# Using PERL with XML - Part I

## By icarus

# Table of Contents

# Perl Of Wisdom

Unless you've spent the last couple of years under a rock, you know about XML, the Extensible Markup Language, and its extended family of related technologies − XSLT, XPath, XLink, WDDX et al. You may even have used it a couple of times, creating XML documents to describe your address book or your CD collection, and marveled at how simple the process is.

It's at this point that most novice developers hit a brick wall. Sure, they know the theory of creating an XML document, and they may even understand why the technology is so widely discussed and praised. But when it comes to actually doing something with it − converting that XML−encoded CD collection into something that can be read by a browser, for example − there's simply not enough information out there to provide guidance on how to take the next step.

Over the next few pages, I will be attempting to rectify this a little bit, with an explanation of how you can convert your XML data into browser−readable HTML. Since XML is nothing but plain text, it makes sense to use a language which specializes in parsing text documents...which is where Perl, that granddaddy of text processing languages, comes in. There are a couple of basic approaches to parsing XML data; this two−part article will explore the Perl implementation of each, together with examples and illustrations.

I'll try and keep it simple − I'm going to use very simple XML sources, so you don't have to worry about namespaces, DTDs and PIs − although I will assume that you know the basic rules of XML markup, and of Perl scripting. So let's get this show on the road.

## Getting Down To Business

Before we get into the nitty−gritty of XML parsing with Perl, I'd like to take some time to explain how all the pieces fit together.

In case you don't already know, XML is a markup language created to help document authors describe the data contained within a document. This description is accomplished by means of tags, very similar in appearance to regular HTML markup. However, where HTML depends on pre−defined tags, XML allows document authors to create their own tags, immediately making it more powerful and flexible. There are some basic rules to be followed when creating an XML file, and a file can only be processed if these rules are followed to the letter.

Once a file has been created, it needs to be converted, or "transformed", from pure data into something a little more readable. XSL, the Extensible Style Language, is typically used for such transformations; it's a powerful language that allows you to generate different output from the same XML data source. For example, you could use different XSL transformations to create an HTML Web page, a WML deck, and an ASCII text file...all from the same source XML.

There's only one problem here: most browsers don't come with an XML parser or an XSL processor. The latest versions of Internet Explorer and Netscape Gecko do support XML, but older versions don't. And this brings up an obvious problem: how do you use an XML data source to generate HTML for these older browsers?

The solution is to insert an additional layer between the client and the server, which takes care of parsing the XML and returning the rendered output to the browser. And that's where Perl comes in − it supports XML parsing, through add−on DOM and XML packages, and even has a package to handle XSL transformations through the Sablotron processor.

As I've said earlier, there are two methods to parse XML data with Perl, and each one has advantages and disadvantages. I'll explain both approaches, together with simple examples to demonstrate how to use them in your own applications.

## Let's Talk About SAX

The first of these approaches is SAX, the Simple API for XML. A SAX parser works by traversing an XML document and calling specific functions as it encounters different types of tags. For example, I might call a specific function to process a starting tag, another function to process an ending tag, and a third function to process the data between them.

The parser's responsibility is simply to parse the document; the functions it calls are responsible for processing the tags found. Once the tag is processed, the parser moves on to the next element in the document, and the process repeats itself.

Perl comes with a SAX parser based on the expat library created by James Clark; it's implemented as a Perl package named XML::Parser, and currently maintained by Clark Cooper. If you don't already have it, you should download and install it before proceeding further; you can get a copy from http://wwwx.netheaven.com/~coopercc/xmlparser/, or from CPAN (http://www.cpan.org/).

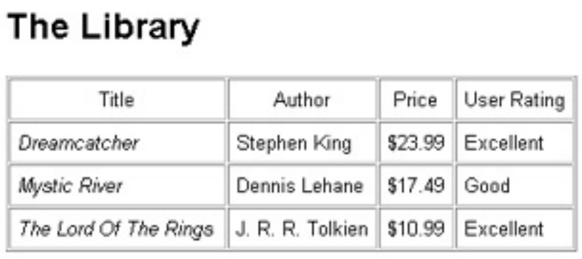
I'll begin by putting together a simple XML file:

```
<?xml version="1.0"?>

<library>
<book>
<title>Dreamcatcher</title>
<author>Stephen King</author>
<genre>Horror</genre>
<pages>899</pages>
<price>23.99</price>
<rating>5</rating>
</book>

<book>
<title>Mystic River</title>
<author>Dennis Lehane</author>
<genre>Thriller</genre>
<pages>390</pages>
<price>17.49</price>
<rating>4</rating>
</book>

<book>
<title>The Lord Of The Rings</title>
<author>J. R. R. Tolkien</author>
<genre>Fantasy</genre>
<pages>3489</pages>
<price>10.99</price>
<rating>5</rating>
</book>
```

```
</library>
```

Once my data is in XML−compliant format, I need to decide what I'd like the final output to look like.

Let's say I want it to look like this:



**The Library**

| Title | Author | Price | User Rating |
|---|---|---|---|
| *Dreamcatcher* | Stephen King | $23.99 | Excellent |
| *Mystic River* | Dennis Lehane | $17.49 | Good |
| *The Lord Of The Rings* | J. R. R. Tolkien | $10.99 | Excellent |

As you can see, this is a simple table containing columns for the book title, author, price and rating. (I'm not using all the information in the XML file). The title of the book is printed in italics, while the numerical rating is converted into something more readable.

Next, I'll write some Perl code to take care of this for me.

## Breaking It Down

The first order of business is to initialize the XML parser, and set up the callback functions.

```
#!/usr/bin/perl

# include package
use XML::Parser;

# initialize parser
$xp = new XML::Parser();

# set callback functions
$xp->setHandlers(Start => \&start, End => \&end, Char =>
\&cdata);

# parse XML
$xp->parsefile("library.xml");
```

The parser is initialized in the ordinary way − by instantiating a new object of the Parser class. This object is assigned to the variable $xp, and is used in subsequent function calls.

```
# initialize parser
$xp = new XML::Parser();
```

The next step is to specify the functions to be executed when the parser encounters the opening and closing tags of an element. The setHandlers() method is used to specify these functions; it accepts a hash of values, with keys containing the events to watch out for, and values indicating which functions to trigger.

```
# set callback functions
$xp->setHandlers(Start => \&start, End => \&end, Char =>
\&cdata);
```

In this case, the user−defined functions start() and end() are called when starting and ending element tags are encountered, while character data triggers the cdata() function.

Obviously, these aren't the only types of events a parser can be set up to handle − the XML::Parser package allows you to specify handlers for a diverse array of events; I'll discuss these briefly a little later.

The next step in the script above is to open the XML file, read it and parse it via the parsefile() method. The parsefile() method will iterate through the XML document, calling the appropriate handling function each time it encounters a specific data type.

```
# parse XML
$xp->parsefile("library.xml");
```

In case your XML data is not stored in a file, but in a string variable − quite likely if, for example, you've generated it dynamically from a database − you can replace the parsefile() method with the parse() method, which accepts a string variable containing the XML document, rather than a filename.

Once the document has been completely parsed, the script will proceed to the next line (if there is one), or terminate gracefully. A parse error − for example, a mismatched tag or a badly−nested element − will cause the script to die immediately.

As you can see, this is fairly simple − simpler, in fact, than the equivalent process in other languages like PHP or Java. Don't get worried, though − this simplicity conceals a fair amount of power.

## Call Me Back

As I've just explained, the start(), end() and cdata() functions will be called by the parser as it progresses through the document. We haven't defined these yet − let's do that next:

```
# keep track of which tag is currently being processed
$currentTag = "";

# this is called when a start tag is found
sub start()
{
# extract variables
my ($parser, $name, %attr) = @_;

$currentTag = lc($name);

if ($currentTag eq "book")
{
print "<tr>";
}
elsif ($currentTag eq "title")
{
print "<td>";
}
elsif ($currentTag eq "author")
{
print "<td>";
}
elsif ($currentTag eq "price")
{
print "<td>";
}
elsif ($currentTag eq "rating")
{
print "<td>";
}

}
```

Each time the parser encounters a starting tag, it calls start() with the name of the tag (and attributes, if any) as arguments. The start() function then processes the tag, printing corresponding HTML markup in place of the XML tag.

I've used an "if" statement, keyed on the tag name, to decide how to process each tag. For example, since I know that <book> indicates the beginning of a new row in my desired output, I replace it with a <tr>, while other elements like <title> and <author> correspond to table cells, and are replaced with <td> tags.

In case you're wondering, I've used the lc() function to convert the tag name to lowercase before performing the comparison; this is necessary to enforce consistency and to ensure that the script works with XML documents that use upper−case or mixed−case tags.

Finally, I've also stored the current tag name in the global variable $currentTag − this can be used to identify which tag is being processed at any stage, and it'll come in useful a little further down.

The end() function takes care of closing tags, and looks similar to start() − note that I've specifically cleaned up $currentTag at the end.

```
# this is called when an end tag is found
sub end()
{
my ($parser, $name) = @_;
$currentTag = lc($name);
if ($currentTag eq "book")
{
print "</tr>";
}
elsif ($currentTag eq "title")
{
print "</td>";
}
elsif ($currentTag eq "author")
{
print "</td>";
}
elsif ($currentTag eq "price")
{
print "</td>";
}
elsif ($currentTag eq "rating")
{
print "</td>";
}

# clear value of current tag
$currentTag = "";
}
```

Note that empty elements generate both start and end events.

So this takes care of replacing XML tags with corresponding HTML tags...but what about handling the data between them?

```
# this is called when CDATA is found
sub cdata()
{
my ($parser, $data) = @_;
my @ratings = ("Words fail me!", "Terrible", "Bad",
"Indifferent", "Good",
"Excellent");

if ($currentTag eq "title")
{
print "<i>$data</i>";
}
elsif ($currentTag eq "author")
{
print $data;
}
elsif ($currentTag eq "price")
{
print "\$$data";
}
elsif ($currentTag eq "rating")
{
print $ratings[$data];
}

}
```

The cdata() function is called whenever the parser encounters data between an XML tag pair. Note, however, that the function is only passed the data as argument; there is no way of telling which tags are around it. However, since the parser processes XML chunk−by−chunk, we can use the $currentTag variable to identify which tag this data belongs to.

Depending on the value of $currentTag, an "if" statement is used to print data with appropriate formatting; this is the place where I add italics to the title, a currency symbol to the price, and a text rating (corresponding to a numerical index) from the @ratings array.

Here's what the finished script (with some additional HTML, so that you can use it via CGI) looks like:

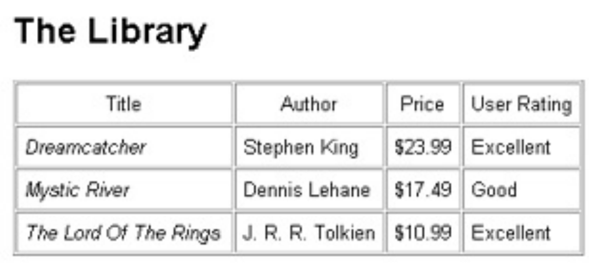
```
#!/usr/bin/perl

# include package
use XML::Parser;

# initialize parser
$xp = new XML::Parser();

# set callback functions
```

```
$xp->setHandlers(Start => \&start, End => \&end, Char =>
\&cdata);

# keep track of which tag is currently being processed
$currentTag = "";

# send standard header to browser
print "Content-Type: text/html\n\n";

# set up HTML page
print "<html><head></head><body>";
print "<h2>The Library</h2>";
print "<table border=1 cellspacing=1 cellpadding=5>";
print "<tr><td align=center>Title</td><td
align=center>Author</td><td
align=center>Price</td><td align=center>User
Rating</td></tr>";

# parse XML
$xp->parsefile("library.xml");

print "</table></body></html>";

# this is called when a start tag is found
sub start()
{
# extract variables
my ($parser, $name, %attr) = @_;

$currentTag = lc($name);

if ($currentTag eq "book")
{
print "<tr>";
}
elsif ($currentTag eq "title")
{
print "<td>";
}
elsif ($currentTag eq "author")
{
print "<td>";
}
elsif ($currentTag eq "price")
{
print "<td>";
}
elsif ($currentTag eq "rating")
{
```

```
print "<td>";
}

}

# this is called when CDATA is found
sub cdata()
{
my ($parser, $data) = @_;
my @ratings = ("Words fail me!", "Terrible", "Bad",
"Indifferent", "Good",
"Excellent");

if ($currentTag eq "title")
{
print "<i>$data</i>";
}
elsif ($currentTag eq "author")
{
print $data;
}
elsif ($currentTag eq "price")
{
print "\$$data";
}
elsif ($currentTag eq "rating")
{
print $ratings[$data];
}

}

# this is called when an end tag is found
sub end()
{
my ($parser, $name) = @_;
$currentTag = lc($name);
if ($currentTag eq "book")
{
print "</tr>";
}
elsif ($currentTag eq "title")
{
print "</td>";
}
elsif ($currentTag eq "author")
{
print "</td>";
}
```

```
elsif ($currentTag eq "price")
{
print "</td>";
}
elsif ($currentTag eq "rating")
{
print "</td>";
}

# clear value of current tag
$currentTag = "";
}

# end
```

And when you run it, here's what you'll see:



**The Library**

| Title | Author | Price | User Rating |
|---|---|---|---|
| *Dreamcatcher* | Stephen King | $23.99 | Excellent |
| *Mystic River* | Dennis Lehane | $17.49 | Good |
| *The Lord Of The Rings* | J. R. R. Tolkien | $10.99 | Excellent |

You can now add new items to your XML document, or edit existing items, and your rendered HTML page will change accordingly. By separating the data from the presentation, XML has imposed standards on data collections, making it possible, for example, for users with no technical knowledge of HTML to easily update content on a Web site, or to present data from a single source in different ways.

## Random Walk

In addition to elements and CDATA, Perl also allows you to set up handlers for other types of XML structures, most notably PIs, entities and notations (if you don't know what these are, you might want to skip this section and jump straight into another, more complex example on the next page). As demonstrated in the previous example, handlers for these structures are set up by specifying appropriate callback functions via a call to the setHandlers() object method.

Here's a quick list of the types of events that the parser can handle, together with a list of their key names (as expected by the setHandlers() method) and a list of the arguments that the corresponding callback function will receive.

```
Key Arguments Event
to callback
--------------------------------------------------------------------------
Final parser handle Document parsing completed

Start parser handle, Start tag found
element name,
attributes

End parser handle, End tag found
element name

Char parser handle, CDATA found
CDATA

Proc parser handle, PI found
PI target,
PI data

Comment parser handle, Comment found
comment

Unparsed parser handle, entity, Unparsed entity found
base, system ID, public
ID, notation

Notation parser handle, notation, Notation found
base, system ID, public
ID

XMLDecl parser handle, XML declaration found
version, encoding,
standalone

ExternEnt parser handle, base, External entity found
```

```
system ID, public ID

Default parser handle, data Default handler
```

As an example, consider the following example, which uses a simple XML document,

```
<?xml version="1.0"?>
<random>
<?perl print rand(); ?>
</random>
```

in combination with this Perl script to demonstrate how to handle processing instructions (PIs):

```
#!/usr/bin/perl

# include package
use XML::Parser;

# initialize parser
$xp = new XML::Parser();

# set PI handler
$xp->setHandlers(Proc => \&pih);

# output some HTML
print "Content-Type: text/html\n\n";
print "<html><head></head><body>And the winning number is: ";
$xp->parsefile("pi.xml");
print "</body></html>";

# this is called whenever a PI is encountered
sub pih()
{
# extract data
my ($parser, $target, $data) = @_;

# if Perl command
if (lc($target) == "perl")
{
# execute it
eval($data);
}
}

# end
```

In this case, the setHandlers() method knows that it has to call the subroutine pih() when it encounters a processing instruction in the XML data; this user−defined pih() function is automatically passed the PI target and the actual command to be executed. Assuming the command is a Perl command − as indicated by the target name − the function passes it on to eval() for execution.

## What's For Dinner?

Here's another, slightly more complex example using the SAX parser, and one of my favourite meals.

```
<?xml version="1.0"?>

<recipe>

<name>Chicken Tikka</name>
<author>Anonymous</author>
<date>1 June 1999</date>

<ingredients>

<item>
<desc>Boneless chicken breasts</desc>
<quantity>2</quantity>
</item>

<item>
<desc>Chopped onions</desc>
<quantity>2</quantity>
</item>

<item>
<desc>Ginger</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Garlic</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Red chili powder</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Coriander seeds</desc>
<quantity>1 tsp</quantity>
</item>

<item>
<desc>Lime juice</desc>
<quantity>2 tbsp</quantity>
```

```
</item>

<item>
<desc>Butter</desc>
<quantity>1 tbsp</quantity>
</item>
</ingredients>

<servings>
3
</servings>

<process>
<step>Cut chicken into cubes, wash and apply lime juice and
salt</step>
<step>Add ginger, garlic, chili, coriander and lime juice in a
separate
bowl</step>
<step>Mix well, and add chicken to marinate for 3−4
hours</step>
<step>Place chicken pieces on skewers and barbeque</step>
<step>Remove, apply butter, and barbeque again until meat is
tender</step>
<step>Garnish with lemon and chopped onions</step>
</process>

</recipe>
```

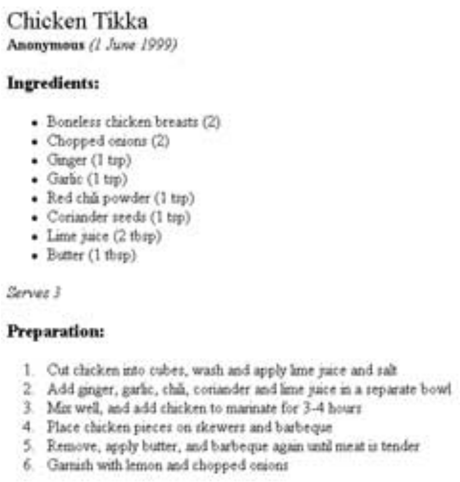
This time, my Perl script won't be using an "if" statement when I parse the file above; instead, I'm going to be keying tag names to values in a hash. Each of the tags in the XML file above will be replaced with appropriate HTML markup.

```
#!/usr/bin/perl

# hash of tag names mapped to HTML markup
# "recipe" => start a new block
# "name" => in bold
# "ingredients" => unordered list
# "desc" => list items
# "process" => ordered list
# "step" => list items

%startTags = (
"recipe" => "<hr>",
"name" => "<font size=+2>",
"date" => "<i>(",
"author" => "<b>",
"servings" => "<i>Serves ",
"ingredients" => "<h3>Ingredients:</h3><ul>",
"desc" => "<li>",
"quantity" => "(",
"process" => "<h3>Preparation:</h3><ol>",
"step" => "<li>"
);

# close tags opened above
%endTags = (
"name" => "</font><br>",
"date" => ")</i>",
"author" => "</b>",
"ingredients" => "</ul>",
"quantity" => ")",
"servings" => "</i>",
"process" => "</ol>"
);

# name of XML file
$file = "recipe.xml";

# this is called when a start tag is found
sub start()
{
# extract variables
my ($parser, $name, %attr) = @_;

# lowercase element name
$name = lc($name);

# print corresponding HTML
if ($startTags{$name})
{
print $startTags{$name};
}
}

# this is called when CDATA is found
sub cdata()
{
my ($parser, $data) = @_;
print $data;
}

# this is called when an end tag is found
sub end()
{
my ($parser, $name) = @_;
```

```
$name = lc($name);
if ($endTags{$name})
{
print $endTags{$name};
}
}

# include package
use XML::Parser;

# initialize parser
$xp = new XML::Parser();

# set callback functions
$xp->setHandlers(Start => \&start, End => \&end, Char =>
\&cdata);

# send standard header to browser
print "Content-Type: text/html\n\n";

# print HTML header
print "<html><head></head><body>";

# parse XML
$xp->parsefile($file);

# print HTML footer
print "</body></html>";

# end
```

In this case, I've set up two hashes, one for opening tags and one for closing tags. When the parser encounters an XML tag, it looks up the hash to see if the tag exists as a key. If it does, the corresponding value (HTML markup) is printed. This method does away with the slightly cumbersome branching "if" statements of the previous example, and is easier to read and understand.

Here's the output:



Chicken Tikka
**Anonymous** *(1 June 1999)*

**Ingredients:**

- Boneless chicken breasts (2)
- Chopped onions (2)
- Ginger (1 tsp)
- Garlic (1 tsp)
- Red chili powder (1 tsp)
- Coriander seeds (1 tsp)
- Lime juice (2 tbsp)
- Butter (1 tbsp)

*Serves 3*

**Preparation:**

1. Cut chicken into cubes, wash and apply lime juice and salt
2. Add ginger, garlic, chili, coriander and lime juice in a separate bowl
3. Mix well, and add chicken to marinate for 3-4 hours
4. Place chicken pieces on skewers and barbeque
5. Remove, apply butter, and barbeque again until meat is tender
6. Garnish with lemon and chopped onions

That's about it for the moment. Over the last few pages, I've discussed using Perl's XML::Parser package to process an XML file and mark up the data within it with HTML tags. However, just as there's more than one way to skin a cat, there's more than one way to process XML data with Perl. In the second part of this article, I'll be looking at an alternative technique of parsing an XML file, this time using the DOM. Make sure you come back for that one!

Note: All examples in this article have been tested on Linux/i586 with Perl 5.005. Examples are illustrative only, and are not meant for a production environment. YMMV!