

XML parser

Software documentation



Author: Tom Kenter, 2012

XML parser

Software documentation

1.	What is it?.....	3
	Low memory usage	3
2.	Where is it?.....	3
3.	What data can it handle?	3
	(Non) valid XML	3
4.	How does it work?.....	4
	Architecture.....	4
	The XML parser.....	4
	Parsing files.....	5
	Parsing strings	5
	Event handlers.....	6
	Information passed to the event handler	6
	Implementing an event handler	7
	Database access	8
5.	Fully working example: xmlParser.pl.....	8
	xmlParser.pl.....	8
	xmlParser::EventHandler::Example.pm	9

1. What is it?

The XML parser is a versatile piece of software, written in Perl, that can be used for all kinds of purposes. Actually, every time some XML(-ish) file has to be processed in whatever way, XML parser could do it.

It has been used to:

- Convert XML files to CoBaLT tokenized files.
All XML formats that are supported by CoBaLT are parsed with the XML parser and converted to a tab based tokenized file that CoBaLT needs.
- Process OED XML for use in the INL Attestation Tool.
This means parsing the OED XML and inserting the relevant data into an Attestation Tool database.
- Parse TEI XML lexica in order to merge them.
- Pretty print XML

Low memory usage

Because the parser goes through the input character by character and only the last tag or text is remembered it does not use a lot of memory and is able to handle very large files (unless of course you do very memory intensive stuff in your event handler code).

2. Where is it?

The core is a Perl module called `xmlParser::xmlParser`.

The current version is in:

`...\Tokenizer_with_XMLFunctionality\xmlParser\xmlParser.pm`

The parser is implemented as an object. It can very easily be incorporated into other programs. But it can easily be used in a stand-alone script as well. See e.g. `xmlParser.pl` (so `.pl` not `.pm`) and `tokenizeXml.pl` (both in `\Tokenizer_with_XMLFunctionality`).

3. What data can it handle?

(Non) valid XML

The XML parser makes no assumptions about the data it handles. XML that is fed to it does not have to be valid. Also, DTD's, namespaces, etc. are ignored completely. In fact, the input doesn't even have to be XML. You can also use plain text as input with an occasional tag in it (like the `.fixed` format)¹.

¹ You could even use tag-less plain text as input or whatever else. This admittedly makes very little sense, but there is nothing in the code that will prevent you from doing so.

If you have regular XML and you want to validate it you can of course use another program like `xmllint` to do the validation beforehand. The `tokenizeXML.pl` script, for example, has an option that does exactly that if you want it to.

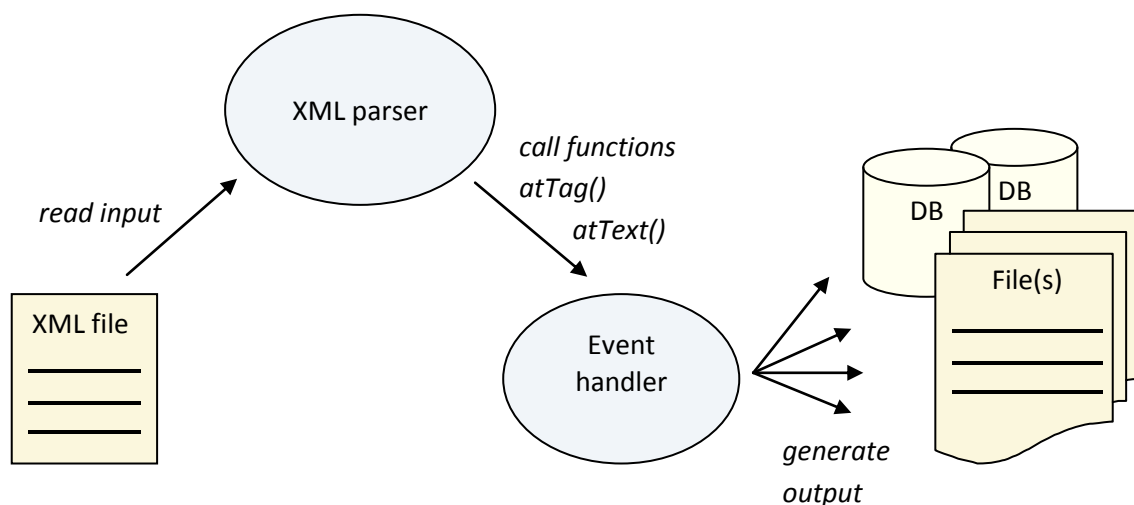
The `xmlParser::xmlParser` code supports both the parsing of entire files and of chunks of data (strings). See relevant sections below.

4. How does it work?

Architecture

XML parser is an event-based sequential access parser, analogous in this respect to Perl's XML::SAX². The parser goes through a file or string character by character. When a complete tag or piece of text has been read, it throws an event. In other words, it calls a function. When a tag has been read, it calls `atTag()`, and when text has been read, it calls `atText()`. These two functions have to be provided per project/use of the parser and it's these two functions that define what the program actually does. They should be implemented in yet another object, in a Perl package, which should be called `xmlParser::EventHandler::myEventHandler.pm` (where *myEventHandler* can be whatever you want to call your event handler. Usually it is the name of the format it is supposed to handle).

In a picture it might look like this:



The XML parser

Actually, the XML parser itself doesn't do a lot. As noted above, it goes through the input and keeps track of the tag or text that is being read at that moment. When a tag or a string of text is read completely, it calls the appropriate event handler method, `atTag()` or `atText()` respectively.

² The reason for not using XML::SAX itself is because it doesn't keep track of character positions. E.g. for the tokenized files of CoBaLT the character positions of the text strings in the XML code are needed. XML::SAX (or other XML parsers) do not provide character position information as from an XML point of view they are irrelevant.

Also, at the start of the file and the end of the file a method is called if its exists. They are called `atStartOfFile()` and `atEndOfFile()` respectively.

If the right event handler code has been provided, all a parser script needs to do is this:

```
use xmlParser::xmlParser;
use xmlParser::EventHandler::myEventHandler.pm

my $oEventHandler = xmlParser::EventHandler::myEventHandler->new();
my $oXmlParser = xmlParser::xmlParser->new();

$oXmlParser->{oEventHandler} = $oEventHandler;

$oXmlParser->setInputFileHandle('myFile.xml');
$oXmlParser->parseFile();
```

Or alternatively you might create the xml parser object in one go:

```
use xmlParser::xmlParser;
use xmlParser::EventHandler::myEventHandler.pm

my $oXmlParser =
  xmlParser::xmlParser->new(oEventHandler =>
    xmlParser::EventHandler::myEventHandler->new());

$oXmlParser->setInputFileHandle('myFile.xml');
$oXmlParser->parseFile();
```

Parsing files

You can call `$oXmlParser->parseFile()` on an entire file.

The code expects an open file handle to be set in `$oXmlParser->{fhInput}`. Also the name of the file should be set in `$oXmlParser->{sInputFileName}`.

The `atStartOfFile()` and `atEndOfFile()` functions will be called, if they are defined, before and after the file is parsed (respectively).

Parsing strings

If you just want to parse just a chunk of data, rather than an entire file, you can use `$oXmlParser->parseString($sXmlString)`.

NOTE that you should give a reference to the string you want to parse as an argument, not the string itself.

The code expects the member variable `$oXmlParser->{sInputFileName}` to have a value. This value is only used for printing errors. E.g. you might want to set it to 'STRING'.

The `atStartOfFile()` and `atEndOfFile()` function will be called, if they are defined, before and after (respectively) the string (in this case) is parsed.

So each time you call `parseString()` these two functions will be called (if they are defined).

Event handlers

Information passed to the event handler

The methods `atTag()` and `atText()` both have one argument which is a reference to a hash representing the tag/text at hand. It is the job of `xmlParser` to fill the hash.

Here is what they look like.

The hash for the tag:


```
{
  sTagText => '',          # The full string as it appeared in the source XML
  sTagName => '',          # The name of the tag
  hrAttributes => {},      # The attributes represented as hash
  iStartPos => 0,          # Character position of the start of the tag
  iEndPos => 1,            # Character position of the end of the tag
}
```

The hash for the text:

```
{
  sText => '',             # The full string as it appeared in the source XML
  iStartPos => 0,          # Character position of the start of the text
  iEndPos => 1,            # Character position of the end of the text
}
```

Example of information passed to the event handler

Let's say we have an XML file that looks like the one below, and the xmlParser code has reached the spot indicated by the arrow.



```
<xml/>
<content documentId="123" title="Example">
  <paragraph>
    This is a paragraph of text.
  </paragraph>
</content>
```

At that point the parser object will call `$oXmlParser->{oEventHandler}->atTag($hrTag)`. And the argument, a reference to a hash representing the tag, will look like this:

```
{
  # The full string as it appeared in the source XML
  sTagText => '<content documentId="123" title="Example">',
  sTagName => 'content',          # The name of the tag
  # The attributes represented as hash
  hrAttributes => {documentId => 123,
                  title => 'Example'
                },
  iStartPos => 7,                 # Character position of the start of the tag
  iEndPos => 49,                  # Character position of the end of the tag
}
```

Implementing an event handler

The event handler code is where it really happens.

The xmlParser code assumes that an event handler object has been set as a member variable of the xmlParser object called `oEventHandler` (see example in XML parser section above). It uses the object in this member variable to call methods of.

As noted earlier, there are four methods that can be called and of course there has to be a constructor for the object. When a new event handler is written the definitions of the constructor and `atTag()` and `atText()` have to be provided. The other two member functions are optional.

- `xmlParser::EventHandler::myEventHandler::new ()`
Obligatory
Constructor of the object. Nothing needs to be done here but for the usual code that creates

an object.

However, you could use the constructor for initializing some member variables.

- `xmlParser::EventHandler:: myEventHandler::atStartOfFile()`
Optional
This method is called before a new file or string is parsed.
- `xmlParser::EventHandler:: myEventHandler::atTag()`
Obligatory
This method is called when a complete tag has been processed. It gets a reference to a hash containing the information about the tag as an argument (see above).
- `xmlParser::EventHandler:: myEventHandler::atText()`
Obligatory
This method is called when a complete string of text has been processed. It gets a reference to a hash containing the information about the text as an argument (see above).
- `xmlParser::EventHandler:: myEventHandler::atEndOfFile()`
Optional
This method is called after the file or string has been parsed.

Database access

If your event handler needs access to an IMPACT lexicon database you can derive your event handler object from `xmlParser::EvenatHandler::databaseFunctions`. This object is also an object, implementing database connectors and some data retrieval and insertion functions. If you derive your event handler from this package it will inherit its functionality and you can add your own stuff as desired.

5. Fully working example: `xmlParser.pl`

You will find this script in:

```
...\Tokenizer_with_XMLFunctionality\xmlParser.pl
```

To run the example you can issue the following command:

```
[/.../Tokenizer_with_XMLFunctionality]  
$ ./xmlParser.pl Example testExample.xml
```

This should produce the following output:

```
[/.../Tokenizer_with_XMLFunctionality]  
$ ./xmlParser.pl Example testExample.xml  
>> At start of file  
Text: 'This is content.' (character positions: 27, 43)  
>> At end of file
```

xmlParser.pl

Usage:


```
./xmlParser.pl FORMAT XML_FILE
```

This script is just a wrap around the xmlParser object. It includes the right event handler package, based on the first argument, and makes a new XML parser object with it. With that all being set up it calls parseFile() and that's it.

xmlParser::EventHandler::Example.pm

The code for this package is in

...\Tokenizer_with_XMLFunctionality\xmlParser\EventHandler\Example.pm.

In the example event handler the only thing that happens is that some text is being printed. Note that the atTag() function is only used to keep track of which tag we are in. If the tag is relevant the text is printed in atText().