# Oracle® Fusion Middleware

# Developing XML Applications for Oracle WebLogic Server



G56960-01
November 2026

ORACLE

Oracle Fusion Middleware Developing XML Applications for Oracle WebLogic Server,

G56960-01

Copyright © 2007, 2026, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

# Contents


## Preface

Audience i
Documentation Accessibility i
Diversity and Inclusion i
Related Documentation ii
Conventions ii

## 1 Introduction

Samples for the XML Developer 1
- XML Examples in the WebLogic Server Distribution 1

Summary of WebLogic Server XML Features 1
- XML Document Parsers 1
- XML Document Transformer 1
- Streaming API for XML (StAX) Implementation 2
- WebLogic XPath API 2
- JAXP Pluggability Layer Implementation 2
- WebLogic Servlet Attributes 2
- XML Registry For Configuring Parsers and Transformers 2
- XML Registry for Configuring External Entity Resolution 2

Selecting XML Parser and Transformer Providers 3
Learning More About XML 3

## 2 XML Overview

What Is XML? 1
How Do You Describe an XML Document? 2
Why Use XML? 3
What Are XSL and XSLT? 3
What Are DOM and SAX? 4
- SAX 4
- DOM 4

What Is the Streaming API for XML (StAX)? 4
What Is JAXP? 5




Copyright © 2007, 2026, Oracle and/or its affiliates.

JAXP Packages 5
Configuring XML Schema Validation with JAXP 6
Common Uses of XML and XSLT 6
Using XML and XSLT to Separate Content from Presentation 7
XML as a Message Format for Business-to-Business Communication 7

## 3 Developing XML Applications with WebLogic Server

Developing XML Applications: Main Steps 1
Parsing XML Documents 1
Parsing XML Documents Using JAXP in SAX Mode 2
Parsing XML Documents Using JAXP in DOM Mode 2
Parsing XML Documents in a Servlet 3
Using the org.xml.sax.DefaultHandler Attribute to Parse a Document 4
Using the org.w3c.dom.Document Attribute to Parse a Document 4
Validating and Non-Validating Parsers 5
Handling Entity Resolution While Parsing an XML Document 6
General Information About External Entities 6
Using the WebLogic Server Entity Resolution Features 7
Using Parsers Other Than the Default 7
Generating New XML Documents 7
Generating XML from a DOM Document Tree 8
Generating XML Documents in a JSP 8
Transforming XML Documents 9
Using JAXP to Transform XML Data 9
Example of Transforming an XML Document Using JAXP 9
Using the JSP Tag to Transform XML Data 10
XSLT JSP Tag Syntax 10
XSLT JSP Tag Usage 12
Example of Using the XSLT JSP Tag in a JSP 13
Using Transformers Other Than the Default Transformer 14

## 4 Using the Streaming API for XML (StAX)

Overview of the Streaming API for XML 1
Description of the Cursor API 1
The XMLStreamReader Interface 1
The XMLStreamWriter Interface 2
Description of the Event Iterator API 2
Main Interfaces and Classes of StAX 3
Parsing XML With the XMLStreamReader Interface: Typical Steps 4
Example of Parsing XML Using StAX 6


Copyright © 2007, 2026, Oracle and/or its affiliates.

Getting the XMLStreamReader Object 9
Determining the Specific XML Event Type 10
Getting the Full Name of an Element 11
Getting the Attributes of an Element 12
Getting the Namespaces of an Element 13
Getting Text Data 14
Getting Location Information 14
Closing the Input Stream 15
Generating XML Using the XMLStreamWriter Interface: Typical Steps 15
Example of Generating XML Using StAX 16
Getting the XMLStreamWriter Object 18
Adding the XML Declaration to the Output Stream 18
Adding Standard XML Events to the Output Stream 19
Adding Attributes and Namespace Declarations to a Start Element 19
Closing the Output Stream 20
Properties Defined for the XMLInputFactory Interface 20
Properties Defined for the XMLOutputFactory Interface 21

5 Using Advanced XML APIs

Using the WebLogic XPath API 1
Using the DOMXPath Class 1
Example of Using the DOMXPath Class 1
Main Steps When Using the DOMXPath Class 4
Using the StreamXPath Class 5
Example of Using the StreamXPath Class 5
Main Steps When Using the StreamXPath Class 7

6 XML Programming Best Practices

When to Use the DOM, SAX, and StAX APIs 1
Increasing Performance of XML Validation 1
When to Use XML Schemas or DTDs 1
Configuring External Entity Resolution for Maximum Performance 2
Using SAX InputSources 2
Improving Performance of Transformations 2

7 XML Programming Techniques

Transmitting XML Data Between a Java Client and WebLogic Server 1
Handling XML Documents in a JMS Application 2


Copyright © 2007, 2026, Oracle and/or its affiliates.

Accessing External Entities That Do Not Have an HTTP Interface 3

## 8 XML Application Scoping

Overview of Application Scoping 1
The weblogic-application.xml File 1
xml 2
parser-factory 3
saxparser-factory 3
document-builder-factory 3
transformer-factory 3
schema-factory 3
xpath-factory 3
xml-input-factory 3
xml-output-factory 3
xml-event-factory 4
entity-mapping 4
entity-mapping-name 4
public-id 4
system-id 4
entity-uri 4
when-to-cache 4
cache-timeout-interval 4
Configuring a Parser or Transformer for an Enterprise Application 5
Configuring an External Entity for an Enterprise Application 5
Configuring the External Entity Cache for an Enterprise Application 6

## 9 Administering WebLogic Server XML

Overview of Administering WebLogic Server XML 1
XML Administration Tasks 1
How the XML Registry Works 1
Parser Selection Within the XML Registry 2
XML Parser and Transformer Configuration Tasks 3
Configuring a Parser or Transformer Other Than the Default 3
Configuring a Parser for a Particular Document Type 3
External Entity Configuration Tasks 3
Configuring External Entity Resolution 3
Configuring the External Entity Cache 4


Copyright © 2007, 2026, Oracle and/or its affiliates.

## A XML Reference


XML APIs A-1
Code Examples A-1
Related WebLogic Server Documentation A-1
Tutorials and Online Courses A-1
Other XML Specifications and Information A-1


## B Using the WebLogic XML Streaming API (Deprecated)


Overview of the WebLogic XML Streaming API B-1
Javadocs for the WebLogic XML Streaming API B-2
Parsing an XML Document: Typical Steps B-2
- Example of Parsing an XML Document B-3
- Getting an XML Input Stream B-5
- Getting a Buffered XML Input Stream B-5
- Filtering the XML Stream B-6
- Creating a Custom Filter B-6
- Iterating Over the Stream B-7
- Determining the Specific XMLEvent Type B-7
- Getting the Attributes of an Element B-10
- Positioning the Stream B-11
- Getting a Substream B-11
- Marking and Resetting a Buffered XML Input Stream B-12
- Closing the Input Stream B-13

Generating a New XML Document: Typical Steps B-13
- Example of Generating an XML Document B-14
- Creating an XML Output Stream B-16
- Adding Elements to the Output Stream B-16
- Adding Attributes to an Element on the Output Stream B-17
- Adding an Input Stream to an Output Stream B-18
- Printing an Output Stream B-18
- Closing the Output Stream B-19




Copyright © 2007, 2026, Oracle and/or its affiliates.

# Preface

This document is a resource for software developers who design and develop applications that include XML processing.

## Audience

The topics in this document are relevant during the design and development phases of a software project. The document also includes topics that are useful in solving application problems that are discovered during test and pre-production phases of a project.

Although this document does include administration and monitoring information useful to developers who want to test their applications in a development environment, the document does not address production-phase administration, monitoring, or performance tuning of XML applications. For links to WebLogic Server® documentation and resources for these topics, see Related Documentation.

It is assumed that the reader is familiar with Web technologies, XML, XSLT, the Java programming language, and the Servlet and JSP APIs of the Jakarta EE specification. This document emphasizes the value-added features provided by WebLogic Server XML and key information about how to use WebLogic Server features and facilities to get an application that performs XML processing up and running.

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

**Access to Oracle Support**

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

## Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# Related Documentation

This document contains XML-specific design and development information.

For comprehensive guidelines for developing, deploying, and monitoring WebLogic Server applications, see the following documents:

- *Developing Applications for Oracle WebLogic Server* is a guide to developing WebLogic Server components (such as Web applications and EJBs) and applications.
- *Developing JAX-WS Web Services for Oracle WebLogic Server* is a guide to developing Web Services that are deployed and run on WebLogic Server.
- *Deploying Applications to Oracle WebLogic Server* is the primary source of information about deploying WebLogic Server applications.

For related information about XML outside the scope of this document, see links listed in Learning More About XML and XML Reference.

### New and Changed WebLogic Server Features

For a comprehensive listing of the new WebLogic Server features introduced in this release, see *What's New in Oracle WebLogic Server*.

# Conventions

The following text conventions are used in this document:

| Convention | Meaning |
|---|---|
| **boldface** | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary. |
| *italic* | Italic type indicates book titles, emphasis, or placeholder variables for text entered by users. |
| `monospace` | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that user enter. |


Copyright © 2007, 2026, Oracle and/or its affiliates.

1

# Introduction

This chapter summarizes the XML features of WebLogic Server, describes the WebLogic Server XML examples, and provides a variety of links to additional XML information.

## Samples for the XML Developer

In addition to this document, Oracle provides a variety of code samples for XML developers. The examples and tutorials illustrate WebLogic Server XML in action, and provide practical instructions on how to perform key XML development tasks.

Oracle recommends that you run some or all of the XML examples before programming your own application that processes XML.

### XML Examples in the WebLogic Server Distribution

WebLogic Server optionally installs API code examples in the *ORACLE_HOME*\wlserver\samples\server directory, where *ORACLE_HOME* represents the directory in which you installed WebLogic Server. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

## Summary of 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 following topics, which describe the features included in the WebLogic Server XML subsystem, are presented in this section:

### XML Document Parsers

WebLogic Server uses, by default, the XML parser that is included in the JDK.

You can also use any other XML parser of your choice by using the WebLogic Remote 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.

For information about parsing XML documents, see Parsing XML Documents.

### XML Document Transformer

WebLogic Server uses, by default, the XML transformer that is included in the JDK.

You can also use any other XML transformer of your choice by using the WebLogic Remote Console to configure it in the XML Registry. You can configure a single instance of WebLogic


Copyright © 2007, 2026, Oracle and/or its affiliates.

Server to use one transformer for a particular application and use another transformer for a different application.

For more information about transforming XML documents, see Transforming XML Documents.

## Streaming API for XML (StAX) Implementation

WebLogic Server includes an implementation of the Streaming API for XML (StAX).

See Using the Streaming API for XML (StAX).

## 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`.

See Using the WebLogic XPath API.

## JAXP Pluggability Layer Implementation

WebLogic Server supports the JAXP APIs provided by the supported JDK. Standard JAXP factory methods select implementations through the JAXP lookup mechanism. For WebLogic-managed XML processing, the XML Registry can supply parser and transformer factories at server or application scope, and can select a parser for a server-registry entry identified by public ID, system ID, or root element.

See Administering WebLogic Server XML, including Configuring a Parser or Transformer Other Than the Default, for the server-wide WebLogic Remote Console workflow; see XML Application Scoping for application configuration.

## 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.

See Parsing XML Documents in a Servlet.

## 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 WebLogic Remote Console to configure the parsers and transformers for an instance of WebLogic Server.

See XML Parser and Transformer Configuration Tasks.

## XML Registry for Configuring External Entity Resolution

WebLogic XML supports external entity resolution through the XML Registry. See External Entity Configuration Tasks.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# Selecting XML Parser and Transformer Providers

The endorsed-standards override mechanism was removed in JDK 9. To select parser or transformer providers for WebLogic-managed XML processing, use the WebLogic Server XML Registry. For standalone JAXP code, use the provider-selection mechanisms documented by `SAXParserFactory.newInstance`, `DocumentBuilderFactory.newInstance`, `TransformerFactory.newInstance`.

# Learning More About XML

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

- A Technical Introduction to XML, available at `https://www.xml.com/pub/a/98/10/guide0.html`
- XML, Java, and the Future of the Web, available at `https://www.xml.com/pub/a/w3j/s3.bosak.html`
- XML 1.0, available at `https://www.w3.org/TR/xml/`
- XML Schema Part 0: Primer, available at `https://www.w3.org/TR/xmlschema-0/`
- SAX API (Java SE 21), available at `https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/org/xml/sax/package-summary.html`
- DOM API (Java SE 21), available at `https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/org/w3c/dom/package-summary.html`
- JAXP API (Java SE 21), available at `https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/module-summary.html`


Copyright © 2007, 2026, Oracle and/or its affiliates.

2

# XML Overview

This chapter describes Extensible Markup Language (XML) technology and the WebLogic Server XML subsystem.
This chapter includes the following sections:

## 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


Copyright © 2007, 2026, Oracle and/or its affiliates.

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: XML Schemas and DTDs.

XML Schemas define the basic requirements for the structure of a particular XML document. A Schema describes the elements and attributes that are valid in an XML document, and the contexts in which they are valid. In other words, a Schema specifies which tags are allowed within certain other tags, and which tags and attributes are optional. Schemas are themselves XML files.

The schema specification is a product of the World Wide Web Consortium (W3C). 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="address_book">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="person" type="personType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="personType">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="addressType" minOccurs="0"/>
      <xs:element name="phone" type="phoneType" minOccurs="0"/>
      <xs:element name="email" type="xs:string" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="gender" type="xs:string" use="required"/>
  </xs:complexType>
  <xs:complexType name="addressType">
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="phoneType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="area_code" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

You can also describe XML documents using Document Type Definition (DTD) files, a technology older than XML Schemas. DTDs are not XML files.

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


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<!DOCTYPE address_book [
<!ELEMENT address_book (person*)>
<!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>
]>
```

An XML document can include a Schema or DTD as part of the document itself, reference an external Schema or DTD, or not include or reference a Schema or DTD 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 need a schema or DTD when they must be validated against those constraints; DTDs can also declare entities and default attribute values. An XML document is considered *valid* if 1) it has an associated Schema or DTD, and 2) it complies with the constraints expressed in the associated Schema or DTD. If, however, an XML document only needs to be *well-formed*, then the document does not have to be accompanied by a Schema or DTD. 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


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 default 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 default parser (the parser included in the JDK) 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 a 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 default parser (the parser included in the JDK) 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 the Streaming API for XML (StAX)?

StAX is a streaming, pull-parsing API for reading and writing XML. It is one of the XML-processing APIs included in JAXP and in the Java SE `java.xml` module. StAX originated as a Java Community Process specification.

StAX gives parsing control to the programmer by exposing a simple iterator-based API and an underlying stream of events; the API includes methods such as `next()` and `hasNext()` that


Copyright © 2007, 2026, Oracle and/or its affiliates.

allow the programmer to ask for the next event rather than handle the event in a callback. This gives the programmer more procedural control over the processing of the XML document.

> **Note**
>
> Previous versions of WebLogic Server included a similar proprietary API called *WebLogic XML Streaming API*. This API was a basis for StAX. Although the WebLogic XML Streaming API is still accessible in this release of WebLogic Server, its functionality has been deprecated as of release 9.0 of WebLogic Server. Programmers should use StAX instead.

For detailed information about using StAX, see Using the Streaming API for XML (StAX).

# 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.

JAXP is part of the Java SE `java.xml` module. Its APIs retain the `javax.xml` package names because they are Java SE APIs. JAXP provides standard APIs for DOM and SAX parsing, StAX streaming, schema validation, XSLT transformation, XPath evaluation, and XML catalogs.

## JAXP Packages

Examples of JAXP packages include:

- `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.

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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

For detailed information on using JAXP to transform XML objects, see Using JAXP to Transform XML Data.

## Configuring XML Schema Validation with JAXP

The default JAXP provider is required to support W3C XML Schema 1.0. XML Schema 1.1 requires a provider that explicitly documents support for it; configure and name that provider before claiming XML Schema 1.1 support.

```
import java.nio.file.Path;
import javax.xml.XMLConstants;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;

SchemaFactory schemaFactory =
    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
schemaFactory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
schemaFactory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
schemaFactory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "file");
Schema schema = schemaFactory.newSchema(
    Path.of("schemas", "Report.xsd").toFile());

SAXParserFactory parserFactory = SAXParserFactory.newInstance();
parserFactory.setNamespaceAware(true);
parserFactory.setSchema(schema);
parserFactory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
SAXParser parser = parserFactory.newSAXParser();
parser.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
parser.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
DefaultHandler validationHandler = new DefaultHandler() {
    @Override
    public void error(SAXParseException exception) throws SAXException {
        throw exception;
    }

    @Override
    public void fatalError(SAXParseException exception) throws SAXException {
        throw exception;
    }
};
parser.parse(Path.of("xml", "report.xml").toFile(), validationHandler);
```

The handler throws on validation errors; the default no-op `DefaultHandler.error` must not be used when the application needs parse failure to enforce validity.

The `file` protocol value on `SchemaFactory` permits every local file path accessible to the process; it is not a directory allowlist. Use it only when trusted schema includes/imports are required, combine it with fixed application-controlled paths and operating-system access controls, and use an empty value when the schema is self-contained. The parser properties separately block the instance document from loading external DTDs or schemas.

# Common Uses of XML and XSLT

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


Copyright © 2007, 2026, Oracle and/or its affiliates.

# 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 client types easily, if necessary.

For additional information about XSLT, see Other XML Specifications and Information .

# 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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

3

# Developing XML Applications with WebLogic Server

This chapter describes how to use the Java programming language and the WebLogic Server XML subsystem to develop XML applications. It is assumed that you know how to use Jakarta Servlets and Jakarta Server Pages (JSPs) to write Java applications.
This chapter includes the following sections.

For information about how to write servlet and JSP applications, see *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

## 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.

   See Parsing XML Documents.

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.

   See Generating New XML Documents.

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.

   See Using JAXP to Transform XML Data.

## 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 Streaming API for XML (StAX) to parse XML documents, see Using the Streaming API for XML (StAX).


Copyright © 2007, 2026, Oracle and/or its affiliates.

You use the WebLogic Remote Console XML Registry to configure the following:

- Per-document-type parsers, which supersede the default parser for the specified document type.
- 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 information on how to use the WebLogic Remote Console for these tasks, see Administering WebLogic Server XML.

These topics are in the following sections:

# 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 java.nio.file.Path;
import javax.xml.XMLConstants;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

MyHandler handler = new MyHandler();
// MyHandler extends org.xml.sax.helpers.DefaultHandler.
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setNamespaceAware(true);
spf.setValidating(true);
spf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
SAXParser sp = spf.newSAXParser();
sp.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "file");
sp.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
sp.parse(Path.of("xml", "file.xml").toFile(), handler);
```

`MyHandler` must throw `SAXParseException` from its `error` and `fatalError` overrides when a validity error must stop processing; the inherited `DefaultHandler` methods are no-ops.

The `file` value permits every local file path accessible to the process; it is not a directory allowlist. Use it only for a trusted DTD at a fixed application-controlled path, together with operating-system access controls. Use an empty value when no external DTD is required, and do not parse untrusted network locations directly.

> **Note**
>
> If you want to use a parser other than the default parser, you must use the WebLogic Remote Console to specify the parser in the XML Registry; otherwise the `SAXParserFactory.newInstance` method returns the default parser.

# 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 java.nio.file.Path;
import javax.xml.XMLConstants;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
dbf.setValidating(true);
dbf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
dbf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "file");
dbf.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
DocumentBuilder db = dbf.newDocumentBuilder();
db.setErrorHandler(new DefaultHandler() {
    @Override
    public void error(SAXParseException exception) throws SAXException {
        throw exception;
    }

    @Override
    public void fatalError(SAXParseException exception) throws SAXException {
        throw exception;
    }
});
Document doc = db.parse(Path.of("xml", "file.xml").toFile());
```

The `file` value permits every local file path accessible to the process; it is not a directory allowlist. Use it only for a trusted DTD at a fixed application-controlled path, together with operating-system access controls. Use an empty value when no external DTD is required, and do not parse untrusted network locations directly.

> **Note**
>
> If you want to use a parser other than the default parser, you must use the WebLogic Remote Console to specify it; otherwise the `DocumentBuilderFactory.newInstance` method returns the default parser.

# Parsing XML Documents in a Servlet

Support for the `setAttribute` and `getAttribute` methods was added to version 2.2 of the Jakarta 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, as described in Using the org.xml.sax.DefaultHandler Attribute to Parse a Document and Using the org.w3c.dom.Document Attribute to Parse a Document.

WebLogic Server provides the `weblogic.servlet.XMLParsingHelper` filter class, but the filter is not installed or mapped automatically for a Web application. The following example enables request-attribute parsing by declaring and mapping the filter in the application's `WEB-INF/web.xml` deployment descriptor:


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
<filter>
  <filter-name>XMLParsingHelper</filter-name>
  <filter-class>weblogic.servlet.XMLParsingHelper</filter-class>
</filter>

<filter-mapping>
  <filter-name>XMLParsingHelper</filter-name>
  <url-pattern>/*</url-pattern>
  <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

For more information on servlet filters, see Filters in *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

## 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,
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
                              HttpServletResponse response)
throws ServletException, IOException  {

try {
   Document doc = (Document) 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.

`SAXParserFactory.setValidating(true)` requests DTD validation. To validate against W3C XML Schema, create a `Schema` with `SchemaFactory` and pass it to `SAXParserFactory.setSchema`; also enable namespace awareness. A parser can still enforce well-formedness without validation.

```
import java.nio.file.Path;
import javax.xml.XMLConstants;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.helpers.DefaultHandler;

SchemaFactory schemaFactory =
    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
schemaFactory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
schemaFactory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
schemaFactory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "file");
Schema schema = schemaFactory.newSchema(
    Path.of("schemas", "address.xsd").toFile());
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
factory.setSchema(schema);
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
SAXParser parser = factory.newSAXParser();
parser.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
parser.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
DefaultHandler validationHandler = new DefaultHandler() {
    @Override
    public void error(SAXParseException exception) throws SAXException {
        throw exception;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    }

    @Override
    public void fatalError(SAXParseException exception) throws SAXException {
        throw exception;
    }
};
parser.parse(Path.of("xml", "address.xml").toFile(), validationHandler);
```

The handler throws on validity errors; a bare `DefaultHandler` does not. The `file` value on `SchemaFactory` permits every local file path accessible to the process and is not a directory allowlist. Use it only for trusted includes/imports at fixed application-controlled paths with operating-system access controls; use an empty value for a self-contained schema. The parser properties separately block instance-document external access.

# 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. When external-entity resolution is enabled and permitted by the JAXP and WebLogic XML Registry settings, a parser can retrieve the referenced resource and substitute its content. For XML from untrusted sources, keep external access disabled. When an application requires a trusted DTD or entity, allow only the required protocol and preferably resolve it to a controlled local resource through the XML Registry or an XML catalog. 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 Jakarta EE 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">
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

## 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.

## Using Parsers Other Than the Default

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

- Accept the default parser 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 XML document type, based on its system or public identifier, or its root element.

# 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 Streaming API for XML (StAX) to generate XML documents, see Using the Streaming API for XML (StAX).


Copyright © 2007, 2026, Oracle and/or its affiliates.

# Generating XML from a DOM Document Tree

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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

# 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 1.2 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`

  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`

  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`

  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`

  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 java.nio.file.Path;
import javax.xml.XMLConstants;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

TransformerFactory factory = TransformerFactory.newInstance();
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");

Path stylesheet = Path.of("stylesheets", "mystylesheet.xsl");
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
Path xmlDocument = Path.of("xml_docs", "myXMLdoc.xml");
Transformer transformer = factory.newTransformer(
    new StreamSource(stylesheet.toFile()));
transformer.transform(new StreamSource(xmlDocument.toFile()),
                      new StreamResult(System.out));
```

The empty external-access values require a self-contained stylesheet. If trusted local `xsl:include` or `xsl:import` resources are required, set `ACCESS_EXTERNAL_STYLESHEET` to `file` and keep those resources under application control.

For an example of how to transform a DOM document into an XML stream, see Using JAXP to Transform XML Data.

## 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.

> **Note**
>
> This feature is deprecated in WebLogic Server 12.1.2.0 and may be removed in a future release.

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`.

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

## 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.

> **Note**
>
> This feature is deprecated in WebLogic Server 12.1.2.0 and may be removed in a future release.

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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

Square brackets indicate optional attributes and are notation, not literal JSP characters. The `uri` attribute and an inline stylesheet body are alternative forms; use only one form on each `x:stylesheet` element.

```
<x:xslt [xml="..."]
        [media="..."]
        [stylesheet="..."]>
  <x:xml>...</x:xml>
  <x:stylesheet [media="..."] uri="..."/>
  <!-- Or supply the stylesheet inline: -->
  <x:stylesheet [media="..."]>
    In-line stylesheet goes here
  </x:stylesheet>
</x:xslt>
```

Table 3-1 describes the attributes of the `x:xslt` tag.

**Table 3-1 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.<br>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.<br>**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.<br>**Note**: It is an error to set both the `media` and `stylesheet` attributes within the same `x:xslt` tag. |

Table 3-2 describes the attributes of the `x:stylesheet` tag.


Copyright © 2007, 2026, Oracle and/or its affiliates.

**Table 3-2 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.<br>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.

> **Note**
>
> This feature is deprecated in WebLogic Server 12.1.2.0 and may be removed in a future release.

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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

- If the `stylesheet` attribute is defined, the XML document is processed using the specified stylesheet.

> **Note**
>
> 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.

## 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.

> **Note**
>
> This feature is deprecated in WebLogic Server 12.1.2.0 and may be removed in a future release.

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");
String myMedia = "wml";
if (clientType != null && clientType.contains("Mozilla")) {
    myMedia = "html";
}
%>

<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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

# Using Transformers Other Than the Default Transformer

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

- Accept the default transformer as the server-wide transformer.
- Configure a transformer other than the default 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 Configuring a Parser or Transformer Other Than the Default.


Copyright © 2007, 2026, Oracle and/or its affiliates.

4

# Using the Streaming API for XML (StAX)

This chapter describes how to use the Streaming API for XML (StAX) to parse and generate XML documents.
The chapter includes the following sections:

## Overview of the Streaming API for XML

The Streaming API for XML (StAX), specified by JSR-173, is distributed as a standalone technology as a part of the Java SE.

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 StAX, 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 StAX is often referred to as *pull parsing*.

StAX includes two APIs, the *cursor API* and the *event-iterator API*, either of which can be used for reading and writing XML. The following sections describe each API and their particular strengths.

### Description of the Cursor API

The basic function of the cursor API is to allow programmers to parse and generate XML as easily and efficiently as possible. Of the two APIs in StAX, this is the one that most programmers would use.

The cursor API iterates over a set of events, such as start elements, comments, and attributes, although the events may be unrealized. The cursor API has two main interfaces: `XMLStreamReader` for parsing XML and `XMLStreamWriter` for generating XML.

### The XMLStreamReader Interface

The cursor API uses the `XMLStreamReader` interface to move a virtual cursor over an XML document and allow access to the data and underlying state through method calls such as `hasNext()`, `next()`, `getEventType()`, and `getText()`. The `XMLStreamReader` interface allows only forward, read-only access to the XML.

Use the `XMLInputFactory` class to create a new instance of the `XMLStreamReader`. You can set a variety of properties when you get a new reader; for details, see Properties Defined for the XMLInputFactory Interface.

When you use the `next()` method of the `XMLStreamReader` interface to parse XML, the reader gets the next parsing event and returns an integer that identifies the type of event just read. Parsing events correspond to sections of an XML document, such as the XML declaration, start and end element tags, character data, white space, comments, and processing instructions. The `XMLStreamConstants` interface specifies the event to which the integer


Copyright © 2007, 2026, Oracle and/or its affiliates.

returned by the `next()` method corresponds. You can also use the `getEventType()` method of `XMLStreamReader` to determine the event type.

The `XMLStreamReader` interface has numerous methods for getting at the specific data in the XML document. Some of these methods include:

- `getLocalName()`—Returns the local name of the current event.
- `getPrefix()`—Returns the prefix of the current event.
- `getAttributeXXX()`—Set of methods that return information about the current attribute event.
- `getNamespaceXXX()`—Set of methods that return information about the current namespace event.
- `getTextXXX()`—Set of methods that return information about the current text event.
- `getPIData()`—Returns the data section of the current processing instruction event.

Only certain methods are valid for each event type; the StAX processor throws a `java.lang.IllegalStateException` if you try to call a method on an invalid event type. For example, it is an error to try to call the `getAttributeXXX()` methods on a namespace event. See the StAX specification, at `http://www.jcp.org/en/jsr/detail?id=173`, for the complete list of events and their valid `XMLStreamReader` methods.

## The XMLStreamWriter Interface

The cursor API uses the `XMLStreamWriter` interface to specify how to generate XML.

Use the `XMLOutputFactory` class to create a new instance of the `XMLStreamWriter`. You can set a property for repairing namespaces and prefixes when you get a new writer; for details, see Properties Defined for the XMLOutputFactory Interface.

The `XMLStreamWriter` interface defines a set of `writeXXX()` methods for writing standard parts of an XML document, such as:

- `writeStartElement()`
- `writeEndDocument()`
- `writeAttribute()`
- `writeNamespace()`
- `writeCData()`

Each part of an XML document, including the attributes and the namespaces, must be explicitly written using these methods.

Use the `flush()` method to write any cached data to the output and the `close()` method to close the writer and free up any resources.

The `XMLStreamWriter`, when generating XML, does not check that the generated document is well-formed; it is the programmer's responsibility to create a well-formed XML document. To print the special characters `&`, `<`, and `>`, use the `writeCharacters()` method.

## Description of the Event Iterator API

The event iterator API is a layer on top of the cursor API. It is easy to extend and facilitates pipelining. Pipelining refers to multiple XML-to-XML transformations. By using the event iterator API, programmers do not have to deserialize and serialize the XML at each stage of the


Copyright © 2007, 2026, Oracle and/or its affiliates.

pipeline; rather, only at each end of the pipeline and use the API methods such as `nextEvent()` to communicate at the middle stages. The event iterator API has two main interfaces: `XMLEventReader` for parsing XML and `XMLEventWriter` for generating XML.

Because the cursor API is the most commonly used API in StAX, this section does not describe in detail how to use the event iterator API, other than showing an example. For details about using this API, see the StAX specification http://www.jcp.org/en/jsr/detail?id=173.

The following example shows a simple program that uses the `XMLEventReader` interface of StAX to parse an XML document. The program takes a single parameter, an XML file, and uses it to create an `XMLEventReader` object. The program then uses the reader to iterate over and print the stream of events.

```
package examples.event;

import java.io.FileReader;
import javax.xml.stream.*;
import javax.xml.stream.events.*;
import javax.xml.stream.util.*;
import javax.xml.namespace.QName;

/**
 * A simple example to iterate over events
 *
 * @author Copyright (c) 2002 by BEA Systems.
 */

public class Parse {
  private static String filename = null;
  private static void printUsage() {
    System.out.println("usage: java examples.event.Parse <xmlfile>");
  }

  public static void main(String[] args) throws Exception {
    try {
      filename = args[0];
    } catch (ArrayIndexOutOfBoundsException aioobe){
      printUsage();
      System.exit(0);
    }

    XMLInputFactory factory = XMLInputFactory.newInstance();
    XMLEventReader r =
      factory.createXMLEventReader(new FileReader(filename));
    while(r.hasNext()) {
      XMLEvent e = r.nextEvent();
      System.out.println("ID:"+e.hashCode()+"["+e+"]");
    }
  }
}
```

# Main Interfaces and Classes of StAX

Table 4-1 describes the main interfaces and classes of the Streaming API for XML


Copyright © 2007, 2026, Oracle and/or its affiliates.

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

| Interface or Class | Used in Cursor or Event Iterator API? | Description |
|---|---|---|
| `XMLInputFactory class` | Both | Factory class used to create an `XMLStreamReader` or `XMLEventReader` instance. |
| `XMLOutputFactory class` | Both | Factory class used to create an `XMLStreamWriter` or `XMLEventWriter` instance. |
| `XMLEventFactory class` | Event Iterator | Factory class used to create an `XMLEvent` instance. |
| `XMLStreamReader interface` | Cursor | Enables an application to advance to the next event, inspect the current event, and determine whether more events remain. |
| `XMLStreamWriter interface` | Cursor | Interface used to generate an XML document. It provides a set of `writeXXX()` methods for generating specific parts of an XML document, such as start elements, attributes, and so on. |
| `XMLEventReader interface` | Event Iterator | Interface used to parse XML events. It enables you to peek at the next event, get the next event, and check for more events. |
| `XMLEventWriter interface` | Event Iterator | Interface used to generate XML. It uses the `add()` method to add XMLEvents to the output stream. |
| `XMLEvent` | Event Iterator | Base interface for handling events. All specific XML events extend from `XMLEvent`, such as `StartElement`, `Attribute`, and so on. |
| `XMLStreamException exception` | Both | Base exception for unexpected processing errors, such as lack of well-formed XML document structure. |

# Parsing XML With the XMLStreamReader Interface: Typical Steps

The following procedure describes the typical steps for using the `XMLStreamReader` interface of the StAX cursor API to parse an XML document. The procedure uses the example from Example of Parsing XML Using StAX in its description.

Call XMLStreamReader.close() to release parser resources, and separately close the underlying InputStream or Reader; XMLStreamReader.close() does not close the source.

If the application intentionally requires a DTD or external entity from a controlled source, keep the feature enabled only with a resolver/catalog that allowlists trusted identifiers and denies every other external fetch.

1. Import the `javax.xml.stream.*` classes.
2. Use the `XMLInputFactory.newInstance()` method to instantiate an `XMLInputFactory`, as shown in the following code excerpt:

```
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import javax.xml.XMLConstants;

XMLInputFactory xmlif = XMLInputFactory.newFactory();
xmlif.setProperty(XMLInputFactory.SUPPORT_DTD, false);
xmlif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
xmlif.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
try (InputStream in = Files.newInputStream(Path.of(filename))) {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    XMLStreamReader xmlr = xmlif.createXMLStreamReader(in);
    try {
        while (true) {
    printEvent(xmlr);
    if (!xmlr.hasNext()) {
        break;
    }
    xmlr.next();
}
    } finally {
        xmlr.close();
    }
}
```

See Properties Defined for the XMLInputFactory Interface for the list of properties you can set.

3. Use the `XMLInputFactory.createXMLStreamReader()` method to instantiate an `XMLStreamReader` object based on an XML document.

   See Getting the XMLStreamReader Object.

4. Parse the XML document, using the `hasNext()` and `next()` methods to step through the XML events, as shown in the following code excerpt:

   ```
   while (true) {
       printEvent(xmlr);
       if (!xmlr.hasNext()) {
           break;
       }
       xmlr.next();
   }
   ```

   In the example, `xmlr` is the `XMLStreamReader` instance and the local `printEvent()` method (not part of the StAX API) determines the specific event type, as described in the next step.

5. While parsing the XML document, determine the current specific event type and take appropriate action. Event types include the start and end of an XML document, the start and end of an XML element, comments, entity references, and so on.

   See Determining the Specific XML Event Type.

6. If the event is START_ELEMENT, use getAttributeCount() and the indexed attribute accessors to inspect its attributes.

   See Getting the Attributes of an Element.

7. If the current event type is a start or end element, optionally get its namespaces.

   See Getting the Namespaces of an Element.

8. If the current event type includes text data, such as a CDATA or comment, optionally get the actual data.

   See Getting Text Data.

9. Optionally, get location information, such as the line number or column number, of the current event.

   See Getting Location Information.

10. Close the parser and its underlying input separately.

    See Closing the Input Stream.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# Example of Parsing XML Using StAX

The following example shows a simple program that uses the `XMLStreamReader` interface of StAX to parse an XML document.

The program takes a single parameter, an XML file, and uses it to create an XMLStreamReader object. The program then uses the reader to iterate over the stream of events, determining the type of each event, such as the start of an XML element, the list of attributes of an element, a processing instruction, and so on. The program prints out information about these events, using internal methods to print out the list of attributes and namespaces when appropriate.

Call XMLStreamReader.close() to release parser resources, and separately close the underlying InputStream or Reader; XMLStreamReader.close() does not close the source.

If the application intentionally requires a DTD or external entity from a controlled source, keep the feature enabled only with a resolver/catalog that allowlists trusted identifiers and denies every other external fetch.

```
package examples.basic;

import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import javax.xml.XMLConstants;
import java.util.Iterator;
import javax.xml.stream.*;
import javax.xml.namespace.QName;

/**
 * This is a simple parsing example that illustrates
 * the XMLStreamReader class.
 *
 * @author Copyright (c) 2003 by BEA Systems.
 */

public class Parse {
  private static String filename = null;
  private static void printUsage() {
    System.out.println("usage: java examples.basic.Parse <xmlfile>");
  }

  public static void main(String[] args) throws Exception {
    try {
      filename = args[0];
    } catch (ArrayIndexOutOfBoundsException aioobe){
      printUsage();
      System.exit(0);
    }

    XMLInputFactory xmlif = XMLInputFactory.newFactory();
    xmlif.setProperty(XMLInputFactory.SUPPORT_DTD, false);
    xmlif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
    xmlif.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
    try (InputStream in = Files.newInputStream(Path.of(filename))) {
        XMLStreamReader xmlr = xmlif.createXMLStreamReader(in);
        try {
            while (true) {
        printEvent(xmlr);
        if (!xmlr.hasNext()) {
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
            break;
        }
        xmlr.next();
    }
        } finally {
            xmlr.close();
        }
    }
}

private static void printEvent(XMLStreamReader xmlr) {

  System.out.print("EVENT:["+xmlr.getLocation().getLineNumber()+"]["+
                   xmlr.getLocation().getColumnNumber()+"] ");

  System.out.print(" [");
  switch (xmlr.getEventType()) {

  case XMLStreamConstants.START_ELEMENT:
    System.out.print("<");
    printName(xmlr);
    printNamespaces(xmlr);
    printAttributes(xmlr);
    System.out.print(">");
    break;
  case XMLStreamConstants.END_ELEMENT:
    System.out.print("</");
    printName(xmlr);
    System.out.print(">");
    break;
  case XMLStreamConstants.SPACE:
  case XMLStreamConstants.CHARACTERS:
    int start = xmlr.getTextStart();
    int length = xmlr.getTextLength();
    System.out.print(new String(xmlr.getTextCharacters(),
                                start,
                                length));
    break;

  case XMLStreamConstants.PROCESSING_INSTRUCTION:
  System.out.print("<?" + xmlr.getPITarget());
  String data = xmlr.getPIData();
  if (data != null && !data.isEmpty()) {
      System.out.print(" " + data);
  }
  System.out.print("?>");
  break;

  case XMLStreamConstants.CDATA:
    System.out.print("<![CDATA[");
    start = xmlr.getTextStart();
    length = xmlr.getTextLength();
    System.out.print(new String(xmlr.getTextCharacters(),
                                start,
                                length));
    System.out.print("]]>");
    break;

  case XMLStreamConstants.COMMENT:
    System.out.print("<!--");
    if (xmlr.hasText())
      System.out.print(xmlr.getText());
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    System.out.print("-->");
    break;

  case XMLStreamConstants.ENTITY_REFERENCE:
    System.out.print(xmlr.getLocalName()+"=");
    if (xmlr.hasText())
      System.out.print("["+xmlr.getText()+"]");
    break;

  case XMLStreamConstants.START_DOCUMENT:
  String version = xmlr.getVersion();
  if (version != null) {
      System.out.print("<?xml version='" + version + "'");
      String declaredEncoding = xmlr.getCharacterEncodingScheme();
      if (declaredEncoding != null) {
          System.out.print(" encoding='" + declaredEncoding + "'");
      }
      if (xmlr.standaloneSet()) {
          System.out.print(" standalone='"
              + (xmlr.isStandalone() ? "yes" : "no") + "'");
      }
      System.out.print("?>");
  }
  break;

  }
  System.out.println("]");
}

private static void printName(XMLStreamReader xmlr){
  if(xmlr.hasName()){
    String prefix = xmlr.getPrefix();
    String uri = xmlr.getNamespaceURI();
    String localName = xmlr.getLocalName();
    printName(prefix,uri,localName);
  }
}

private static void printName(String prefix,
                              String uri,
                              String localName) {
  if (uri != null && !("".equals(uri)) ) System.out.print("['"+uri+"']:");
  if (prefix != null) System.out.print(prefix+":");
  if (localName != null) System.out.print(localName);
}

private static void printAttributes(XMLStreamReader xmlr){
  for (int i=0; i < xmlr.getAttributeCount(); i++) {
    printAttribute(xmlr,i);
  }
}

private static void printAttribute(XMLStreamReader xmlr, int index) {
  String prefix = xmlr.getAttributePrefix(index);
  String namespace = xmlr.getAttributeNamespace(index);
  String localName = xmlr.getAttributeLocalName(index);
  String value = xmlr.getAttributeValue(index);
  System.out.print(" ");
  printName(prefix,namespace,localName);
  System.out.print("='"+value+"'");
}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  private static void printNamespaces(XMLStreamReader xmlr){
    for (int i=0; i < xmlr.getNamespaceCount(); i++) {
      printNamespace(xmlr,i);
    }
  }

  private static void printNamespace(XMLStreamReader xmlr, int index) {
    String prefix = xmlr.getNamespacePrefix(index);
    String uri = xmlr.getNamespaceURI(index);
    System.out.print(" ");
    if (prefix == null)
      System.out.print("xmlns='"+uri+"'");
    else
      System.out.print("xmlns:"+prefix+"='"+uri+"'");
  }
}
```

# Getting the XMLStreamReader Object

Use the `XMLInputFactory.createXMLStreamReader()` method to instantiate an `XMLStreamReader` object based on an XML document, as shown in the following code excerpt:

```
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import javax.xml.XMLConstants;

XMLInputFactory xmlif = XMLInputFactory.newFactory();
xmlif.setProperty(XMLInputFactory.SUPPORT_DTD, false);
xmlif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
xmlif.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
try (InputStream in = Files.newInputStream(Path.of(filename))) {
    XMLStreamReader xmlr = xmlif.createXMLStreamReader(in);
    try {
        while (true) {
    printEvent(xmlr);
    if (!xmlr.hasNext()) {
        break;
    }
    xmlr.next();
}
    } finally {
        xmlr.close();
    }
}
```

Call XMLStreamReader.close() to release parser resources, and separately close the underlying InputStream or Reader; XMLStreamReader.close() does not close the source.

If the application intentionally requires a DTD or external entity from a controlled source, keep the feature enabled only with a resolver/catalog that allowlists trusted identifiers and denies every other external fetch.

In the example, `xmlif` is the `XMLInputFactory` instance.

The various signatures of the `createXMLStreamReader()` method allow for the following XML document formats as parameters:

- `java.io.InputStream` (shown in the example)
- `java.io.Reader`


Copyright © 2007, 2026, Oracle and/or its affiliates.

- `javax.xml.transform.Source` (specified in the JAXP API https://docs.oracle.com/en/java/javase/17/docs/api/java.xml/javax/xml/transform/Source.html)

## Determining the Specific XML Event Type

To determine the specific event type while parsing an XML document, use either the `XMLStreamReader.next()` or `XMLStreamReader.getEventType()` methods. The `next()` method reads the next event and returns an integer which identifies the read event type; the `getEventType()` method simply returns the integer identifying the current event type. The `XMLStreamConstants` superinterface of `XMLStreamReader` defines the event type constants, shown in the following list:

- `XMLStreamConstants.ATTRIBUTE`
- `XMLStreamConstants.CDATA`
- `XMLStreamConstants.CHARACTERS`
- `XMLStreamConstants.COMMENT`
- `XMLStreamConstants.DTD`
- `XMLStreamConstants.END_DOCUMENT`
- `XMLStreamConstants.END_ELEMENT`
- `XMLStreamConstants.ENTITY_DECLARATION`
- `XMLStreamConstants.ENTITY_REFERENCE`
- `XMLStreamConstants.NAMESPACE`
- `XMLStreamConstants.NOTATION_DECLARATION`
- `XMLStreamConstants.PROCESSING_INSTRUCTION`
- `XMLStreamConstants.SPACE`
- `XMLStreamConstants.START_DOCUMENT`
- `XMLStreamConstants.START_ELEMENT`

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

```
switch (xmlr.getEventType()) {

case XMLStreamConstants.START_ELEMENT:
  System.out.print("Start Element\n");
  break;

case XMLStreamConstants.END_ELEMENT:
  System.out.print("End Element\n");
  break;

case XMLStreamConstants.SPACE:
  System.out.print("Space\n");
  break;

case XMLStreamConstants.CHARACTERS:
  System.out.print("Characters\n");
  break;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
case XMLStreamConstants.PROCESSING_INSTRUCTION:
  System.out.print("Processing Instrcutions\n");
  break;

case XMLStreamConstants.CDATA:
  System.out.print("CDATA\n");
  break;

case XMLStreamConstants.COMMENT:
  System.out.print("Comment\n");
  break;

case XMLStreamConstants.DTD:
  System.out.print("DTD\n");
  break;

case XMLStreamConstants.ENTITY_REFERENCE:
  System.out.print("Entity Reference\n");
  break;

case XMLStreamConstants.ENTITY_DECLARATION:
  System.out.print("Entity Declaration\n");
  break;

case XMLStreamConstants.START_DOCUMENT:
  System.out.print("Start Document\n");
  break;

case XMLStreamConstants.END_DOCUMENT:
  System.out.print("End Document\n");
  break;

}
```

## Getting the Full Name of an Element

The full name of an element includes its prefix, namespace URI, and local name; use the `getPrefix()`, `getNamespaceURI()`, and `getLocalName()` methods of the `XMLStreamReader` interface, respectively, to get this information once you determine that the current event is a start or end element.

For example, assume the `case` statement for a start element event in the sample program looks like the following:

```
case XMLStreamConstants.START_ELEMENT:

  System.out.print("<");
  printName(xmlr);
  printNamespaces(xmlr);
  printAttributes(xmlr);
  System.out.print(">");
  break;
```

> **Note**
>
> The `printNamespaces()` and `printAttributes()` methods are discussed in other sections.


Copyright © 2007, 2026, Oracle and/or its affiliates.

The two local `printName()` methods can use the `getXXX()` methods as follows:

```
private static void printName(XMLStreamReader xmlr){
  if(xmlr.hasName()){
    String prefix = xmlr.getPrefix();
    String uri = xmlr.getNamespaceURI();
    String localName = xmlr.getLocalName();
    printName(prefix,uri,localName);
  }
}

private static void printName(String prefix,
                              String uri,
                              String localName) {
  if (uri != null && !("".equals(uri)) ) System.out.print("['"+uri+"']:");
  if (prefix != null) System.out.print(prefix+":");
  if (localName != null) System.out.print(localName);
}
```

## Getting the Attributes of an Element

Call the attribute accessors when the current event is START_ELEMENT (or ATTRIBUTE when a provider exposes that event type). Attribute values are not available from END_ELEMENT.

Use the `getAttributeCount()` method to return the number of attributes of the current element and use the count in a loop that iterates over the list of attributes. The method does not include namespaces in the count. Additional `getAttributeXXX()` methods return the prefix, namespace URI, local name, and value for a particular attribute.

For example, assume the `case` statement for a start element event in our sample program looks like the following:

```
case XMLStreamConstants.START_ELEMENT:

  System.out.print("<");
  printName(xmlr);
  printNamespaces(xmlr);
  printAttributes(xmlr);
  System.out.print(">");
  break;
```

> **Note**
>
> The `printName()` and `printNamespaces()` methods are discussed in other sections.

The following local `printAttributes()` method shows one way of iterating through the list of attributes; because attribute indices are zero-based, the `for` loop starts at 0:

```
private static void printAttributes(XMLStreamReader xmlr){

  for (int i=0; i < xmlr.getAttributeCount(); i++) {
    printAttribute(xmlr,i);
  }
}
```

The following local `printAttribute()` method shows how to print out all the information for a particular attribute:


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
private static void printAttribute(XMLStreamReader xmlr, int index) {
  String prefix = xmlr.getAttributePrefix(index);
  String namespace = xmlr.getAttributeNamespace(index);
  String localName = xmlr.getAttributeLocalName(index);
  String value = xmlr.getAttributeValue(index);
  System.out.print(" ");
  printName(prefix,namespace,localName);
  System.out.print("='"+value+"'");
}
```

The `printName()` method is described in Getting the Full Name of an Element.

## Getting the Namespaces of an Element

Once you determine that the current event is a start element, end element, or namespace, use the `getNamespaceXXX()` methods of the `XMLStreamReader` interface to get the list of namespaces declared for the event.

> **Note**
>
> You can use the `getNamespaceXXX()` methods *only* on start element, end element, and namespace events; a `java.lang.IllegalStateException` is thrown if you try to execute the methods on any other type of event.

Use the `getNamespaceCount()` method to return the number of namespaces declared for the current event, and use the count in a loop that iterates over the list. If the current event is an end element, the count refers to the number of namespaces that are about to go out of scope. Additional `getNamespaceXXX()` methods return the prefix and namespace URI for a particular namespace.

For example, assume the `case` statement for a start element event in our sample program looks like the following:

```
case XMLStreamConstants.START_ELEMENT:

  System.out.print("<");
  printName(xmlr);
  printNamespaces(xmlr);
  printAttributes(xmlr);
  System.out.print(">");
  break;
```

> **Note**
>
> The `printName()` and `printAttributes()` methods are discussed in other sections.

The following local `printNamespaces()` method shows one way of iterating through the list of namespaces for the start element; because namespace indices are zero-based, the `for` loop starts at 0:

```
private static void printNamespaces(XMLStreamReader xmlr){
  for (int i=0; i < xmlr.getNamespaceCount(); i++) {
    printNamespace(xmlr,i);
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
  }
}
```

The following local `printNamespace()` method shows how to print out all the information for a particular namespace:

```
private static void printNamespace(XMLStreamReader xmlr, int index) {
  String prefix = xmlr.getNamespacePrefix(index);
  String uri = xmlr.getNamespaceURI(index);
  System.out.print(" ");
  if (prefix == null)
    System.out.print("xmlns='"+uri+"'");
  else
    System.out.print("xmlns:"+prefix+"='"+uri+"'");
}
```

The `getNamespacePrefix()` method returns null for the default namespace declaration.

## Getting Text Data

The `XMLStreamReader` interface includes various `getTextXXX()` methods for getting text data from events such as comments and CDATA.

Use the `getTextStart()` method to get the offset into the text character array where the first character of the current text event is stored. Use the `getTextLength()` method to get the length of the sequence of characters within the text character array. Finally, use the `getTextCharacters()` method to return this character array for the current event. The character array contains text information about only the current event; as soon as you call the `next()` method to read the next event on the input stream, the character array is filled with new information.

The following example shows how to print out text data for the CDATA event:

```
case XMLStreamConstants.CDATA:
  System.out.print("<![CDATA[");
  start = xmlr.getTextStart();
  length = xmlr.getTextLength();
  System.out.print(new String(xmlr.getTextCharacters(),
                              start,
                              length));
  System.out.print("]]>");
  break;
```

If you want to first check that the character event actually has text, use the `hasText()` method, as shown in the following example:

```
case XMLStreamConstants.COMMENT:
  System.out.print("<!--");
  if (xmlr.hasText())
    System.out.print(xmlr.getText());
  System.out.print("-->");
  break;
```

## Getting Location Information

The `Location` interface of the StAX API provides methods for getting location information about an event, such as the line number or column number, as well as the public ID and system ID of the XML being parsed. Use the `getLocation()` method of the `XMLStreamReader` interface to return a `Location` object for the current event, as shown in the following example:


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
System.out.print("EVENT:["+xmlr.getLocation().getLineNumber()+"]["+
                    xmlr.getLocation().getColumnNumber()+"] ");
```

## Closing the Input Stream

Call XMLStreamReader.close() to release parser resources, and separately close the underlying InputStream or Reader; XMLStreamReader.close() does not close the source.

If the application intentionally requires a DTD or external entity from a controlled source, keep the feature enabled only with a resolver/catalog that allowlists trusted identifiers and denies every other external fetch.

```
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import javax.xml.XMLConstants;

XMLInputFactory xmlif = XMLInputFactory.newFactory();
xmlif.setProperty(XMLInputFactory.SUPPORT_DTD, false);
xmlif.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES, false);
xmlif.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
try (InputStream in = Files.newInputStream(Path.of(filename))) {
    XMLStreamReader xmlr = xmlif.createXMLStreamReader(in);
    try {
        while (true) {
    printEvent(xmlr);
    if (!xmlr.hasNext()) {
        break;
    }
    xmlr.next();
}
    } finally {
        xmlr.close();
    }
}
```

# Generating XML Using the XMLStreamWriter Interface: Typical Steps

The following procedure describes the typical steps for using the `XMLStreamWriter` interface of the StAX cursor API to generate a new XML document.

1. Import the `javax.xml.stream.*` classes.
2. Use the `XMLOutputFactory.newInstance()` method to instantiate an `XMLOutputFactory`, as shown in the following code excerpt:

   ```
   XMLOutputFactory xmlof = XMLOutputFactory.newInstance();
   ```

   See Properties Defined for the XMLOutputFactory Interface for the list of properties you can set.
3. Use the `XMLOutputFactory.createXMLStreamWriter()` method to instantiate an `XMLStreamWriter` object, passing it the name of the file or object that will contain the XML.

   See Getting the XMLStreamWriter Object.
4. Add the XML declaration to the output. See Adding the XML Declaration to the Output Stream.


Copyright © 2007, 2026, Oracle and/or its affiliates.

5. Add standard XML objects, such as start elements, comments, and characters, to the output. See Adding Standard XML Events to the Output Stream.
6. Add attributes and namespace declarations to a start element. See Adding Attributes and Namespace Declarations to a Start Element.
7. After writing the end of the document, flush the XMLStreamWriter, close it to release resources, and then close the underlying OutputStream or Writer. Closing the StAX writer does not close that destination. See Closing the Output Stream.

# Example of Generating XML Using StAX

The following example shows a simple program that uses the `XMLStreamWriter` interface of StAX to generate an XML document.

The program first creates an instance of an `XMLStreamWriter`, specifying that the output be written to the file `outFile.xml` in the current directory. Then, using various `writeXXX()` methods, it builds an XML file that looks like the following:

```
<?xml version='1.0' encoding='utf-8'?>

<!--this is a comment-->
<person xmlns:one="http://namespaceOne" gender="f">
    <one:name hair="pigtails" freckles="yes">Pippi Longstocking</one:name>
</person>
```

The `XMLStreamWriter` interface does not check for that an XML document is well-formed; it is the programmer's responsibility to ensure that, for example, each start element has a corresponding end element, and so on. The example also shows how to use the `writeCharacters("\n")` method to add new lines to the output to make the XML more readable when writing to a text file.

After writing the end of the document, call XMLStreamWriter.flush() to write cached data, close the XMLStreamWriter to release its resources, and then close the underlying OutputStream or Writer. Closing the StAX writer does not close that destination.

```
package examples.basic;

import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Iterator;
import javax.xml.stream.*;
import javax.xml.namespace.QName;

/**
 * This is a simple example that illustrates how to use the
 * the XMLStreamWriter class to generate XML.
 *
 * The generated XML file looks like this:
 *
 *   <?xml version='1.0' encoding='utf-8'?>
 *
 *   <!--this is a comment-->
 *   <person xmlns:one="http://namespaceOne" gender="f">
 *       <one:name hair="pigtails" freckles="yes">Pippi Longstocking</one:name>
 *   </person>
 *
 *
 * @author Copyright (c) 2003 by BEA Systems.
 */
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
public class Generate {

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

    //
    // Get an output factory
    //
    XMLOutputFactory xmlof = XMLOutputFactory.newInstance();
    System.out.println("FACTORY: " + xmlof);

    //
    // Instantiate a writer
    //
    try (OutputStream out = Files.newOutputStream(Path.of("outFile.xml"))) {
        XMLStreamWriter xmlw = xmlof.createXMLStreamWriter(out, "UTF-8");
        try {
            // Write the default XML declaration
        xmlw.writeStartDocument();
        xmlw.writeCharacters("\n");
        xmlw.writeCharacters("\n");

        // Write a comment
        xmlw.writeComment("this is a comment");
        xmlw.writeCharacters("\n");

        // Write the root element "person" with a single attribute "gender"
        xmlw.writeStartElement("person");
        xmlw.writeNamespace("one", "http://namespaceOne");
        xmlw.writeAttribute("gender","f");
        xmlw.writeCharacters("\n");

        // Write the "name" element with some content and two attributes
        xmlw.writeCharacters("    ");
        xmlw.writeStartElement("one", "name", "http://namespaceOne");
        xmlw.writeAttribute("hair","pigtails");
        xmlw.writeAttribute("freckles","yes");
        xmlw.writeCharacters("Pippi Longstocking");

        // End the "name" element
        xmlw.writeEndElement();
        xmlw.writeCharacters("\n");

        // End the "person" element
        xmlw.writeEndElement();

        // End the XML document
        xmlw.writeEndDocument();
            xmlw.flush();
        } finally {
            xmlw.close();
        }
    }

  }

}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

## Getting the XMLStreamWriter Object

Use the `XMLOutputFactory.createXMLStreamWriter()` method to instantiate an `XMLStreamWriter` object based on an XML document, as shown in the following code excerpt:

```
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;

try (OutputStream out = Files.newOutputStream(Path.of(filename))) {
    XMLStreamWriter xmlw = xmlof.createXMLStreamWriter(out, "UTF-8");
    try {
        // Existing writeStartDocument/write.../writeEndDocument calls.
        xmlw.flush();
    } finally {
        xmlw.close();
    }
}
```

After writing the end of the document, call XMLStreamWriter.flush() to write cached data, close the XMLStreamWriter to release its resources, and then close the underlying OutputStream or Writer. Closing the StAX writer does not close that destination.

In the example, `xmlof` is the `XMLOutputFactory` instance.

The various signatures of the `createXMLStreamWriter()` method allow for the following XML document formats as parameters:

- `java.io.OutputStream` (shown in the example)
- `java.io.Writer`
- `javax.xml.transform.Result` (specified in the JAXP API https://docs.oracle.com/en/java/javase/17/docs/api/java.xml/javax/xml/transform/Result.html)

## Adding the XML Declaration to the Output Stream

Use the `XMLStreamWriter.writeStartDocument()` method to add the XML declaration as the first line of the XML document, as shown in the following code excerpt:

```
xmlw.writeStartDocument();
```

With no arguments, the method writes the default XML declaration:

```
<?xml version='1.0' encoding='utf-8'?>
```

If you want to specify a different encoding or XML version, use the following flavors of the `writeStartDocument()` method:

- `writeStartDocument(java.lang.String version)`
- `writeStartDocument(java.lang.String encoding, java.lang.String version)`

Setting the encoding with the `writeStartDocument()` method does not set the actual encoding of the underlying output; it simply specifies what value is written for the `encoding` attribute of the XML declaration. To actually set the encoding of the output, you must specify the `encoding` parameter when creating the instance of the `XMLStreamWriter` with the appropriate `XMLOutputFactory.createXMLStreamWriter()` method.


Copyright © 2007, 2026, Oracle and/or its affiliates.

## Adding Standard XML Events to the Output Stream

Use the `XMLStreamWriter.write`*`XXX`*`()` methods to add standard XML events, such as start elements, end elements, comments, CDATA, entity references, and so on to the output stream.The *`XXX`* refers to the particular event, such as `writeStartElement()`, `writeEndElement()`, `writeComment()`, `writeCData()`, and so on. You can create most elements by passing the name or text data as a `String`.

The `XMLStreamWriter` interface does not validate your data, nor does it check that the document is well-formed; it is the programmer's responsibility to ensure that, for example, each start element has a corresponding end element, and so on. It is also up to the programmer to ensure that the start and end element events are correctly nested. To make the output XML more human-readable when writing to a text file, use the `writeCharacters("\n")` method to add new lines in appropriate places.

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

```
<!-- This is a comment -->
<name>Jane Doe</name>
```

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

```
xmlw.writeComment("This is a comment");
xmlw.writeCharacters("\n");

xmlw.writeStartElement("name");
xmlw.writeCharacters("Jane Doe");
xmlw.writeEndElement();
xmlw.writeCharacters("\n");
```

## Adding Attributes and Namespace Declarations to a Start Element

Use the `writeAttribute()` method right after a start element event to add attributes to the element. You can specify a prefix for the attribute, as well as the URI it is bound to, or specify no prefix at all.

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

```
<person gender="f">
```

The Java code to produce this XML is as follows:

```
xmlw.writeStartElement("person");
xmlw.writeAttribute("gender","f");
xmlw.writeCharacters("\n");
```

Use the `writeNamespace()` method to write a namespace to the output stream. It is up to the programmer to ensure that the current event allows namespace writing, such as start element; if the current event does not allow namespace writing, a `javax.xml.stream.XMLStreamException` is thrown. Use appropriate flavors of other `writeXXX()` methods to specify a prefix for an event and the URI to which it is bound.

For example, the following XML output shows a namespace declaration for the `<person>` element, and the `one` prefix specified for the `<one>` child element:

```
<person xmlns:one="http://namespaceOne" gender="f">
    <one:name hair="pigtails" freckles="yes">Pippi Longstocking</one:name>
</person>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

The Java code to produce this XML is as follows:

```
// Write the root element "person" with a single attribute "gender"
xmlw.writeStartElement("person");
xmlw.writeNamespace("one", "http://namespaceOne");
xmlw.writeAttribute("gender","f");
xmlw.writeCharacters("\n");

// Write the "name" element with some content and two attributes
xmlw.writeCharacters("    ");
xmlw.writeStartElement("one", "name", "http://namespaceOne");
xmlw.writeAttribute("hair","pigtails");
xmlw.writeAttribute("freckles","yes");
xmlw.writeCharacters("Pippi Longstocking");

// End the "name" element
xmlw.writeEndElement();
xmlw.writeCharacters("\n");

// End the "person" element
xmlw.writeEndElement();
```

## Closing the Output Stream

After writing the end of the document, call XMLStreamWriter.flush() to write cached data, close the XMLStreamWriter to release its resources, and then close the underlying OutputStream or Writer. Closing the StAX writer does not close that destination.

```
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;

try (OutputStream out = Files.newOutputStream(Path.of(filename))) {
    XMLStreamWriter xmlw = xmlof.createXMLStreamWriter(out, "UTF-8");
    try {
        // Existing writeStartDocument/write.../writeEndDocument calls.
        xmlw.flush();
    } finally {
        xmlw.close();
    }
}
```

# Properties Defined for the XMLInputFactory Interface

Table 4-2 lists the standard properties you can set when using the `XMLInputFactory` to generate an `XMLStreamReader` or `XMLEventReader` object.

All properties in Table 4-2 are preceded with `javax.xml.stream`, such as `javax.xml.stream.isValidating`.

**Table 4-2 Standard XMLInputFactory Properties**

| Property | Description | Return Type | Default Value |
|---|---|---|---|
| `isValidating` | Specifies whether implementation-specific DTD validation is enabled or disabled. | Boolean | False |
| `isNamespaceAware` | Specifies whether namespace processing is enabled or disabled. Used for XML 1.0 support. | Boolean | True |


Copyright © 2007, 2026, Oracle and/or its affiliates.

**Table 4-2 (Cont.) Standard XMLInputFactory Properties**

| Property | Description | Return Type | Default Value |
|---|---|---|---|
| `isCoalescing` | Specifies whether to coalesce adjacent character data. | Boolean | False |
| `isReplacingEntityReferences` | Specifies whether internal entity references should be replaced with their replacement text and reported as characters. | Boolean | True |
| `isSupportingExternalEntities` | Specifies whether to resolve external parsed entities.<br>Applications that process untrusted input must set this property explicitly; do not infer its value from the provider default. | Boolean | Unspecified (provider dependent) |
| `supportDTD` | Specifies whether the processor used is one that supports or does not support DTDs. | Boolean | True |
| `reporter` | Specifies the implementation of `javax.xml.stream.XMLReporter` that should be used. Specifies the implementation of `javax.xml.stream.XMLReporter` that should be used. | XMLReporter | Null |
| `resolver` | Specifies the implementation of `javax.xml.stream.XMLResolver` that should be used. | XMLResolver | Null |
| `allocator` | Specifies the implementation of `javax.xml.stream.util.XMLEventAllocator` that should be used. | util.XMLEventAllocator | Null |

# Properties Defined for the XMLOutputFactory Interface

Table 4-3 lists the standard properties you can set when using the `XMLOutputFactory` to generate an `XMLStreamWriter` or `XMLEventWriter` object.

The standard XMLOutputFactory property is javax.xml.stream.isRepairingNamespaces. Its default is false. A provider may define additional properties; use isPropertySupported(String) before setting a provider-specific property.

**Table 4-3 Standard XMLOutputFactory Properties**

| Property | Description | Return Type | Default Value |
|---|---|---|---|
| `isRepairingNamespaces` | Specifies that the writer use default namespace prefix declarations.<br>There are strict rules about how the StAX processor repairs namespaces and prefixes when generating XML. For details, see the StAX specification at `http://www.jcp.org/en/jsr/detail?id=173`. | Boolean | False |


Copyright © 2007, 2026, Oracle and/or its affiliates.

5

# Using Advanced XML APIs

This chapter describes how to use the WebLogic XPath API.
This chapter includes the following section:

## 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 `weblogic.xml.xpath` in the *Java API Reference for Oracle WebLogic Server*.

### 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>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    </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.
 */

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);
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    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

  }

}
```

When adapting this controlled local-file example to untrusted XML, use the following alternative parser setup:

```
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import javax.xml.XMLConstants;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;

String file = "purchaseorder.xml";
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
factory.setXIncludeAware(false);
factory.setExpandEntityReferences(false);
Document doc;
try (InputStream in = Files.newInputStream(Path.of(file))) {
    doc = factory.newDocumentBuilder().parse(in);
}
// Continue with the existing DOMXPath declarations and evaluations against doc.
```

A controlled workflow requiring a trusted DTD or schema should use an allowlisting resolver or catalog instead of removing the restrictions wholesale.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# 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);
   ```

   When adapting this controlled local-file example to untrusted XML, use the following alternative parser setup:

   ```
   import java.io.InputStream;
   import java.nio.file.Files;
   import java.nio.file.Path;
   import javax.xml.XMLConstants;
   import javax.xml.parsers.DocumentBuilderFactory;
   import org.w3c.dom.Document;

   String file = "purchaseorder.xml";
   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
   factory.setNamespaceAware(true);
   factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
   factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
   factory.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
   factory.setXIncludeAware(false);
   factory.setExpandEntityReferences(false);
   Document doc;
   try (InputStream in = Files.newInputStream(Path.of(file))) {
       doc = factory.newDocumentBuilder().parse(in);
   }
   // Continue with the existing DOMXPath declarations and evaluations against doc.
   ```

   A controlled workflow requiring a trusted DTD or schema should use an allowlisting resolver or catalog instead of removing the restrictions wholesale.

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`.


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `weblogic.xml.xpath` in the *Java API Reference for Oracle WebLogic Server*.

# Using the StreamXPath Class

> **Note**
>
> StreamXPath remains available for compatibility with applications that use weblogic.xml.stream. The underlying XMLInputStream API is deprecated in favor of standard StAX. StreamXPath implements a subset of XPath; unsupported constructs include several axes and core functions documented in the StreamXPath API reference. For new streaming designs, start with the standard StAX APIs and use this WebLogic API only when its compatibility behavior and XPath subset meet the application's requirements.

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

StreamXPath uses deprecated stream types and a defined XPath subset. Review the compatibility guidance in Using the StreamXPath Class before using this example.

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'>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    <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.
 */

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 stream.

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

    // Create an XPathStreamFactory.
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    XPathStreamFactory factory = new XPathStreamFactory();

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

StreamXPath uses deprecated stream types and a defined XPath subset. Review the compatibility guidance in Using the StreamXPath Class before using this example.

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`.

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


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `weblogic.xml.xpath` in the *Java API Reference for Oracle WebLogic Server*.


Copyright © 2007, 2026, Oracle and/or its affiliates.

6

# XML Programming Best Practices

This chapter describes the best programming practices to use when creating Java applications that process XML data.
This chapter includes the following sections:

## When to Use the DOM, SAX, and StAX APIs

You can parse an XML document with the DOM, SAX, or StAX APIs. This section describes when to use each API.

Use DOM when the application needs random access, repeated traversal, or in-memory modification and representative documents fit within the application's measured memory and latency budget. Benchmark with production-size documents and expected concurrency; there is no portable element-count cutoff.

SAX: Use SAX for single-pass, event-driven processing when the application is structured around parser callbacks and does not need an in-memory tree. Document depth and repeated element names are not selection limits; the handler must maintain whatever context the application requires.

StAX: Use StAX for single-pass streaming when application code should control when the parser advances. The cursor API offers low-overhead pull processing; the event API offers event objects and iterator-style composition. Like SAX, StAX can keep memory use low, but it is a distinct pull model rather than an API based on SAX.

## Increasing Performance of XML Validation

Validation has a runtime cost, so enable it according to the document contract and trust boundary. When a protocol or application requires conformance, validate the complete document. For DTD validation, call `SAXParserFactory.setValidating(true)` or `DocumentBuilderFactory.setValidating(true)`. For XML Schema validation, create a `Schema` and set it on the parser factory with `setSchema(Schema)`.

If full DTD or XML Schema validation is not required, check the specific business invariants the application depends on and describe those checks as business-rule validation, not XML validity. Business-rule checks do not establish DTD or XML Schema conformance.

Measure representative workloads before changing validation strategy. Configure external-access controls separately; enabling validation does not authorize unrestricted retrieval of schemas or DTDs.

## When to Use XML Schemas or DTDs

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

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. A SOAP message must not contain a Document Type Declaration. Use XML Schema to describe and, where required, validate application payload structures.


Copyright © 2007, 2026, Oracle and/or its affiliates.

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. Because DTDs are less expressive than Schemas, they are easier to write and manage.

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

# Configuring External Entity Resolution for Maximum Performance

Oracle 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.

Enable external resolution only when the application requires a known external resource. Map approved public/system identifiers to trusted local registry entries. Configure the calling parser's resolver and external-access policy to fail closed for unknown identifiers; do not assume that the registry mapping alone rejects an identifier that has no mapping. For untrusted XML that does not require DTDs or external entities, disable those features and external access. Do not allow a missing registry entry to fall back to an unrestricted network fetch.

# 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.

Oracle 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.

Measure representative transformations before changing implementation. For repeated XSLT, compile the stylesheet once with TransformerFactory.newTemplates(Source), reuse the thread-safe Templates object, and create a separate Transformer for each operation or thread. Configure the factory to restrict external DTD and stylesheet access for untrusted inputs. Choose DOM only when the transformation requires random access or mutation and its


Copyright © 2007, 2026, Oracle and/or its affiliates.

measured memory cost is acceptable; use streaming sources/results where the stylesheet and processor allow them.


Copyright © 2007, 2026, Oracle and/or its affiliates.

7

# XML Programming Techniques

This chapter describes specific XML programming techniques for developing a Jakarta EE application with WebLogic Server that processes XML data.
This chapter includes the following sections:

## Transmitting XML Data Between a Java Client and WebLogic Server

In a typical Jakarta EE 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 Jakarta EE 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 and read the response, use `java.net.http.HttpClient`. The following example sends the complete XML file using HTTP POST and reads the response as bytes.

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

```
import java.io.IOException;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;
import java.nio.file.Path;
import java.time.Duration;

public final class Client {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: java Client <URL> <XML-file>");
            return;
        }
        HttpClient client = HttpClient.newBuilder()
            .connectTimeout(Duration.ofSeconds(10))
            .build();
        HttpRequest request = HttpRequest.newBuilder(URI.create(args[0]))
            .timeout(Duration.ofSeconds(30))
            .header("Content-Type", "application/xml")
            .header("Accept", "application/xml")
            .POST(HttpRequest.BodyPublishers.ofFile(Path.of(args[1])))
            .build();
        HttpResponse<byte[]> response = client.send(
            request, HttpResponse.BodyHandlers.ofByteArray());
        if (response.statusCode() < 200 || response.statusCode() >= 300) {
            throw new IOException("HTTP " + response.statusCode());
        }
        System.out.write(response.body());
    }
}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

The example posts the XML file given in the argument list, checks the HTTP status, and writes the response bytes without decoding or truncating them. In production, use HTTPS and the application's required authentication.

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:

```
<%@ page
import="java.io.PrintWriter,javax.xml.XMLConstants,javax.xml.parsers.DocumentBuilder,java
x.xml.parsers.DocumentBuilderFactory,org.w3c.dom.Document" %>
<%
response.setContentType("application/xml");
response.setCharacterEncoding("UTF-8");
DocumentBuilderFactory fact = DocumentBuilderFactory.newInstance();
fact.setNamespaceAware(true);
fact.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);
fact.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
fact.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
fact.setXIncludeAware(false);
fact.setExpandEntityReferences(false);
DocumentBuilder db = fact.newDocumentBuilder();
Document doc = db.parse(request.getInputStream());
// Keep the existing processing that uses doc here.
PrintWriter responseWriter = response.getWriter();
responseWriter.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
%>
```

Apply an application/container request-size limit. If this endpoint intentionally accepts a controlled trusted DTD or schema, use an allowlisting resolver or catalog instead of opening arbitrary external access.

For detailed information on programming WebLogic servlets and JSPs, see *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*.

# Handling XML Documents in a JMS Application

WebLogic Server provides the following extensions to some Jakarta Messaging (JMS) classes to handle XML documents in a JMS application:

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

Use the WebLogic JMS_BEA_SELECT selector function to evaluate XPath against an XMLMessage. For example: `JMS_BEA_SELECT('xpath', '/Order/Manifest/Item[2]/ID/text()') = '208'`. The first argument selects XPath evaluation, the second is the XPath expression, and the selector compares the returned value.

To create a WebLogic XMLMessage, obtain the provider session as weblogic.jms.extensions.WLSession and call createXMLMessage(). WLQueueSession and WLTopicSession are domain-specific compatibility interfaces that extend WLSession.


Copyright © 2007, 2026, Oracle and/or its affiliates.

For detailed information on using `XMLMessage` objects in your JMS application, see *Developing JMS Applications for Oracle WebLogic Server*.

# 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 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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

8

# XML Application Scoping

This chapter describes application scoping and how to configure parsers, transformers, external entities, and the external entity cache for a particular application.
This chapter includes the following sections:

## 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, xpath, schema, xmlInput, xmlOutput, xmlEntity 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 WebLogic Remote Console to configure parser, transformer, and external entity resources for a WebLogic Server instance, as described in 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 Jakarta EE deployment descriptor is called `application.xml`.

The following sample configures an application-scoped entity mapping and uses the default XML provider selection. The `parser-factory` element is optional; omit it to use WebLogic Server's default XML Registry/JAXP provider selection. Configure `parser-factory` only when the application requires a custom provider. Each configured value must name a public, concrete provider class with an accessible public no-argument constructor that is visible to the application class loader, is an implementation of the corresponding standard JAXP factory type, and is compatible with WebLogic Server. Do not configure JDK-internal `com.sun.org.apache.*.internal` implementation classes.

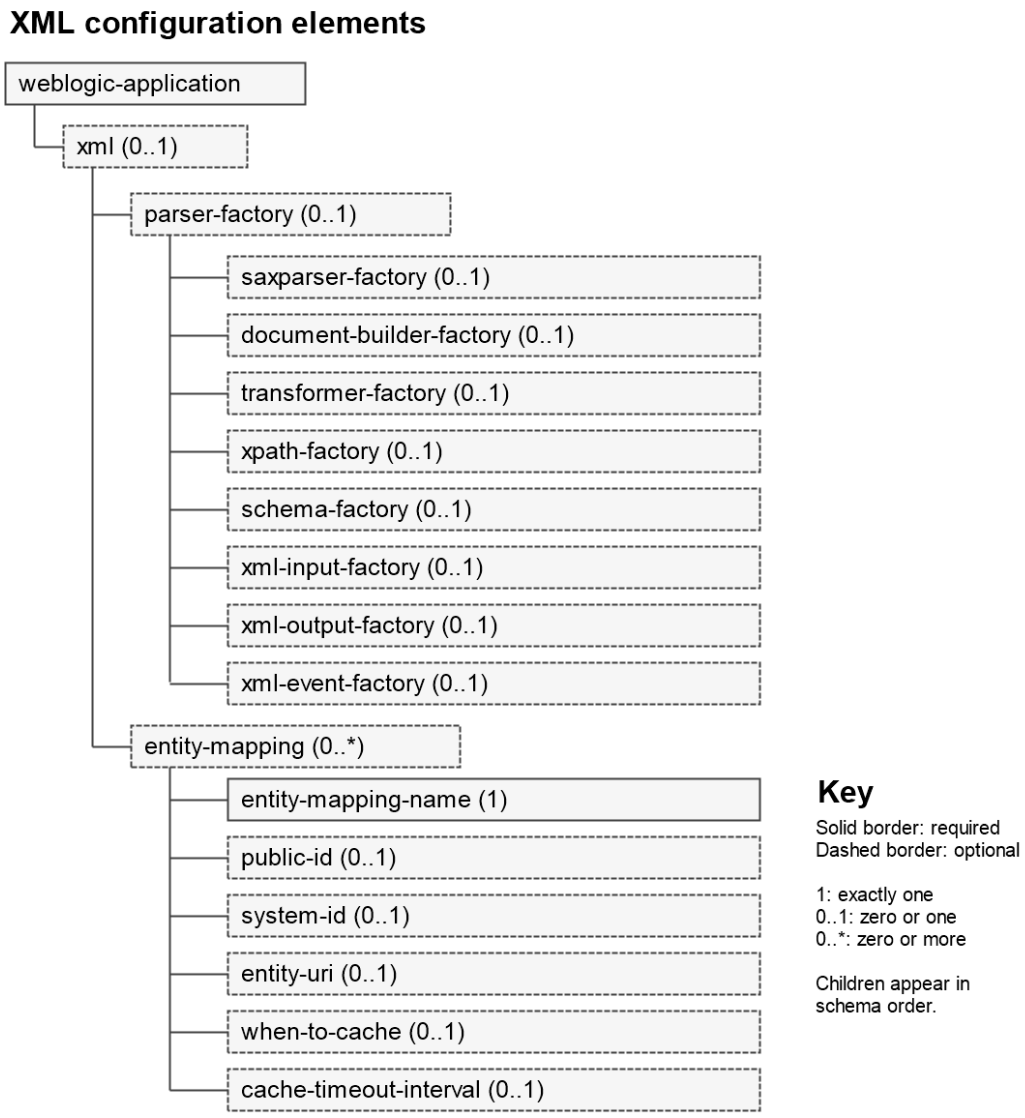
```
<weblogic-application>
  ...
  <xml>
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    <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>. Figure 8-1 describes the sub-elements of the <xml> element; the sections following this figure describe each element.

**Figure 8-1 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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

## 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.

## schema-factory

Element that enables applications deployed to WebLogic Server to configure and obtain a `Schema` object used to validate XML documents using schema. If this element is not specified, the default schema factory specified for the WebLogic Server instance is used.

## xpath-factory

Element that enables applications deployed to WebLogic Server to configure and obtain a `XPath` object used to search XML elements. If this element is not specified, the default XPath factory specified for the WebLogic Server instance is used.

## xml-input-factory

Element that enables applications deployed to WebLogic Server to configure and obtain a `XMLEventReader/XMLStreamReader` object used to read XML streams. If this element is not specified, the default XML input factory specified for the WebLogic Server instance is used.

## xml-output-factory

Element that enables applications deployed to WebLogic Server to configure and obtain a `XMLEventWriter/XMLStreamWriter` object used to write XML streams. If this element is not specified, the default XML output factory specified for the WebLogic Server instance is used.


Copyright © 2007, 2026, Oracle and/or its affiliates.

## xml-event-factory

Element that enables applications deployed to WebLogic Server to configure and obtain a `XMLEvent` object used to parse or build XML streams. If this element is not specified, the XMLEvent 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 path of the mapped entity relative to the application XML registry directory, `lib/xml/registry`, in the EAR archive or exploded enterprise application. For example, `dtds/car.dtd` resolves to `lib/xml/registry/dtds/car.dtd`.

## when-to-cache

For an application-scoped entity mapping, use one of the following values:

- `cache-on-reference`—WebLogic Server caches the external entity the first time it is resolved;
- `cache-never`—WebLogic Server does not cache the external entity.

The default is `cache-on-reference`. Do not use `cache-at-initialization` for an application-scoped mapping; in this release, application registry initialization does not preload the entry.

## cache-timeout-interval

Element that specifies the number of seconds after which the cached external entity becomes stale, or out-of-date. If the cached copy has been in the cache for longer than this interval, WebLogic Server retrieves it again the next time the entity is requested, from the specified URL or path under the application XML registry directory, `lib/xml/registry`.

The default value for this field is 120 seconds.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# Configuring a Parser or Transformer for an Enterprise Application

You can specify that an XML application use a parser or transformer different from the default 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 default, for an enterprise application, follow these steps:

> **Note**
>
> The following factory types follow the same process detailed below: `XPathFactory`, `SchemaFactory`, `XMLInputFactory`, `XMLOutputFactory` and `XMLEventFactory`.

1. Use the `parser-factory` subelement of `xml` to configure application-specific factory classes. The values in the following syntax example are placeholders; replace each one with the fully qualified name of a public, concrete provider class with an accessible public no-argument constructor that is visible to the application class loader, is an implementation of the corresponding standard JAXP factory type, and is compatible with WebLogic Server. Do not use JDK-internal `com.sun.org.apache.*.internal` implementation classes.

   ```
   <parser-factory>
     <saxparser-factory>com.example.xml.CustomSAXParserFactory</saxparser-factory>
     <document-builder-factory>com.example.xml.CustomDocumentBuilderFactory</document-
   builder-factory>
     <transformer-factory>com.example.xml.CustomTransformerFactory</transformer-factory>
   </parser-factory>
   ```

   The three values in this syntax example are placeholders, not provider classes supplied by WebLogic Server. Replace only the factory types that the application needs to override. Package the selected provider classes as described in the following steps.

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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 cache an external entity referenced by a URL when it is first referenced. You can also map an entity to a local path under the application XML registry directory, `lib/xml/registry`. Caching avoids repeated remote retrieval. After a cached entity exceeds its configured timeout, WebLogic Server attempts to retrieve it again the next time the entity is requested and caches the resulting copy.

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-on-reference</when-to-cache>
  <cache-timeout-interval>300</cache-timeout-interval>
</entity-mapping>
```

In the example, `car.dtd` is stored in `lib/xml/registry/dtds` under the EAR archive or exploded application. WebLogic Server caches the DTD when it is first referenced. After the cached copy exceeds 300 seconds, WebLogic Server attempts to retrieve the entity again the next time it is requested and caches the resulting copy.


Copyright © 2007, 2026, Oracle and/or its affiliates.

9

# Administering WebLogic Server XML

This chapter describes administering XML in WebLogic Server using the XML Registry, which is accessed through the WebLogic Remote Console.
This chapter includes the following sections:

## Overview of Administering WebLogic Server XML

You access the XML Registry through the WebLogic Remote Console and use it to configure WebLogic Server for XML applications. To access the WebLogic Remote Console, see Get Started in the *Oracle WebLogic Remote Console Online Help*.

### XML Administration Tasks

You create, configure, and use the XML Registry using the WebLogic Remote Console. Using the WebLogic Remote 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 default parser.
- An XML parser per document type.
- An alternative server-wide transformer instead of the default 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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

If an instance of WebLogic Server does not have an XML Registry associated with it, then the default parser and transformer are used when parsing or transforming documents. The default parser and transformer are those included in the JDK.

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 parse or transform your XML applications. WebLogic Server follows an ordered lookup when determining which parser class to load:

1. If the current enterprise application configures the requested factory in `weblogic-application.xml`, use the application-scoped factory.
2. Otherwise, use the parser defined for the document type in the XML registry associated with the WebLogic Server instance.
3. Otherwise, use the alternative server-wide parser defined in that XML registry.
4. Otherwise, use the default parser supplied through JAXP by the JDK.

The same order applies to transformers, except that there is no document-type-specific transformer step.

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.

For application-scoped configuration, see The weblogic-application.xml File in XML Application Scoping.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# XML Parser and Transformer Configuration Tasks

By default, WebLogic Server is configured to use the default parser and transformer to parse and transform XML documents. The default parser and transformer are those included in the JDK. 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 default, you must use the XML Registry to configure them.

## Configuring a Parser or Transformer Other Than the Default

First, you have to create an XML registry that defines SAX and DOM parsers and transformers. Then, you associate (or *plug-in*) the new XML Registry with an instance of WebLogic Server so that the server starts to use the new parsers and transformer.

> **Note**
>
> You can plug-in only those parsers and transformers that are compatible with the default WebLogic Server parser transformer. The default parser and transformer are those that are included in the JDK.

Follow the steps outlined in Create an XML Registry in the *Oracle WebLogic Remote 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 to identify the document type.

> **Note**
>
> 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.

To configure a parser for a particular document type, see Step 4 under Create an XML Registry in the *Oracle WebLogic Remote 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:


Copyright © 2007, 2026, Oracle and/or its affiliates.

- 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.

To configure a trusted local copy of an external entity with WebLogic Remote Console:

1. On the Administration Server host, create `DOMAIN_HOME/xml/registries/<registry-name>` if it does not already exist. Copy the trusted entity below that directory. Choose a relative path for the Entity URI; for example, if the registry is named `MyXMLRegistry` and the Entity URI is `dtds/catalog.dtd`, store the file as `DOMAIN_HOME/xml/registries/MyXMLRegistry/dtds/catalog.dtd`. Ensure that the Administration Server process can read the file.
2. In the Edit Tree, go to Services > XML Registries. Click New, enter `<registry-name>`, and click Create.
3. In the navigation tree, expand the new registry and select XML Entity Spec Registry Entries. Click New.
4. Enter a unique Name. In Public Id and/or System Id, enter the external identifier or identifiers exactly as they appear in the XML document. In Entity URI, enter the relative path chosen in step 1. Click Create. Map only identifiers and local entity files that you trust.
5. Associate the registry with every server that must use it. In the Edit Tree, go to Environment > Servers and select the server. In WebLogic Remote Console 3.0.5, select Services > XML; in XML Registry, select `<registry-name>`, and click Save. A server can have only one XML registry, although one registry can be associated with multiple servers.
6. Open the Shopping Cart, review the pending changes, and click Commit Changes.
7. Restart each affected server when WebLogic Remote Console displays the Server or App Restart Required icon. The new server-registry association does not take effect until the restart completes.
8. Verify the mapping with an XML document whose public or system identifier matches the entry and confirm that the parser reads the trusted local entity without retrieving the external URI.

See XML Resources in WebLogic Remote Console 3.0.5 for registry creation and server-association details.

## 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


Copyright © 2007, 2026, Oracle and/or its affiliates.

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, see Create an XML Entity Cache in the *Oracle WebLogic Remote Console Online Help*.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# A
# XML Reference

This appendix provides links to additional information about the XML specifications, application programming interfaces (APIs), and tools supported by WebLogic Server.
This appendix includes the following sections:

## XML APIs

- SAX API (Java SE 21), available at https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/org/xml/sax/package-summary.html
- DOM Level 2 Specification, available at https://www.w3.org/TR/DOM-Level-2/
- JAXP API (Java SE 21), available at https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/module-summary.html

## Code Examples

XML code examples and supporting documentation are optionally included in the WebLogic Server *ORACLE_HOME*\wlserver\samples\server directory, where *ORACLE_HOME* represents the directory in which you installed WebLogic Server. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

## Related WebLogic Server Documentation

- *Developing JAX-WS Web Services for Oracle WebLogic Server*
- *Developing Jakarta Enterprise Beans Using Deployment Descriptors*
- *Developing JMS Applications for Oracle WebLogic Server*
- *Developing Web Applications, Servlets, and JSPs for Oracle WebLogic Server*

## Tutorials and Online Courses

- A Technical Introduction to XML, available at https://www.xml.com/pub/a/98/10/guide0.html
- XML, Java, and the Future of the Web, available at https://www.xml.com/pub/a/w3j/s3.bosak.html
- XML Schema Part 0: Primer, available at https://www.w3.org/TR/xmlschema-0/
- Java API for XML Processing (JAXP), Java SE 21, available at https://docs.oracle.com/en/java/javase/21/docs/api/java.xml/module-summary.html

## Other XML Specifications and Information

- XML 1.0, available at https://www.w3.org/TR/xml/


Copyright © 2007, 2026, Oracle and/or its affiliates.

- XML Schema Part 1, available at https://www.w3.org/TR/xmlschema-1/
- XML Schema Part 2, available at https://www.w3.org/TR/xmlschema-2/
- Namespaces in XML, available at https://www.w3.org/TR/xml-names/
- XSL 1.0, available at https://www.w3.org/TR/2001/REC-xsl-20011015/
- XPath 1.0, available at https://www.w3.org/TR/xpath-10/
- XLink 1.1, available at https://www.w3.org/TR/xlink11/
- XPointer Framework, available at https://www.w3.org/TR/xptr-framework/
- Jakarta XML Binding 4.0, available at https://jakarta.ee/specifications/xml-binding/4.0/
- W3C, available at https://www.w3.org/
- XML.com, available at https://www.xml.com/


Copyright © 2007, 2026, Oracle and/or its affiliates.

B

# Using the WebLogic XML Streaming API (Deprecated)

This appendix describes how to use the WebLogic XML Streaming API to parse and generate XML documents.

> **Note**
>
> The WebLogic XML Streaming API has been deprecated as of release 9.0 of WebLogic Server. The API remains available in this release for compatibility. Do not use it for new applications; use the standard StAX API. You should instead use the Streaming API for XML (StAX), a standard specification from the Java Community Process. For details, see Using the Streaming API for XML (StAX).

This appendix includes the following sections:

## 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.

For a complete example of parsing an XML document using the streaming API, see the *`ORACLE_HOME`*`\wlserver\samples\server` directory, where *`ORACLE_HOME`* represents the directory in which you installed WebLogic Server. For more information about the WebLogic Server code examples, see Sample Applications and Code Examples in *Understanding Oracle WebLogic Server*.

Table B-1 describes the main interfaces and classes of the WebLogic Streaming API.


Copyright © 2007, 2026, Oracle and/or its affiliates.

**Table B-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`
- `weblogic.xml.stream.util`

# 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.
3. Iterate over the stream, returning generic `XMLEvent` types. See Iterating Over the Stream.
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.
5. Get the attributes of an element. See Getting the Attributes of an Element.
6. Position the stream by skipping over event, skipping to a particular event, and so on. See Positioning the Stream.
7. Get the children of an element. See Getting a Substream.


Copyright © 2007, 2026, Oracle and/or its affiliates.

8. Close the stream. See Closing the Input Stream.

# 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
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
 *  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
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
   * @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)));
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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.

## 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));
```

Table B-2 describes the filters provided by the WebLogic XML Streaming API. They are part of the `weblogic.xml.stream.util` package.

**Table B-2 Filters Provided by WebLogic XML Streaming API**

| 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 for a full list of event types.<br>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")` |
| `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 NamespaceTypeFilter("http://namespace.org", XMLEvent.START_ELEMENT)`<br>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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

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.ElementFilter;
import weblogic.xml.stream.XMLEvent;

public class SuperDooperFilter implements ElementFilter {
  private final String name;

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

  public boolean accept(XMLEvent event) {
    return event.hasName()
        && name.equals(event.getName().getLocalName());
  }
}
```

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. Figure B-1 shows the hierarchy of the `XMLEvent` interface and its subinterfaces.


Copyright © 2007, 2026, Oracle and/or its affiliates.

**Figure B-1 Hierarchy of the XMLEvent Interface and Its SubInterfaces**

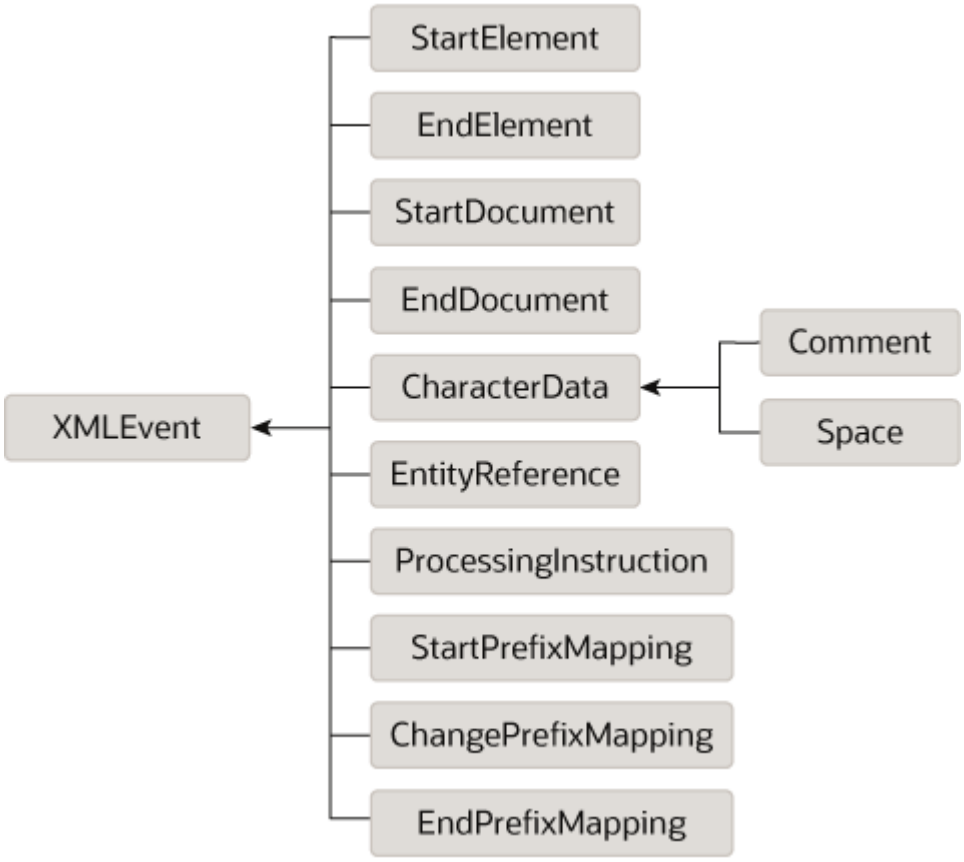



Table B-3 lists the subclasses and fields of the `XMLEvent` class that you can use to identify a particular event while parsing the XML stream

**Table B-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. |
| `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. |


Copyright © 2007, 2026, Oracle and/or its affiliates.

**Table B-3 (Cont.) 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 |
|---|---|---|---|
| `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");
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
    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:

```
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;
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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 `Attribute.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

Table B-4 describes the methods of the `XMLInputStream` interface that you can use to skip ahead to specific locations in the stream.

**Table B-4 Methods Used to Position the Input Stream**

| Method of XMLInputStream | Description |
|---|---|
| `skip()` | Positions the input stream to the next stream event.<br>**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.<br>Examples of event types are `XMLEvent.START_ELEMENT` and `XMLEvent.END_DOCUMENT`. Refer to Table B-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


Copyright © 2007, 2026, Oracle and/or its affiliates.

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.


Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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.
3. Add events to the XML output stream. See Adding Elements to the Output Stream.
4. Add attributes to the XML output stream. See Adding Attributes to an Element on the Output Stream.
5. Add an input stream to the output stream. See Adding an Input Stream to an Output Stream.
6. Print the output stream. See Printing an Output Stream.
7. Close the output stream. See Closing the Output Stream.


Copyright © 2007, 2026, Oracle and/or its affiliates.

# 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
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

```
   * @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();
  }
}
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

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`.

> **Tip**
>
> The `XMLOutputStream` does not validate your XML.


Copyright © 2007, 2026, Oracle and/or its affiliates.

> **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"));
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

## 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();
```


Copyright © 2007, 2026, Oracle and/or its affiliates.

# 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();
```


Copyright © 2007, 2026, Oracle and/or its affiliates.