external entity is referenced in an XML document.
- Specify that a managed WebLogic Server cache external entities that are referenced with a URL or a pathname relative to the Administration server, either at server-startup or when the entity is first referenced.

  Caching the external entity in the managed WebLogic Server saves the remote access time and provides a local backup in the event that the Administration server cannot be accessed while an XML document is being parsed, due to the network or the Administration server being down.

  You can configure the expiration date a cached entity, at which point WebLogic Server re-retrieves the entity from the URL or Administration server and re-caches it.

Note: In the following procedure it is assumed that you are going to create a new XML registry, add the necessary external entity resolution registry entries, and associate it with a server. If you have already associated an existing XML registry with your server, skip to step 5.

To configure external entity resolution, perform the following steps:

1. Start the WebLogic Administration Server and invoke the Administration Console in your browser.

   See “Overview of Administering WebLogic Server XML” on page 4-1 for information on invoking the Administration Console.

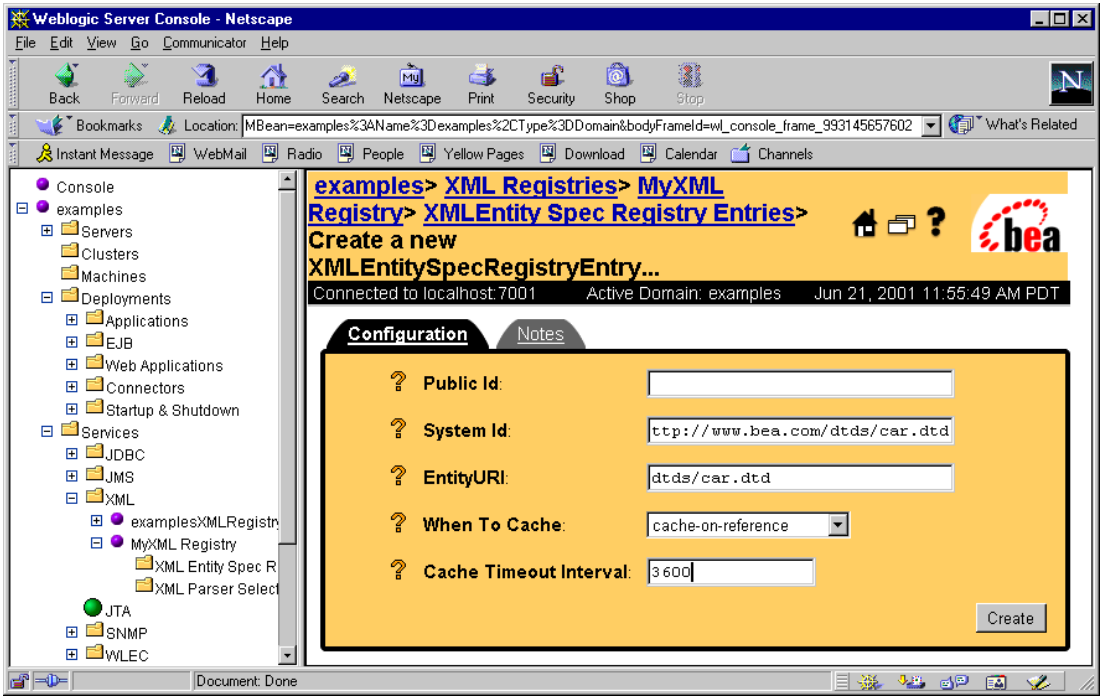
2. Right-click the XML node under the Services node in the left pane and select Configure a new XML Registry from the drop-down menu. The window to create a new XML registry is displayed, as shown in Figure 4-1.

3. In the Name field, enter a unique registry name. If you want to configure default parsers and transformer for your server, enter the factory class names in the DocumentBuilderFactory, SaxParserFactory, and TransformerFactory fields. Otherwise, leave these fields blank.

4. Click the Create button. The XML Registry is created and listed under the XML node in the left pane.

5. Under the XMl node in the left pane, right-click the XML Entity Spec Registry Entry node under your XML registry. Select Configure a New XMLEntitySpecRegistryEntry from the drop-down menu. A blank window for entering entity resolution information appears in the right pane, as shown in the following figure:

Figure 4-4 Configuring External Entities using the Administration Console



6. Enter either the `System Id` or `Public Id` that is used to reference the external entity in the XML document. For example, for the following `car.xml` file, enter `http://www.bea.com/dtds/car.dtd` for the `System Id`:

Listing 4-2 car.xml File

```
<?xml version="1.0"?>
<!-- This XML document describes a car -->
<!DOCTYPE CAR PUBLIC "-//BEA Systems, Inc.//DTD for cars//EN"
"http://www.bea.com/dtds/car.dtd">
<CAR>
<MAKE>Toyota</MAKE>
<MODEL>Corrolla</MODEL>
<YEAR>1998</YEAR>
<ENGINE>1.5L</ENGINE>
```

```
<HP>149</HP>
</CAR>
```

7. In the EntityURI field, enter one of the following two entity paths:

   a. The pathname of the copy of the entity file in the Administration Server. This pathname must be relative to the registries entity directory, which is *BEAHome*/wlserver6.1/config/*domain*/xml/registries/*reg_name* in the domain configuration directory, where *BEAHome* is the top-level directory in which the WebLogic Server software is installed, *domain* is the name of your WebLogic Server domain, and *reg_name* is the name of the new XML Registry. For example, for the `car.xml` file, you might enter `dtds/car.dtd` in the EntityURI field.

   b. A URL that points to an external entity out on the Web or an entity stored in a repository. For example, enter `http://java.sun.com/j2ee/dtds/application_1_2.dtd` to reference the DTD for the `application.xml` file used to describe J2EE Enterprise Applications or use `jdbc:` to reference an entity in a database.

      Use the following protocol declarations to specify an external entity: `http://`, `file://`, `jdbc:`, or `ftp://`.

8. Select one of the following options from the WhenToCache list box:

   - `cache-on-reference`—WebLogic Server caches the external entity referenced by a URL the first time the entity is referenced in an XML document.
   - `cache-at-initialization`—WebLogic Server caches the entity when the server starts.
   - `defer-to-registry-setting`—WebLogic Server uses the default caching setting. See "Configuring the External Entity Cache" on page 4-15 for information on configuring default caching settings.

9. In the CacheTimeoutInternal field, enter the number of seconds after which the cached external entity becomes stale, or out-of-date. WebLogic Server re-retrieves the external entity from the specified URL or pathname relative to the Administration server if the cached copy has been in the cache for longer than this amount.

The default value for this field is -1, which means that the global timeout value for WebLogic Server is used. See "Configuring the External Entity Cache" on page 4-15 for information on configuring global cache timeout settings.

10. Click the Create button. The XMLEntitySpec registry entry is created.
11. In the left pane under the Servers node, click the name of the server with which you want to associate the new XML registry.
12. In the right pane, select the Services tab.
13. Select the XML tab. The window to configure XML properties of WebLogic Server appears in the right pane, as shown in Figure 4-2.
14. In the XML Registry field, select the XML registry name that you want to associate with this server, and click the Apply button.
15. Restart your server so the new settings to take effect.
16. If you specified that a local copy of the entity be used, rather than caching the one from the Web, copy the entity file into the entity directory. For example, you would copy the `car.dtd` file to the directory *`BEAHome`*`/wlserver6.1/config/`*`domain`*`/xml/registries/`*`reg_name`*`/dtds`, where *`BEAHome`* is the top-level directory in which the WebLogic Server software is installed, *`domain`* is the name of your WebLogic Server domain, and *`reg_name`* is the name of the new XML Registry.

Configuring the External Entity Cache

You can configure the following properties of the external entity cache:

- Size, in KB, of the cache memory. The default value for this property is 500 KB.
- Size, in MB, of the persistent disk cache. The default value for this property is 5 MB.
- Number of seconds after which external entities in the cache become stale after they have been cached by WebLogic Server. This is the default value for the entire server - you can override this value for specific external entities when you configure the entity. The default value for this property is 120 seconds (2 minutes).

To configure the external entity cache, follow these steps:

1. Start the WebLogic Administration server and invoke the Administration Console in your browser.

   See "Overview of Administering WebLogic Server XML" on page 4-1 for information on invoking the Administration Console.

2. Under the Servers node in the left pane, click the name of the WebLogic Server for which you want to configure the external entity cache.
3. Select the Services tab in the right pane.
4. Select the XML tab. The window to configure XML properties of WebLogic Server appears in the right pane, as shown in Figure 4-2.
5. In the Cache Memory Size field, enter the size, in KB, of the cache memory.
6. In the Cache Disk Size field, enter the size, in MB, of the persistent disk cache.
7. In the Cache Timeout Interval field, enter the number of seconds after which entities become stale.
8. Click the Apply button.

Monitoring the External Entity Cache

A set of statistics that describes the external entity cache is available for you to use to monitor the effectiveness of the cache. These statistics describe:

- The current state of the cache.
- The cumulative activity for the current session.
- The cumulative activity since the cache was created, typically when WebLogic Server started.

To access the statistics, use the J2EE Java Management Extension (JMX) specification with the WebLogic Server Management API to create and deploy Management Beans (or MBeans) to monitor entity external caching in WebLogic Server. Use the following MBean interfaces:

- `weblogic.management.runtime.EntityCacheCumulativeRuntimeMBean`

- `weblogic.management.runtime.EntityCacheCurrentStateRuntimeMBean`
- `weblogic.management.runtime.EntityCacheRuntimeMBean`

The WebLogic Server *Management API* is fully documented online in JavaDocs.

The following table describes the methods you can use to get statics on the current state of the external entity cache.

Table 4-1 Current State of Cache Statistics

Method	Description
getMemoryUsage	Returns the number of bytes used to store all memory-resident entries.
getDiskUsage	Returns the number of bytes used to store all disk resident entries.
getTotalCurrentEntries	Returns the number of total entries in the cache.
getTotalPersistentCurrentEntries	Returns the number of persistent entries in the cache.
getTotalTransientCurrentEntries	Returns the number of transient entries in the cache.
getAvgPercentTransient	Returns the percent of entries which are transient.
getAvgPercentPersistent	Returns the percent of entries which are persistent.
getAvgTimeout	Returns the average timeout value for the entries.
getMinEntryTimeout	Returns the smallest timeout value for any current entry.
getMaxEntryTimeout	Returns the largest timeout value for any current entry.
getAvgPerEntryMemorySize	Returns the average memory size of the current entries.
getMaxEntryMemorySize	Returns the largest memory size for any current entry.
getMinEntryMemorySize	Returns the smallest memory size for any current entry.
getAvgPerEntryDiskSize	Returns the average disk size of the current entries.

The following table describes the methods you can use to get statics on the cumulative activity of the external entity cache.

Table 4-2 Cumulative Activity of the Cache

Method	Description
getTotalCurrentEntries	Returns the number of total entries in the cache.
getTotalPersistentCurrentEntries	Returns the number of persistent entries in the cache.
getTotalTransientCurrentEntries	Returns the number of transient entries in the cache.
getAvgPercentTransient	Returns the percent of entries which are transient.
getAvgPercentPersistent	Returns the percent of entries which are persistent.
getAvgTimeout	Returns the average timeout value for the entries.
getMinEntryTimeout	Returns the smallest timeout value for any current entry.
getMaxEntryTimeout	Returns the largest timeout value for any current entry.
getAvgPerEntryMemorySize	Returns the average memory size of the current entries.
getMaxEntryMemorySize	Returns the largest memory size for any current entry.
getMinEntryMemorySize	Returns the smallest memory size for any current entry.
getAvgPerEntryDiskSize	Returns the average disk size of the current entries.
getTotalNumberMemoryPurges	Returns the number of memory purges done.
getTotalItemsMemoryPurged	Returns the total number of items purged in all memory purges.
getAvgEntrySizeMemoryPurged	Returns the average size in bytes of items memory purged.
getMostRecentMemoryPurge	Returns the time of the most recent memory purge.
getMemoryPurgesPerHour	Returns the average number of memory purges per hour.
getTotalNumberDiskPurges	Returns the number of disk purges done.
getTotalItemsDiskPurged	Returns the total number of items purged in all disk purges.

Table 4-2 Cumulative Activity of the Cache

Method	Description
getAvgEntrySizeDiskPurged	Returns the average size in bytes of items disk purged.
getMostRecentDiskPurge	Returns the time of the most recent disk purge.
getDiskPurgesPerHour	Returns the average number of disk purges per hour.
getTotalNumberOfRejections	Returns the number of entries that have been rejected.
getTotalSizeOfRejections	Returns the total size in bytes of all items rejected.
getPercentRejected	Returns the percent of inserts that were rejected.
getTotalNumberOfRenewals	Returns the number of times an stale entry was renewed.

5 XML Reference

The following sections describe the XML specifications, application programming interfaces (APIs), and tools supported by WebLogic Server:

- Extensible Markup Language (XML) 1.0 Specification
- Simple API for XML (SAX) 2.0
- Document Object Model (DOM) Level 2 API
- W3C XML Namespaces 1.0 Recommendation
- Java API for XML Processing (JAXP) 1.1
- Apache Xerces Java Parser API
- Apache Xalan XML Stylesheet Language Transformer (XSLT) API
- Additional Resources

Extensible Markup Language (XML) 1.0 Specification

The W3C Recommendation for XML provides the following abstract:

"The Extensible Markup Language (XML) is a subset of SGML that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML."

The complete XML specification is available at http://www.w3.org/TR/REC-xml/.

Simple API for XML (SAX) 2.0

The SAX API is platform-independent and language-neutral. It is a standard interface for event-based XML parsing that was developed collaboratively by the members of the XML-DEV mailing list.

SAX applications process an XML document by creating a parser object and associating handlers with XML events. Once these tasks are done, the parser can read through the document as events occur, and pass them to the handlers. Events represent the entities within the document, such as start of document, end of document, start of element, and end of element. The SAX interface provides a simple coding model and is useful for processing XML documents with a relatively simple hierarchical structure. Hence, the SAX interface provides what is required by the bundled parser to parse XML documents.

For more information on SAX, see http://www.saxproject.org/. For Javadoc documentation, see SAX (Simple API for XML) at http://e-docs.bea.com/wls/docs61/xerces/index.html.

Document Object Model (DOM) Level 2 API

The DOM API is a platform- and language-neutral interface It allows programs and scripts to access and update the content, structure, and style of XML documents dynamically. DOM gives you access to the information stored in your XML document as a hierarchical object model, much like a tree with the document's root element as the tree's root node. Using the DOM interface, you can access different parts of XML documents, navigate through them, and make changes and additions to them.

When an application invokes a DOM parser, the parser processes the entire document, creating an in-memory object model, which the application can process in any fashion it chooses. The DOM approach is most useful for more complex documents because it does not require a developer to interpret every element.

For more information on DOM, see the DOM (Document Object Model) Level 2 Specification at http://www.w3.org/TR/DOM-Level-2/. For Javadoc documentation, see DOM (Document Object Model) at http://e-docs.bea.com/wls/docs61/xerces/index.html.

W3C XML Namespaces 1.0 Recommendation

The following abstract is taken from the W3C XML Namespace Recommendation:

"XML namespaces provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents by associating them with namespaces identified by Universal Resource Identifier (URI) references."

The XML Namespaces 1.0 Recommendation is available on the Internet at http://www.w3.org/TR/REC-xml-names/.

Java API for XML Processing (JAXP) 1.1

JAXP is Sun's Java API for XML parsing and transforming. JAXP provides basic support for parsing, manipulating, and transforming XML documents through a standardized set of Java platform APIs. Thus, applications that use JAXP to process XML documents are portable across platforms.

JAXP does not replace either the SAX or DOM API. Instead, it adds some convenience methods that are designed to make applications that use the SAX and DOM APIs portable.

JAXP 1.1 consists of the `javax.xml.parsers` and `javax.xml.transform` packages that contain the interfaces, classes, and methods for parsing and transforming XML data.

The JAXP specification is available on the Internet at `http://java.sun.com/xml/`. The JAXP Javadoc is available at http://java.sun.com/xml/jaxp/dist/1.1/docs/api/index.html.

Apache Xerces Java Parser API

The Apache Xerces Java Parser package includes the API documentation for SAX and DOM, the two most common interfaces for programming XML. In addition, the parser provides documentation for classes that are not part of the SAX and DOM APIs, but are useful for writing parser programs.

For more information about the Apache Xerces Java parser, refer to http://xml.apache.org/xerces-j/index.html. For Javadoc documentation, refer to Apache Xerces Java Parser at http://e-docs.bea.com/wls/docs61/xerces/index.html.

Apache Xalan XML Stylesheet Language Transformer (XSLT) API

The Apache Xalan-Java XSLT transformer is used for transforming XML documents. It implements the W3C Recommendation 16 November 1999 XSL Transformations (XSLT) Version 1.0. XSLT is a stylesheet language for transforming XML documents into other XML documents, HTML documents, or other document types. The language includes the XSL Transformation vocabulary and XPath, a language for addressing parts of an XML document. An XSL stylesheet describes how to transform the tree of nodes in the XML input into another tree of nodes.

For more information about the Apache Xalan XSLT transformer, refer to http://xml.apache.org/xalan-j/index.html. For Javadoc documentation, refer to Apache Xalan XSLT Transformer at http://e-docs.bea.com/wls/docs61/Xalan/index.html.

Additional Resources

This section lists various resources that are available online to help you learn about programming with WebLogic XML:

- Code Examples
- Related WebLogic Documentation
- General XML Information
- Tutorials and Online Courses
- Other XML Specifications

Code Examples

XML code examples and supporting documentation are included in the WebLogic Server distribution at *BEA Home*\wlserver6.1\samples\examples\xml, where *BEA Home* is the directory in which the WebLogic Server software is installed.

Related WebLogic Documentation

- *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs61/webServices/index.html
- *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs61/ejb/index.html
- *Programming WebLogic JMS* at http://e-docs.bea.com/wls/docs61/jms/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs61/jsp/index.html
- *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs61/servlet/index.html
- *Programming WebLogic Server for Wireless Services* at http://e-docs.bea.com/wls/docs61/wireless/index.html

General XML Information

- W3C (World Wide Web Consortium) at http://www.w3c.org.

- XML.com at http://www.xml.com.
- XML FAQ at http://www.ucc.ie/xml/.
- XML.org, The XML Industry Portal at http://www.xml.org/.
- W3C: Extensible Stylesheet Language at http://www.w3.org/Style/XSL/.

Tutorials and Online Courses

- A Technical Introduction to XML at http://www.xml.com/pub/a/98/10/guide0.html.
- XML Authoring Tutorial at http://www.xml.com/pub/r/32.
- Working with XML and Java at http://java.sun.com/xml/tutorial_intro.html.
- Tutorials for using the Java 2 platform and XML technology at http://developerlife.com/.
- Developing XML Solutions with JavaServer Pages Technology at http://java.sun.com/products/jsp/html/JSPXML.html.
- XML, Java, and the Future of the Web at http://www.xml.com/pub/a/w3j/s3.bosak.html.
- Chapter 14 of the XML Bible: XSL Transformations at http://metalab.unc.edu/xml/books/bible/updates/14.html.
- XSL Tutorial by Miloslav Nic at http://zvon.vscht.cz/HTMLonly/XSLTutorial/Books/Book1/index.html.
- XML Schema Part 0: Primer at http://www.w3.org/TR/2000/CR-xmlschema-0-20001024/.

Other XML Specifications

- Extensible Stylesheet Language (XSL) 1.0 Specification at http://www.w3.org/TR/xsl/.

- JSR-000031 XML Data Binding Specification at http://java.sun.com/aboutJava/communityprocess/jsr/jsr_031_xmld.htm.
- XML Path Language (XPath) Version 1.0 Specification at http://www.w3.org/TR/xpath.
- XML Linking Language (XLink) Specification at http://www.w3.org/TR/xlink.
- XML Pointer Language (XPointer) Specification at http://www.w3.org/TR/WD-xptr.
- XML Schema Part 1: Structures at http://www.w3.org/TR/xmlschema-1/.
- XML Schema Part 2: Datatypes at http://www.w3.org/TR/xmlschema-2/.

Index

G

H

I

J

M

V

W

X