

Using XML in Delphi applications. Part II

Sergey N. Kuchеров
(October 12, 2002)

Document Object Model (DOM) ▪ Simple API for XML (SAX) ▪ Microsoft XML Core Services ▪
Implementing XML Parser using Microsoft SAX interface

The Part I was about creating a basic XML object model and parser. The code is simple, however it does not support the whole set of W3C XML standards. While it is possible to make the code to cover at least XML 1.0 specifications, I give you a very good reason why you do not want to do this. XML is a dynamic standard, which is constantly evolving. Unless you are willing to dedicate time to keep your code up to date, use a standard solution and let somebody else to do coding.

In many cases the benefits to use a standard solution instead of a proprietary one can be significant. The two most commonly used standards in today's XML programming industry are Document Object Model (DOM), and Simple XML API (SAX).

Document Object Model (DOM)

Part I contains an example how to process the following simple XML document:

```
<?xml version="1.0"?>
<movie>
  <title>The Matrix</title>
  <producer>
    <FirstName>Bruce</FirstName>
    <LastName>Berman</LastName>
  </producer>
  <country>USA</country>
  <language>English</language>
</movie>
```

The following simple code uses a custom parser and simple object model. The model enables programmers to access it through an easy and intuitive interface:

```
var
  Node: TxmlNode;
  Parser: TxmlParser;
  title, producer: string;
begin
  Parser := TxmlParser.Create;
  Node := Parser.LoadFile('test.xml');
  title := Node['title'].Value;
  producer := Node['producer/FirstName'].Value +
    ' ' + Node['producer/LastName'].Value;
end;
```

Document Object Model (DOM) is a W3C² standard, which defines interface for a universal object model. The standards regulates only interface, therefore DOM can be implemented on any platform using any programming language. Technically, DOM is not intended to represent only XML documents. Initially DOM was designed for HTML parsers. As result the interface is optimized for a program, which has to scan through every single element of the document to produce a viewable web page.

Right now DOM is the only official standard for managing XML data in programs. It is widely used in Java, C++, and even .NET environments. W3C also supports a couple of standards (*XPath* and *XQuery*) for referencing XML documents in scripts.

The following code shows how to use DOM interface provided by Microsoft XML Core Services⁴ to read and process the simple XML document above:

```
var
  Doc: IXMLDOMDocument;
  Element: IXMLDOMElement;
  List: IXMLDOMNodeList;
  Path, ID, title, producer: string;
begin
  Doc := CreateOleObject('Microsoft.XMLDOM')
    as IXMLDomDocument;

  Doc.load('test.xml');
  Element := Doc.documentElement;
  List := Element.getElementsByTagName('title');
  title := List.item[0].text;
  Path := 'producer/FirstName';
  List := Element.getElementsByTagName(Path);
  producer := List.item[0].text;
  Path := 'producer/LastName';
  List := Element.getElementsByTagName(Path);
  producer := producer + ' ' + List.item[0].text;
end;
```

Any other DOM implementation will not change this code much, because DOM specifies the object model. Unlike the simple object model introduced in the Part I, DOM makes code more complicated. Even after removing the redundant *Path* variable the code is still a headache.

Now, when XML is widely used as a format to store and transfer data DOM interface is still dominates in Java, C++, and even .NET. Unfortunately, you have to make your choice of object model before you start any serious project.

Simple API for XML

SAX stands for Simple API for XML³. Unlike DOM, SAX does not implement any object model. The purpose of SAX is to parse XML code and to call foreign function to process it. The main benefit of SAX is an ability to separate object model from parser. It is very useful, when XML standards change frequently and you do not want to rewrite your code every month to comply with latest W3C ideas.

Microsoft XML Core Services

Microsoft XML Core Services library contains SAX interface, which can be used by any Windows program. All code examples have been tested using MSXML version 4.0 SP1. You can download latest copy of MSXML from Microsoft web site⁴.

Make sure the MSXML is installed on your computer before you create Delphi unit and import MSXML interface. To do this just open “system32\msxml4.dll” file from Delphi IDE. Delphi will recognize this file as a type library. You can save it as a Delphi unit to your own directory.

The next step is to create a parser, which will use MSXML SAX features. To keep our examples simple let us implement only core SAX interface. In our case it will be *IVBSAXContentHandler*. The parser will also need a SAX reader, which will read XML text:

```
TxmlParserMS = class(TInterfacedObject,
                    IVBSAXContentHandler)
private
    FXMLReader: IVBSAXXMLReader;
...
end;
```

XML Reader calls parser’s functions to process individual XML elements. To establish contact with parser reader needs a reference to the handler interface. In our case this interface represented by the *TxmlParserMS* itself. The best place for this assignment is a constructor:

```
constructor TxmlParserMS.Create;
begin
    FXMLReader := CoSAXXMLReader.Create;
    FXMLReader._Set_contentHandler(Self);
end;
```

To implement content handler interface we have to define at least three methods. First one is *startElement*. SAX parser will call this method every time it finds a new element to parse. Our code is responsible for creating XML nodes and SAX interface will provide all necessary information. Variable *Ex* refers to an active XML node – the node parser currently working with. Every time when parser calls *startElement* method, a new child node will be created:

```
procedure TxmlParserMS.startElement;
var
    i: Integer;
    E: TxmlNode;
begin
    if Ex=nil then begin
        XDoc.Name := strQName;
        E := XDoc;
    end else begin
        E := TxmlNode.Create;
        E.Name := strQName;
    end;
    if Ex<>nil then Ex.AddChild(E);
    Ex := E;
end;
```

At the end of an element SAX parser will call the *endElement* method, where we can change active node back to the parent element:

```
procedure TxmlParserMS.endElement;
begin
    if Ex.Parent <> nil then Ex := Ex.Parent;
end;
```

Finally, *Characters* event provide host program with text data found between tags. In our example just add this text to *Value* property:

```
procedure TxmlParserMS.characters;
begin
    Ex.Value := Ex.Value + strChars;
end;
```

The SAX parser provides calling program with familiar interface to create XML node from XML string. *XDoc* variable will keep the root element and *FXMLReader* will do actual work parsing the XML code:

```
function TxmlParserMS.Parse(XML: widestring):
                                TxmlNode;
begin
    Ex := nil;
    XDoc := TxmlNode.Create;
    FXMLReader.Parse(XML);
    Result := XDoc;
end;
```

In Part I we have made a decision to keep parser in a separate class. Because of that the changes in our program will be minimal. Just replace *TxmlParser* with *TxmlParserMS* and the program is ready to go. If you are planning to distribute your program, then do not forget to include MSXML runtime. Your program will not work without installed Microsoft XML Core Services.

You can find this article and full source code of the examples on my web site: <http://www.skch.net>.

References

- [1] Borland Delphi
<http://www.borland.com/delphi/>
- [2] The World Wide Web Consortium (W3C)
<http://www.w3.org>
- [3] Simple API for XML (SAX)
<http://www.saxproject.org/>
- [4] Microsoft XML Core Services
<http://msdn.microsoft.com/xml>