StAX'ing up XML, Part 1: An introduction to Streaming API for XML (StAX)

First, explore its cursor-based API that pulls XML as a token (or event) stream

Peter Nehrer
November 29, 2006

The Streaming API for XML (StAX) is the latest standard for processing XML in the Java™ language. As a stream-oriented approach, it often proves a better alternative to other methods, such as DOM and SAX, both in terms of performance and usability. This article, the first in a three part series, provides an overview of StAX and describes its cursor-based API for processing XML.

View more content in this series

StAX overview

Since its inception, the Java API for XML Processing (JAXP) provided two methods for processing XML -- the Document Object Model (DOM) method, which uses a standard object model to represent XML documents, and the Simple API for XML (SAX) method, which uses application-supplied event handlers to process XML. A streaming alternative to these approaches was proposed in JSR-173: Streaming API for XML (StAX). Its final release was published in March 2004 and it became part of JAXP 1.4 (to be included in the upcoming Java 6 release).

As its name reveals, StAX places emphasis on streaming. In fact, what distinguishes StAX from other approaches is the application's ability to process XML as a stream of events. The idea of handling XML as a set of events is not entirely new (in fact, it is already present in SAX); however, the difference is that StAX allows the application code to pull these events one after another, rather than having to provide a handler that receives events from the parser at the parser's convenience.

StAX actually consists of two sets of XML processing API, each providing a different level of abstraction. The cursor-based API allows the application to work with XML as a stream of tokens (or events); the application can examine the parser's state and obtain information about the last parsed token, then advance to the next token, and so on. This is a rather low-level API; while considerably efficient, it does not provide an abstraction of the underlying XML structure. The higher-level iterator-based API allows the application to process XML as a series of event objects,
each of which communicates a piece of the XML structure to the application. All the application needs to do is determine the type of the parsed event, cast it to the corresponding concrete type, and use its methods to get information pertaining to the event.

The basics

In order to use either API, the application must first obtain a concrete XMLInputFactory. In the classic JAXP style, this is done using the Abstract Factory pattern; the XMLInputFactory class provides static newInstance methods, which are responsible for locating and instantiating a concrete factory. To configure this instance, you can set custom or pre-defined properties (whose names are defined in class XMLInputFactory). Finally, to use the cursor-based API, the application obtains an XMLStreamReader by calling one of the createXMLStreamReader methods. Alternatively, to use the event iterator-based API, the application calls one of the createXMLEventReader methods to obtain an XMLEventReader (see Listing 1).

Listing 1. Obtaining and configuring the default XMLInputFactory

```java
// get the default factory instance
XMLInputFactory factory = XMLInputFactory.newInstance();
// configure it to create readers that coalesce adjacent character sections
factory.setProperty(XMLInputFactory.IS_COALESCING, Boolean.TRUE);
XMLStreamReader r = factory.createXMLStreamReader(input);
// ...
```

Both XMLStreamReader and XMLEventReader allow the application to iterate over the underlying XML stream on its own. The difference between the two approaches lies in how they expose pieces of the parsed XML InfoSet. The XMLStreamReader acts as a cursor that points just beyond the most recently parsed XML token and provides methods for obtaining more information about it. This approach is very memory-efficient as it does not create any new objects. However, business application developers might find XMLEventReader slightly more intuitive because it is actually a standard Java Iterator that turns the XML into a stream of event objects. Each event object in turn encapsulates information pertaining to the particular XML structure it represents. Part 2 of this series will provide a detailed description of the event iterator-based API.

As to which API style to use depends on the situation. The event iterator-based API represents a more object-oriented approach than the cursor-based API. As such, it is easier to apply in modular architectures, because the current parser state is reflected in the event object; thus, an application component does not need access to the parser/reader while processing the event. Furthermore, it is possible to create an XMLEventReader from an XMLStreamReader using XMLInputFactory's createXMLEventReader(XMLStreamReader) method.

StAX also defines a serialization API, a feature that has been sorely missing in Java's standard XML processing support. Like its parsing counterpart, it is also a streaming API that comes in two flavors -- the lower-level XMLStreamWriter that works with tokens, and the higher-level XMLEventWriter that works with event objects. XMLStreamWriter provides methods for writing individual XML tokens (such as opening and closing tags or element attributes) without checking their well-formedness. XMLEventWriter, on the other hand, allows the application to add full XML event objects to the output. In Part 3 you will explore the StAX serialization API in detail.
Why use StAX?

Before you commit to learning a new XML processing API, you might wonder if it is worth the trouble. In fact, the pull-based approach employed by StAX gives it several important advantages over other methods. First, regardless of the API style used, it is the application that calls the reader (parser), not the other way around. By retaining control of the parsing process, you can simplify the calling code to handle precisely the content it expects, and to choose to simply stop parsing when it encounters something unexpected. Furthermore, because this method is not based on handler callbacks, the application does not need to maintain a simulated parser state like it might need to when using SAX.

StAX also retains the benefits that SAX provides over DOM. By shifting focus from a resulting object model to the parsed stream itself, applications gain the ability to process theoretically infinite XML streams, since events are inherently transient and do not need to accumulate in memory. This is of particular importance to a class of applications that use XML as a messaging protocol rather than to represent document content, such as Web Services or Instant Messaging applications. For example, it is of little use to a Web Service router servlet to be handed a DOM if all it does is translate it to an application-specific object model and then simply discard it. Using StAX to go straight to the application model is more efficient. For an Extensible Messaging and Presence Protocol (XMPP) client, using DOM is plain impossible - an XMPP client/server stream is incrementally generated in real time from user-entered messages. To wait for the stream's closing tag (in order to finalize building the DOM) means waiting until the conversation ends. By processing XML as a series of events, the application can react to each event in a manner that is most appropriate (for example, display the incoming instant message and so on.)

Due to its bi-directional nature, StAX also supports chained processing very well, especially at the event level. The ability to accept events (from whichever source) is encapsulated in interface XMLEventConsumer, which the XMLEventWriter extends. Thus, you can write the application modularly to read XML events from XMLEventReader (which is also a plain Iterator and can be treated as such), process them, and pass them on to an event consumer (which can then further extend the processing chain if need be). As you will learn in Part 2, you can also customize the XMLEventReader by using an application-supplied filter (a class implementing the EventFilter interface) or by decorating an existing XMLEventReader using EventReaderDelegate.

To put it all in perspective, StAX brings the application closer to the underlying XML than either DOM or SAX. By using StAX, not only can the application build up the object model it needs (rather than having to deal with the standard DOM), it can do so at its own convenience, rather than only after getting a call-back from the parser.

The next section delves into the details of the cursor-based API and how to use it to efficiently process XML streams.

The cursor-based API

When using the cursor-based API, the application processes XML by advancing a logical cursor over a stream of XML tokens. The cursor-based parser is essentially a state machine transitioning from one well-defined state to another as a result of an event. In this case, the triggering event is
an XML token that is parsed when the application advances the parser along the token stream using the appropriate method. In each state, you can use a set of methods to obtain information about the latest event. Typically, not all methods are applicable in all states.

To use the cursor-based approach, the application must first obtain an `XMLStreamReader` from the `XMLInputFactory` by calling one of its `createXMLStreamReader` methods. There are several versions of this method, each of which supports a different type of input. For example, it is possible to create an `XMLStreamReader` to parse a plain `java.io.InputStream`, a `java.io.Reader`, but also a JAXP Source (`javax.xml.transform.Source`). In theory, the last option should make it easier to interact with other JAXP technologies, such as SAX and DOM.

**Listing 2. Creating an XMLStreamReader to parse an InputStream**

```java
URL url = new URL(uri);
InputStream input = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader r = factory.createXMLStreamReader(uri, input);
// process the stream
// ...
r.close();
input.close();
```

The `XMLStreamReader` interface essentially defines the cursor-based API (though the token constants are defined in its super-type, interface `XMLStreamConstants`). It is called cursor-based because the reader acts as a cursor over the underlying token stream. The application can advance the cursor forward along the token stream and examine the token at the cursor location.

`XMLStreamReader` provides several methods to navigate the token stream. To determine the type of token (or event) that the cursor is currently pointing at, the application can call `getEventType()`. This method returns one of the token constants defined in interface `XMLStreamConstants`. To move on to the next token, the application can call `next()`. This method also returns the type of the parsed token -- the same value that would be returned by a subsequent call to `getEventType()`. This method (and other reader-advancing methods) may only be called as long as method `hasNext()` returns true (that is, there are more tokens to be parsed).
Listing 3. Common usage pattern for processing XML using XMLStreamReader

```java
// create an XMLStreamReader
XMLStreamReader r = ...;
try {
    int event = r.getEventType();
    while (true) {
        switch (event) {
            case XMLStreamConstants.START_DOCUMENT:
                // add cases for each event of interest
                // ...
                break;
            case XMLStreamConstants.END_DOCUMENT:
                // add cases for each event of interest
                // ...
                break;
        }
        if (!r.hasNext())
            break;
        event = r.next();
    }
} finally {
    r.close();
}
```

A few other methods can cause the reader to advance. Method `nextTag()` will skip any white-
space, comment, or processing instruction until a `START_ELEMENT` or `END_ELEMENT` is reached. This
method is useful when parsing element-only content; if it encounters non-white-space text before
a tag is found (other than comments or processing instructions), it throws an exception. Method `getElementText()` will return all text content of an element between its opening and closing tags
(that is, between `START_ELEMENT` and `END_ELEMENT`). It throws an exception if it finds any nested
elements.

You will notice the terms "tokens" and "events" are used interchangeably in this context. While
the documentation for the cursor-based API talks about events, it is easier to think of the input
source as a stream of tokens. It is also less confusing, since there is a whole other event-based
API style (where events are proper objects). However, `XMLStreamReader`'s events are not all tokens
per se. For instance, the `START_DOCUMENT` and `END_DOCUMENT` events require no matching tokens.
The former event occurs before parsing begins, and the latter after no more parsing can be done
(for example, after parsing the last element's closing tag, the reader is in the `END_ELEMENT` state;
however, after attempting to parse more tokens and finding none, the reader transitions to the
`END_DOCUMENT` state).

Processing XML documents

In each parser state, the application can use the applicable methods to get information about
it. For instance, methods `getNamespaceContext()` and `getNamespaceURI()` can get the current
namespace context and the namespace URI currently in effect, respectively, regardless of the
current event type. Similarly, `getLocation()` can get information about the location of the current
event. Methods `hasName()` and `hasText()` can find out if the current event has a name (such as an
element or an attribute), or text (such as characters, comments, or CDATA), respectively. Methods `isStartElement()`, `isEndElement()`, `isCharacters()`, and `isWhiteSpace()` are convenience
shortcuts for determining the nature of the current event. Lastly, method `require(int, String, String)` can assert the expected parser state; it will throw an exception unless the current event is
of the specified type and the local name and namespace, if specified, match the current event.
Listing 4. Using attribute-related methods available when current event is
START_ELEMENT

```java
if (reader.getEventType() == XMLStreamConstants.START_ELEMENT) {
    System.out.println("Start Element: " + reader.getName());
    for(int i = 0, n = reader.getAttributeCount(); i < n; ++i) {
        QName name = reader.getAttributeName(i);
        String value = reader.getAttributeValue(i);
        System.out.println("Attribute: " + name + "=" + value);
    }
}
```

Right after its creation, XMLStreamReader starts in the START_DOCUMENT state (that is, getEventType() will return START_DOCUMENT). Take this into account when you process tokens. Unlike an iterator, the cursor need not be advanced first (using next()) in order to get into a valid state. Similarly, the application should not attempt to advance after the reader transitions to its final state -- END_DOCUMENT. Once in this state, method hasNext() will return false.

The START_DOCUMENT event provides methods to obtain information about the document itself, such as getEncoding(), getVersion(), and isStandalone(). The application can also obtain named property values by calling getProperty(String); however, some properties are only defined in specific states (for instance, properties javax.xml.stream.notations and javax.xml.stream.entities return any notation and entity declarations, respectively, if the current event is DTD).

In START_ELEMENT and END_ELEMENT, you can use methods related to element name and namespace (such as getName(), getLocalName(), getPrefix(), and getNamespaceXXX()); attribute-related methods (getAttributeXXX()) are also available in START_ELEMENT.

ATTRIBUTE and NAMESPACE are also recognized as standalone events, though one would not encounter them while parsing a typical XML document. They could, however, be encountered when an ATTRIBUTE or NAMESPACE node is returned as a result of an XPath query.

In text-based events (such as CHARACTERS, CDATA, COMMENT, and SPACE), obtain text using the various getTextXXX() methods. You can retrieve the target and data of a PROCESSING_INSTRUCTION using get PITarget() and getPIData(), respectively. ENTITY_REFERENCE and DTD also support getText(); ENTITY_REFERENCE also getLocalName().

Once parsing is complete, the application closes the reader to release any resources that it acquired during the process. Note that this does not close the underlying input source.

Listing 5 provides a complete example of using the cursor-based API to process an XML document. First, the default instance of XMLInputFactory is obtained and an XMLStreamReader created to parse the given input stream. Next, the reader's state is iteratively examined and depending on the current event type, specific information is reported (such as element name and its attributes if in the START_ELEMENT state). Finally, the reader is closed when END_DOCUMENT is reached.
Listing 5. Complete example of using XMLStreamReader to parse an XML document

```java
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader r = factory.createXMLStreamReader(input);
try {
    int event = r.getEventType();
    while (true) {
        switch (event) {
        case XMLStreamConstants.START_DOCUMENT:
            out.println("Start Document.");
            break;
        case XMLStreamConstants.START_ELEMENT:
            out.println("Start Element: " + r.getName());
            for(int i = 0, n = r.getAttributeCount(); i < n; ++i)
                out.println("Attribute: " + r.getAttributeName(i) + "=" + r.getAttributeValue(i));
            break;
        case XMLStreamConstants.CHARACTERS:
            if (r.isWhiteSpace())
                break;
            out.println("Text: " + r.getText());
            break;
        case XMLStreamConstants.END_ELEMENT:
            out.println("End Element:" + r.getName());
            break;
        case XMLStreamConstants.END_DOCUMENT:
            out.println("End Document.");
            break;
        }
        if (!r.hasNext())
            break;
        event = r.next();
    }
} finally {
    r.close();
}
```

Advanced uses of XMLStreamReader

It is also possible to create a filtered XMLStreamReader by calling XMLInputFactory's createFilteredReader method with the base reader and an application-defined filter (that is, an instance of class implementing StreamFilter). While navigating a filtered reader, the filter is consulted whenever the base reader advances to the next token. If the filter approves of the current event, it is exposed to the filtered reader. If not, the token is skipped and the next one is tested, and so on. This approach allows developers to create cursor-based XML processors that handle a simplified subset of the parsed content and reuse them in conjunction with filters for various extended content models.

To perform more sophisticated stream manipulation, subclass StreamReaderDelegate and override the appropriate methods. An instance of this subclass can then be used to wrap a base XMLStreamReader, thus giving the application a modified view of the base XML stream. Use this technique to perform simple transformations of an XML stream, such as filtering out or substituting certain tokens, or even augmenting the stream with new ones.
In Listing 6, you wrap a base `XMLStreamReader` with a custom `StreamReaderDelegate` and override its `next()` method to skip over `COMMENT` and `PROCESSING_INSTRUCTION` events. When using the resulting reader, the application does not need to worry about ever encountering these types of tokens.

**Listing 6. Using a custom StreamReaderDelegate to filter out comments and processing instructions**

```java
URL url = new URL(uri);
InputStream input = url.openStream();

XMLInputFactory f = XMLInputFactory.newInstance();
XMLStreamReader r = f.createXMLStreamReader(uri, input);
XMLStreamReader fr = new StreamReaderDelegate(r) {
    public int next() throws XMLStreamException {
        while (true) {
            int event = super.next();
            switch (event) {
            case XMLStreamConstants.COMMENT:
            case XMLStreamConstants.PROCESSING_INSTRUCTION:
                continue;
            default:
                return event;
            }
        }
    }
}

try {
    int event = fr.getEventType();
    while (true) {
        switch (event) {
        case XMLStreamConstants.COMMENT:
        case XMLStreamConstants.PROCESSING_INSTRUCTION:
            // this should never happen
            throw new IllegalStateException("Filter failed!");
        default:
            // process XML normally
            
        }
        if (!fr.hasNext())
            break;
        event = fr.next();
    }
} finally {
    fr.close();
}

input.close();
```

**Beyond cursor-based processing**

As you can see, cursor-based API is all about efficiency. All state information is available directly from the stream reader and no extra objects are created. This is especially useful in applications where performance and low memory footprint are highly important.

The benefits of pull-based XML parsing have been known for some time. In fact, StAX itself has been derived from an approach called *XML Pull Parsing*. The XML Pull Parser API is similar to the
cursor-based API provided by StAX; the parser state can be examined for information about the last parsed event, then advanced to the next one, and so on. No event iterator-based alternative API was provided. This approach is quite light-weight and particularly suitable for resource-constrained environments, such as J2ME. However, few implementations provided enterprise-level features such as validation and thus XML Pull has never caught on among enterprise Java developers.

Based on experience with previous pull parser implementations, the creators of StAX opted to include an object-oriented alternative to the cursor-based API. Even though the `XMLEventReader` interface seems deceptively simple, the event iterator-based approach offers an important advantage over the cursor-based method. By turning parser events into first-class objects, it allows the application to process them in an object-oriented fashion. This promotes better modularity and code reuse across multiple application components.

**Listing 7. Parsing XML with StAX XMLEventReader**

```java
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLEventReader reader = inputFactory.createXMLEventReader(input);
try {
    while (reader.hasNext()) {
        XMLEvent e = reader.nextEvent();
        if (e.isCharacters() && ((Characters) e).isWhiteSpace())
            continue;
        out.println(e);
    }
} finally {
    reader.close();
}
```

**Summary**

In this article you were introduced to StAX and its lower level cursor-based API. Part 2 will take a more in-depth look at the event iterator API.
Related topics

- **JSR 173: Streaming API for XML**: Read the Java Specification Request proposing a Java-based API for pull-parsing XML.
- **XML Pull Parsing**: Explore this site dedicated to promotion and education of pull-based XML parsing.
- **All about JAXP, Part 1** (Brett McLaughlin, developerWorks, May 2005): Learn how to work with the JAXP API's parsing and validation features.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.