**Parsing XML**

Parsing XML by hand is full of traps and pitfalls. Although it’s relatively easy for a person to read, dissecting the XML into its component parts within an application is quite difficult.

However, you don’t have to worry about writing your own. There are loads of different solutions for parsing XML; all have their advantages, ranging from speed and accessibility to the interface. A parser falls into one of two categories only when it comes to accessing the XML document it has parsed.

In this section, we’ll look at the two different parser types and some examples of the parser solutions available to parse and provide an interface to an XML document.

**Parser Types**

All XML processing tools have a basic parser mechanism. It reads the XML and identifies the tags, their attributes, and all the other components of the XML file before passing it to a separate component. The other component then does the work of modeling the information and providing an interface that allows you to access the information and, if possible, edit it.

There are many different XML parsers available for all the different languages. A quick check reveals four different systems under Python and no fewer than sixteen under Perl. Each falls into one of two groups: It provides either an event-driven interface or a tree-based interface.

**Event-Driven Interface**

If you split an XML document into its component parts, it’s easy to identify and parse the document. As the document is processed, each particular element is treated as an *event*.

In order for the event-driven parser to work, you need to associate a particular function with the type of element that is identified in the XML document. Then, when the document is being parsed, the function is called each time a recognized element is identified. For example, each time a start tag is seen, the start tag handler function is called; each time character data is identified, the character data function is called.

This all gives rise to the term event-driven. Each time you see an element (a tag, a processing instruction, and so forth), you raise an event, which is turn processed by an event handler.

For example, given the following XML file:

<contact refid="23456">

<firstname>Martin</firstname>

<surname>Brown</surname>

</contact>

an event-based parser would raise the following events:

* Found start element contact with attribute refid and value 23456.
* Found start element firstname.
* Found character data Martin.
* Found end element firstname.
* Found start element surname.
* Found character data Brown.
* Found end element surname.
* Found end element /contact.

The exact implementation will vary according to the parser you are using, but the basic sequence is there. Note, by the way, that the events only highlight that a tag has been identified; the tag name is supplied to the event handler function. This is necessary because you don’t know what the tag names are in advance. It’s up to the script parsing the document to make a decision about what to do with a specific tag.

Because event-driven systems read an XML document in sequence without ever holding the entire document in memory, they are generally very fast and efficient. The downside is that because you read the XML document from start to finish, you have no way of moving within the document to another position. If that’s a requirement of the parsing process, you’ll need to record information manually as you go along.

Event-driven parsers are ideal for processing XML data for use elsewhere, such as during conversion to HTML or when reading the data from the file for insertion into a database. Other things event-driven parsers are good at include the following:

* cument searches—You can process an XML document until you find the tag or character data you are looking for.
* Conversion—HTML is just one example, but anything that requires the raw XML to be translated into another format is generally best done with event-driven parsers because you translate the information on-the-fly to its new format.
* Minor modifications—It may seem pointless, but you can read and regenerate XML with a parser. During the parsing process, you can change minor words, character data contents, and even reform XML. Event-driven parsers are great for cleaning and reformatting an XML document.
* Simple validation—The whole document isn’t in memory, so you can’t do all the checks necessary to validate the information completely, but simple problems such as spelling errors and general well-formedness can be checked.
* Building an internal structure—You can use event-driven parsers to build up a complex internal representation of the XML document. In a moment, we’ll look at the tree-based parser; event-driven parsers are often used to build the tree structure used by tree-based interfaces.

The downside to the event-driven parser is that because you don’t hold the entire document in memory, you cannot make decisions or modifications that require you to jump around the document. For example, if you wanted to reorder or change the structure of the document, you would have to record the structure first, which kind of defeats the object of reading the XML document sequentially.

This lack also means that you can’t verify the document beyond the simple checks already discussed, and you can’t cross reference the contents of the document between XML elements.

Despite all of these apparent problems, event-driven parsing is the most powerful and also one of the easiest to use. It doesn’t take a lot of work to get an event-driven parser working and, unless you need that cross-referencing facility, the speed and memory benefits of the event-driven parser far exceed its limitations.

**Tree-Based**

Logically, the individual elements of an XML document are similar to components of a tree. For example, the following extends our earlier contact example:

<contact refid="23456">

<name>

<firstname>Martin</firstname>

<surname>Brown</surname>

</name>

<address>

<house>29</house>

<street>The Road</street>

<town>The Town</town>

<city>The City</city>

<postcode>AB12 34CD</postcode>

<country>UK</country>

</address>

</contact>

The main trunk is contact; name is a branch that contains the first name and surname; address is a second branch that in turn contains further branches (or leaves, since they are at the end of a branch) containing the individual details of the address.

A tree-based parser does exactly what we’ve described above: It parses an XML document and turns the document into an internal representation that closely matches a tree.

If the event-driven method is sequential access, then the tree-based method is random access. Once the document has been parsed, you can access any element of the tree, change the order (grafting one branch from one position to another), and of course change the contents. For example, to change the country in our example, you just need to change the value of the country branch of the address branch of the document tree.

Scripting languages are ideally suited to the tree-based method because most support the complex structures and easy referencing and linking of information required to build a convenient tree model.

**Parser Solutions**

There are literally hundreds of different XML parsers and parser libraries available. In fact, long before XML actually became an official standard, there were a number of different parsers and other tools available.

It wasn’t very long before it became clear that some sort of standard needed to be produced. Two standard toolkits, both originally written in Java, now exist: Simple API for XML (SAX) and Document Object Model (DOM). SAX is the standard for event-driven parsers, and DOM is the standard for tree-based parsers.

We’ll also look at one other parser, Expat, which is not a standard, but is one of the most widely used parsers available for working with XML within the confines of a scripting language.

**Expat**

Expat was written by James Clark and is an event-driven parser for XML documents. Expat was originally written in C. As a result, it has the flexibility of being incorporated into a number of different scripting languages through their normal extension mechanisms, unlike many Java-based tools. This means that Expat is probably the most popular and widely supported of all the XML parsers that you’ll be seeing in this book.

Expat lends itself well to most parsing tasks. Some solutions even use Expat as the basis of a full SAX or DOM interface.

**Simple API for XML (SAX)**

The Simple API for XML (SAX) really just defines the frontend interface for processing XML documents. In the background is an XML parser that is responsible for reading the information and identifying the different elements.

SAX itself is an event-driven XML parser; to actually process a document, you must first create the methods or functions that will handle the different elements of the document. The SAX standard is based on the original Java implementation, called org.xml.sax, and defines the names of the methods and the process behind supplying the parser with information.

In practice there is very little difference between using SAX and using any other event-driven XML parser, including Expat. However, the big difference between a SAX-compliant parser and the other solutions is that the methods you create and the XML elements that can be handled remain the same. In fact, aside from the language-specific semantics of the language you are using, migrating from SAX under one language to SAX under another should be completely painless.

**Document Object Model (DOM)**

The Document Object Model (DOM) is a W3C standard for a tree-based API for processing and working with XML documents. As with SAX, it was originally a Java/JavaScript solution, but it has since grown into a general specification for working with documents in tree form.

Unlike SAX, with DOM you do not define functions to be called when particular elements are found. Instead, the DOM specification requires that methods be created to enable you to modify and create branches within an XML tree structure. Most DOM implementations define a basic set of functions to do this for you.

For example, within both the Perl and Python implementations, a nested data structure is generated, with each branch having a combination of methods and properties that make up the interface for manipulating and working with the XML document in its tree form.

The minor irony with most DOM implementations is that they will often use SAX or a similar event-driven parser to build the tree before it’s exposed and made available to the programmer.

Within the DOM specification, the individual elements of an XML document are identified as *nodes;* you use these nodes to access the data from the document. The different node types that should be supported by your DOM implementation are shown in [Table 5.1](http://acm.books24x7.com/viewer_r.asp?bookid=3214&chunkid=383741444#wbpch05P88).

| Table 5.1: DOM Node Types | |
| --- | --- |
| **Name** | **Children** |
| Document | Element (the root XML element), ProcessingInstruction, Comment, DocumentType |
| DocumentFragment | Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference |
| DocumentType | None |
| EntityReference | Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference |
| Element | Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference |
| Attr | Text, EntityReference |
| ProcessingInstruction | None |
| Comment | None |
| Text | None |
| CDATASection | None |
| Entity | Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference |
| Notation | None |

The exact implementation and interface used are entirely dependent on the extension or module you are using. Many will class themselves as DOM-compliant if they adhere to the names and general structure as outlined in the DOM specification. Others will just identify themselves as DOM-compatible or DOM-like if they are close.