Package SemiXML to describe and use semi XML

Marcel Timmerman <mt1957@gmail.com >

Abstract

This manual explains the use of a set of classes to convert *semi-xml* text into *XML* typed languages. A description of the syntax is given and the configuration controlling this process. Furthermore, the dynamic aspects of this language is explained as well as how to extend this language with your own classes.

The manual will also explain how the programs work which are installed with this package.

The latest version of this document is generated on date2018-04-14.

Table of Contents

1. Introduction of package *SemiXML*

Welcome to the SemiXML package. The *SemiXML* package comprises of a set of Perl 6 modules to convert a text written in a language coined *semi-xml* into one of the *XML* family of languages such as *HTML*, *SOAP*, *Docbook* or *SVG*. There are dynamic ways to generate elements by calling methods in external modules. This is also the way to extend the language by creating your own methods.

In the package there is also a program called *sxml2xml* which uses *SemiXML::Sxml* in the package *SemiXML* to transform *semi-xml* text from a given file into XML after which it can be written to another file. The generated XML can also be send to any program for conversion to other formats or for checking. Examples are *xsltproc*, *xmllint*, *rnv*, *wkhtmltopdf*, *xep* etcetera. There is a second program *xml2sxml* in the making. It is used to reverse engineer XML into sxml. The methods used to do this can also insert xml into the tree directly as a 'literal' element node or convert it completely to the internal node tree representation.

Furthermore there are modules designed to generate fragments of html, docbook or other specific constructs like Epub and testing reports. Much of this code need more additions and some refinements but it shows the strength of this package.

This language is for people who want to write XML in a more readable way and want to add dynamic constructs to insert information much the same way as *PHP* does in html.

1.1. Advantages using this language

- The language has a better readability because the end tag of an elements is absent. The end of an element is just the ending character of the content. Also, when attribute values are simple, i.e. without spaces, quoting is not necessary.
- When output is generated, the library consults the configuration to see if an element is inlined (e.g. *b* in html) or is self-closing (e.g. *br* and *meta* in html). There two more situation which need attention. These are space preserving elements like *pre*(in html) and *programlisting*(from docbook) and no conversion elements which is needed for *script* and *style*(both from html).
- In this language, constructs are introduced to call methods coded in modules to generate XML elements. An example of this is shown in the example above to insert a date. This date will always be the current date when the text is processed.
- The language is extensible. Developers can add their own libraries to insert XML elements. Humble ideas such as inserting simple HTML elements to alleviate repetitive tasks or more elaborate projects to insert tables with data loaded from a database.

1.2. Disadvantages

- There is an extra level of processing. If the XML text is simple, you should not be bothered writing stuff in this language. Especially when the dynamic constructs aren't used.
- Pointing to the proper spot in the sxml text when an error occurs has been proven difficult because the only thing the parser has is the element format, attributes and matching brackets. When the parser arrives at the end of the document it may miss a closing bracket or seeing one too many. It is not easy to show where the bracket is missing or where there is one typed too many. However, a few solutions are implemented to help finding the error if there is one. This also needs some attention.

Finally an example will speak for itself when Sxml input is compared with the generated HTML. The generated html shown here is pretty printed but would normally be as compact as possible.

Example 1. Input *semi-xml* text

```
$html [
  $head [
    $title [ My First Sxml Document ]
  ]

  $body [
    $h1 [ My First Sxml Document ]
    $p class=hello [
      Hello world! And it is said on $!SxmlCore.date.
    ]
  ]
]
```

Example 2. Generated html text

```
<html>
  <head>
    <title>My First Sxml Document</title>
  </head>
  <body>
    <h1>My First Sxml Document</h1>
    <p class="hello">Hello world! And it is said on 2016-03-22.</p>
  </body>
</html>
```

1.4. Overview

- *Install and setup*. This part will help you to start by showing how to install the package and make a start with your first document.
- *The package*. An overview of the parts comprising the package. This section will tell something about the parsing modules, the included XML generating modules and the programs.
- *The language*. Here the *semi-xml* language is explained, the terminology of parts in the document and the used syntax.
- *Configuration*. The toml configuration file will be explained. This section explains how to control the output of the result and where control files are found. Defaults are also explained.
- *The programmers view*. Developers can write new modules with methods to process tasks not yet captured by this package. This section will talk about classes, data and methods.
- *The programs*. The program *sxml2xml* is used to read *semi-xml* from a file and saved or send away. Explanation of arguments and prelude options can be found here. While not available yet *xml2sxml* is used to transform *XML* to *semi-xml*
- *Examples*. Many short examples to show its use of sxml.

2. Install and setup

To install the package and programs the *zef perl6* module installer must be used. How to install perl6 and zef is not documented in this manual and one must look for the per6 documentation elsewhere.

The command is simple and is shown below;

```
> zef install SemiXML
```

If you want to skip testing add an option

```
> zef --/test install SemiXML
```

Another facility which is very helpful when editing is syntax highlighting. A highlighting package is made for the atom editor. For the moment not yet in the atom repository but only on github. For the time being, one can do the following to get this working;

```
> git clone https://github.com/MARTIMM/language-sxml.git
> apm link .
```

3. The package

There are several parts in this package. First there is the core consisting of several modules. These modules handle tasks to load the configuration, instantiating the external modules aiding the dynamic parts while parsing, the parsing of sxml text, saving results to a file or sending it to programs for further processing and much more.

We can devide the package in several pieces;

- There is a configuration file to set defaults.

3.2. Configuration

The configuration is needed to control the several phases of the Translation process. For exmple, there is the addition of extra text upfront of the xml result text such as xml description, doctype and header messages. Other parts control the output to a file or sending the result to another program. The format of the configuration file and how the data is processed is described in a separate chapter.

3.3. External modules

External modules can be written to add functionality to the language. This package comes with few modules to insert text, read text from files, specific HTML or Docbook tasks, Epub generation and test report documents. The modules Delivered in this package and how to write such a module is explained also in a separate chapter.

3.4. Parsing and translation

Parsing is the the process of taking the text and checking it for the *semi-xml* language. Together with this process translation of the text takes place. How this translation takes place is depending on the way parsing is done. Roughly it starts with the top level element and arriving at the first bracket '[' it may see a mix of text and new elements before the closing bracket ']'. The new elements content (between the elements brackets) is looked into in turn. So the translation of the deepest level must be done first before the second deepest etc. up to the top level. This knowledge is not important when using the simple element form and the predefined methods. However, when designing your own methods your method will get the processed XML of the content between its brackets.

3.5. Saving results

After the translation the result is extended with some extra textlines. This result can be saved into a file on disk. The other option to saving is sending the result to program for further processing. For example if the result happens to be HTML, this can be converted to PDF using a program like wkhtmltopdf.

4. The language

4.1. The document

A document always starts with an element as you probably know from any XML dialect. At the top level there may be only one element. The first example shows the most simple HTML document ever.

```
$html
```

Something to mention here is that there is no need to add a XML description, Doctype or default namespace. These are all controlled by the configuration data.

A more complex example shows a part of RSS or Really Simple Syndication.

```
$rss version=2.0 [
  $channel [
    $title [W3Schools Home Page]
    $link [https://www.w3schools.com]
    $description [Free web building tutorials]
    $item [
      $title [RSS Tutorial]
      $link [https://www.w3schools.com/xml/xml_rss.asp]
      $description [New RSS tutorial on W3Schools]
    ]
  ]
]
```

As you might have guessed it, the example is taken from w3schools where all sorts of XML languages are described.

The example shows how nesting is done using brackets ('[' and ']') and that between brackets there can be a mix of text and other elements. What we learn from this example is:

- *Elements*. A simple element is created by using a dollar in front of an identifier like *$rss*.
- *Attributes*. Elements may have any number of attributes and are written as a key value pair separated by an equal ('='). like *version=2.0* on the $rss element. The key name cannot be repeated on the same element, only one attribute of the duplicated key names will survive. The values do not have to be quoted if there are no spaces in the value. If there are any, single and double quote characters (' and ") can be used.

4.1.1. Comment

Comments are possible on several places and starts with '#'. To be more exact, all space before '\#', the character '\#' and characters after it up to the end of line including the end of line characters are removed.

4.1.2. Escaping characters

When you want to use a character which is used as a delimiter ('[') or start of an element ('$') you must escape that character to prevent the action that takes place when encountering that character. Writing '\' before that character escapes them from that interpretation. Examples are '\\[' or '\\\$'.

4.1.3. Unicode characters

All available Unicode characters from the utf-8 set may be used in the text. These can be inserted directly by cut and paste, using special keyboards, compose keys, entity codes such symbol © as &copy;, or using numeric codes symbol & &#9986; or &\#x2702;. The entity code must be known by the specific XML code using e.g. the DTD of the language.

4.1.4. Container types

There are several ways to contain the text between delimiters depending on the contents or even the absence of content. Take, for example the HTML break element <br/>. This element does not have content. We could write this as *$br []*. But we can leave off the brackets to make it shorter like so; *$br*. There are however situations where you must use the brackets when the following text is in a argument=value format to prevent them to be interpreted as such. E.g. *$br [] after=break*.

As described above under *Escaping characters* it is possible to insert characters which are otherwise interpreted as something special. However, it gets annoying when there are many of them like in a piece of JavaScript or Perl code. There is a solution for that luckily, just enclose the text between '{' and '}'. Nothing gets interpreted between these delimiters as well as comments are kept in the text. That works for simple code and is easier to type. But these brackets can be used a lot in code so there is another possibility. Use '«' and '»' for that. Not many program languages beside perl6 use these characters. It might come in handy to program a compose key for this, otherwise it becomes a pain to insert those characters.

In html the text is justified automatically except in certain sections such as within <pre> elements. Sxml does not have any notion of these sections and treat those as xml and all excess of space is removed. To cope with these sections another set of container delimiters are used. These are *[...]*. With these brackets, all indenting and newlines are fixed and protected.

Multiple content bodies are also possible when there are parts which need protection and in other parts, elements are needed. E.g. *$p [The line;]« my Int $x = 10; »[is an $b[Int] declaration]*

4.1.5. Element types

Some examples of elements are already shown at the start of this chapter like *$html* for instance. It is written as a name with a dollar prefixed to it. Normally, spacing around an element is minimized but at some places there must be a space before or after the element. Examples from html are <a>, <b> and <strong> among other inline elements. ... F Table config .... to have a space on both sides, on the left side or on the right side respectively.

Another type is a special one. It is defined as *$!module-key.method-name* where the module-key is mapped to a real module in the configuration file. One key is predefined: SxmlCore and is mapped to SxmlLib::SxmlCore. The methods defined in that module are explained in a separate chapter. How a module is initialized and called for its services is also enaugh stuff to have an extra chapter.

4.1.6. Attribute types

4.1.7. Core element methods

4.2. Configuration

The configuration is defined in a separate file which is in the TOML format. The data from the configuration is used to e.g. control the output of the result. Other usages are referencing libraries to be used while parsing the document and to specify te dependncy list. The information is specified in a number of tables. Each of these tables are used in one specific phase of the whole process. The phases in this transformation process and the used tables are;

- *Pre transformation phase*. In this phase dependencies, if any, can be checked and solved before transformation starts. The *X* table is used to find the dependencies.
- *Transformation phase*. This is the phase where parsing and transformation of the document takes place. The tables used here are the libraries table *L* and modules *M* table to initialize objects needed in this proces.
- *Post transformation phase*. When the document is translated, the result can be prefixed with extra data. The table used for this is the prefix table *P*. When the result is generated on a server and needs to be returned, extra data must be prefixed. These are the so called message headers of which that of the http protocol are the most commonly used headers. The table *H* is used for this.
- *Storage phase*. The next phase is storing the data in a file. For this the staorage table *S* is used where the files basename, path and extension is found.
- .

The prelude consists of a series of key-value pairs. The keys are defined as a series of catagories and subcatagories. The value can be anything. The key value pair is separated by a colon ':' and the pair ends in a ';'. The prelude part is also optional. In that case defaults will be applied.

Below you see an example of two options to let the xml prelude be written as well as a doctype at the start of the result.

4.2.1. Options used by the module *SemiXML::Sxml*

module

option/doctype/definition

```
option/doctype/definition: [
  <!ENTITY company "Acme Mc Carpenter, Inc">
  <!ENTITY program "sxml2xml">
  <!ENTITY library "Semi-xml">
  <!ENTITY nbsp " ">
]
```

option/doctype/show

option/http-header

option/xml-prelude/encoding

option/xml-prelude/show

option/xml-prelude/version

output/fileext

output/filename

output/filepath

output/program

```
output/program/pdf:
  | xsltproc --encoding utf-8 --xinclude stylesheet.xsl -
  | xep -fo - -pdf sxml2xml.pdf
  ;
```

5. The programmers view

5.1. Classes data and methods

5.2. Substitution data

5.3. Methods

6. Using program *sxml2xml*

6.1. Program arguments

```
sxml2xml [--run=<run-selector>] <sxml-file>
```

6.2. Prelude options

6.2.1. dependencies/files

The value of this option is a list of paths to semi-xml documents which must be processed after the current one.

6.3. Unix she-bang usage

7. Syntax

```
<sxml-syntax> ::= <prelude-section>? <document>;

<prelude-section> ::= '---' <new-line> <key-value-pairs>* '---' <new-line>;
<key-value-pairs> ::= <key-name> ':' <value> ';';
<key-name> ::= <key-part> ('/' <key-part>)*;
<key-part> ::= <letter> (<letter>|<number>)*;

<document> ::= <prefix> <element> ( <attribute> '=' <attr-value> )*
               '[' <body-start-control> <body> <body-end-control>
               ']';

<prefix> ::= '$.' | '$';

<element> ::= <identifier>;
<attribute> ::= <identifier>;
<attr-value> ::= '"' <ws-text> '"' | "'" <ws-text> "'"
                 | <non-ws-text>;

<identifier> ::=  <letter> (<letter>|<number>)*
                  ( '-' (<letter> (<letter>|<number>)+) )*;

<body-start-control> ::= '!=' | '\!' | '=';
<body-end-control> ::= '\!';
```

8. Examples

Index