XML for Perl developers, Part 2: Advanced XML parsing techniques using Perl
A look at tree parsing and event-driven parsing

Skill Level: Intermediate

Jim Dixon (jddixon@gmail.com)
Writer
Freelance

06 Feb 2007

This series is a guide to those who need a quick XML-and-Perl solution. Part 1 looked at XML::Simple, a tool to integrate XML into a Perl application. This second article in the series introduces the Perl programmer to the two major schools of XML parsing: tree parsing and event-driven parsing.

Introduction

For a surprisingly broad range of Perl applications, the XML tool of choice is XML::Simple, which was the subject of Part 1 of this series (see Resources). XML::Simple converts XML input files into an easily manipulated Perl data structure, and writes such data structures back out as XML. Keep in mind, however, that this approach doesn't work under certain circumstances.

XML::Simple is not the best way to go when you need to build a representation of the XML document in memory and then either search it or transform it in fairly complex or unpredictable ways, which is where tree parsing comes in. If the XML document won't fit in memory or is a stream of unknown length, you can't use XML::Simple. You must use an event-driven parser. Most people find event-driven parsers a bit strange at first, but once you are used to this style of parsing, SAX might become your tool of preference.

The rest of this article will look at these two advanced ways of using Perl to parse XML.

Getting started
You will need some open source Perl modules to follow this article. Most often you
will get these in one of two ways: if you are on Windows, use ppm; if your operating
system is UNIX® or Linux™, go to CPAN (see Resources for links). If you aren't
familiar with these repositories, Part 1 in this series can introduce you to them.

Listing 1 shows you how to get the modules under UNIX/Linux. This is of course
better done as root, to make these modules available to all accounts on the system.
These modules have dependencies, some of which might not be present on your
system. If cpan is configured correctly (follow=yes), the dependencies will be
installed automatically.

**Listing 1. Getting modules used in this article from CPAN**

```
$ perl -MCaron -e shell
cpan> install XML::LibXML XML::SAX::Base XML::SAX::ExpatXS XML::SAX::Writer
cpan> quit
```

Under Windows, it's even simpler, as seen in Listing 2. Once again installation is
better done by an admin account.

**Listing 2. Getting the modules using PPM**

```
$ ppm install XML::LibXML XML::SAX::Base XML::SAX::ExpatXS XML::SAX::Writer
```

**Tree parsing**

Most programmers will probably find it comfortable to view XML as a tree structure.
This view of XML was formalized as the Document Object Model, the DOM, in a
process lasting many years; DOM Level 3 was reached in 2002.

The DOM represents an XML document as a tree of doubly-linked nodes, with the
first child at each level linked up to its parent and across to siblings. A large set of
functions is defined on the tree, with implementations in the major programming
languages.

Although you can navigate a DOM tree by following the links, it is generally more
efficient in terms of programmer time to use the XPath protocol. This is a
sublanguage allowing navigation to nodes, retrieval of sets of nodes, and so forth.

See Resources for pointers to the DOM specification itself and more readable
introductions to the DOM spec, XPath, and related protocols.

Many Perl modules can parse XML documents into DOM trees. Of these Petr
Pajas's XML::LibXML is one of the best (see Resources). It wraps the Gnome
project's libxml2, which is a multi-faceted package including a DOM parser, a partial
implementation of XPath and an implementation of SAX2 (discussed below).

Listing 3 is the XML file you worked with in Part 1 of this series (see Resources),
where you parsed it with XML::Simple, made changes to the representation as a Perl data structure, and then used XML::Simple to convert it back to XML in text form.

Listing 3. Rosie’s pet shop, pets.xml

```xml
<?xml version='1.0'?>
<pets>
  <cat>
    <name>Madness</name>
    <dob>1 February 2004</dob>
    <price>150</price>
  </cat>
  <dog>
    <name>Maggie</name>
    <dob>12 October 2002</dob>
    <price>75</price>
    <owner>Rosie</owner>
  </dog>
  <cat>
    <name>Little</name>
    <dob>23 June 2006</dob>
    <price>25</price>
  </cat>
</pets>
```

Using XML::LibXML to parse this is straightforward (see Listing 4 and the output from the program in Listing 5). A simple $parser->parse_file creates an XML tree structure to the DOM model. You define a simple Perl subroutine to add a subelement to a node in the tree and then use this to construct the subtree representing an individual pet. This subroutine, addPet(), is then used to add a couple of new pets, a gerbil and a hamster, to your inventory.

Listing 4. A XML::LibXML parse of Rosie's stock

```perl
#!/usr/bin/perl -w
use strict;
use XML::LibXML;

my $parser = XML::LibXML->new;
my $doc = $parser->parse_file('pets.xml')
or die "can't parse Rosie's stock file: $@";
my $root = $doc->documentElement();
sub addSubElm($$$) {
    my ($pet, $name, $body) = @_;  
    my $subElm = $pet->addNewChild('', $name);
    $subElm->addChild($doc->createTextNode($body));
}
sub addPet($$$) {
    my ($type, $name, $dob, $price) = @_;  
    # addNewChild is non-compliant; could use addSibling instead
    my $pet = $root->addNewChild('', $type);
    addSubElm ($pet, 'name', $name);
    addSubElm ($pet, 'dob', $dob);
    addSubElm ($pet, 'price', $price);
}
addPet('gerbil', 'nasty', '15 February 2006', '5');
addPet('hamster', 'boris', '5 July 2006', '7.00');
my @nodeList = $doc->getElementsByTagName('price');
foreach my $priceNode (@nodeList) {
    my $curPrice = $priceNode->textContent;
    my $newPrice = sprintf "%6.2f", $curPrice * 1.2;
    my $parent = $priceNode->parentNode;
```
my $newPriceNode = XML::LibXML::Element->new('price');
$newPriceNode->addChild ( $doc->createTextNode( $newPrice ) );
$parent->replaceChild ( $newPriceNode, $priceNode );
}
print $doc->toString(1);  # pretty print

To illustrate your mastery of the DOM, you then get a list of references to price
nodes in the tree and increase each price by 20%. Because you can represent text
(the prices) within an element by more than one text node, the simplest way to do
this is to get the price from the node, increase it and reformat it, and then entirely
replace the original node, rather than try to change it in place. This is of course
noticeably more complex than the same transformation in the Perl code in Part 1.

Listing 5. Tree Parser output, tidied up

```
<?xml version="1.0"?><pets>
  <cat>
    <name>Madness</name> <dob>1 February 2004</dob> <price>180.00</price>
  </cat>
  <cat>
    <name>Maggie</name> <dob>12 October 2002</dob> <price>90.00</price>
    <owner>Rosie</owner>
  </cat>
  <cat>
    <name>Little</name> <dob>23 June 2006</dob> <price>30.00</price>
  </cat>
  <gerbil>
    <name>nasty</name><dob>15 February 2006</dob><price>6.00</price>
  </gerbil>
  <hamster>
    <name>boris</name><dob>5 July 2006</dob><price>8.40</price>
  </hamster>
</pets>
```

This is typical when you deal with XML using a more conventional tree parser. Source as text in XML format is transformed into a DOM tree. To navigate the tree, either walk the nodes, following links from one to another, or use XPath-like commands to retrieve sets of references to nodes. You can then edit the nodes using those references. Then you can write the tree back to disk or pretty-print it.

For smaller and simpler trees, using XML::Simple is generally cheaper in terms of engineering costs. However, if the XML document is at all complicated, the availability of methods like getElementsByTagName tips the balance in favor of XML::LibXML. While this method might run more slowly than hand-crafted Perl and XML::Simple, you don't have to write it and you don't have to debug it.

Event-based parsing: SAX

The Simple API for XML (SAX), takes an entirely different approach to parsing, one that initially has a higher overhead. SAX conceives of a document as a series of events, and requires that you tell it how to respond to each. Such events include
The Perl SAX 2.1 Binding in Resources has a full list. For any document the Perl programmer must make available a set of handler methods, one for each type of event.

While this might seem a recipe for tedium and repetition, actually it's an opportunity, as you will see shortly.

While XML::LibXML has a SAX interface, it remains a DOM parser, so it reads an entire document into memory and then offers an event-oriented interface to it. This can often be useful, but it can't deal with documents that won't fit into memory or are XML streams, like Jabber/XMPP. So you will use XML::SAX::ExpatXS. This module wraps James Clark's venerable expat parser, and is very stable and fast.

Suppose you have a new pet shop, similar to that used in Part 1 of this series. Listing 6 represents part of that shop's inventory.

**Listing 6. Lizzie's Petatorium, pets2.xml**

```xml
<stock>
  <item type="iguana" cost="124.42" location="stockroom" age="1"/>
  <item type="pig" cost="15" location="floor" age="0.5"/>
  <item type="parrot" cost="700" location="cage" age="6"/>
  <item type="pig" cost="117.50" location="floor" age="3.2"/>
</stock>
```

To parse this using SAX2, you need something to handle the events produced by the parser. The simplest event handler is a writer which outputs some text on each event. The code in Listing 7 parses your new XML.

**Listing 7. SAX parse of pets2.xml**

```perl
#!/usr/bin/perl -w
#use strict;
use XML::SAX::ParserFactory;
use XML::SAX::Writer;
my $writer = XML::SAX::Writer->new;
$XML::SAX::ParserPackage = "XML::SAX::ExpatXS";
my $parser = XML::SAX::ParserFactory->parser(Handler => $writer);
eval { $parser->parse_file('pets2.xml') };
```

die "can't parse Lizzie's stock file: $@" if $@;

The XML then produces the output shown in Listing 8.

**Listing 8. SAX parser output**

```xml
<stock>
  <item cost='124.42' location='stockroom' type='iguana' age='1' />
  <item cost='15' location='floor' type='pig' age='0.5' />
  <item cost='700' location='cage' type='parrot' age='6' />
  <item cost='117.50' location='floor' type='pig' age='3.2' />
</stock>
```
Here are a few things to watch out for when you use ExpatXS:

- Make sure that all of your tools are either SAX or SAX2: don't try to mix and match. If you use XML::Handler::YAWriter instead of XML::SAX::Writer in Listing 7, you won't receive any error messages, but the output will be largely garbage. Because ExpatXS is a SAX2 parser, you must use a SAX2 writer with it.

- To check for parser errors, wrap the parse in an `eval` and then test `$@` not `$!`.

- You must set up handlers before you use them. It's important to understand that whereas a programmer visualizes a SAX parser as a pipeline (a point elaborated on below) proceeding from left to right, initialization must be done from right to left. That is, the pipeline looks like `P > W`, so you need to initialize in the reverse order, `W` and then `P`.

Drivers and filters

The genius of SAX starts here. SAX defines an event stream: the parser generates a series of events, passing each to a handler. Imagine an abstract module that can look like either or both. Like a parser, it can generate SAX events. But it's also a handler, one that can deal with any standard SAX event simply by switching hats, taking on its parser role, and passing the event on to the next handler. That is, it defines a set of default methods which just pass on events. The module handles these methods is XML::SAX::Base.

To define any conceivable SAX event handler, a programmer extends XML::SAX::Base and overrides whatever methods are of interest. Other events are just passed on. Such event handlers can be chained together, so that you can build pipelines, just like the UNIX command line. There are well defined interfaces on the handlers and exceedingly well-defined content: XML.

What is more, you can take the same approach at both ends of the pipe. At first cut, the generator is a SAX2 parser, consuming an XML document and generating events. Actually the generator can be anything that generates SAX events. You can for example write a module that reads a table in a database and outputs a stream of SAX events. (This exists too, as XML::Generator::DBI.)

Conventionally the other end of the pipe consumes SAX events and outputs a document. XML::SAX::Writer does just this. But the handler could just as easily write to a database (XML::SAX::DBI).

All of this provides two major benefits. First, it encourages the development of SAX handlers that transform the event stream in simple ways. This has happened; there are now hundreds of open-source Perl modules that implement the SAX 2.1 binding (see Resources). Secondly, it means that designers can focus their efforts on specifying handlers that provide the minimal functionality necessary to do a job in conjunction with existing handlers. Both exchange cheap machine resources for
expensive programmer time.

XML::SAX::Base in more detail

Designing handlers using Kip Hampton's XML::SAX::Base involves two simple steps. First, the handler must extend the base class. Secondly, the programmer must override base methods as necessary. Then you can either abandon the event or must invoke the method overridden in the based class. It is essential that the handler invoke methods in the superclass rather than methods in the overriding module (see Listing 9).

Listing 9. Using XML::SAX::Base

```perl
package XyzHandler;
use base qw(XML::SAX::Base); # extend it
sub start_element { # override methods as necessary
    my $self = shift;
    my $data = shift; # parameter is a reference to a hash
    $self->SUPER::start_element($data);
}
```

Summary

In this article, Part 2 of this three-part series, I gave you a necessarily brief overview of the very complex world of XML parsing.

First it showed you how to convert an XML document into a tree of objects in memory. Initially most programmers find this approach more natural, and it is indeed more convenient in many ways so long as the data will fit in memory.
Then it introduced you to SAX and event-based parsing, the approach you must take if your XML document is very large or is an unending stream. As it turns out, the tools developed to deal with these conditions lend themselves to an entirely different style of programming, one that turns out to be very rich: the SAX pipeline.

The next article in this series will show how you can use both of these approaches -- DOM and SAX parsing -- in more complex applications.
Resources

Learn

- **Tutorial on the use of references in Perl**: Study complete documentation about all aspects of Perl references and nested data structures.


- **Effective XML processing with DOM and XPath in Perl** (Parand Darugar, developerWorks, October 2001): Examine how to make effective and efficient use of DOM.

- **High-order perl** (Mark Jason Dominus, 2005): Read a book about functional programming techniques in Perl and how to write functions that can modify and manufacture other functions.

- **IBM XML certification**: Find out how you can become an IBM-Certified Developer in XML and related technologies.

- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.

- **developerWorks technical events and webcasts**: Stay current with technology in these sessions.

Get products and technologies

- **XML::LibXML** is one of the best Perl modules for parsing XML documents.

- **Document Object Model spec**: Get the details on a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents.

- **Perl SAX 2.1 binding**: Get a document that describes the version of SAX used by Perl modules.

- **Perl**: Get the most recent version and put it in action.

- **The huge CPAN Perl library**: Visit the Comprehensive Perl Archive Network for links to all of the modules mentioned in this article.

- **PPM, Perl Package Manager for Windows**: Get a tool that allows you to install, remove, upgrade, and otherwise manage the use of common Perl CPAN modules (like Tk and DBI) with ActivePerl.

- **Grant McLean's XML::Simple**: Try the XML::Simple module for a simple API layer on top of an underlying XML parsing module.

- **XML specification**: Explore this complete description of the Extensible Markup Language (XML).

- **Introduction to XML** (Doug Tidwell, developerWorks, August 2002): For a gentler introduction to XML, take this tutorial that covers how XML developed,
how it's shaping the future of electronic commerce, a variety of XML programming interfaces and standards, and two case studies that show how companies solve business problems with XML.

- **XPath 1.0**: Get the specification for a language to navigate the DOM tree.
- **XSLT 1.0 specification**: Learn about transforming one XML document into another.
- **Dare to script tree-based XML with Perl**: Find out how to work with tree-based document models (Parand Darugar, developerWorks, July 2000): Get a solid introduction to tree-based XML parsing with Perl.
- **IBM trial software**: Build your next development project with trial software available for download directly from developerWorks.

**Discuss**

- **XML zone discussion forums**: Participate in any of several XML-centered forums.
- **developerWorks blogs**: Get involved in the developerWorks community.

**About the author**

Jim Dixon
Jim Dixon is an independent contractor recently returned to San Francisco, where he advises Web 2.0 startups on the wonders of Perl and Ruby. Earlier in life he was technical lead at a UK/US Internet service provider for seven years and developed a lot of Java/J2EE software.

**Trademarks**

UNIX is a registered trademark of The Open Group in the United States and other countries.
Linux is a trademark of Linus Torvalds in the United States, other countries, or both. Other company, product, or service names may be trademarks or service marks of others.