BEA WebLogic Server™

Programming WebLogic XML

Version 8.1
Revised: November 2, 2005

Contents

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 Application Scoping," describes how to configure parsers, transformers, and external entities for a particular Enterprise application.
- Chapter 4, "Using the WebLogic XML Streaming API," describes in detail how to use the WebLogic XML Streaming API in your Java applications to parse an XML document.
- Chapter 5, "Using Advanced XML APIs," describes how to use advanced XML APIs, such as the WebLogic XPath API to perform XPath matching against an XML document and the WebLogic XML Digital Signature API to digitally sign SOAP messages.
- Chapter 6, "XML Programming Best Practices," describes some best practices to follow when creating Java applications that handle XML documents.

- Chapter 7, "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 8, "Administering WebLogic Server XML," describes the Administration Console XML Registry and how to perform XML configuration tasks.
- Chapter 9, "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.

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 and Chapter 9, "XML Reference."

Contact Us!

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

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

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

Convention	Usage

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

`monospace italic text`	Variables in code. *Example*: `String` *`CustomerName`*`;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Examples*: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.

`[ ]`	Optional items in a syntax line. *Example*:
	`java utils.MulticastTest -n name -a address`
	`     [-p portnumber] [-t timeout] [-s send]`

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

Convention	Usage
. . .	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.

CHAPTER 1

XML Overview

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

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 `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 XML Schemas.

Document Type Definitions (DTDs) define the basic requirements for the structure of a particular 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 standard Java application programming interfaces (APIs) for parsing XML data. Both are supported by the WebLogic Server built-in parser. 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 platform-independent language neutral 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. For detailed information about the differences between the two versions, refer to http://www.saxproject.org/.

DOM

DOM stands for the *Document Object Model*. It is platform- and language-neutral interface that allows programs and scripts to access and update the content, structure, and style of XML documents dynamically. 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 XML Streaming?

In addition to SAX and DOM, you can also parse an XML document using the XML streaming API.

The WebLogic XML Streaming API provides an easy and intuitive way to parse and generate XML documents. It is based upon the SAX API, but enables a procedural, stream-based handling of XML documents rather than requiring you to write SAX event handlers, which can get complicated when you work with complex XML documents. In other words, the streaming API gives you more control over parsing than the SAX API.

The XML Streaming API uses the WebLogic FastParser when parsing documents.

For detailed information on using the WebLogic XML Streaming API, see Chapter 4, "Using the WebLogic XML Streaming API."

Note: Unlike DOM and SAX, XML Streaming is not yet part of the Java API for XML Processing (JAXP).

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 pluggability layer that allows programmers to use any compliant parser or transformer.

WebLogic Server implements JAXP to facilitate XML application development and the work required to move XML applications built on WebLogic Server to other Web application servers. 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 2-11.

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 "Other XML Specifications and Information" on page 9-3.

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, 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
- WebLogic XML Streaming API
- JAXP Pluggability 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 for Parsing and Transforming XML Documents

XML Document Parsers

WebLogic Server includes the following two parsers:

Table 1-1 Parsers Included With WebLogic Server

Parser	Description
Built-in	A validating parser based on the Apache Xerces parser version 2.1.0. You can use the built-in parser in either Simple API For XML (SAX) mode or Document Object Model (DOM) mode using the JAXP API. The package name of the built-in WebLogic Server parser is `weblogic.apache.xerces.*`. For detailed information on this parsers, see its Javadoc. If you have *not* used the XML Registry to configure a different built-in parser for WebLogic Server, and you use JAXP in your application to obtain a parser, this built-in parser is the one get.
WebLogic FastParser	A high-performance non-validating XML parser specifically designed for processing small to medium size documents, such as SOAP and WSDL files associated with WebLogic Web services. The FastParser supports SAX-style parsing only. Configure WebLogic Server to use FastParser if your application mostly handles small to medium size (up to 10,000 elements) XML documents. For detailed information on using WebLogic FastParser, refer to "Using the WebLogic FastParser" on page 2-8.

You can also use any other XML parser of your choice by using the Administration Console to configure it in the XML Registry. 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

The built-in XSLT transformer included in WebLogic Server is the same one that is included in the JDK 1.4.1 that is shipped with WebLogic Server: Version 2.2.D11 of the Apache Xalan XSL transformer.

If you have not used the XML Registry to configure a different built-in transformer for WebLogic Server, and you use JAXP in your application to obtain a transformer, this built-in transformer is the one get. The package name of this transformer is org.apache.xalan.*.

You can use this built-in XSLT transformer or other XSLT transformers in your XML application to transform XML documents into other XML documents, HTML, and so on. For more information about transforming XML documents, see "Using JAXP to Transform XML Data" on page 2-11.

Difference in Built-In Transformer Between Versions 8.1 and Previous of WebLogic Server

The built-in transformer in Versions 7.0 and previous of WebLogic Server was one that was *based* on Apache's Xalan XSLT transformer and whose package name started with `weblogic.apache.xalan.*`. In Version 8.1 of WebLogic Server, this transformer has been deprecated. Instead, the built-in transformer is the same one that is shipped in JDK 1.4.1: Apache's Xalan 2.2.D11.

For backward compatibility, the `weblogic.apache.xalan.*` transformer is still available in Version 8.1 of WebLogic Server, although BEA highly recommends you do not use it since it will not be available in future versions. If, however, you need to temporarily continue using this transformer, you must use the Administration Console to configure a transformer other than the built-in for your WebLogic Server instance by updating, or creating a new, XML Registry. Use the following transformer factory:

```
weblogic.apache.xalan.processor.TransformerFactoryImpl
```

For detailed information on using the Administration Console to configure the XML Registry for WebLogic Server, see *Configuring a Parser or Transformer Other Than the Built-In* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/xml.html#xml002.

WebLogic XML Streaming API

The WebLogic XML Streaming API provides an easy and intuitive way to parse and generate XML documents. It is based upon the SAX API, but provides a more procedural, stream-based handling of XML documents rather than having to write SAX event handlers, which can get complicated when dealing with complex XML documents. In other words, the streaming API gives you more control over parsing than the SAX API.

For detailed information on using the WebLogic XML Streaming API, see Chapter 4, "Using the WebLogic XML Streaming API."

JAXP Pluggability 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. This is different from the JAXP 1.1 specification which specifies the use of system properties to plug in parsers and transformers.

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 instance 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 to process a particular application.

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. 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. 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 8-5.

Code Examples for Parsing and Transforming XML Documents

WebLogic Server includes examples of parsing and transforming XML documents.

The examples are located in the *`WL_HOME`*`\samples\server\examples\src\examples\xml` directory, where *`WL_HOME`* refers to the top-level WebLogic Platform directory.

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

Endorsed Standards Override Mechanism for DOM/SAX: Not Supported

WebLogic Server does not support switching the server's DOM and SAX interfaces using the endorsed standards override mechanism.

An *endorsed standard* is a Java API defined through a standards process other than the Java Community Process (JCP). For more information, see Endorsed Standards Override Mechanism.

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. Thus you can choose how to edit 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 online help.

You can download BEA XML Editor from *dev2dev Online* at http://dev2dev.bea.com/index.jsp.

Learning About XML

To learn about XML, see the following online courses and tutorials. Chapter 9, "XML Reference," provides links to more information.

- 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/docs81/servlet/index.html and *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs81/jsp/index.html.

- "Developing XML Applications: Main Steps" on page 2-1
- "Parsing XML Documents" on page 2-2
- "Generating New XML Documents" on page 2-8
- "Transforming XML Documents" on page 2-11

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 New XML Documents" on page 2-8.

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 2-11.

Parsing XML Documents

This section describes how to parse XML documents using JAXP in both DOM and SAX mode and how to parse XML documents from a servlet.

Note: For detailed instructions on using the WebLogic XML Streaming API to parse XML documents, see Chapter 4, "Using the WebLogic XML Streaming API."

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 8, "Administering WebLogic Server XML."

For a complete example of parsing an XML document in SAX mode, see the `*WL_HOME*\samples\server\examples\src\examples\xml\sax` directory, where `*WL_HOME*` refers to the top-level WebLogic Platform 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 8-4.

For a complete example of parsing an XML document in SAX mode, see the `*WL_HOME*\samples\server\examples\src\examples\xml\sax` directory, where `*WL_HOME*` refers to the top-level WebLogic Platform 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 8-4.

For a complete example of parsing an XML document in DOM mode, see the `WL_HOME\samples\server\examples\src\examples\xml\dom` directory, where `WL_HOME` refers to the top-level WebLogic Platform 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 the `setAttribute` and `getAttribute` methods to parse XML documents. Use the `setAttribute` method for SAX mode parsing and the `getAttribute` method for DOM mode parsing.

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 "For instructions on how to use the XML Registry to configure parsing options, see "XML Parser and Transformer Configuration Tasks" on page 8-3." on page 2-8, is non-validating.

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.

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 the retrieve-and-cache 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 8-5.

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 another parser of your choice (such as a different version of the Apache Xerces parser) 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 8-3.

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 8-3. Set the `SAXParserFactory` field to `weblogic.xml.babel.jaxp.SAXParserFactoryImpl`.

Generating New XML Documents

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

Note: For detailed instructions on using the WebLogic XML Streaming API to generate XML documents, see Chapter 4, "Using the WebLogic XML Streaming API."

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()` method 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`.

Transforming XML Documents

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.) This section describes how to transform XML documents using JAXP and from within a JSP using JSP tags.

Using JAXP to Transform XML Data

Version 1.1 of JAXP provides pluggable 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.

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 2-10.

Converting Your XML Code From Using 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 using 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 JAXP 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.

Table 2-1 Equivalent Xalan and JAXP Classes and Interfaces

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.

Table 2-2 x:xslt JSP Tag Attributes

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.
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.

Table 2-3 x:stylesheet JSP Tag Attributes

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 *`WL_HOME`*`\server\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, and 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 8-3.

XML Application Scoping

The following sections describe how to configure parsers, transformers, external entities, and the external entity cache for a particular application:

- "Overview of Application Scoping" on page 3-1
- "The weblogic-application.xml File" on page 3-2
- "Configuring a Parser or Transformer for an Enterprise Application" on page 3-5
- "Configuring an External Entity for an Enterprise Application" on page 3-6
- "Configuring the External Entity Cache for an Enterprise Application" on page 3-7

Overview of Application Scoping

Application scoping refers to configuring resources for a particular enterprise application rather than for an entire WebLogic Server configuration. In the case of XML, these resources include parser, transformer, external entity, and external entity cache configuration. The main advantage of application scoping is that it isolates the resources for a given application to the application itself. Using application scoping, you can configure different parsers for different applications, store the DTDs for an application within the `EAR` file or exploded enterprise directory, and so on.

Another advantage of using application scoping is that by associating the resources with the EAR file, you can run this EAR file on another instance of WebLogic Server without having to configure the resources for that server.

To configure XML resources for a particular application, you add information to the `weblogic-application.xml` deployment descriptor file located in the `META-INF` directory of the `EAR` file or exploded enterprise application directory.

Note: You use the Administration Console to configure parser, transformer, and external entity resources for a WebLogic Server instance, as described in Chapter 8, "Administering WebLogic Server XML."

The weblogic-application.xml File

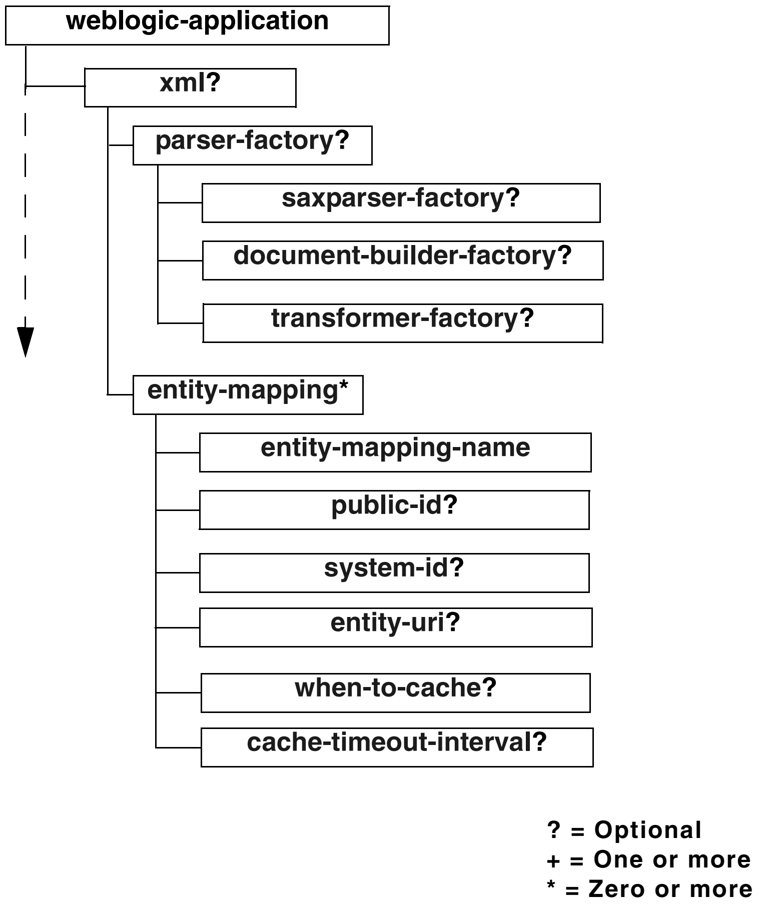
The `weblogic-application.xml` file is the WebLogic Server-specific deployment descriptor for an enterprise application. It contains configuration information about the XML, JDBC, and EJB resources used by an enterprise application. The standard J2EE deployment descriptor is called `application.xml`.

The following sample `weblogic-application.xml` file shows how to configure XML resources for an enterprise application; the body of the various elements are shown in bold:

```
<weblogic-application>
  ...
  <xml>
     <parser-factory>
        <saxparser-factory>
           weblogic.xml.babel.jaxp.SAXParserFactoryImpl
        </saxparser-factory>
        <document-builder-factory>
           org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
        </document-builder-factory>
        <transformer-factory>
          org.apache.xalan.processor.TransformerFactoryImpl
        </transformer-factory>
     </parser-factory>
     <entity-mapping>
       <entity-mapping-name>My Mapping</entity-mapping-name>
       <public-id>-//BEA Systems, Inc.//DTD for cars//EN</public-id>
       <system-id>http://www.bea.com/dtds/car.dtd</system-id>
       <entity-uri>dtds/car.dtd</entity-uri>
     </entity-mapping>
  </xml>
</weblogic-application>
```

The main element for configuring XML resources is `<xml>`. The following diagram describes the sub-elements of the `<xml>` element; the sections following the diagram describe each element.

Figure 3-1 Sub-Elements of the <xml> Element in weblogic-application.xml



xml

The main element for configuring XML resources, such as parsers, transformers, external entities, and the external entity cache for an enterprise application.

parser-factory

The parent element for specifying a particular parser or transformer for an enterprise application.

saxparser-factory

Element that specifies the factory class to be used for SAX style parsing in this application. If this element is not specified, the default SAX parser factory specified for the WebLogic Server instance is used.

document-builder-factory

Element that specifies the factory class to be used for DOM style parsing in this application. If this element is not specified, the default DOM parser factory specified for the WebLogic Server instance is used.

transformer-factory

Element that specifies the factory class to be used when transforming documents using the `javax.xml.transform` packages in this application. If this element is not specified, the default XSLT transformer factory specified for the WebLogic Server instance is used.

entity-mapping

The parent element for mapping an entity declaration in an XML file to a local copy of the entity, such as a DTD or Schema.

entity-mapping-name

Element that specifies the name of the entity mapping declaration.

public-id

Element that specifies the public ID of the entity, such as:

```
 -//BEA Systems, Inc.//DTD for cars//EN.
```

system-id

Element that specifies the system ID of the entity, such as:

`http://www.bea.com/dtds/car.dtd`

entity-uri

Element that specifies the URI of the entity. The path is relative to the main directory of the EAR archive or the exploded directory.

For example, `dtds/car.dtd` indicates that there is a directory called `dtds` in the main EAR archive (parallel to the `META-INF` directory) and it contains a file called `car.dtd`.

when-to-cache

Element that specifies when you should cache the external entity. Valid values are:

- `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.
- `cache-never`—WebLogic Server never caches the external entity.

The default value is `cache-on-reference`.

cache-timeout-interval

Element that specifies 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 main directory of the EAR archive or exploded directory if the cached copy has been in the cache for longer than this interval.

The default value for this field is 120 seconds.

Configuring a Parser or Transformer for an Enterprise Application

You can specify that an XML application use a parser or transformer different from the built-in parser or transformer configured for WebLogic Server by updating the `weblogic-application.xml` file of the EAR file or exploded directory that contains the XML application.

To configure a parser or transformer, other than the built-in, for an enterprise application, follow these steps:

1. Use the `<parser-factory>` sub-element of the `<xml>` element to configure factory classes for both SAX and DOM style parsing and for XSLT transformations for the enterprise application, as shown in the following example:

   ```
   <parser-factory>
      <saxparser-factory>
         weblogic.xml.babel.jaxp.SAXParserFactoryImpl
      </saxparser-factory>
      <document-builder-factory>
         org.apache.xerces.jaxp.DocumentBuilderFactoryImpl
      </document-builder-factory>
      <transformer-factory>
        org.apache.xalan.processor.TransformerFactoryImpl
      </transformer-factory>
   </parser-factory>
   ```

   The application corresponding to this `weblogic-application.xml` file uses the `weblogic.xml.babel.jaxp.SAXParserFactoryImpl` factory class for SAX style parsing, the `org.apache.xerces.jaxp.DocumentBuilderFactoryImpl` factory class for DOM style parsing, and the `org.apache.xalan.processor.TransformerFactoryImpl` class for XSLT transformations.

2. If you want the parser or transformer classes to be local to the EAR archive, put the JAR file that contains the classes anywhere you want in the EAR file, then update the `Class-Path` variable in the `META-INF/MANIFEST.MF` file.

   For example, if you put the parser or transformer classes in a JAR file called `myparser.jar` in the directory `lib/xml,` update the `MANIFEST.MF` file as shown:

   ```
   Manifest-Version: 1.0
   Created-By: 1.3.1_01 (Sun Microsystems Inc.)
   Class-Path: lib/xml/myparser.jar
   ```

3. If you want to store the parser or transformer classes in a location other than the EAR archive, be sure that you update the WebLogic Server CLASSPATH variable to include the full pathname of the JAR file that contains the classes.

Configuring an External Entity for an Enterprise Application

You can store a local copy of an external entity, such as a DTD, in the EAR archive or exploded directory rather than always retrieving it from the Web.

To configure an external entity for an enterprise application:

1. Create the directory `lib/xml/registry` under the main directory of the EAR archive.
2. Copy the external entity, such as a DTD, to the directory.
3. Update the `weblogic-application.xml` file, using the `<entity-mapping>` sub-element of the `<xml>` element to map the name of the entity to entity declarations in any XML files processed by the application, as shown in the following example:

   ```
   <entity-mapping>
     <entity-mapping-name>My Mapping</entity-mapping-name>
     <public-id>-//BEA Systems, Inc.//DTD for cars//EN</public-id>
     <system-id>http://www.bea.com/dtds/car.dtd</system-id>
     <entity-uri>dtds/car.dtd</entity-uri>
   </entity-mapping>
   ```

   In the example, a local copy of a DTD called `car.dtd` is stored in the `lib/xml/registry/dtds` directory under the main directory of the EAR archive or exploded directory. The public ID of the entity is `-//BEA Systems, Inc.//DTD for cars//EN` and the system id is `http://www.bea.com/dtds/car.dtd`. Whenever the application is parsing an XML file and it encounters an entity declaration using either one of the IDs, it will substitute the `car.dtd` file.

Note: Specify an `<entity-mapping>` element for each entity declaration for which you want to map a local copy of the entity.

Configuring the External Entity Cache for an Enterprise Application

You can specify that WebLogic Server cache external entities that are referenced with a URL or a pathname relative to the main directory of the EAR archive, either at server-startup or when the entity is first referenced.

Caching the external entity 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 of a cached entity, at which point WebLogic Server re-retrieves the entity from the URL or directory of the EAR and re-caches it.

Use the `<when-to-cache>` and `<cache-timeout-interval>` subelements of the `<entity-mapping>` element to configure external entity caching for an enterprise application, as shown in the following example:

```
<entity-mapping>
  <entity-mapping-name>My Mapping</entity-mapping-name>
  <public-id>-//BEA Systems, Inc.//DTD for cars//EN</public-id>
  <system-id>http://www.bea.com/dtds/car.dtd</system-id>
  <entity-uri>dtds/car.dtd</entity-uri>
 <when-to-cache>cache-at-initialization</when-to-cache>
  <cache-timeout-interval>300</cache-timeout-interval>
</entity-mapping>
```

In the example, the `car.dtd` is stored in the `lib/xml/registry/dtds` directory under the main directory of the EAR archive or exploded directory. WebLogic Server caches a copy of the DTD in its memory when it first starts up, and then refreshes the cached copy if it is stored for longer than 300 seconds.

CHAPTER 4

Using the WebLogic XML Streaming API

The following sections describe how to use the WebLogic XML Streaming API to parse and generate XML documents:

- "Overview of the WebLogic XML Streaming API" on page 4-1
- "Javadocs for the WebLogic XML Streaming API" on page 4-3
- "Parsing an XML Document: Typical Steps" on page 4-3
- "Generating a New XML Document: Typical Steps" on page 4-18

Overview of the WebLogic XML Streaming API

The WebLogic XML Streaming API provides an easy and intuitive way to parse and generate XML documents. It is similar to the SAX API, but enables a procedural, stream-based handling of XML documents rather than requiring you to write SAX event handlers, which can get complicated when you work with complex XML documents. In other words, the streaming API gives you more control over parsing than the SAX API.

When a program parses an XML document using SAX, the program must create event listeners that listen to parsing events as they occur; the program must react to events rather than ask for a specific event. By contrast, when you use the streaming API, you can methodically step through an XML document, ask for certain types of events (such as the start of an element), iterate over the attributes of an element, skip ahead in the document, stop processing at any time, get sub-elements of a particular element, and filter out elements as desired. Because you are asking for events rather than reacting to them, using the streaming API is often referred to as *pull parsing*.

You can parse many types of XML documents with the streaming API, such as XML files on the operating system, DOM trees, and sets of SAX events. You convert these XML documents into a stream of events, or an `XMLInputStream`, and then step through the stream, pulling events such as the start of an element, the end of the document, and so on, off the stack as needed.

The WebLogic Streaming API uses the WebLogic FastParser as its default parser.

For a complete example of parsing an XML document using the streaming API, see the *`WL_HOME`*`\samples\server\examples\src\examples\xml\orderParser` directory, where *`WL_HOME`* refers to the top-level WebLogic Platform directory.

The following table describes the main interfaces and classes of the WebLogic Streaming API.

Table 4-1 Interfaces and Classes of the XML Streaming API

Interface or Class	Description
`XMLInputStreamFactory`	Factory used to create `XMLInputStream` objects for parsing XML documents.
`XMLInputStream`	Interface used to contain the input stream of events.
`BufferedXMLInputStream`	Extension of the `XMLInputStream` interface to allow marking and resetting of the stream.
`XMLOutputStreamFactory`	Factory used to create `XMLOutputStream` objects for generating XML documents.
`XMLOutputStream`	Interface used write events.
`ElementFactory`	Utility to create instances of the interfaces used in this API.
`XMLEvent`	Base interface for all types of events in an XML document, such as the start of an element, the end of an element, and so on.
`StartElement`	Most important of the `XMLEvent` sub-interfaces. Used to get information about a start element in an XML document.
`AttributeIterator`	Object used to iterate over the set of attributes of an element.
`Attribute`	Object that describes a particular attribute of an element.

Javadocs for the WebLogic XML Streaming API

The following Javadocs provide reference material for the WebLogic XML Streaming API features described in this chapter as well as additional features not explicitly documented:

- weblogic.xml.stream at http://e-docs.bea.com/wls/docs81/javadocs/weblogic/xml/stream/package-summary.html
- weblogic.xml.stream.util at http://e-docs.bea.com/wls/docs81/javadocs/weblogic/xml/stream/util/package-summary.htm l

Parsing an XML Document: Typical Steps

The following procedure describes the typical steps for using the WebLogic XML Streaming API to parse and manipulate an XML document.

The first two steps are required. The next steps you take depend on how you want to process the XML file.

1. Import the `weblogic.xml.stream.*` classes.
2. Get an XML stream of events from an XML file, a DOM tree, or a set of SAX events. You can also filter the XML stream to get only certain types of events, names of specific elements, and so on. See “Getting an XML Input Stream” on page -7.
3. Iterate over the stream, returning generic `XMLEvent` types. See “Iterating Over the Stream” on page -10.
4. For each generic `XMLEvent` type, determine the specific event type. Event types include the start of an XML document, the end of an element, an entity reference, and so on. See “Determining the Specific XMLEvent Type” on page -10.
5. Get the attributes of an element. See “Getting the Attributes of an Element” on page -14.
6. Position the stream by skipping over event, skipping to a particular event, and so on. See “Positioning the Stream” on page -15.
7. Get the children of an element. See “Getting a Substream” on page -16.
8. Close the stream. See “Closing the Input Stream” on page -18.

Example of Parsing an XML Document

The following program shows an example of using the XML Streaming API to parse an XML document.

The program takes a single parameter, an XML file, that it converts into an XML input stream. It then iterates over the stream, determining the type of each event, such as the start of an XML element, the end of the XML document, and so on. The program prints out information for three types of events: start elements, end elements, and the character data that forms the body of an element. The program does nothing when it encounters the other types of events, such as comments or start of the XML document.

Note: The code in bold font is described in detail in the sections following the example.

```
package examples.xml.stream;

import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.AttributeIterator;
import weblogic.xml.stream.ChangePrefixMapping;
import weblogic.xml.stream.CharacterData;
import weblogic.xml.stream.Comment;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.EndDocument;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.EntityReference;
import weblogic.xml.stream.ProcessingInstruction;
import weblogic.xml.stream.Space;
import weblogic.xml.stream.StartDocument;
import weblogic.xml.stream.StartPrefixMapping;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.EndPrefixMapping;
import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLInputStreamFactory;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLStreamException;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class ComplexParse {

  /**
   * Helper method to get a handle on a stream.
   * Takes in a name and returns a stream.  This
   * method usese the InputStreamFactory to create an
   * instance of an XMLInputStream
   * @param name The file to parse
   * @return XMLInputStream the stream to parse
```

```
 */
public XMLInputStream getStream(String name)
  throws XMLStreamException, FileNotFoundException
{
  XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
  XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
  return stream;
}

/**
 *  Determines the type of event, such as the start
 *  of an element, end of a document, and so on.  If the
 *  event is of type START_ELEMENT, END_ELEMENT, or
 *  CHARACTER_DATA, the method prints out appropriate info;
 *  otherwise, it does nothing.
 *  @param event The XML event that has been parsed
 */
public void parse(XMLEvent event)
  throws XMLStreamException
{
  switch(event.getType()) {
  case XMLEvent.START_ELEMENT:
    StartElement startElement = (StartElement) event;
    System.out.print("<" + startElement.getName().getQualifiedName() );
    AttributeIterator attributes = startElement.getAttributesAndNamespaces();
    while(attributes.hasNext()){
      Attribute attribute = attributes.next();
      System.out.print(" " + attribute.getName().getQualifiedName() +
                        "='" + attribute.getValue() + "'");
    }
    System.out.print(">");
    break;
  case XMLEvent.END_ELEMENT:
    System.out.print("</" + event.getName().getQualifiedName() +">");
    break;
  case XMLEvent.SPACE:
  case XMLEvent.CHARACTER_DATA:
    CharacterData characterData = (CharacterData) event;
    System.out.print(characterData.getContent());
    break;
  case XMLEvent.COMMENT:
    // Print comment
    break;
  case XMLEvent.PROCESSING_INSTRUCTION:
    // Print ProcessingInstruction
    break;
  case XMLEvent.START_DOCUMENT:
    // Print StartDocument
    break;
    case XMLEvent.END_DOCUMENT:
      // Print EndDocument
      break;
    case XMLEvent.START_PREFIX_MAPPING:
      // Print StartPrefixMapping
      break;
    case XMLEvent.END_PREFIX_MAPPING:
      // Print EndPrefixMapping
      break;
    case XMLEvent.CHANGE_PREFIX_MAPPING:
      // Print ChangePrefixMapping
      break;
    case XMLEvent.ENTITY_REFERENCE:
      // Print EntityReference
      break;
    case XMLEvent.NULL_ELEMENT:
      throw new XMLStreamException("Attempt to write a null event.");
    default:
      throw new XMLStreamException("Attempt to write unknown event
                                   ["+event.getType()+"]");
    }
  }
  /**
   * Helper method to iterate over a stream
   * @param name The file to parse
   */
  public void parse(XMLInputStream stream)
    throws XMLStreamException
  {
    while(stream.hasNext()) {
      XMLEvent event = stream.next();
      parse(event);
    }
    stream.close();
  }

  /** Main method.   Takes a single argument: an XML file
   *  that will be converted into an XML input stream.
   */
  public static void main(String args[])
    throws Exception
  {
    ComplexParse complexParse= new ComplexParse();
    complexParse.parse(complexParse.getStream(args[0]));
  }
}
```

Getting an XML Input Stream

You can use the XML Streaming API to convert a variety of objects, such as XML files, DOM trees, or SAX events, into a stream of events.

The following example shows how to create a stream of events from an XML file:

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
```

First you create a new instance of the `XMLInputStreamFactory`, then use the factory to create a new `XMLInputStream` from the XML file referred to in the `name` variable.

The following example shows how to create a stream from a DOM tree:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(false);
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(new java.io.File(file));
XMLInputStream stream =
XMLInputStreamFactory.newInstance().newInputStream(doc);
```

Getting a Buffered XML Input Stream

After you finish iterating over an `XMLInputStream` object, you cannot access the stream again. If, however, you need to process the stream again, such as send it to another application or iterate over it again in some other way, use a `BufferedXMLInputStream` object rather than a plain `XMlInputStream` object.

Use the `newBufferedInputStream()` method of the `XMLInputStreamFactory` class to create a buffered XML input stream, as shown in the following example:

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
BufferedXMLInputStream bufstream =
    factory.newBufferedInputStream(factory.newInputStream(new
    FileInputStream(name)));
```

You can use the `mark()` and `reset()` methods of the `BufferedXMLInputStream` object to mark a particular spot in the stream, continue processing the stream, then reset the stream back to the marked spot. See “Marking and Resetting a Buffered XML Input Stream” on page 4-17 for more information.

Filtering the XML Stream

Filtering an XML stream refers to creating a stream that contains only specified types of events. For example, you can create a stream that contains only start elements, end elements, and the character data that make up the body of an XML element. Another example is filtering an XML stream so that only elements with a specified name appear in the stream.

To filter an XML stream, you specify a filter class as the second parameter to the `XMLInputStreamFactory.newInputStream()` method. You specify the events that you want in the XML stream as parameters to the filter class. The following example shows how to use the `TypeFilter` class to specify that you want only start and end XML elements and character data in the resulting XML stream:

```
import weblogic.xml.stream.util.TypeFilter;

XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new FileInputStream(name),
                       new TypeFilter(XMLEvent.START_ELEMENT |
                                      XMLEvent.END_ELEMENT |
                                      XMLEvent.CHARACTER_DATA));
```

The following table describes the filters provided by the WebLogic XML Streaming API. They are part of the `weblogic.xml.stream.util` package.

Table 4-2

Name of Filter	Description	Sample Usage
`TypeFilter`	Filter an XML stream based on specified event types, such as `XMLEvent.START_ELEMENT`, `XMLEvent.END_ELEMENT`, and so on. See “Determining the Specific XMLEvent Type” on page -10 for a full list of event types. TypeFilter takes an integer bitmask as input; you OR the values to create this bitmask, as shown in the sample.	`new TypeFilter (XMLEvent.START_ELEMENT | XMLEvent.END_ELEMENT | XMLEvent.CHARACTER_DATA)`
`NameFilter`	Filter an XML stream based on the name of an element in the XML document.	`new NameFilter ("Book")`

Table 4-2

Name of Filter	Description	Sample Usage
`NameSpaceFilter`	Filter an XML stream based on the specified namespace URI.	`new NameSpaceFilter ("http://namespace.org")`
`NamespaceTypeFilter`	Filter an XML stream based on specified event types and namespace URI. This filter combines the functionality of `TypeFilter` and `NameSpaceFilter`.	`new NamespaceFilter ("http://namespace.org", XMLEvent.START_ELEMENT)` The example returns a stream where all start elements have the specified namespace.

Creating a Custom Filter

You can also create your own filter if the ones included in the API do not meet your needs.

1. Create a class that implements the `ElementFilter` interface and contains a method called `accept(XMLEvent)`. This method tells the `XMLInputStreamFactory.newInputStream()` method whether to add a particular event to the stream or not, as shown in the following example:

```
package my.filters;

import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.ElementFilter;
import weblogic.xml.stream.events.NullEvent;

public class SuperDooperFilter implements ElementFilter {

  protected String name;

  public SuperDooperFilter(String name)
  {
    this.name = name;
  }

  public boolean accept(XMLEvent e) {
    if (name.equals(e.getName().getLocalName()))
      return true;
    return false;
  }
}
```

2. In your XML application, be sure to import the new filter class:

```
import my.filters.SuperDooperFilter
```

3. Specify the filter as the second parameter to the `newInputStream()` method, passing to the filter class the types of events you want to appear in the XML stream in whatever format required by your filter class:

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new FileInputStream(name),
                        new SuperDooperFilter(param));
```

Iterating Over the Stream

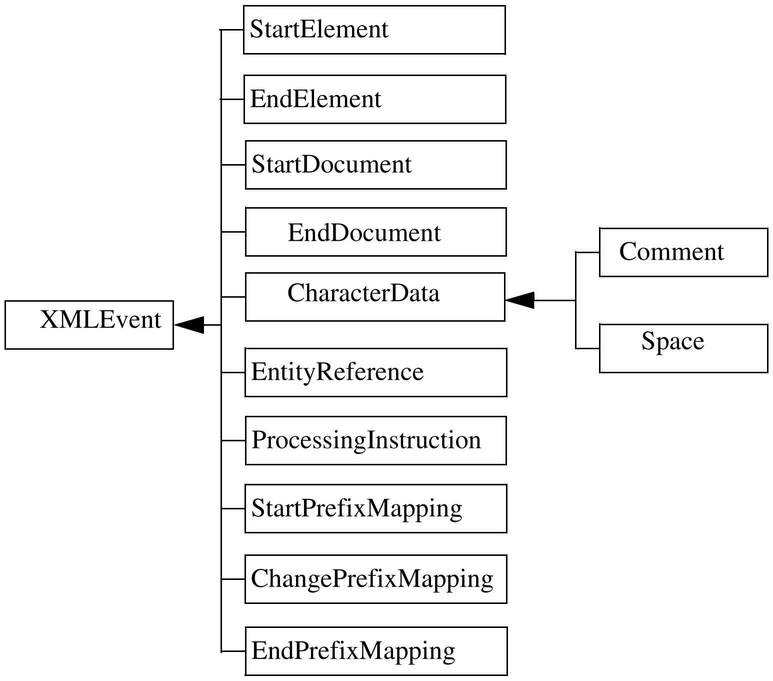
Once you have a stream of events, the next step is to methodically step through it using the `XMLInputStream.next()` and `XMLInputStream.hasNext()` methods, as shown in the following example:

```
while(stream.hasNext()) {
  XMLEvent event = stream.next();
  System.out.print(event);
}
```

Determining the Specific XMLEvent Type

The `XMLInputStream.next()` method returns an object of type `XMLEvent`. `XMLEvent` has subinterfaces that further classify what this event might be, such as the start of the XML document, the end of an element, an entity reference, and so on. The `XMLEvent` interface also contains corresponding fields, or constants, as well as a set of methods that you can use to identify the actual event. The following diagram shows the hierarchy of the `XMLEvent` interface and its subinterfaces.

Figure 4-1 Hierarchy of the XMLEvent Interface and Its SubInterfaces



The following table lists the subclasses and fields of the `XMLEvent` class that you can use to identify a particular event while parsing the XML stream.

Table 4-3 Subclasses and Fields of the XMLEvent Class

XMLEvent Subclass	Field of the XMLEvent Class used to Identify Subclass	Method used to Identify Subclass	Description of the Subclass Event
ChangePrefixMapping	CHANGE_PREFIX_MAPPING	isChangePrefixMapping	Signals that a prefix mapping has changed from an old namespace to a new namespace.
CharacterData	CHARACTER_DATA	isCharacterData	Signals that the returned `XMLEvent` object contains the character data from the body of the element.

Table 4-3 Subclasses and Fields of the XMLEvent Class

XMLEvent Subclass	Field of the XMLEvent Class used to Identify Subclass	Method used to Identify Subclass	Description of the Subclass Event
Comment	COMMENT	isComment	Signals that the returned `XMLEvent` object contains an XML comment.
EndDocument	END_DOCUMENT	isEndDocument	Signals the end of the XML document.
EndElement	END_ELEMENT	isEndElement	Signals the end of an element in the XML document.
EndPrefixMapping	END_PREFIX_MAPPING	isEndPrefixMapping	Signals that a prefix mapping has gone out of scope.
EntityReference	ENTITY_REFERENCE	isEntityReference	Signals that the returned `XMLEvent` object contains an entity reference.
ProcessingInstruction	PROCESSING_INSTRUCTION	isProcessingInstruction	Signals that the returned `XMLEvent` object contains a processing instruction.
Space	SPACE	isSpace	Signals that the returned `XMLEvent` object contains whitespace.
StartDocument	START_DOCUMENT	isStartDocument	Signals the start of an XML document.
StartElement	START_ELEMENT	isStartElement	Signals the start of a element in the XML document.
StartPrefixMapping	START_PREFIX_MAPPING	isStartPrefixMapping	Signals that a prefix mapping has started its scope.

The following example shows how to use the Java `case` statement to determine the particular type of event that was returned by the `XMLInputStream.next()` method. For simplicity, the example simply prints that an event has been found; later sections show further processing of the event.

```
switch(event.getType()) {
case XMLEvent.START_ELEMENT:
  // Start of an element
  System.out.println ("Start Element\n");
  break;

case XMLEvent.END_ELEMENT:
  // End of an element
  System.out.println ("End Element\n");
  break;

case XMLEvent.PROCESSING_INSTRUCTION:
  // Processing Instruction
  System.out.println ("Processing instruction\n");
  break;

case XMLEvent.SPACE:
  // Whitespace
  System.out.println ("White space\n");
  break;

case XMLEvent.CHARACTER_DATA:
  // Character data
  System.out.println ("Character data\n");
  break;

case XMLEvent.COMMENT:
  // Comment
  System.out.println ("Comment\n");
  break;

case XMLEvent.START_DOCUMENT:
  // Start of the XML document
  System.out.println ("Start Document\n");
  break;

case XMLEvent.END_DOCUMENT:
  // End of the XML Document
  System.out.println ("End Document\n");
  break;

case XMLEvent.START_PREFIX_MAPPING:
  // The start of a prefix mapping scope
```

```
      System.out.println ("Start prefix mapping\n");
      break;

    case XMLEvent.END_PREFIX_MAPPING:
      // The end of a prefix mapping scope
      System.out.println ("End prefix mapping\n");
      break;

    case XMLEvent.CHANGE_PREFIX_MAPPING:
      // Prefix mapping has changed namespaces
      System.out.println ("Change prefix mapping\n");
      break;

    case XMLEvent.ENTITY_REFERENCE:
      // An entity reference
      System.out.println ("Entity reference\n");
      break;
    default:

      throw new XMLStreamException("Attempt to parse unknown event
                                  [" + event.getType() + "]");
    }
```

Getting the Attributes of an Element

To get the attributes of an element in an XML document, you must first cast the `XMLEvent` object that was returned by the `XMLInputStream.next()` method to a `StartElement` object.

Because you do not know how many attributes an element might have, you must first create an `AttributeIterator` object to contain the entire list of attributes, and then iterate over the list until there are no more attributes. The following example describes how to do this as part of the `START_ELEMENT` case of the `switch` statement shown in "Iterating Over the Stream" on page -10:

```
case XMLEvent.START_ELEMENT:

    StartElement startElement = (StartElement) event;
    System.out.print("<" + startElement.getName().getQualifiedName() );
    AttributeIterator attributes = startElement.getAttributesAndNamespaces();
    while(attributes.hasNext()){
      Attribute attribute = attributes.next();
      System.out.print(" " + attribute.getName().getQualifiedName() +
                       "='" + attribute.getValue() + "'");
    }
    System.out.print(">");
    break;
```

The example first creates a `StartElement` object by casting the returned `XMLEvent` to `StartElement`. It then creates an `AttributeIterator` object using the method

`StartElement.getAttributesAndNamespaces()`, and iterates over the attributes using the `AttributeIterator.hasNext()` method. For each `Attribute`, it uses the `Attributes.getName().getQualifiedName()` and `Attribute.getValue()` methods to return the name and value of the attribute.

You can also use the `getNamespace()` and `getAttributes()` methods to return just the namespaces or attributes on their own.

Positioning the Stream

The following table describes the methods of the `XMLInputStream` interface that you can use to skip ahead to specific locations in the stream.

Table 4-4 Methods Used to Position the Input Stream

Method of XMLInputStream	Description
`skip()`	Positions the input stream to the next stream event. **Note:** The next event might not necessarily be an actual element in the XML file; for example, it could be a comment or white space.
`skip(int)`	Positions the input stream to the next event of this type. Examples of event types are `XMLEvent.START_ELEMENT` and `XMLEvent.END_DOCUMENT`. Refer to Table 4-3 for the full list of event types.
`skip(XMLName)`	Positions the input stream to the next event of this name.
`skip(XMLName, int)`	Positions the input stream to the next event of this name and type.
`skipElement()`	Skips to the next element (does not skip to the sub-elements of the current element).
`peek()`	Checks the next event without actually reading it from the stream.

The following example shows how you can modify the basic code for iterating over an input stream to skip over the character data in the body of an XML element:

```
while(stream.hasNext()) {
  XMLEvent peek = stream.peek();
```

```
      if (peek.getType() == XMLEvent.CHARACTER_DATA ) {
        stream.skip();
        continue;
      }
      XMLEvent event = stream.next();
      parse(event);
    }
```

The example shows how to use the `XMLInputStream.peek()` method to determine the next event on the stream. If the type of event is `XMLEvent.CHARACTER_DATA`, then skip the event and go to the next one.

Getting a Substream

Use the `XMLInputStream.getSubStream()` method to get a copy of the next element, including all its subelements. The `getSubstream()` method returns an `XMLInputStream` object. Your position in the parent stream (or the stream from which you called `getSubStream()`) does not move. In the parent stream, if you want to skip the element you just got with `getSubStream()`, use the `skipElement()` method.

The `getSubStream()` method keeps a count of the `START_ELEMENT` and `END_ELEMENT` events it encounters, and as soon as the number is equal (or in other words, as soon as it finds the complete next element) it stops and returns the resulting substream as an `XMLInputStream` object.

For example, assume that you are using the XML Streaming API to parse the following XML document, but you are only interested in the substream delineated by the `<content>` and `</content>` tags:

```
<book>
  <title>The History of the World</title>
  <author>Juliet Shackell</author>
  <publisher>CrazyDays Publishing</publisher>
  <content>
      <chapter title='Just a Speck of Dust'>
        <synopsis>The world as a speck of dust</synopsis>
        <para>Once the world was just a speck of dust...</para>
      </chapter>
      <chapter title='Life Appears'>
        <synopsis>Move over dust, here comes life.</synopsis>
        <para>Happily, the dust got a companion: life...</para>
      </chapter>
   </content>
</book>
```

The following code fragment shows how you can skip to the `<content>` start element tag, get the substream, and parse it using a separate `ComplexParse` object:

```
 if (stream.skip( ElementFactory.createXMLName("content")))
   {
      ComplexParse complexParse = new ComplexParse();
      complexParse.parse(stream.getSubStream());
   }
```

When you call this method on the previous XML document, you get the following output:

```
<content>
      <chapter title='Just a Speck of Dust'>
        <synopsis>The world as a speck of dust</synopsis>
        <para>Once the world was just a speck of dust...</para>
      </chapter>
      <chapter title='Life Appears'>
        <synopsis>Move over dust, here comes life.</synopsis>
        <para>Happily, the dust got a companion: life...</para>
      </chapter>
</content>
```

Marking and Resetting a Buffered XML Input Stream

If you are using a `BufferedXMLInputStream` object, you can use the `mark()` and `reset()` methods to mark the stream at a particular spot, process the stream, and then subsequently reset the stream back to the marked spot. These methods are useful if you want to further manipulate the stream after initially iterating over it.

Note: If you read a buffered stream without marking it, you cannot access what you've just read. In other words, just because the stream is buffered, it does not automatically mean you can reread it. You must mark it first.

The following example shows a typical use of the `BufferedXMLInputStream` object:

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
BufferedXMLInputStream bufstream =
factory.newBufferedInputStream(factory.newInputStream(new
            FileInputStream(name)));
```

```
// mark the start of the stream
bufstream.mark();

// process it locally
bufferedParse.parse(bufstream);

// reset the stream to the mark
bufstream.reset();

// send stream off to another application
ComplexParse complexParse =  new ComplexParse();
complexParse.parse(bufstream);
```

Closing the Input Stream

It is good programming practice to explicitly close the XML input stream when you are finished with it.   To close an input stream, use the `XMLInputStream.close()` method, as shown in the following example:

```
// close the input stream
input.close();
```

Generating a New XML Document: Typical Steps

The following procedure describes the typical steps for using the WebLogic XML Streaming API to generate a new XML document.

The first two steps are required. The next steps you take depend on how you want to generate the XML file.

1. Import the `weblogic.xml.stream.*` classes.
2. Create an XML output stream to which to write the XML document. See “Creating an XML Output Stream” on page -21.
3. Add events to the XML output stream. See “Adding Elements to the Output Stream” on page -22.
4. Add attributes to the XML output stream. See “Adding Attributes to an Element on the Output Stream” on page -23.
5. Add an input stream to the output stream. See “Adding an Input Stream to an Output Stream” on page -23.
6. Print the output stream. See “Printing an Output Stream” on page -24.

7. Close the output stream. See “Closing the Output Stream” on page -25.

Example of Generating an XML Document

The following program shows an example of using the XML Streaming API to generate an XML document.

The program first creates an output stream based on a `PrintWriter` object, then adds elements to the output stream to create a simple XML purchase order, described in the comments of the program. The program also shows how to add an input stream based on a separate XML file to the output stream.

Note: The topics following the example describe it in more detail.

```
package examples.xml.stream;

import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLOutputStream;
import weblogic.xml.stream.XMLInputStreamFactory;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.ElementFactory;
import weblogic.xml.stream.XMLStreamException;
import weblogic.xml.stream.XMLOutputStreamFactory;


import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.PrintWriter;

/**
 * Program that prints out a very simple purchase order that looks
 * like the following:
 *
 * <purchase_order>
 *  <name>Juliet Shackell</name>
 *  <item id="1234" quantity="2">Fabulous Chair</item>
 *  <!-- this is a comment-->
 *  <another_file>
 *    This comes from another file called "another_file.xml"
 *  </another_file>
 * </purchase_order>
 *
 * In the preceding XML file, the <another_file> element is actually another
 * XML file that is passed as an argument to the program, converted into an
 * XMLInputStream, then added to the output stream.
 */
public class PrintPurchaseOrder {

  /**
   * Helper method to get a handle on a stream.
   * Takes in a name and returns a stream.  This
   * method uses the InputStreamFactory to create an
   * instance of an XMLInputStream
   * @param name The file to parse
   * @return XMLInputStream the stream to parse
   */
  public XMLInputStream getInputStream(String name)
    throws XMLStreamException, FileNotFoundException
  {
    XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
    XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
    return stream;
  }
  public static void main(String args[])
    throws Exception
  {
    PrintPurchaseOrder printer = new PrintPurchaseOrder();
    //
    // Create an output stream.
    //
    XMLOutputStreamFactory factory = XMLOutputStreamFactory.newInstance();
    XMLOutputStream output = factory.newOutputStream(new
                                    PrintWriter(System.out,true));

   // add the <purchase_order> root element
   output.add(ElementFactory.createStartElement("purchase_order"));
   output.add(ElementFactory.createCharacterData("\n"));

   // add the <name> element
   output.add(ElementFactory.createStartElement("name"));
   output.add(ElementFactory.createCharacterData("Juliet Shackell"));
   output.add(ElementFactory.createEndElement("name"));
   output.add(ElementFactory.createCharacterData("\n"));

   // add the <item> element along with the id and quantity attributes
   output.add(ElementFactory.createStartElement("item"));
   output.add(ElementFactory.createAttribute("id","1234"));
   output.add(ElementFactory.createAttribute("quantity","2"));
   output.add(ElementFactory.createCharacterData("Fabulous Chair"));
   output.add(ElementFactory.createEndElement("item"));
   output.add(ElementFactory.createCharacterData("\n"));
```

```
     // add a comment
    output.add("<!-- this is a comment-->");
    output.add(ElementFactory.createCharacterData("\n"));

   // create an input stream from each XML file argument then add it to the output
    for (int i=0; i < args.length; i++)
    //
    // Get an input stream and add it to the output stream
    //
    output.add(printer.getInputStream(args[i]));

     // Finally, end the root "purchase_order" element
    output.add(ElementFactory.createEndElement("purchase_order"));
    output.add(ElementFactory.createCharacterData("\n"));

     //
     // Print the results to the screen
     //
     output.flush();

    // Close the output streams
    output.close();
  }
}
```

The preceding program produces the following output:

```
<purchase_order>
  <name>Juliet Shackell</name>
  <item id="1234" quantity="2">Fabulous Chair</item>
  <!-- this is a comment-->
  <another_file>
     This is from another file.
  </another_file>
</purchase_order>
```

Creating an XML Output Stream

One of the first steps in generating an XML document using the Weblogic XML Streaming API is to create an output stream which holds the document as it is being built. Creating an XML output stream is similar to creating an input stream: you first create an instance of the `XMLOutputStreamFactory` and then create an output stream with the `XMLOutputStreamFactory.newOutputStream()` method, as shown in the following example:

```
XMLOutputStreamFactory factory = XMLOutputStreamFactory.newInstance();
XMLOutputStream output = factory.newOutputStream(new
                           PrintWriter(System.out,true));
```

The following example shows how to create an `XMLOutputStream` based on a DOM tree:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setValidating(false);
dbf.setNamespaceAware(true);
Document doc = dbf.newDocumentBuilder().newDocument();
XMLOutputStream out =
     XMLOutputStreamFactory.newInstance().newOutputStream(doc);
```

You can use the `XMLOutputStreamFactory.newOutputStream()` method to create an output stream based on the following four Java objects, depending on what the final form of the XML document will be (such as a file on the operating system, a DOM tree, and so on):

- `java.io.OutputStream`
- `java.io.Writer`
- `org.xml.sax.ContentHandler`
- `org.w3c.dom.Document`

Adding Elements to the Output Stream

Use the `XMLOutputStream.add(XMLEvent)` method to add elements to the output stream. Use the `ElementFactory` to create the particular element.

The `ElementFactory` interface includes methods to create each type of element; the general format is `ElementFactory.create`*`XXX`*`()` where *`XXX`* refers to the particular element, such as `createStartElement()`, `createCharacterData()`, and so on. You can create most elements by passing the name as a `String` or as an `XMLName`.

Warning: The `XMLOutputStream` does not validate your XML.

Note: Each time you create a start element, you must explicitly also create an end element at some point. The same rule applies to creating a start document.

For example, assume you want to create the following snippet of XML:

```
<name>Juliet Shackell</name>
```

The Java code to add this element to an output stream is as follows:

```
output.add(ElementFactory.createStartElement("name"));
output.add(ElementFactory.createCharacterData("Juliet Shackell"));
output.add(ElementFactory.createEndElement("name"));
output.add(ElementFactory.createCharacterData("\n"));
```

The final `createCharacterData()` method adds a newline character to the output stream. This is optional, but useful if you want to create human-readable XML.

Adding Attributes to an Element on the Output Stream

Use the `XMLOutputStream.add(Attribute)` to add attributes to an element you have just created. Use the `ElementFactory.createAttribute()` method to create a particular attribute.

For example, assume you want to create the following snippet of XML:

```
<item id="1234" quantity="2">Fabulous Chair</item>
```

The Java code to add this element to an output stream is as follows:

```
output.add(ElementFactory.createStartElement("item"));
output.add(ElementFactory.createAttribute("id","1234"));
output.add(ElementFactory.createAttribute("quantity","2"));
output.add(ElementFactory.createCharacterData("Fabulous Chair"));
output.add(ElementFactory.createEndElement("item"));
output.add(ElementFactory.createCharacterData("\n"));
```

Note: Be sure you add attributes to an element *after* you create the start element but *before* you create the corresponding end element. Otherwise, although your code will compile successfully, you will get a runtime error when you try to run the program. For example, the following code returns an error because the attributes are added to the `<item>` element *after* the element has been explicitly ended:

```
output.add(ElementFactory.createStartElement("item"));
output.add(ElementFactory.createEndElement("item"));
output.add(ElementFactory.createAttribute("id","1234"));
output.add(ElementFactory.createAttribute("quantity","2"));
output.add(ElementFactory.createCharacterData("Fabulous Chair"));
output.add(ElementFactory.createCharacterData("\n"));
```

Adding an Input Stream to an Output Stream

When creating an XML output stream, you might want to add an existing XML document, such as an XML file or a DOM tree, to the output stream. To do this, you must first convert the XML document to an XML input stream, then use `XMLOutputStream.add(XMLInputStream)` method to add the input stream to the output stream.

The following example first shows a method called getInputStream() that creates an XML input stream from an XML file and then how to use the method to add the created input stream to an output stream:

```
  /**
   * Helper method to get a handle on a stream.
   * Takes in a name and returns a stream.  This
   * method uses the InputStreamFactory to create an
   * instance of an XMLInputStream
   * @param name The file to parse
   * @return XMLInputStream the stream to parse
   */

  public XMLInputStream getInputStream(String name)
    throws XMLStreamException, FileNotFoundException
  {
    XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
    XMLInputStream stream = factory.newInputStream(new FileInputStream(name));
    return stream;
  }

....

   // create an input stream from each XML file argument then add it to the output

    for (int i=0; i < args.length; i++)
    //
    // Get an input stream and add it to the output stream
    //
    output.add(printer.getInputStream(args[i]));
```

Printing an Output Stream

Use the `XMLOutputStream.flush()` method to print out the XML output stream to whatever object you created it from. For example, if you created an XML output stream from a `PrintWriter` object, then the `flush()` method prints the stream to the standard output.

Note: If you are writing to an XMLOutputStream based on a DOM tree, you must execute the `flush()` method before you can manipulate the DOM.

The following example shows how to print an output stream:

```
    //
    // Print the results to the screen
    //
    output.flush();
```

Closing the Output Stream

It is good programming practice to explicitly close the XML output stream when you are finished with it. To close an output stream, use the `XMLOutputStream.close()` method, as shown in the following example:

```
// close the output stream
output.close();
```

Using Advanced XML APIs

The following sections provide information about using the following advanced XML APIs:

- "Using the WebLogic XPath API" on page 5-1
- "Using the WebLogic XML Digital Signature API" on page 5-10

Using the WebLogic XPath API

The WebLogic XPath API contains all of the classes required to perform XPath matching against a document represented as a `DOM`, an `XMLInputStream`, or an `XMLOutputStream`. Use the API if you want to identify a subset of XML elements within an XML document that conform to a given pattern.

For additional API reference information about the WebLogic XPath API, see the weblogic.xml.xpath Javadoc at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

Using the DOMXPath Class

This section describes how to use the `DOMXPath` class of the WebLogic XPath API to perform XPath matching against an XML document represented as a `DOM`. The section first provides an example and then a description of the main steps used in the example.

Example of Using the DOMXPath Class

The sample Java program at the end of this section uses the following XML document in its matching:

```
<?xml version='1.0' encoding='us-ascii'?>

<!-- "Purchaseorder". -->

<purchaseorder
    department="Sales"
    date="01-11-2001"
    raisedby="Pikachu"
    >
    <item
        ID="101">
        <title>Laptop</title>
        <quantity>5</quantity>
        <make>Dell</make>
    </item>
    <item
        ID="102">
        <title>Desktop</title>
        <quantity>15</quantity>
        <make>Dell</make>
    </item>
    <item
        ID="103">
        <title>Office Software</title>
        <quantity>10</quantity>
        <make>Microsoft</make>
    </item>
</purchaseorder>
```

The Java code example is as follows:

```
package examples.xml.xpath;

import java.io.IOException;
import java.util.Iterator;
import java.util.Set;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;
import weblogic.xml.xpath.DOMXPath;
import weblogic.xml.xpath.XPathException;

/**
 * This class provides a simple example of how to use the DOMXPath
 * API.
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

public abstract class DOMXPathExample {

  public static void main(String[] ignored)

    throws XPathException, ParserConfigurationException,

           SAXException, IOException

  {

    // create a dom representation of the document

    String file = "purchaseorder.xml";

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

    factory.setNamespaceAware(true); // doesn't matter for this example

    DocumentBuilder builder = factory.newDocumentBuilder();

    Document doc = builder.parse(file);

    // create some instances of DOMXPath to evaluate against the

    // document.

    DOMXPath totalItems = // count number of items

      new DOMXPath("count(purchaseorder/item)");

    DOMXPath atLeast10 =  // titles of items with quantity >= 10

      new DOMXPath("purchaseorder/item[quantity >= 10]/title");

    // evaluate them against the document

    double count = totalItems.evaluateAsNumber(doc);

    Set nodeset = atLeast10.evaluateAsNodeset(doc);

    // output results

    System.out.println(file+" contains "+count+" total items.");
    System.out.println("The following items have quantity >= 10:");

    if (nodeset != null) {

      Iterator i = nodeset.iterator();

      while(i.hasNext()) {

        Node node = (Node)i.next();

        System.out.println("  "+node.getNodeName()+

                           ": "+node.getFirstChild().getNodeValue());

      }

    }

    // note that at this point we are free to continue using evaluate

    // atLeast10 and totalItems against other documents

  }

}
```

Main Steps When Using the DOMXPath Class

The following procedure describes the main steps to use the `DOMXPath` class to perform XPath matching against an XML document represented as a `DOM`:

1. Create an `org.w3c.dom.Document` object from an XML document, as shown in the following code excerpt:

   ```
   String file = "purchaseorder.xml";
   DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
   DocumentBuilder builder = factory.newDocumentBuilder();
   Document doc = builder.parse(file);
   ```

2. Create a `DOMXPath` object to represent the XPath expression you want to evaluate against the DOM.

   The following example shows an XPath expression that counts the items in a purchase order:

   ```
    DOMXPath totalItems =
         new DOMXPath("count(purchaseorder/item)");
   ```

   The following example shows an XPath expression that returns the titles of items whose quantity is greater or equal to 10:

```
DOMXPath atLeast10 =
     new DOMXPath("purchaseorder/item[quantity >= 10]/title");
```

3. Evaluate the XPath expression using one of the `DOMXPath.evaluateAs`*`XXX`*`()` methods, where *`XXX`* refers to the data type of the returned data, such as `Boolean`, `Nodeset`, `Number`, or `String`.

   The following example shows how to use the `evaluateAsNumber()` method to evaluate the `totalItems` XPath expression:

   ```
   double count = totalItems.evaluateAsNumber(doc);
   System.out.println(file+" contains "+count+" total items.");
   ```

   The following example shows how to use the `evaluateAsNodeset()` method to return a `Set` of `org.w3c.dom.Nodes` which you can iterate through in the standard way:

   ```
   Set nodeset = atLeast10.evaluateAsNodeset(doc);

   System.out.println("The following items have quantity >= 10:");
     if (nodeset != null) {
        Iterator i = nodeset.iterator();
        while(i.hasNext()) {
          Node node = (Node)i.next();
          System.out.println("  "+node.getNodeName()+
                                  ": "+node.getFirstChild().getNodeValue());
        }
     }
   ```

For additional API reference information about the WebLogic XPath API, see the weblogic.xml.xpath Javadoc at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

Using the StreamXPath Class

The example in this section shows how to use the `StreamXPath` class of the WebLogic XPath API to perform XPath matching against an `XMLInputStream`. The section first provides an example and then a description of the main steps used in the example. Although the example shows how to match only against an `XMLInputStream`, you can use similar code to match against an `XMLOutputStream`.

Example of Using the StreamXPath Class

The sample Java program at the end of this section uses the following XML document in its matching:

```
<?xml version='1.0' encoding='us-ascii'?>

<!-- "Stock Quotes". -->
```

```
<stock_quotes>
  <stock_quote symbol='BEAS'>
    <when>
      <date>01/27/2001</date>
      <time>3:40PM</time>
    </when>
    <price type="ask"     value="65.1875"/>
    <price type="open"    value="64.00"/>
    <price type="dayhigh" value="66.6875"/>
    <price type="daylow"  value="64.75"/>
    <change>+2.1875</change>
    <volume>7050200</volume>
  </stock_quote>
  <stock_quote symbol='MSFT'>
    <when>
      <date>01/27/2001</date>
      <time>3:40PM</time>
    </when>
    <price type="ask" value="55.6875"/>
    <price type="open"    value="50.25"/>
    <price type="dayhigh" value="56"/>
    <price type="daylow"  value="52.9375"/>
    <change>+5.25</change>
    <volume>64282200</volume>
  </stock_quote>
</stock_quotes>
```

The Java code for the example is as follows:

```
package examples.xml.xpath;

import java.io.File;
import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLInputStreamFactory;
import weblogic.xml.stream.XMLStreamException;
import weblogic.xml.xpath.StreamXPath;
import weblogic.xml.xpath.XPathException;
import weblogic.xml.xpath.XPathStreamFactory;
import weblogic.xml.xpath.XPathStreamObserver;
```

```
/**
 * This class provides a simple example of how to use the StreamXPath
 * API.
 *
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

public abstract class StreamXPathExample {

  public static void main(String[] ignored)

    throws XPathException, XMLStreamException

  {

    // Create instances of StreamXPath for two xpaths we want to match

    // against this tream.

    StreamXPath symbols =

      new StreamXPath("stock_quotes/stock_quote");

    StreamXPath openingPrices =

      new StreamXPath("stock_quotes/stock_quote/price[@type='open']");

    // Create an XPathStreamFactory.

    XPathStreamFactory factory = new XPathStreamFactory();

    // Create and install two XPathStreamObservers.  In this case, we

    // just use to anonymous classes to print a message when a

    // callback is received.  Note that a given observer can observe

    // more than one xpath, and a given xpath can be observed by

    // mutliple observers.

    factory.install(symbols, new XPathStreamObserver () {

      public void observe(XMLEvent event) {

        System.out.println("Matched a quote: "+event);

      }

      public void observeAttribute(StartElement e, Attribute a) {} //ignore

      public void observeNamespace(StartElement e, Attribute a) {} //ignore

    });
    // Note that we get matches for both a start and an end elements,
    // even in the case of <price/> which is an empty element - this
    // is the behavior of the underlying streaming parser.
    factory.install(openingPrices, new XPathStreamObserver () {
      public void observe(XMLEvent event) {
        System.out.println("Matched an opening price: "+event);
      }
      public void observeAttribute(StartElement e, Attribute a) {} //ignore
      public void observeNamespace(StartElement e, Attribute a) {} //ignore
    });
    // get an XMLInputStream on the document
    String file = "stocks.xml";
    XMLInputStream sourceStream = XMLInputStreamFactory.newInstance().
      newInputStream(new File(file));
    // use the factory to create an XMLInputStream that will do xpath
    // matching against the source stream
    XMLInputStream matchingStream = factory.createStream(sourceStream);
    // now iterate through the stream
    System.out.println("Matching against xml stream from "+file);
    while(matchingStream.hasNext()) {
      // we don't do anything with the events in our example - the
      // XPathStreamObserver instances that we installed in the
      // factory will get callbacks for appropriate events
      XMLEvent event = matchingStream.next();
    }
  }
}
```

Main Steps When Using the StreamXPath Class

The following procedure describes the main steps to use the `StreamXPath` class to perform XPath matching against an XML document represented as an `XMLInputStream`:

1. Create a `StreamXPath` object to represent the XPath expression you want to evaluate against the `XMLInputStream`.

   The following example shows an XPath expression that searches for stock quotes in an XML document:

   ```
   StreamXPath symbols =
      new StreamXPath("stock_quotes/stock_quote");
   ```

   The following example shows an XPath expression that matches stock quotes using their opening price:

   ```
   StreamXPath openingPrices = new
   StreamXPath("stock_quotes/stock_quote/price[@type='open']");
   ```

2. Create an `XPathStreamFactory`. Use this factory class to specify the set of `StreamXPath` objects that you want to evaluate against an `XMLInputStream` and to create observers (using the `XPathStreamObserver` interface) used to register a callback whenever an XPath match occurs. The following example shows how to create the `XPathStreamFactory`:

   ```
   XPathStreamFactory factory = new XPathStreamFactory();
   ```

3. Create and install the observers using the `XPathStreamFactory.install()` method, specifying the XPath expression with the first `StreamXPath` parameter, and an observer with the second `XPathStreamObserver` parameter. The following example shows how to use an anonymous class to implement the `XPathStreamObserver` interface. The implementation of the `observe()` method simply prints a message when a callback is received. In the example, the `observeAttribute()` and `observeNamespace()` methods do nothing.

   ```
   factory.install(symbols, new XPathStreamObserver () {
     public void observe(XMLEvent event) {
       System.out.println("Matched a quote: "+event);
     }
     public void observeAttribute(StartElement e, Attribute a) {}
     public void observeNamespace(StartElement e, Attribute a) {}
    }
   );
   ```

4. Create an `XMLInputStream` from an XML document:

   ```
   String file = "stocks.xml";
   ```

```
XMLInputStream sourceStream =
    XMLInputStreamFactory.newInstance().newInputStream(new File(file));
```

5. Use the `createStream()` method of the `XPathStreamFactory` to create a new `XMLInputStream` that will perform XPath matching against the original `XMLInputStream`:

   ```
    XMLInputStream matchingStream =
       factory.createStream(sourceStream);
   ```

6. Iterate over the new XMLInputStream. During the iteration, if an XPath match occurs, the registered observer is notified:

   ```
   while(matchingStream.hasNext()) {
     XMLEvent event = matchingStream.next();
   }
   ```

For additional API reference information about the WebLogic XPath API, see the weblogic.xml.xpath Javadoc at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

Using the WebLogic XML Digital Signature API

The WebLogic XML Digital Signature API contains classes to digitally sign and validate SOAP messages. The API is contained in the `weblogic.webservice.core.soap` package.

The following table lists the two main classes of the XML Digital Signature API.

Table 5-1 Classes of the WebLogic XML Didital Signature API

Class	Description
XMLSignature	Main class used to sign and validate a SOAP message represented as a `java.io.InputStream`. Use the `sign()` method to digitally sign the message using the client's digital certificate and private key. Use the `validate()` method to ensure that the SOAP message is valid.
ValidateResult	Class that represents the result of validating a signed SOAP message. Use the `getCertificates()` method to return a list of digital certificates found in the signed XML document.

When using the API, a thrown `XMLSignatureInvalidException` means that the signature is invalid and a thrown `XMLSignatureMalformedException` means that the SOAP message is unprocessable.

Warning: The WebLogic XML Digital Signature API works only with RSA encryption.

The following sample client application shows a simple way to use the WebLogic XML Digital Signature API. The Java code in bold shows how to use the `XMLSignature.sign()` method to read an unsigned SOAP message, represented as a `java.io.InputStream`, and then write the resulting signed SOAP message, represented as a `java.io.OutputStream` , using the client's private key and digital certificates. The program also shows how to validate the signed SOAP message and output the digital certificates found in the message.

```
import weblogic.webservice.core.soap.XMLSignature;
import weblogic.webservice.core.soap.ValidateResult;

import java.security.cert.X509Certificate;
import java.security.PrivateKey;
import java.security.Key;
import java.security.KeyStore;
import java.io.FileInputStream;
import java.io.FileOutputStream;

/**
* Sample client application that shows how to sign and validate a SOAP message.
*
* Copyright (c) 2003 by BEA Systems. All Rights Reserved.
*/

public class SignSOAPMessage {

  private static final String CLIENT_KEYSTORE = "clientkeystore";
  private static final String KEYSTORE_PASS = "mypasswd";
  private static final String KEYNAME = "clientkey";
  private static final String KEYPASS = "clientpass";

  private static final String signedXML = "soap-signed.xml";

  private static Key getPrivateKey(String keyname,
                                   String password,
                                   String keystore) throws Exception {
    KeyStore ks = KeyStore.getInstance("JKS");

    ks.load(new FileInputStream(keystore), KEYSTORE_PASS.toCharArray());

    Key result = ks.getKey(keyname, password.toCharArray());

    return result;

  }

  private static X509Certificate getCertificate(String keyname,
                                                String keystore) throws Exception {
    KeyStore ks = KeyStore.getInstance("JKS");

    ks.load(new FileInputStream(keystore), KEYSTORE_PASS.toCharArray());
    X509Certificate result = (X509Certificate) ks.getCertificate(keyname);

    return result;
  }

  public static void main(String[] args) throws Exception {
    X509Certificate clientcert = getCertificate(KEYNAME, CLIENT_KEYSTORE);
    PrivateKey clientprivate = (PrivateKey) getPrivateKey(KEYNAME,
        KEYPASS,  CLIENT_KEYSTORE);
    ValidateResult result;
    if (args[1].equals("sign")) {
      XMLSignature.sign(new FileInputStream(args[0]), new
FileOutputStream(signedXML), clientcert, clientprivate);
      result = XMLSignature.validate(new FileInputStream(signedXML));
    } else {
      result = XMLSignature.validate(new FileInputStream(args[1]));
    }

    System.out.println("Cert: " + result.getCertificates());

  }
}
```

CHAPTER 6

XML Programming Best Practices

The following sections discuss best programming practices when creating Java applications that process XML data:

When to Use the DOM, SAX, and Streaming APIs

You can parse an XML document with the DOM, SAX, or Streaming API. This section describes the pros and cons of each API.

The DOM API is good for small documents, or those that contain under a thousand elements. Because DOM constructs a tree of your XML data, it is ideal for editing the structure of your XML document by adding or deleting elements.

The DOM API parses the *entire* XML document and converts it into a DOM tree before you can begin processing it. This cost might be beneficial if you know that you need to access the entire document. If you occasionally need to access only part of the XML document, the cost could decrease the performance of your application with no added benefit. In this case the SAX or streaming API is preferable.

The SAX API is the most lightweight of the APIs. It is ideal for parsing shallow documents (documents that do not contain much nesting of elements) with unique element names. SAX uses a callback structure; this means that programmers handle parsing events as the API is reading an XML document, which is a relatively efficient and quick way to parse.

However, the callback nature of SAX also means that it is not the best API to use if you want to modify the structure of your XML data. Additionally, programming your application to handle callbacks is not always straight-forward and intuitive.

The streaming API is based on SAX, so all the reasons for using SAX also apply to the streaming API. In addition, the streaming API is more intuitive to use than SAX, because programmers ask for events rather than react to them as they happen. The streaming API is also best if you plan to pass the entire XML document as a parameter; it is easier to pass an input stream than it is to pass SAX events. Finally, the streaming API was designed to be used when binding XML data to Java objects.

Increasing Performance of XML Validation

If the performance of your XML application decreases due to a parser validation issue, and you need to validate your XML documents, you might improve the performance of your application by writing your own customized code that validates the data as it is being received or parsed, rather than using the `setValidating()` method of the `DocumentBuilderFactory` or `SaxParserFactory`.

When you turn on validation while parsing an XML document with SAX or DOM, the parser might do more validation of the document than you really need, thus decreasing the overall performance of the application. Instead, consider choosing certain points during the parsing of the document when you want to check that the XML document is valid, and add your own Java code at those points.

For example, assume you are writing an application that uses the WebLogic XML Streaming API to processes an XML purchase order. Because you know that the first element of the document should be `<purchase_order>`, you can quickly verify that the document *appears* to be valid by pulling the first element off the stream and checking its name. This check does not ensure that the entire XML document is valid, of course, but you can continue checking for known elements as you pull elements from the stream. These quick checks are much faster than using the standard `setValidating()` methods.

When to Use XML Schemas or DTDs

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

The current trend is to use Schemas to describe XML documents. Schemas are much more expressive than DTDs because the set of available data types to describe XML elements is much richer and you can describe more specifically what is valid in an XML document. In addition, you can only use Schemas, and not DTDs, in SOAP messages. Because SOAP is the main messaging protocol used in Web services, consider using Schemas to describe any XML documents that might be used as either input or output parameters to Web services.

Still, DTDs have a few advantages. DTDs are more widely supported than Schemas, although that is changing rapidly. Because DTDs are less expressive than Schemas, they are easier to write and manage.

However, BEA Systems recommends that you use Schemas to describe your XML documents.

Configuring External Entity Resolution for Maximum Performance

BEA Systems highly recommends you store external entities locally whenever possible rather than always retrieving the entity over the network. Storage improves the performance of your applications because it is much faster to look up an entity on the same machine as WebLogic Server than it is to look it up over a network connection.

For detailed information on configuring external entity resolution for WebLogic Server, see "External Entity Configuration Tasks" on page 8-5.

Using SAX InputSources

When you use the SAX API to parse an XML document, you first create an `InputSource` object from the XML document and then pass the `InputSource` object to the `parse()` method. You can create the `InputSource` object from either a `java.io.InputStream` or `java.io.Reader` object based on your XML data.

BEA recommends that you create an `InputSource` from a `java.io.InputStream` object whenever possible. When passed an `InputStream` object, the SAX parser auto-detects the character encoding of the XML data and automatically instantiates an `InputStreamReader` object for you, using the correct character encoding. In other words, the parser does all the character encoding work for you, which is more likely to be error-free at runtime than if you decide to specify the character encoding yourself.

Improving Performance of Transformations

XSLT is a language for transforming an XML document into a different format, such as another XML document, HTML, WML, and so on. To use XSLT, you create a stylesheet that defines how each element in the input XML document should be transformed in the output document.

Although XSLT is a powerful language, creating stylesheets for complex transformations can be very complicated. In addition, the actual transformation requires a lot of resources and might decrease the performance of your application.

Therefore, if your transformations are complex, consider writing your own transformation code in your application rather than using XSLT stylesheets. Also consider using the DOM API. First parse the XML document, manipulate the resulting DOM tree as needed, then write out the new document, using custom Java code to transform it into its final format.

CHAPTER 7

XML Programming Techniques

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

- "Transmitting XML Data Between A Java Client and WebLogic Server" on page 7-1
- "Handling XML Documents in a JMS Application" on page 7-3
- "Accessing External Entities That Do Not Have an HTTP Interface" on page 7-4
- "Retrieving XML Document Header Information" on page 7-4

Transmitting XML Data Between A Java Client and WebLogic Server

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 XML data from a Java client to a WebLogic Server-hosted servlet or JSP which then returns the data to the client, 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 transmit XML data between the program and 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 shows how to open a URL connection to the JSP by 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 show 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/docs81/servlet/index.html and *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs81/jsp/index.html

Handling XML Documents in a JMS Application

WebLogic Server provides the following extensions to some Java Message Service (JMS) classes to handle XML documents in a 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/docs81/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 a URL, that returns the entity. See "External Entity Configuration Tasks" on page 8-5 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 an interface. 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.

Retrieving XML Document Header Information

Suppose you want only XML document header information, such as the root element, system ID, or public ID, instead of all the actual data within the document. Fully parsing the document is unnecessary and 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 by using the `weblogic.xml.sax.XMLInputSource` class, which is Weblogic Server's extension to the `org.xml.sax.InputSource` class.

Warning: The `weblogic.xml.sax.XMLInputSource` class is deprecated as of Version 8.1 of WebLogic Server. Instead, you should use the WebLogic XML Streaming API to get the root element, public and system ids, and namespace URI of an XML document. For more information, see Chapter 4, “Using the WebLogic XML Streaming API.”

The following example segment shows how to use the `XMLInputSource` 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.

Administering WebLogic Server XML

The following sections describe XML administration with WebLogic Server:

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 the 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. Using the Administration Console XML Registry has several benefits:

- 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 type.
- 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.

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 make the following types of entries for a given XML registry:

- Configure parsers and transformers
- 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 8.1, the built-in XML parser is one based on Apache Xerces (package name `weblogic.apache.`xerces.*) and the built-in transformer is the Apache Xalan included in the JDK 1.4.1 (package name `org.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.

Warning: In version 8.1 of WebLogic Server, you can plug in only the following versions of the Apache Xerces parser:

- Xerces 2.2.0
- Xerces 2.3.0
- Xerces 2.4.0

In addition, you can plug in only those versions of the Apache Xalan transformer that are compatible with the preceding versions of the Apache Xerces parser.

1. Start the WebLogic Administration Server and invoke the Administration Console in your browser. See "Overview of Administering WebLogic Server XML" on page 8-1 for information on invoking the Administration Console.
2. Follow the steps outlined in *Configuring a Parser or Transformer Other Than the Built-In* of the Administration Console Online Help.

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.

Warning: WebLogic Server searches only the first 1000 bytes of an XML document when attempting to identify its document type. If it does not find a DOCTYPE identifier in those first 1000 bytes, it stops searching the document and uses the parser configured for the WebLogic Server instance to parse the document.

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.

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 8-1 for information on invoking the Administration Console.

2. Follow the steps outlined in *Configuring a Parser for a Particular Document Type* of the Administration Console Online Help.

External Entity Configuration Tasks

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 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 a Managed 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 for 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.

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 8-1 for information on invoking the Administration Console.

2. Follow the steps outlined in *Configuring External Entity Resolution* of the Administration Console Online Help.

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 8-1 for information on invoking the Administration Console.

2. Follow the steps outlined in *Configuring the External Entity Cache* in the Administration Online Help.

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 monitor 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 8-1 for information on invoking the Administration Console.

2. Follow the steps outlined in *Monitoring the External Entity Cache* in the Administration Console Online Help.

XML Reference

The following sections provide links to additional information about the XML specifications, application programming interfaces (APIs), and tools supported by WebLogic Server:

XML APIs

- SAX 2.0 API at http://www.saxproject.org/
- DOM (Document Object Model) Level 2 Specification at http://www.w3.org/TR/DOM-Level-2/
- JAXP API 1.1 specification at http://java.sun.com/xml/
- Apache Xerces Java Parser at http://xml.apache.org/xerces-j/index.html
- Apache Xalan XSLT transformer at http://xml.apache.org/xalan-j/index.html

Code Examples

XML code examples and supporting documentation are included in the WebLogic Server distribution at `WL_HOME\samples\server\examples\src\examples\xml`, where `WL_HOME` refers to the top-level WebLogic Platform directory.

Related WebLogic Server Documentation

- *Programming WebLogic Web Services* at http://e-docs.bea.com/wls/docs81/webServices/index.html
- *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81/ejb/index.html
- *Programming WebLogic JMS* at http://e-docs.bea.com/wls/docs81/jms/index.html
- *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs81/jsp/index.html
- *Programming WebLogic HTTP Servlets* at http://e-docs.bea.com/wls/docs81/servlet/index.html
- *Programming WebLogic Server for Wireless Services* at http://e-docs.bea.com/wls/docs81/wireless/index.html

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 and Information

- XML 1.0 specification at http://www.w3.org/TR/REC-xml/
- 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/
- W3C XML Namespaces 1.0 Recommendation at http://www.w3.org/TR/REC-xml-names/
- 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
- 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/

Index