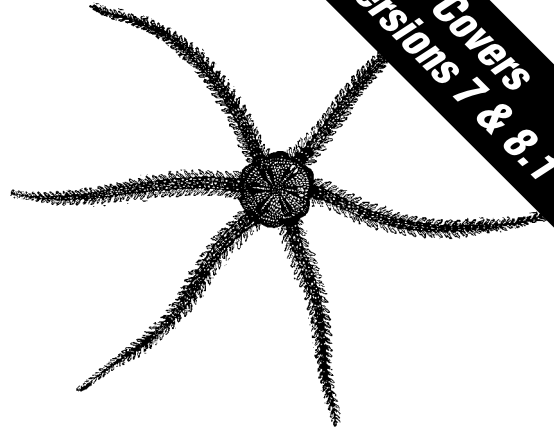


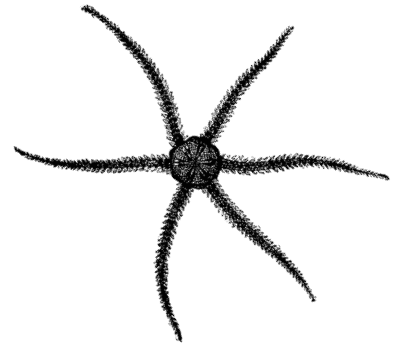
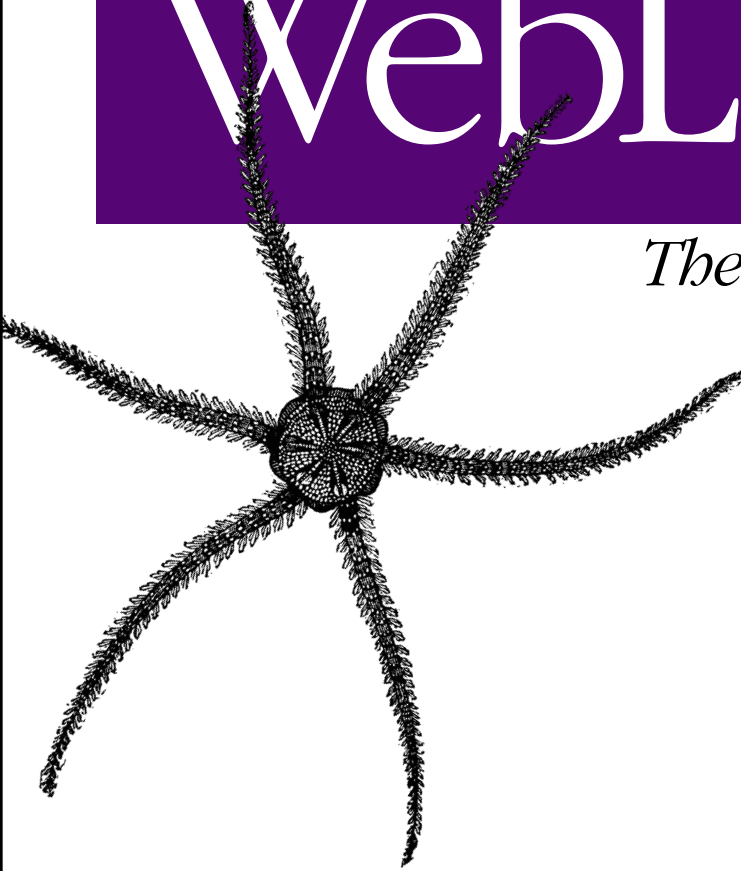
*Development, Deployment & Maintenance*

**Covers  
Versions 7 & 8.1**



# WebLogic™

*The Definitive Guide*



**O'REILLY®**

*Jon Mountjoy & Avinash Chugh*

WebLogic provides extensive support for XML and XML-related APIs. Not surprisingly, it comes equipped with modern SAX and DOM parsers based on Apache's Xerces 2.1.0, and an XSLT transformer based on the Xalan 2.2 libraries that is shipped with the JDK 1.4.1 distribution. It also is shipped with a FastParser, which is a high-performance nonvalidating SAX parser. WebLogic's FastParser is optimized for small to medium-size documents, typical of a lot of SOAP and WSDL documents.

WebLogic implements the standard JAXP 1.1 layer, which provides a generic way to access any parser or transformer. Its XML Registry integrates with the JAXP implementation. This allows you to override the built-in parsers and transformers and specify alternative implementations that should be used under different circumstances. For example, you can set up different parsers to be based on the root element or document type information. These XML Registries can be configured on a per-server basis, giving you good control of what XML parsers should be used when. Of course, all these adjustments can be made without changing a single line of Java code.

XML documents often contain references to *external entities*. External entities, such as DTDs, are pieces of text external to the document being parsed. An XML Registry can be used to configure local mappings for these DTDs. If this is done, WebLogic can bypass a potentially time-consuming fetch for the DTD whenever XML data needs to be parsed. The registry also has a configurable cache, which can be used on the results of external entity resolution.

WebLogic's Streaming API offers a parsing paradigm very different from either the DOM or SAX models. Like the SAX model, the program views an XML document as a number of parser events that signify important structural elements within the document. Unlike SAX, the Streaming API supports a *pull model* for parsing XML data, whereby the program must explicitly extract XML events from the stream.

The Streaming API lets you iterate over the stream, pulling off events and processing the XML data depending on the events you encounter. You also can skip ahead over

a number of XML events, or just ignore them altogether. You even can apply a filter to the XML stream to ensure that only specific events are allowed through the filter. Finally, you can use the Streaming API to generate XML data and direct this output to various destinations.

WebLogic also provides an XPath API that lets you match XPath queries against a DOM or an XML input stream. WebLogic supports two convenience features as well: a JSP tag library that provides easy access to XSL transformers from within a JSP page, and special-purpose servlet request attributes that automatically can parse the message body of an incoming request.

Finally, WebLogic provides a standalone, Java-based XML editor, which is a simple tool for creating and editing XML documents. It supports both a plain-text view and a tree view of an XML document. The XML Editor also lets you validate a document against a DTD or an XML Schema. WebLogic's JMS implementation includes support for a new type of message that can hold XML data. This is covered in Chapter 8.

BEA has created other XML technologies, not exclusively for use in WebLogic Server, which also deserve a mention. This includes a Streaming API for XML implementation, and XMLBeans, which is an XML Schema to Java Object mapping tool. You can learn more about these from BEA's dev2dev web site.

## JAXP

The Java API for XML Processing (JAXP) defines a generic API for processing XML documents and transforming an XML source. The `javax.xml.parsers` package contains classes and interfaces needed to parse XML documents in SAX 2.0 and DOM 2.0 modes, while the `javax.xml.transform` package contains the classes and interfaces needed for transforming XML data (a source) into another format (a result). An XML document can manifest itself in several ways: a stream (`File`, `InputStream`, or `Reader`), SAX events, or a DOM tree representation. The aim of JAXP is to provide applications with a portable way for parsing and transforming XML documents. WebLogic Server is shipped with JAXP 1.1 classes and interfaces—by default, this implementation of JAXP is configured to use WebLogic's built-in parsers and transformers.

The JAXP specification demands that you set the appropriate system properties in order to plug in a custom XML parser (or transformer). However, WebLogic deviates from this model. Instead of using system properties, WebLogic allows you to determine the actual parsers used in two ways:

- You can create an XML Registry that lets you configure server-specific and document-specific parsers and transformers. XML Registries are domain resources configured using the Administration Console.
- You can define application-scoped XML settings in the WebLogic-specific *weblogic-application.xml* deployment descriptor for an EAR. Here you can specify parser and transformer factories that apply to an enterprise application and all its constituent modules.

Both of these mechanisms allow you to configure the parsers and transformers used by your applications at deployment time. This means that you can transparently alter the configuration without changing a single line of code.

In the following sections, we'll show you how to use WebLogic's JAXP to create SAX and DOM parsers, and a transformer based on an XSLT stylesheet. We also will look at WebLogic's support for resolving external entities.

## SAX

The Simple API for XML (SAX) is an event-based API for parsing XML documents. The SAX interface defines various events that are triggered while an XML parser is reading an XML document. A custom handler can listen for these events and process the XML document in an easy, event-oriented fashion. A SAX handler implements the relevant callback methods and responds to various events—such as when it encounters start and end tags, character data, or a processing instruction. As the SAX parser reads through an XML document, it invokes the methods on the handler class to mark each particular event. In this regard, the SAX interface is unlike other XML APIs because it relies on a *push model*; the data is pushed to the application as it is encountered.

Example 18-1 shows how to retrieve a SAX parser using the JAXP interface. In order to parse an XML document with SAX, you need to create an instance of a SAX parser. The JAXP interface defines a `SAXParserFactory`, which manufactures a SAX parser for you. The `newInstance()` method in the `SAXParserFactory` class creates a new factory object. Call the `newSAXParser()` method on this factory instance to create a new SAX parser.

*Example 18-1. Retrieving and using a SAX parser*

```
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
//...
//Obtain an instance of SAXParserFactory
SAXParserFactory spf = SAXParserFactory.newInstance();
//Obtain a SAX parser from the factory
SAXParser sp = spf.newSAXParser();
//Parse an XML document
sp.parse(new java.io.StringBufferInputStream("<a><b></b></a>"),
    new org.xml.sax.helpers.DefaultHandler() {
        public void startElement (String uri, String localName,
            String qName, Attributes attr
            { System.err.println(qName); });
    });
```

You can now use the SAX parser to parse an XML document. In this example, the SAX handler registered with the parser responds whenever a start tag is encountered. For each start tag, it simply prints out the name of the tag.

By default, WebLogic Server uses its built-in SAX parser factory: `weblogic.apache.xerces.jaxp.SAXParserFactoryImpl`. Later, we'll see how you can override this setting and configure WebLogic to use an alternative SAX parser.

## DOM

An XML document is a tree of elements—data elements can be nested within other elements and optionally have attributes attached. The Document Object Model (DOM) defines an object hierarchy that represents the structure of an XML document as a recursive tree of elements. This means that you need to read and parse the entire XML document in order to build this data structure. Because a DOM representation of an XML document needs to be held in memory, many DOM implementations can be memory-intensive.

As its name suggests, DOM parsing uses a hierarchical, object-based model. XML parsing in DOM mode is useful when you need frequent, random access to different parts of the document, or if you want to manipulate its structure in complex ways. However, the DOM API is not suitable for applications that need to parse XML data incrementally; in these cases, you should use SAX.

Example 18-2 shows how you can use the JAXP interface to create a DOM parser. In order to parse an XML document in DOM mode, you need to obtain an instance of `DocumentBuilder`. Once again, the JAXP interface provides a factory, called the `DocumentBuilderFactory`, which manufactures a DOM parser for you. The `newInstance()` method on the `DocumentBuilderFactory` class creates a new DOM parser factory. Call the `newDocumentBuilder()` method on the `DocumentBuilderFactory` instance to create a new DOM parser.

*Example 18-2. Retrieving and using a DOM parser*

```
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
//...
//Obtain an instance of the DocumentBuilderFactory
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
// Get a DOM parser
DocumentBuilder db = dbf.newDocumentBuilder();
System.err.println(db.getDOMImplementation().getClass().getName());
//Parse the document
Document doc = db.parse("*.xml");
System.err.println(doc.getDocumentElement().getNodeName());
```

Once you create a `DocumentBuilder` object, you can parse an XML document and build a DOM tree. The `parse()` method returns an `org.w3c.dom.Document` object, which represents the hierarchical view of the XML document. Now that you have built the DOM tree in memory, you can access the elements within the `Document`, or perhaps even modify the structure. In the earlier example, we simply print the name of the root element of the document.

By default, WebLogic uses the built-in DOM parser factory. Later, we will see how you can configure WebLogic to use a third-party DOM parser.

## XSL Transformers

XSL Transformations (XSLT) are XML-formatted rules that define how one XML document can be transformed into another. You typically would assemble templates specifying a particular transformation and place these in an XSLT stylesheet. Then, an XSLT processor uses the templates defined in the stylesheet to process matching elements in the input XML source and writes the template conversion into an output tree. The source and sink (or output target) of an XSL Transformation can be a stream, a DOM tree, or SAX events. JAXP provides a standard way for acquiring an XSL transformer. The same XSL transformer can process XML data that originates from a variety of sources and can write the output of the transformation to a variety of sinks. For instance, using the same XSLT stylesheet, you could easily transform a stream of input SAX events to a DOM tree representation of the output XML document.

The JAXP interface enables you to capture XSLT stylesheets in two ways:

- A `Transformer` object that is created via the `newTransformer()` factory method on an instance of the `TransformerFactory`.
- A `Templates` instance that encapsulates runtime-compiled information about the XSLT stylesheet. You can create a `Templates` object via the `newTemplates()` method on an instance of the `TransformerFactory`.

A `Templates` object can be shared across multiple concurrent threads and may be used multiple times during its lifetime. However, an XSL Transformer isn't thread-safe and must not be shared across multiple concurrent threads. Typically, a single `Templates` object will spawn multiple `Transformer` instances, one for each thread that needs to use the XSLT stylesheet. Example 18-3 shows how you can create an XSL transformer for a given stylesheet.

*Example 18-3. Using JAXP to access a transformer*

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
//...
TransformerFactory tf = TransformerFactory.newInstance();
Transformer t =
    tf.newTransformer(new StreamSource(new java.io.File("foo.xml")));

// apply stylesheet to a DOM tree and write the
// output of transformation to System.out
t.transform(new DOMSource(doc), new StreamResult(System.out));
```

Alternatively, you could create a `Templates` object that captures all the transformation instructions specified in the stylesheet. For example, you could create a `Templates` instance in the `init()` method of a servlet:

```
TransformerFactory tf = TransformerFactory.newInstance();
Templates tmpl =
    tf.newTemplates(new StreamSource(new java.io.File("foo.xml")));
```

Then, when you need to apply the XSLT stylesheet to an XML document (for instance, in the `doPost()` method of a servlet), you need only to create a new `Transformer` instance using the same `Templates` object:

```
// Needs to be applied to an XML document
Transformer t = tmpl.newTransformer();
t.transform(domSource, streamResult);
```

By default, WebLogic uses its built-in factory for processing XSLT stylesheets: `org.apache.xalan.processor.TransformerFactoryImpl`.

## External Entity Resolution

External entities are portions of text external to the XML file being parsed. An external entity resembles a macro replacement facility. The replacement text can either be *parsed*, which means the text is incorporated into the XML document, or *unparsed*, which means the declaration points to external data. A typical external entity declaration uses the `SYSTEM` keyword and the URI of the substitution text:

```
<!ENTITY header SYSTEM "http://www.oreilly.com/templates/header.xml">
```

Alternatively, the declaration also could specify a relative URL:

```
<!ENTITY header SYSTEM "/templates/header.xml">
```

Whenever a parser encounters an entity reference `&header;`, it replaces the reference with the actual contents of the document located at the specified URI. A DTD reference is another example of an external entity reference. An XML document uses a `DOCTYPE` declaration to reference a DTD:

```
<!DOCTYPE oreilly SYSTEM "http://www.oreilly.com/dtds/wl.dtd">
<oreilly> <!-- --> </oreilly>
```

The preceding declaration indicates that the root element for the document is `oreilly`, and its DTD is located at `http://www.oreilly.com/dtds/wl.dtd`. In addition, the `DOCTYPE` declaration may use a *public identifier*, which is a publicly declared name that identifies the associated DTD. When a validating XML parser processes an XML document that includes a `DOCTYPE` declaration, the parser needs to fetch the DTD file referenced by the URI.

The SAX interface allows you to associate a custom entity resolver with the parser, which can intercept parser requests for external entities (including an external DTD) and determine how the entity references are resolved. To do this, you need to register an instance of the `org.xml.sax.EntityResolver` interface with the `XMLReader` before

you begin to read from an XML source. Whenever the SAX parser encounters an entity reference, it will invoke the `resolveEntity()` method on the registered `EntityResolver` instance.



SAX's `XMLReader` interface is encapsulated by the JAXP `SAXParser` class.

Typically, an application uses an `EntityResolver` to substitute a reference to a remote URI with a local copy. The `resolveEntity()` method returns an input source for the XML entity based on either a character or a byte stream. If the method returns `null`, the parser tries to open a connection to the URI referenced by the system identifier. A custom entity resolver is very useful if your application parses XML documents that need to be retrieved from a database or other nonstandard locations.

WebLogic's XML Registry provides several enhancements that improve the performance of external entity resolution:

- It allows you to map an external entity to a local (or remote) URI that contains a copy of the substitution text for the entity.
- It allows you to retrieve a copy of the remote resource associated with the external entity and cache the text, either in memory or on disk.
- It allows you to specify the duration after which a cached item becomes stale and needs to be refreshed.

Each entry in the XML Registry uses a public or system identifier to identify the external entity. Each entry also may specify various caching options for entity resolution. For instance, you could instruct the server to fetch an external entity the first time it is required, and cache it for 120 seconds. Whenever an XML parser reads a document that uses an entity reference configured in the registry, WebLogic Server fetches a local or a cached copy of the substitution text.

## Built-in Processors

The WebLogic distribution comes equipped with a number of standard XML parsers and transformers: the Apache-based Xerces and Xalan libraries, which provide the SAX and DOM parsers as well as the XSLT transformer. You also can use WebLogic's own `FastParser`, which is a fast, nonvalidating SAX parser.

### WebLogic's FastParser

WebLogic's `FastParser` is a high-performance, nonvalidating SAX parser. It has been designed for processing small to medium-size documents—documents that contain no more than 10,000 elements. This is typical of many web service applications, in

which SOAP and WSDL documents tend to be small. The Streaming API, which we'll cover later, is based on WebLogic's FastParser.

Because the FastParser supports the SAX interface, you can use it instead of WebLogic's default SAX parser, Apache Xerces. By configuring an XML Registry to use the FastParser factory, you ensure that your JAXP code automatically uses the FastParser for processing an XML document.

Clearly, you cannot use the nonvalidating FastParser to validate XML documents.

## Using JAXP-Compliant Parsers

The default parsers supplied with WebLogic are based on Apache's Xerces 2.1.0 and Xalan 2.2 libraries. They provide full support for SAX and DOM parsing, as well as XSL transformers. However, you are not required to use these parsers. For instance, you could upgrade the Xerces parser to a newer version that supports the current W3C XML Schema recommendation.

All parser implementations work on a factory design pattern, as we saw in the examples earlier in this chapter. The JAXP interface hides the actual parser class being used, but it also can be exposed in certain situations. For instance, the print statement in Example 18-2 yields `weblogic.apache.xerces.dom.DOMImplementationImpl` as its output. This indicates that we were using WebLogic's default DOM parser, based on Xerces.

When you configure WebLogic to use a particular parser or transformer implementation, it is important to know the fully qualified class names of the factories that manufacture the parsers or transformers. Table 18-1 lists the class names for WebLogic's default DOM, SAX, and transformer factories.

Table 18-1. Default XML factories for WebLogic Server

XML factory	Class name
DOM parser	<code>weblogic.apache.xerces.jaxp.DocumentBuilderFactoryImpl</code>
SAX parser	<code>weblogic.apache.xerces.jaxp.SAXParserFactoryImpl</code>
XSL Transformer	<code>org.apache.xalan.processor.TransformerFactoryImpl</code>
FastParser	<code>weblogic.xml.babel.jaxp.SAXParserFactoryImpl</code>

In some cases, you may want to bypass the JAXP layer and instantiate a parser directly. For instance, you could instantiate WebLogic's DOM parser directly as follows:

```
import weblogic.apache.xerces.parsers.*;
...
DocumentBuilderFactory dbf = DocumentBuilderFactoryImpl.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
```

However, it also means that your applications are less portable and you need more code changes when you need to change the DOM parser implementation. The

following code snippet shows how you can use the Xerces-derived SAX parser implementation directly:

```
import weblogic.apache.xerces.parsers.*;
...
SAXParserFactory spf = SAXParserFactoryImpl.newInstance();
SAXParser sp = spf.newSAXParser();
```

Ideally, you should use the generic JAXP interfaces to create an XML parser or an XSL transformer. If you want to use another library, you should configure WebLogic separately with the desired parser factories. In fact, the XML Registry allows you to configure this on a per-server basis.

In some situations, you may need to use more than one parser implementation. For example, you may want to use the FastParser for SOAP documents, and a standard SAX parser for other XML processing within your application. There are two ways in which to achieve this:

- Use the XML Registry to target a parser to a particular document, based on its root element (for instance). This allows you to continue using the standard JAXP interface to retrieve parsers and processors.
- Bypass the JAXP interface altogether and directly use the API provided by the parser implementation. This approach means sacrificing application portability.

## The XML Registry

An *XML Registry* provides domain-wide configuration settings for XML parsers and XSL transformers, as well as resolution of different external entities. It is a domain resource that can be administered on a per-server basis using the Administration Console. Once you associate an XML Registry with a server instance, the settings apply to all applications running on that server. A WebLogic domain may define multiple XML Registries, but one XML Registry only may be assigned to a server instance. Of course, the same XML Registry may be shared by multiple servers in the domain. If a server instance doesn't have an XML Registry targeted to it, any server-side application that uses the JAXP interfaces will use the default parsers and XSL transformers shipped with your WebLogic distribution.

An XML Registry allows you to configure alternative parsers and transformers, instead of WebLogic's built-in parsers and transformers.



The XML parsers and XSL transformers you've defined in the registry are available only to server-side applications that use the standard JAXP interface. The XML Registry doesn't affect applications that directly use the API provided by the actual parser implementation.

As we saw in earlier examples, an application that relies on JAXP does not need to use any code specific to the parser or transformer. The XML Registry lets you plug in different parsers and transformers without changing any code. The parser implementation returned by JAXP depends on the following conditions:

- If you have defined application-scoped parsers or transformer factories for an application EAR, WebLogic will use these configuration settings to determine the parser or transformer.
- If an application EAR doesn't have any such application-scoped XML configuration, WebLogic will look for an XML Registry that may be targeted to the server. If it exists, then the following occurs:
  - If the XML Registry defines a parser specific to the XML document being parsed, WebLogic will use this configured value.
  - Otherwise, WebLogic will choose from the default parsers defined in the registry.
- If there is neither an application-scoped configuration nor any XML Registry targeted to the server, WebLogic will use its built-in parsers.

Thus, an XML Registry is a server-specific, domain-wide resource. It determines the actual parser and transformer implementations used by *all* applications running on a server, provided they use the JAXP interface and don't have an application-scoped configuration! An XML Registry consists of the following:

- A list of default factories that will be used to create a parser or transformer.
- A list of external entity resolvers that map external entities to possible local URIs, with options for caching the entities as well.
- A list of XML factories that will be used for particular XML applications. Each such XML document is identified either by its root element or by its public and system identifiers.

An XML Registry acts as a deploy-time parser configuration for server-side applications running on a particular server instance. Document-specific parsers provide a simple yet powerful way to transparently alter the actual parser, without any change to the code.

## Creating an XML Registry

In order to create an XML Registry, open the Administration Console, move to the Services/XML node in the left pane, and then select the “Configure a new XML Registry” option from the right pane. You will need to supply a name for the registry and the fully qualified class names for the SAX and DOM parser factories, as well as an XSL transformer. If any of these fields are left blank, WebLogic's default parsers will be used. As it is, the fields are initialized with the default values for WebLogic's built-in parser and transformer factories. After creating an XML Registry, select the Target

and Deploy tab to associate the registry with particular server instances and make it available.

Suppose you've set `webllogic.xml.babel.jaxp.SAXParserFactoryImpl` as the SAX parser factory for an XML Registry, and the registry is targeted to server A. If no other configuration overrides this setting, any server-side application running on server A that uses JAXP will automatically use the `FastParser` as its SAX parser.

## Configuring Document-Specific Parsers

Once you configure the default parsers for an XML Registry, you can further specify document-specific parser factories. You can configure this by setting up a new Parser Select Registry Entry. This option is available from the Parser Select Entries node under the selected XML Registry. Once again, you will need to specify the fully qualified class names of the XML factories (you can ignore the defunct Parser Class Name field). In addition, you need to associate these XML factories with a specific document. You can specify the document type information in two ways:

- You can supply the public or system identifier that corresponds to a DTD. If a server-side application parses a document that includes a DTD reference with the same public or system ID, it will use the associated parser factories.
- You can supply the name of a root element. Because XML is case-sensitive, be sure to use the correct case for the tag name. If the XML document defines a namespace, include the namespace-prefix for the root element.

Remember, the Parser Select Entries associated with an XML Registry apply only to server-side applications that use the JAXP interface to acquire parser factories. When an application is about to parse a document, WebLogic tries to determine the document type by searching through the first 1000 characters of the document. If it does find a public or system identifier, or a root element that matches one of the parser select entries, WebLogic uses the parser specified for that document type.

This document-based selection of a parser is useful when you want to use parsers that are more optimal for specific document types (e.g., the `FastParser` for SOAP messages). Another benefit of document-specific parsers is that you can override the default XML configuration transparently, without requiring any code changes. However, because WebLogic needs to inspect the document type for any XML document, this feature may carry a small performance penalty.

## Configuring External Entity Resolution

An XML Registry also can define a number of entity resolution mappings. Each mapping associates an external entity with either a local file or contents of a remote URL. It also provides additional cache settings that determine when the external entity is fetched, and the length of time it will be cached. Creating an entity resolution mapping requires a little more effort than defining a document-specific parser. Select an

XML Registry entry and then select the Entity Spec Entries option. You can now configure an entity resolution mapping by mapping a public or system identifier to an entity URI.

The URI specifies the location from which the external entity can be fetched. Its value is either the path to a local copy of the external entity, or a URL that refers to a remote copy. If the entity URI refers to a local file, the path is interpreted relative to the directory associated with the XML Registry: *domainRoot/xml/registries/registryName*, where *registryName* is the name of the registry. You have to create this directory manually. It will stock local copies of files that will be used to resolve external entities configured in the XML Registry.

As an example, let's add external entity resolution to our XML Registry, *MyRegistry*. Start by creating a directory in the domain root called *xml/registries/MyRegistry*. Now create a file, called *ext.txt*, which holds the text `<side><in/></side>`. Next, configure an external entity mapping, with a system identifier of `example`, and specify a URI of *ext.txt*. We have now effectively mapped an external entity with a system identifier of `example` to substitution text contained in the *ext.txt* file. We can test this configuration by creating a server-side application (servlet, JSP, etc.) that parses an XML fragment that includes this entity reference:

```
// Grab the SAX parser using JAXP
SAXParserFactory spf = SAXParserFactory.newInstance();
SAXParser sp = spf.newSAXParser();

sp.parse(new java.io.StringBufferInputStream(
    "<!DOCTYPE outside[ <!ENTITY a SYSTEM \"example\" ]> <outside>&a;</outside>"
), new org.xml.sax.helpers.DefaultHandler() {
    public void startElement (String uri, String lName,
        String qName, Attributes attr){
        System.err.println(qName);}; }
);
```

Here, the SAX handler simply prints the name of each element encountered during the parse. If the external entity is resolved successfully, the resulting XML should be:

```
<outside><side><in/></side></outside>
```

The parse yields the following output, as expected:

```
outside
side
in
```

So, an Entity Spec Entry allows you to map an external entity to a local file that holds the replacement XML. This kind of mapping also is useful for DTD references, which are treated like external entities. For example, you could create another mapping under *MyRegistry* that associates a document type with system identifier *http://oreilly.com/dtds/foo* to a local entity URI */dtds/foo.dtd*.

```
<!DOCTYPE someroot SYSTEM "http://oreilly.com/dtds/foo">
<!-- rest of XML document -->
```

Then, any XML document that includes a DTD reference with the same system ID will resolve the DTD to the local copy held under the */xml/registries/MyRegistry* folder.

## Caching Entities

WebLogic provides a caching facility that improves the performance of external entity resolution. You can configure WebLogic's support for caching by adjusting when the external entity is fetched, and the period after which it is considered stale. The When to Cache field for an Entity Spec Entry determines when an external entity is fetched. If you select an XML Registry from the left pane of the Administration Console and then choose the Configuration tab from the right pane, you can set a value for the When to Cache field. WebLogic permits the following values for this setting:

`cache-on-reference`

This setting ensures that WebLogic caches the item after it has been referenced for the first time while parsing a document.

`cache-at-initialization`

This setting ensures that WebLogic caches the item when the server starts up.

`cache-never`

This setting instructs the server never to cache the item

`defer-to-registry-setting`

This setting instructs the cache to use the value set in the XML Registry's main configuration page.

The When to Cache field can take any one of the first three values explained earlier. By default, the XML Registry is configured to cache an external entity when it is first referenced, and if an entity resolution mapping doesn't override the XML Registry setting, it will inherit the value of its cache setting.

Finally, you can adjust the Cache Timeout Interval setting for an entity resolution mapping, which determines the duration (in seconds) after which the cached entity is considered stale. A subsequent request to a cached entity that has become stale causes WebLogic to fetch the resource from the location specified by the URI. Otherwise, the server will continue to use the cached value of the entity resolution. Although you may specify a timeout interval for each entity mapping, you also can specify a value of -1 for the timeout interval. In this case, the actual timeout will be determined by the value of the Cache Timeout Interval setting associated with the server instance that the XML Registry is targeted to. All entity mappings with a timeout value of -1 will inherit the cache timeout setting for the targeted server itself.

The Cache Timeout Interval server setting can be found by clicking on the server in the left pane of the Administration Console and then selecting the Services/XML node. This setting determines the timeout period for all entity mappings whose

Cache Timeout Interval has a value of -1. Three other settings on this screen are of interest:

#### *XML Registry*

This setting determines the name of the XML Registry targeted to the server instance—you can choose from any one of the XML Registries defined in the domain. Of course, you can change this value by selecting an XML Registry and assigning the server from the Targets panel. Make sure that you do not target more than one XML Registry to the same server.

#### *Cache Memory Size*

WebLogic can cache some of the external entities in memory. This setting specifies how much memory (in kilobytes) to set aside for this cache. It defaults to 500 KB.

#### *Cache Disk Size*

When the memory cache has reached its maximum allotted size, WebLogic persists the little-used external entities to disk. This setting determines the maximum size (in megabytes) for the disk cache. It defaults to 5 MB.

Using these settings, you can specify on a per-server basis the size of the cache for external entities, both in memory and on disk, and when to refresh a cached external entity.

The final option on this screen, Monitor XML Entity Cache, allows you to monitor how the cache is being used. Select this option if you need to access usage statistics such as the total number of cached entries, the frequency of timeouts, and the resource usage.

## **XML Application Scoping**

The XML Application Scoping mechanism in WebLogic allows you to configure XML resources such as parsers, transformers, and entity resolvers on a *per-application* basis. This is different from the XML Registry settings that we covered earlier—they apply to a server instance and all applications running on it. An application-scoped XML configuration has two major benefits:

- It allows you to configure different parsers for different applications. You can covertly change the parsers that will be used by the enterprise application simply by editing a deployment descriptor.
- It makes the resultant EAR file less dependent on the server configuration. If you do not specify an application-scoped factory, the application is at the mercy of the target server. You need to ensure that all servers that will host the enterprise application are configured identically. A scoped XML configuration defined for an application EAR removes this dependence.

To use this mechanism, you have to include an XML deployment descriptor, *weblogic-application.xml*, within the *META-INF* directory of the application EAR file, as shown in Example 18-4. You also can use this descriptor to configure application-specific parameters, JDBC pools, security settings, EJB-wide settings, etc. For now, we focus only on the XML configuration settings.

*Example 18-4. A typical weblogic-application.xml configuration*

```
<weblogic-application>
  <!-- ... rest of weblogic-application ... -->
  <xml>
    <parser-factory>
      <saxparser-factory>
        weblogic.xml.babel.jaxp.SAXParserFactoryImpl
      </saxparser-factory>
      <document-builder-factory>
        weblogic.apache.xerces.jaxp.DocumentBuilderFactoryImpl
      </document-builder-factory>
      <transformer-factory>
        org.apache.xalan.processor.TransformerFactoryImpl
      </transformer-factory>
    </parser-factory>
    <entity-mapping>
      <entity-mapping-name>My Mapping</entity-mapping-name>
      <public-id>--//O'Reilly & Associates//DTD WL//EN</public-id>
      <system-id>http://www.oreilly.com/dtds/wl.dtd</system-id>
      <entity-uri>dtds/wl.dtd</entity-uri>
      <when-to-cache>cache-at-initialization</when-to-cache>
      <cache-timeout-interval>300</cache-timeout-interval>
    </entity-mapping>
  </xml>
</weblogic-application>
```

Example 18-4 shows how the *weblogic-application.xml* descriptor file allows you to set up application-scoped XML resources. The configuration is split into two parts—a parser factory configuration that specifies the default parsers and transformers to use, and multiple entity mappings that associate external entities with local or remote URIs. The entity mappings also include cache settings, which are analogous to the external entity resolution mappings that may be specified for an XML Registry. The easiest way to set up this configuration is to use the WebLogic Builder tool.

## Configuring Factories

The parser factory configuration is quite straightforward. If the parser-factory element declares any XML or transformer factories, these settings will be used to manufacture a parser (or transformer) whenever the application uses JAXP. If you don't specify a factory in the *weblogic-application.xml* descriptor file, the factory in the XML Registry will be used instead. If there is no application-scoped factory setting, nor any corresponding setting in the XML Registry targeted to the server, WebLogic will then use its built-in XML factory.

To specify an XML factory, you need to provide its fully qualified class name. You can define any of the following three subelements:

#### saxparser-factory

This element specifies the factory to use for generating SAX parsers. In Example 18-4, we configured the application to use the WebLogic FastParser.

#### document-builder-factory

This element specifies the factory to use for generating DOM parsers. In Example 18-4, we configured the application to use the default built-in DOM parser factory.

#### transformer-factory

This element specifies the factory to use for generating XSL transformers. In Example 18-4, we configured the application to use the default built-in XSL transformer factory.

Now any server-side component in the application that uses the JAXP interface will automatically use the factory settings declared in the *weblogic-application.xml* descriptor file:

```
SAXParserFactory spf = SAXParserFactory.newInstance();
//Create a parser using the factory
SAXParser sp = spf.newSAXParser();
```

Given the XML configuration in Example 18-4, the preceding code for retrieving a SAX parser will use WebLogic's FastParser.

## Configuring Entity Resolution

As Example 18-4 illustrates, the *weblogic-application.xml* descriptor may also define multiple entity-mapping elements. Each entity-mapping element specifies a name for identification purposes, and includes the following optional subelements:

#### public-id

This element specifies the public identifier of the external entity.

#### system-id

This element specifies the system identifier of the external entity.

#### entity-uri

This element specifies the location of a file that holds the substitution text for the external entity. The file path is relative to the root directory of the EAR. When parsing an XML document, WebLogic uses the *entity-uri* setting to resolve a reference to an external entity with a matching public or system identifier.

Given the configuration in Example 18-4, any request for an external entity matching the specified system or public ID will resolve to the file *dt ds/wl.dtd*, where this path is relative to the root directory of the EAR.

Just like entity resolution mappings in an XML Registry, you can specify the cache settings for an application-scoped entity mapping:

`when-to-cache`

This element determines when the external entity should be cached. It can accept three valid values:

`cache-on-reference`

This setting ensures that WebLogic will cache this entity the first time the entity is referenced. This is the default value for the `when-to-cache` setting.

`cache-at-initialization`

This setting ensures that WebLogic will cache this entity during server initialization.

`cache-never`

This setting guarantees that WebLogic will never cache this entity.

`cache-timeout-interval`

If an item is cached, this setting determines the duration (in seconds) after which the cached entity should be considered stale. After a cached entity becomes stale, the next request for the entity causes WebLogic to retrieve it again from its location. The default value for this setting is 120 seconds.

## WebLogic's Streaming API

WebLogic's Streaming API offers a simple and intuitive way to parse and generate XML data. Compared to the SAX or DOM parsing models, it presents a fundamentally different viewpoint on an XML document. As the name suggests, parsing with the Streaming API is based around a stream. This stream is, in fact, a stream of XML events generated as the XML document is parsed. These events are similar to the events defined in the SAX API because they represent the same fundamental information about the XML data.

BEA is involved in the standardization of the Streaming API for XML (StAX), which has an API very similar to that described here. As a result, you can consider using BEA's implementation of StAX, available from the dev2dev web site.

When an XML document is parsed in SAX mode, the program registers a handler that can listen for SAX events as they occur. The SAX parser then automatically invokes the different callback methods of the event listener. In contrast, a program using the Streaming API pulls events off a stream, whereby each event represents some fundamental information about the XML being parsed. The Streaming API supports events that mark the occurrence of start and end tags, character data, whitespace characters, processing instructions, and several other document characteristics. These parser events enable you to step through the XML document, filter out certain event types, perhaps skip ahead in the document, and stop processing at

any point. Parsing an XML document entails iterating over the stream of events and processing the XML data depending on the type of parse event. Thus, parsing with the Streaming API is demand-driven because you need to explicitly iterate over the stream and extract the events of interest. For this reason, it is often referred to as *pull* parsing.

Assume that `xmlInput` is a string variable holding the XML fragment shown in Example 18-5.

*Example 18-5. Sample XML*

```
<out>
  Hello <inOne> <simple/> </inOne>
  <inTwo> World </inTwo>
</out>
```

The following piece of code illustrates how you can parse the XML data using the Streaming API:

```
import weblogic.xml.stream.*;
//
String xmlInput = /* as in Example 18-5 */;
// Create a stream
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new StringReader(xmlInput));
// Iterate over the stream
while (stream.hasNext())
    // Ask for the next event off the stream
    XMLEvent e = stream.next();
    // Do something with the event
    System.out.print(e.getTypeAsString());
}
stream.close(); // Always close your streams
```

Notice how we create the XML input stream, then iterate over the stream of events and process each event that is encountered. Here, the program simply lists the type of XML event generated during the parse. Example 18-6 lists the output generated as a result of running this program. (Note that we have indented the list for readability purposes.)

*Example 18-6. Events generated while parsing the sample XML*

```
START_DOCUMENT
START_ELEMENT
CHARACTER_DATA
START_ELEMENT SPACE
START_ELEMENT
END_ELEMENT
SPACE
END_ELEMENT
SPACE
START_ELEMENT
```

*Example 18-6. Events generated while parsing the sample XML (continued)*

```
CHARACTER_DATA
END_ELEMENT
END_ELEMENT
END_DOCUMENT
```

You now can see the similarity between the events generated using the Streaming API and SAX events generated by a SAX parser. The Streaming API generates events that indicate the start and end of the document, the start and end of an XML element, whitespace characters, and character data used in a tag's body. Later, we shall look at all the XML event types that can be generated when parsing with the Streaming API.

The Streaming API provides a number of useful enhancements to iterating over a stream of events:

- Instead of handling all events, you can apply a filter to the stream so that only specific event types permeate through.
- The Streaming API also allows you to skip a number of events, or skip until a particular event occurs.

You also can use the Streaming API to generate XML data. In this case, the entire process is reversed. Instead of requesting events from the stream, you create an XML output stream and then write elements to this stream. The Streaming API provides an `ElementFactory` class that manufactures the XML elements that you'll need. When writing to the XML output stream, you need to construct the XML document in a serial fashion. For each element, you need to create the start tag, its attributes, the body that may include other elements, and finally the end tag.

These features make the Streaming API a very useful addition to the current arsenal of SAX and DOM parsers. Each model has its own niche—the Streaming API is best suited when you need to process only a subset of the events generated during the parse. Because the Streaming API relies on WebLogic's `FastParser`, you cannot use it to validate an XML document. Remember, the Streaming API is proprietary and not yet supported by the JAXP interface. This means you cannot use the JAXP interface to create a streaming parser. Therefore, neither the XML Registry nor the application-scoped parser factories can impact how you use the Streaming API.

## Creating a Stream

With the Streaming API, the parsing occurs implicitly as you iterate through the stream of events. So, you don't even explicitly create a streaming parser; instead, you create an `XMLInputStream` instance, which then acts as source of parse events:

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(someSource);
```

This stream can be created from a number of different sources, including the following:

`java.io.File`, `java.io.InputStream`, or `java.io.Reader`

The source XML data is read from a file, byte stream, or character-based reader (as we saw in the earlier example).

`org.w3c.dom.Document` or `org.w3c.dom.Node`

The source XML data is read from a DOM tree, perhaps created previously by a DOM parser.

`XMLInputStream`

It may seem odd that the source for a stream may be a stream itself, but as we shall see later in this chapter, the Streaming API allows you to snap off a substream to be used for a different parse. The substream can be created by calling the `getSubStream()` method on an existing stream.

Once you create a stream, you can start parsing the XML data by iterating over the stream. The `XMLInputStream` provides the familiar iterator pattern for retrieving the next parse event:

```
while(stream.hasNext()) {
    XMLEvent event = stream.next();
    // Do something with the event
}
```

## Events

As an XML document is parsed, the `XMLInputStream` object generates a stream of parser events. A typical handler will determine the event's type and then process the event accordingly. As we have already seen, the Streaming API supports a number of different types of events. In fact, the Streaming API provides an interface that corresponds to each event type, and all these interfaces extend the `XMLEvent` interface (one way or the other). Table 18-2 provides a complete list of `XMLEvent` subinterfaces provided by the Streaming API.

Table 18-2. `XMLEvent` subinterfaces

XMLEvent subclass	Description	Constant identifier
<code>StartDocument</code>	Indicates the start of an XML document	<code>START_DOCUMENT</code>
<code>EndDocument</code>	Indicates the end of an XML document	<code>END_DOCUMENT</code>
<code>StartElement</code>	Indicates the start tag for an element has been encountered	<code>START_ELEMENT</code>
<code>EndElement</code>	Indicates the end tag for an element has been encountered	<code>END_ELEMENT</code>
<code>CharacterData</code>	Indicates character data from the body of an element has been encountered	<code>CHARACTER_DATA</code>
<code>Space</code>	Indicates whitespace characters have been encountered	<code>SPACE</code>

Table 18-2. XMLEvent subinterfaces (continued)

XMLEvent subclass	Description	Constant identifier
Comment	Indicates an XML comment has been encountered	COMMENT
ProcessingInstruction	Indicates an XML processing instruction has been encountered	PROCESSING_INSTRUCTION
StartPrefixMapping	Indicates prefix mapping has started its scope (triggered before the StartElement event)	START_PREFIX_MAPPING
EndPrefixMapping	Indicates prefix mapping has ended its scope (triggered after the EndElement event)	END_PREFIX_MAPPING
ChangePrefixMapping	Indicates transition from one prefix mapping to another	CHANGE_PREFIX_MAPPING
EntityReference	Indicates an entity reference has been encountered	ENTITY_REFERENCE

The Constant Identifier column indicates the name of an integer constant in the XMLEvent interface that identifies each event type. The `getType()` method on an XMLEvent object returns a value that matches one of these constants.

The XMLEvent interface also defines an `is<XMLEvent>()` method for each of the XML events, which allows you to identify the actual event subclass. Once you determine its type, you can cast the XMLEvent object to the correct subinterface and process the event accordingly. For example, when a new namespace prefix is introduced, the XML stream returns a `StartPrefixMapping` event. It provides various methods for accessing the namespace data—e.g., the `getNamespaceUri()` and `getPrefix()` methods. Similarly, a `StartElement` instance has methods for accessing the attributes of an element:

```
//alternative check:
// if (event.getType() == XMLEvent.START_ELEMENT) {
// ...
// }
if (event.isStartElement()) {
    StartElement startElement = (StartElement) event;
    AttributeIterator attributes = startElement.getAttributesAndNamespaces();
    while(attributes.hasNext()){
        Attribute attribute = attributes.next();
        System.out.print("Name of attr: " + attribute.getName().getQualifiedName());
        System.out.print("Value of attr: " + attribute.getValue());
    }
}
```

The preceding piece of code illustrates how you can access an element's attributes once you have encountered its start tag.

## Filtering a Stream

At this point, we know how to create an XML stream from a document, step over the stream of events, and provide custom handling depending on the event's type. The

Streaming API also enables you to filter an XML stream so that only the events of interest are pulled from the stream. This means that when you do iterate over a filtered stream, you need to deal with only those events that have passed the filter. WebLogic's Streaming API allows you to register your interest in several ways. You can apply a filter based on an event's type, a subset of the elements, or the URI/type of a namespace. You even can apply a custom filter, whereby you decide which XML events will pass through the filter.

Earlier in this section, we saw an example of how you can use the `newInputStream()` methods on an `XMLInputStreamFactory` instance to create a stream from an XML document. The Streaming API supports an alternative two-argument version of the same methods. In this case, you use the second parameter to supply a filter. It could be a custom filter you have created or one of the default filters provided with WebLogic Server. The default filters are available in the `weblogic.xml.stream.util` package. Each of the `TypeFilter`, `NameFilter`, `NameSpaceFilter`, and `NamespaceTypeFilter` classes implement the `ElementFilter` interface.

The `TypeFilter` class takes a bit mask of all the event types that you want to let through. For instance, if you need to retrieve only the character and whitespace data in a document, you could apply the type filter as follows:

```
XMLInputStream stream =
    factory.newInputStream(someSource,
        new TypeFilter(XMLEvent.CHARACTER_DATA | XMLEvent.SPACE));
```

Now when you iterate over the XML stream, you'll encounter only events for whitespace and character data:

```
while (stream.hasNext()) {
    XMLEvent e = stream.next ();
    switch (e.getType()) {
        case XMLEvent.SPACE: //Handle whitespace here and break
        case XMLEvent.CHARACTER_DATA: //Handle character data here and break
        default: // You will never reach here
    }
}
```

The `NameFilter` class filters the stream based on the name of the element. For example, if you need to deal with only `inOne` elements in the XML data, you would apply a name filter as follows:

```
XMLInputStream stream =
    factory.newInputStream(someSource, new NameFilter("inOne"));
```

The `NameSpaceFilter` class lets you filter a stream based on the URI of a namespace, while the `NamespaceTypeFilter` class allows you to filter on both the namespace and type of an element. For example, if you want to retrieve only the XSLT start elements of an XSLT document, you would create a `NamespaceTypeFilter` as follows:

```
XMLInputStream stream = factory.newInputStream(someSource,
    new NamespaceTypeFilter ("http://www.w3.org/1999/XSL/Transform",
        XMLEvent.START_ELEMENT));
```

## Custom Filters

Custom filters can be applied to an XML stream in the same way as built-in filters. In order to create a custom filter, you need to register an instance of a class that implements the `ElementFilter` interface with the XML stream:

```
package weblogic.xml.stream;
public interface ElementFilter {
    public boolean accept(XMLElement event);
}
```

A custom filter needs to implement the `accept()` method, which determines whether an incoming `XMLEvent` can be let through.

Example 18-7 shows how to implement a custom filter that wraps multiple filters. The wrapping filter accepts an element if it is accepted by any one of its component filters.

*Example 18-7. A custom filter*

```
package com.oreilly.weblogic.xml.filters;

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

public class OrFilter implements ElementFilter {
    protected ElementFilter [] filters;

    public OrFilter(ElementFilter [] filters) {
        this.filters = filters;
    }
    public setFilters(ElementFilter [] filters) {
        this.filters = filters;
    }

    // Only permit elements that pass any of the filters
    public boolean accept(XMLEvent e) {
        for (int i=0; i<filters.length; i++)
            if (filters[i].accept(e))
                return true;
        return false;
    }
}
```

We then can apply an `OrFilter` to an XML stream so that only elements with the name `a` or `b` are let through:

```
ElementFilter myFilter = new OrFilter(
    new ElementFilter[] {new NameFilter("a"), new NameFilter("b")});
XMLInputStream stream = factory.newInputStream(someSource, myFilter);
```

## Positioning the Stream

The ability to skip ahead while iterating over a stream of parse events is a powerful feature of the Streaming API. You can skip ahead by *n* events or until a particular element has been encountered. Later in this chapter, we will examine how you can use this feature in conjunction with a buffered XML stream, whereby you can mark a position within the stream and later reset the stream to an earlier mark.

WebLogic provides three mechanisms for skipping within a stream. All of these methods are invoked on an `XMLInputStream` instance:

`skip()` and `skip(int)`

These methods allow you to skip ahead by one or a specified number of events. It does not matter what type of events are present in the stream. This method will just skip over as many events as you specify.

`skip(XMLName)` and `skip(XMLName, int)`

These methods allow you to skip ahead to an event with the specified name, or an event with the specified name and type. For instance, you could skip to the next end tag for element `b` as follows:

```
skip(ElementFactory.createXMLName("b"), XMLEvent.END_ELEMENT)
```

`skipElement()`

This method simply skips over the start/end tag pair for the next element, avoiding all its subelements. If the current XML element contains no subelements, the method skips ahead past its end tag. For instance, if you create an XML stream using the following XML fragment "`<c> foo </c><a> <b/>bar </a>`", you can expect the following behavior:

- If you invoke the `skipElement()` method just after processing the `StartElement` event for `a`, the stream will skip past the space and element `b`, and the next event will be the `CharacterData` event marking `bar`.
- If you invoke the `skipElement()` method just after processing the `StartElement` event for `c`, the stream will skip past the character data, and the next event will be the `StartElement` event marking the start tag for `a`.

An XML stream supports an additional `peek()` method, which allows you to look ahead at the next event. Because this method returns the next `XMLEvent`, you then can make a decision based on the event's type or any information that you can extract from it.

## Substreams

At any point while you are stepping over an existing XML stream, you can invoke the `getSubStream()` method on the `XMLInputStream` instance to return a copy of the next element and all of its subelements. The new XML substream will generate all XML events between (and including) the start/end tag pair for the next element. The parent stream remains unaltered and you can continue to iterate over the existing stream

as before. If you want to step over the element that generated the substream, you can invoke the `skipElement()` method. The `getSubStream()` method returns a new `XMLInputStream` instance, which now can be used as the basis for parsing the next element and all its subelements. This substream extends over all XML events and stops only after it encounters an `EndElement` event that matches the initial `StartElement` event for the substream.

## Buffered Streams

A buffered input stream can be created by wrapping an `XMLInputStream` instance by a `BufferedXMLInputStream` instance. When using a buffered XML stream, you can mark a particular position in the stream and later reset the stream back to the marked spot. Effectively, this feature allows you to reparse the stream—this is very useful in situations in which you need to process the same XML fragment more than once.

The following code sample shows how you can mark a position within a stream, process that stream, and later reset it back to the earlier position:

```
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
BufferedXMLInputStream bstream =
    factory.newBufferedInputStream(factory.newInputStream(someSource));
skip(4); // go somewhere
// mark the current position
bstream.mark();
// perform some work on the stream
workOne(bstream);
// go back to our mark
bstream.reset();
// perform some more work on the stream
workTwo(bstream);
```

## Creating Output Streams

WebLogic's Streaming API also enables you to generate XML documents on the fly. Here you need to create an XML output stream and send various XML events to this stream. The Streaming API has an `ElementFactory` class, which provides factory methods for the various elements of an XML document: character data, comments, attributes, start and end tags, etc. Because XML data is structured hierarchically, you will be sending a linear stream of events based on a flattened representation of the XML. This means that for each element you need to construct a start tag, add any attributes, then build its body (which may include other elements) and finally its end tag.

The following code shows how to create an XML output stream, write XML data to the stream, and finally flush the contents of the stream:

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

```
// ...
output.add(ElementFactory.createCharacterData("avi"));
// ...
output.flush();
output.close();
```

You can construct an output stream from a number of different sinks:

`java.io.OutputStream` and `java.io.Writer`

The XML data is written to the binary stream or character writer, as you would expect. In the earlier example, the XML output stream wraps the writer `System.out`, so the XML data is written to the console screen.

`org.xml.sax.ContentHandler`

The output is written to a SAX content handler, which eventually generates a stream of SAX events.

`org.w3c.dom.Document`

The XML output is written to a DOM document. The following code snippet shows how to create an XML output stream that wraps a DOM tree:

```
XMLOutputSteamFactory factory = XMLOutputSteamFactory.newInstance();
Document doc =
    DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
XMLOutputSteam output = factory.newOutputSteam(doc);
```

Do not forget to use the `flush()` method on the output stream. Only then will the current contents of the XML output stream be written to the actual sink. For instance, if you are writing to a DOM document, you must flush the output stream before you start manipulating the DOM tree. If you don't flush the XML stream, you may have to suffer the consequences of a partially constructed DOM!

## Writing to the Stream

The XML output stream provides various `add()` methods that enable you to generate XML data and write various elements to the stream. All elements are created via factory methods provided by the `ElementFactory` class:

```
output.add(ElementFactory.createStartElement("myelement"));
output.add(ElementFactory.createAttribute("a", "1"));
output.add(ElementFactory.createCharacterData("Hello World"));
output.add(ElementFactory.createEndElement("myelement"));
```

The preceding code sample generates the following XML data:

```
<myelement a='1'>Hello World</myelement>
```

The output stream is nonvalidating, so you need to guarantee that the XML document is well-formed. For instance, you need to ensure that all elements are properly nested and that each start tag for an element has a matching end tag.

In fact, the `add()` method supports adding four types of objects: plain markup, elements, attributes, and an `XMLInputStream`. You can supply an `XMLInputStream` to the

add() method—this allows you to easily insert XML data from another source. Recall an XMLInputStream instance can wrap several different sources: a file, character reader, or DOM tree. The next example shows how you can insert XML data that has been parsed from a file:

```
output.add(ElementFactory.createStartElement("example"));
XMLInputStreamFactory factory = XMLInputStreamFactory.newInstance();
XMLInputStream stream = factory.newInputStream(new FileInputStream(somefile));
output.add(stream);
output.add(ElementFactory.createEndElement("example"));
```

As you can see, writing to the XML output stream is quite straightforward. The ElementFactory class can manufacture all of the obvious elements in an XML document: start and end tags, attributes, character data, processing instructions, etc. It is quite cumbersome having to use the XMLOutputStream to write XML data. You need to generate the hierarchical XML data in a serial fashion, much like how the XML input stream delivers its XML events during a parse. This is quite *unlike* the more natural approach of constructing a DOM tree.

## WebLogic's XPath API

WebLogic XPath API lets you match XPath expressions against an XML document, represented either as a DOM tree or an XMLInputStream. The API is contained in the weblogic.xml.xpath.\* package. Though the XPath syntax and semantics follow the W3C standard, there is still no standard Java API for manipulating XPath expressions, so this is a welcome API.

### Using the API with a DOM

To use the API, construct DOMXPath instances representing your XPath. Here are a few examples:

```
DOMXPath threeDebt = new DOMXPath("sum(//co[@type='3']/debt)");
DOMXPath haveFirst = new DOMXPath("count(//co[@type='1']) > 0");
DOMXPath biggestDebt = new DOMXPath("//co[debt > 1000]/debt");
```

We will use the following XML as test input:

```
<world>
  <co type='1'><debt>100</debt></co>
  <co type='3'><debt>1000</debt></co>
  <co type='3'><debt>2000</debt></co>
</world>
```

You can match the XPath expressions against either a DOM node or the entire document itself. Use a node if you intend to just search a portion of the document. To evaluate the XPath expression, you must invoke one of the following evaluate methods on the DOMXPath instance:

```
public boolean evaluateAsBoolean(Document|Node);
public java.util.Set evaluateAsNodeset(Document|Node);
```

```
public double evaluateAsNumber(Document|Node);
public String evaluateAsString(Document|Node);
```

The actual evaluate method you invoke will depend on the type of the expected result(s) of your XPath expression. For example, we would invoke the earlier XPath expressions as follows:

```
/** evaluate the total third-world debt */
double e_debt = threeDebt.evaluateAsNumber(doc);
/** evaluate if there are any developed countries */
boolean e_haveFirst = haveFirst.evaluateAsBoolean(doc);
/** find the countries with a serious debt problem */
Set biggest = biggestDebt.evaluateAsNodeSet(doc);
```

The `evaluateAsNodeSet()` method returns a set of `org.w3c.dom.Node` objects that match the XPath expression, so, you could iterate over the set of nodes as follows:

```
if (biggest != null) {
    Iterator i = biggest.iterator();
    while (i.hasNext()) {
        Node n = (Node)i.next();
        // Process the node
    }
}
```

The result of running this little test will indicate that the total third-world debt (`e_debt`) is 3000, that it is true that there is a first-world country, and that there is one member in the set of countries with a major debt problem.

## Using the API with a Stream

Matching XPath expressions against an XML document that is read as an XML stream is just a little more involved. You would expect this because the document that you are matching against is read only incrementally. Thus, instead of waiting for the XPath expression to complete its evaluation, the API lets you register a set of “observers” before the XML stream is processed. These observers then are invoked whenever an XPath match is found as you incrementally parse the XML stream, in the order in which the observers were assigned. Because you’re going to match an XPath expression against an XML stream, you need to create a `StreamXPath` instance (instead of a `DOMXPath` object) to represent your XPath expression:

```
StreamXPath pops = new StreamXPath ("//co/pop");
```

Now you must create an `XPathStreamFactory` instance and register one or more observers that are invoked whenever something within the XML stream matches the XPath expression:

```
XPathStreamFactory factory = new XPathStreamFactory();
factory.install(pops,
    new XPathStreamObserver () {
        public void observe(XMLEvent event) {
            System.out.println("Population event matched: "+event);
        }
    });
```

```

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

```

In this case, we've registered an observer that responds to any XML events that match the XPath expression. This means that as we pull events of the XML stream, WebLogic automatically calls the appropriate `observe()` methods whenever an XML event matching the XPath expression `//co/pop` is found. Remember, if you configure a `StreamXPath` instance with multiple observers, they will be called (during the parse) in the order in which they were installed.

Once you've installed the observers with the XPath expression, you then can initiate the evaluation. To achieve this, you need to simply use the `XPathStreamFactory` instance to construct an XML stream that can trigger the XPath observers, whenever you pull XML events that may match the XPath expression. In fact, the `XPathStreamFactory` instance provides the `createStream()` method, which accepts a single parameter: an `XMLInputStream` or an `XMLOutputStream` object. This method then returns an XML stream that can match the events in the source stream with the installed XPath observers. The following example shows how to enable XPath matching on an XML stream:

```

//src represents the location of an XML document
XMLInputStream sourceStream =
    XMLInputStreamFactory.newInstance().newInputStream(new File(src));
XMLInputStream matchingStream =
    factory.createStream(sourceStream);

```

Now when you iterate over this XML stream, you get the same XML events as if you had iterated over the source stream, but in addition, the events are matched against the configured observers and the appropriate `observe()` methods are invoked. The following example shows how you would evaluate the XPath expression against the source stream:

```

while(matchingStream.hasNext()) {
    XMLEvent event = matchingStream.next();
    // Do nothing if you are only interested in the XPath observations
}

```

As you can see, we simply pull XML events from the stream—the configured observers are executed transparently during this iteration. Note that because the Streaming API observes a set of events, a node such as `debt` will have two events associated with it (i.e., the start and end events). Both of these events will match the XPath expression, and both will trigger the appropriate `observe()` methods.

One of the disadvantages of using the Streaming XPath API is that you cannot use the full XPath expression syntax. For instance, if you attempt to match the following XPath expression against an XML stream, you will surely encounter a `weblogic.xml.xpath.XPathUnsupportedException`:

```

StreamXPath cost = new StreamXPath ("//co[debt > 1000]/debt");

```

The reason should be obvious. Since the Streaming XPath API iterates over a stream of XML events without any “look ahead” capability, it cannot possibly match against child elements. For this reason, you cannot use the *child* axis in the XPath predicate. The same reasoning can be applied to several other XPath expressions, leading to the following natural limitations:

- You may not use the XPath functions `last()`, `size()`, `id()`, `lang()`, and `count()` when constructing a `StreamXPath` object. In addition, the `string()` function is not fully supported because the string value of a node with offspring depends on its child nodes.
- The Streaming XPath API doesn’t support XPath predicates that use the following axes: `self`, `child`, `descendant`, `descendant-or-self`, `following`, `following-sibling`, `attribute`, and `namespace`.

## Miscellaneous Extensions

WebLogic provides a number of miscellaneous extensions that ease the processing of XML data. For instance, WebLogic extends the standard SAX input source. Instances of the `weblogic.xml.sax.XMLInputSource` class enable you to retrieve document header information such as the name of the root element or the public and system identifiers:

```
package weblogic.xml.sax;
public class XMLInputSource extends org.xml.sax.InputSource {
    public String getNamespaceURI();
    public String getPublicId();
    public String getRootTag();
    public String getSystemId();
    //...
}
```

This is useful when you need to decide how to process the XML data without having to complete a lengthy parse of a potentially large document. The following code snippet shows how to retrieve the root tag of an incoming XML document:

```
XMLInputSource xis =
    new XMLInputSource(new java.io.StringBufferInputStream("<theroot></theroot>"));
System.err.println(xis.getRootTag());
```

In addition, WebLogic provides a JMS extension whereby XML messages may be filtered on the value of an XPath expression. We covered this in Chapter 8.

## Parsing XML in a Servlet

WebLogic provides a proprietary but convenient way for parsing the message body of an HTTP POST request made to a servlet. It allows you to use the `setAttribute` and `getAttribute` methods on the `HttpServletRequest` object to parse XML documents. However, it is not a feature supported by other J2EE-compliant servlet

engines. WebLogic provides two special-purpose attributes in the request object. When you retrieve the value of the particular attribute, instead of returning its value, WebLogic parses the message body of the HTTP request.

WebLogic automatically uses the JAXP interface to create the appropriate parser, and lets the parser run through the contents of the body of the HTTP POST. To use the DOM parser, you need to retrieve the value of the `org.w3c.dom.Document` attribute from the HTTP request object and cast the return value to `Document`:

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

To use the SAX parser, you need to assign a SAX handler instance as the value of the `org.xml.sax.helpers.DefaultHandler` request attribute. WebLogic then will automatically parse the message body of the HTTP POST request using the SAX handler you've supplied:

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

Example 18-8 illustrates how a simple servlet can parse the message body of an incoming POST request. In response, the servlet outputs the names of all the start elements found in the XML document.

*Example 18-8. Parsing the body of a POST request*

```
import weblogic.servlet.XMLProcessingException;
import org.xml.sax.helpers.DefaultHandler;
// ...
public void doPost(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    try {
        final PrintWriter out = response.getWriter();
        DefaultHandler printName = new DefaultHandler() {
            public void startElement (String uri, String lName, String qName,
                                     Attributes attr){
                out.println(qName);};
        };
        request.setAttribute("org.xml.sax.helpers.DefaultHandler",
                             printName);

        } catch(XMLProcessingException ex) {
            ex.printStackTrace();
        }
    }
}
```

A Java client may send a POST request to the servlet, using an instance of the `java.net.URLConnection` class. Example 18-9 shows how a program can send XML data to an HTTP servlet. Once the client has established a connection with the servlet and delivered the XML data, it prints the response data returned from the servlet.

Example 18-9. Sending data to the XML servlet

```
// Connect to our servlet
URL u = new URL("http://10.0.10.10:7061/B");
URLConnection uc = (URLConnection) u.openConnection();
uc.setRequestProperty("Content-Type", "text/xml");
uc.setDoOutput(true);
uc.setDoInput(true);
// Send some XML
PrintWriter pw = new PrintWriter(uc.getOutputStream());
pw.print("<fine>I am <dandy/>You are</fine>");
pw.close();
// Read the result
BufferedReader in =
    new BufferedReader(new InputStreamReader(uc.getInputStream()));
String inputLine;
while ((inputLine = in.readLine()) != null) {
    System.out.println(inputLine);
}
in.close();
```

Running this code will yield the following output in the console window:

```
fine
dandy
```

## Using the JSP Tag Library for XSL Transformations

WebLogic supplies a JSP tag library to help with XSL Transformations from within a JSP page. You can make the tag library available to a web application by placing the tag library *xmlx-tags.jar* under the *WEB-INF/lib* folder of the web application. Before this, however, you need to extract the tag library JAR from the *WL\_HOME\server\ext\xmlx.zip* file. Finally, you need to specify a *taglib* element in the standard *web.xml* descriptor file to make the tag library visible to the web application:

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

Given this declaration, you now can reference the tag library from within a JSP page as follows:

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

The tag library defines the main *xslt* tag, and two tags that can be used within its body: the *xml* and *stylesheet* tags. The *xslt* tag can be used in several different ways:

- The XML data can be supplied within the body of the *xml* tag.
- You can reference multiple XSLT stylesheets and, at runtime, determine which one is used for the transformation.
- If the JSP uses an empty *xslt* tag, the XML data is grabbed from the URL used to access the JSP.

In all these cases, the JSP tag automatically uses the JAXP interface to acquire an XSL transformer and then apply the correct stylesheet to the XML data.

## Using an interceptor

You can set up a mechanism whereby a request for an XML file automatically triggers an XSLT conversion, and the result of the transformation is then returned back to the client. In this case, the XML file references the XSLT stylesheet that ought to be used for the transformation. The trick is to map a servlet (or JSP) to a URL pattern so that the servlet (or JSP) can intercept the requests for the XML file and run the transformation. In this case, we'll use a JSP to intercept requests for the XML file(s). The JSP page intercepts the HTTP request. A simple JSP tag then automatically passes the requested file through an XSLT processor and sends the output of the transformation back to the client.

The JSP page looks quite simple—it merely includes an empty `xslt` tag. The following code snippet lists the source for the *interceptor.jsp* file:

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

Next, you need to register this JSP as a servlet and map it to a URL pattern—say, */xslt/\**. Do this by modifying the standard *web.xml* descriptor for the web application:

```
<!-- web.xml entry -->
<servlet>
  <servlet-name>interceptor</servlet-name>
  <jsp-file>interceptor.jsp</jsp-file>
</servlet>
<servlet-mapping>
  <servlet-name>interceptor</servlet-name>
  <url-pattern>/xslt/*</url-pattern>
</servlet-mapping>
```

This will ensure that all requests that match the URL pattern */xslt/\** are now delegated to the interceptor JSP. Because the *interceptor.jsp* contains an `xslt` tag with an empty body, the tag implementation will fetch the contents of the resource automatically. The location of the resource is determined implicitly by the rest of the URI that follows the servlet path. The JSP tag will read the XML data from the resource and then use the referenced XSLT stylesheet to execute the transformation, before returning the output back to the client. Now, suppose the web application contains an XML file *test.xml*:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="test.xsl"?>
<a>Hello World!</a>
```

Here, the XML file includes a reference to the XSLT stylesheet *test.xsl*:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

```

<xsl:template match="a">
  <html><h1>
    <xsl:value-of select="."/>
  </h1></html>
</xsl:template>
</xsl:stylesheet>

```

If you place both *test.xml* and *test.xsl* under the root of the web application, a request for the URL *http://server:port/webapp/xsl/test.xml* will be redirected to the interceptor JSP automatically. The JSP will invoke the `xslt` tag, which will fetch the resource using the URI that follows the servlet path (i.e., */test.xml*) and send this file through the configured XSLT processor. The HTML generated as a result of the transformation then will be returned to the client:

```

<html><h1>
  <a>Hello World!</a>
</h1></html>

```

Of course, you can reduce the cost of XSLT processing by relying on WebLogic's support for caching, via either the JSP cache tags or the cache filters.

### Using the XSLT tag

The XSLT tag also can be used in a more traditional manner—i.e., in nonintercept mode. The following code sample shows how to execute a transformation when the URIs for the XML document and the XSLT stylesheet are attributes of the `xslt` tag:

```

<%@ taglib uri="xmlx.tld" prefix="x"%>
<x:xslt xml="test.xml" stylesheet="test.xsl"/>

```

In this case, the result of the XSL transformation also will be the output of the `xslt` tag. You can specify a number of subelements within the body of the `xslt` tag:

- You can specify the source XML data within the body of an `xml` tag. If the `xslt` tag doesn't define an `xml` attribute or include an `xml` tag, the XML data is grabbed from the URI that follows the servlet path, as described earlier.
- You can specify one or more stylesheet elements, whereby each tag declares a `media` attribute and a `uri` attribute that references an XSLT stylesheet. You also can define the XSL templates within the body of the stylesheet tags.

If the `xslt` tag declares a `media` attribute, its value determines which one of the stylesheets is eventually used. Example 18-10 illustrates how to use the `media` attribute to select from one of the stylesheets. Here, the JSP decides based on whether the output is meant for an HTML browser or a WAP browser (which supports WML).

*Example 18-10. Using the xslt tag*

```

<%@ taglib uri="xmlx.tld" prefix="x"%>
<%
  String mediaType = "html"; // Usually dynamically computed
  String content = "<a>Hello World!</a>";

```

Example 18-10. Using the `xslt` tag (continued)

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

In this case, the JSP generates the same output as *interceptor.jsp*:

```
<html><h1>  
  <a>Hello World!</a>  
</h1></html>
```

The `media` attribute acts as a selector variable for stylesheets listed in the body of the `xslt` tag. This way, the JSP can decide at runtime which stylesheet should be used for the XSL transformation.