A Portable XML

by Primoz Gabrijelcic

For every programmer there comes a time to write code to read an XML document. At least it seems so, in this day and age. The first problem the programmer must solve is which of the many XML libraries should be used.

Like many others, I went through that process and found a tool that suits me fine, called OmniXML. It is written in Delphi and comes with complete source. It is also reasonably open-source (parts are released under the Mozilla Public License and parts are pure freeware) and as such is ready for inclusion into large projects. You can download it from www.omnixml.com (you will need to do this to follow my examples here).

The core unit, OmniXML.pas, which contains the XML representation interfaces, parser and writer, was written by a single programmer, Miha Remec (he is also the guy behind the www.omnixml.com website). He started writing it in 2000, because he was missing a native Delphi DOM parser, one that would represent the DOM the same way as it was designed. The best Delphi parser around at that time was OpenXML, but it used classes to represent XML elements, not interfaces. OmniXML uses interfaces, derived from the `IXMLNode` (as specified by the DOM). That also makes it almost completely compatible with the MSXML parser, which uses the same approach.

From the start OmniXML was developed using the 'write what you need' approach and it shows. There are still some placeholder methods which contain only a comment *Not Yet Implemented*. There are only a few such methods, however, and they don't include any important parts of the DOM standard.

The Parser

The most important part of any XML library is undoubtedly a parser. Managing interfaces in memory is pretty trivial, writing a document back to persistent storage is simple, but parsing existing data and breaking it into an internal representation can be really challenging.

OmniXML can read an XML document from three different sources: a file, wide string, or a `TStream`. Only when processing a `TStream` does OmniXML have any hard work: the first two sources are simply wrapped into a stream and passed to the stream parser.

The parser expects all incoming data to be in the Unicode 16-bit format (UTF-16). In real life that will not always be the case. To convert the input into a UTF-16 form, the parser reads the data through the `IUnicodeStream` interface. All parts of the parser use this interface for reading, they never access the stream itself.

The `IUnicodeStream` implementation differs according to the operating system. On Windows, the `TGpTextStream` class converts various codepages and UTF-8 into `WideChars`. As it uses the Windows API for codepage processing, it cannot be directly used on Linux. The plan is to use `libiconv` on Linux, but currently the Linux version only supports UTF-16 input, read by the `TUnicodeStream`, which is a wrapper around `TMemoryStream`.

The parser is distributed between the DOM elements. Each element implements a method `ReadFromStream` which contains an internal state engine. `ReadFromStream` reads the data `WideChar` by `WideChar` and constructs an internal representation of the XML document. As soon as it detects that another element should be created, it creates a representing interface, pushes the current `WideChar` back to the reader (so it will be read again by the new element), and calls the newly created interface's `ReadFromStream`. In more technical terms, the parser is an LR(1) parser with a state engine distributed across several layers.

Input processing always starts from the top element, `IXMLDocument` (representing the entire XML document). `IXMLDocument`'s part of the parser reads a few characters to find out what it is reading, then it creates another interface (typically `IXMLElement`) and calls its `ReadFromStream` method. `IXMLElement` will read a few characters, and…well, you get the point.

➤ *Listing 1: Parsing IXMLText node.*

```
procedure TXMLText.ReadFromStream(const Parent: TXMLNode;
  const InputStream: IUnicodeStream);
type
  TParserState = (psText);
var
  ReadChar: WideChar;
  PState: TParserState;
begin
  // [43] content ::= CharData? ((element | Reference |
  // CDSect | PI | Comment) CharData?)* /* */
  // [14] CharData ::= [^<&]* - ([^<&]* ']]>' [^<&]*)
  PState := psText;
  // read next available character
  while InputStream.ProcessChar(ReadChar) do begin
    case PState of
      psText:
        case ReadChar of
          '<':
            begin
              InputStream.UndoRead;
              // 2002-12-20 (mr): speed optimization
              // add #text node only when some text exists
              if InputStream.OutputBufferLen > 0 then begin
                if not FOwnerDocument.PreserveWhiteSpace
                  then
                  NodeValue := ShrinkWhitespace(NodeValue +
                    InputStream.GetOutputBuffer)
                else
                  NodeValue := NodeValue +
                    InputStream.GetOutputBuffer;
                if NodeValue = '' then
                  Parent.RemoveChild(Self);
              end else
                Parent.RemoveChild(Self);
              Exit;
            end;
          '&': InputStream.WriteOutputChar(
            Reference2Char(InputStream));
        else
          InputStream.WriteOutputChar(ReadChar);
        end;
    end;
  end;
end;
```

```
<?xml version="1.0"?>
<rss version="0.91">
  <channel>
    <title>OpenBSD Journal</title>
    <item><title>Slovenian user's list</title></item>
    <item><title>gcc 3.3.2 imported into CURRENT</title></item>
    <item><title>Status on USB 2.0?</title></item>
    <item><title>OWASP Top Ten for PHP</title></item>
    <item><title>OpenBSD under Bochs?</title></item>
  </channel>
</rss>
```

➤ *Figure 1: Trivial RSS document.*

In the search for a reasonably-sized example, I selected `IXMLText.ReadFromStream` (`IXMLText` being the interface concerned with text nodes, ie the actual data stored between opening and closing XML tags). As you can see in Listing 1, its `ReadFromStream` method will read characters from the stream in a `while` loop. Most of the time, it will just store the character that was just read for later use (the `else` part of the `case` statement). There are only two cases that deserve a special processing: the `&` character (introducing a reference, for example `&lt`) and a `<`, which signals either the start of a subnode or the end of the current node. In both cases, `ReadFromStream` stores the text it has just read into the `NodeValue` property, pushes the `<` back into the input stream (so it can be read by some other `ReadFromStream`) and exits.

RSS Reader

To show OmniXML in action I decided to code a small RSS reader (stored as the project RSSReader in this month's download). It is only a small project that doesn't do much: it iterates through the channels in the RSS file and reads the title for each channel. Furthermore, the code only supports the old RSS 0.9 specification. (For details on the RSS format see the excellent article by Bob Swart, published in Issue 99.)

The structure of the RSS file can get quite complicated, but that is not our concern today. To understand the sample code you can pretend that the RSS document in question is quite simple, like the example in Figure 1 (borrowed from www.deadly.org).

An RSS document contains one or more channels, stored under the node `channel`. Each channel has a name (node `title`) and zero or more active items (node `item`), each having its own title.

To read this information, the code (see Listing 2) first creates a top level XML interface, `IXMLDocument`, by calling the `CreateXMLDoc` helper function. Then it loads the RSS file into that interface (that is the moment when all the `ReadFromStream` procedures are called).

Next we have to iterate over all the `channel` nodes under the top level `rss` node. The right way to do this is to call the `SelectNodes` method on the interface representing the `rss` node. To get *that* interface, we merely access `xml.DocumentElement`, which will return the first node in the document (ie, the `rss` node).

The result of the `SelectNodes` method is the `IXMLNodeList` interface, an interface that can manage a list of nodes. Besides some other methods, it offers us a way to access (`Item`) and count (`Length`) stored nodes. We now merely have to loop over the resulting `IXMLNodeList` with a simple `for` loop and access each channel in turn.

The channel title is stored immediately under the `channel` node. To get the `title` node, we can use `SelectSingleNode`, which is a simpler cousin to `SelectNodes` that returns only one node, or nil if the node doesn't exist.

To get the value of the `title` node you have to call the `Text` function. The result is, of course, a Unicode string, but in this example I simply allowed Delphi to convert it back to an 8-bit string.

Now that we have the title, we simply repeat the `SelectNodes` approach and iterate over all the items in the channel. Of course, an item could be title-less, although such entry in the RSS feed wouldn't make much sense, and we should cater for that possibility.

Creating Documents From Scratch

Creating an XML document is a simple but boring process. The demo program RSSWriter (see Listing 3) shows how to create a simple RSS document with one channel and two items in it. As you can see, the mantra is 'create an element, sets its text, then insert it at the right place'. At the end we can simply access the `XML` property of the `IXMLDocument`, which will return complete document, converted into a wide string.

The `IXMLDocument` interface also declares two functions to save a document into a file or a stream. Both are capable of applying some basic formatting to the output (ie

➤ *Listing 1: Parsing IXMLText node.*

```
procedure TForm1.LoadItems(const rssFileName: string);
var
  channel : IXMLNode;
  channels: IXMLNodeList;
  iChannel: integer;
  iItem   : integer;
  items   : IXMLNodeList;
  title   : IXMLNode;
  xml     : IXMLDocument;
begin
  xml := CreateXMLDoc;
  if not xml.Load(rssFileName) then
    ListBox1.Items.Add('Not an XML document: '+rssFileName)
  else begin
    channels := xml.DocumentElement.SelectNodes('channel');
    for iChannel := 0 to channels.Length-1 do begin
      channel := channels.Item[iChannel];
      title := channel.SelectSingleNode('title');
      if assigned(title) then
        ListBox1.Items.Add('['+title.Text+']')
      else
        ListBox1.Items.Add('[]');
      items := channel.SelectNodes('item');
      for iItem := 0 to items.Length-1 do begin
        title := items.Item[iItem].SelectSingleNode(
          'title');
        if assigned(title) then
          ListBox1.Items.Add('  <'+title.Text+'>')
        else
          ListBox1.Items.Add('  <>');
      end; //for iItem
    end; //for iChannel
  end;
end; { TForm1.LoadItems }
```

saving the document with each node starting in a new line, either indented or left-aligned).

The process of manually creating an XML document can be simplified with some wrapper methods. Don't bother writing them, though: OmniXML contains three units to help you. Which brings us nicely to the...

OmniXMLUtils

The most basic of the accompanying libraries (and the most important, at least in my view) is a collection of various helper procedures and functions. You could expect to find many of those functions in the base OmniXML classes, but you won't, because Miha wanted to keep `IXMLDocument` (and the other interfaces) DOM-compatible and not cluttered with various additions and extensions. That's why all such helpers have found their place in `OmniXMLUtils`.

There are far too many procedures in this unit to name them all. They can be divided into three large groups, based on the level they are acting upon: document, node, or node data.

The first group is the smallest. The most important members are the functions to load an XML document from a string, wide string, stream, registry, resource or file (all with names starting with `XMLLoadFrom...`) and to save it to a string, wide string, stream, registry, or file (`XMLSaveTo...`). These functions take care of hiding small differences between OmniXML and MSXML. There is also a function that will return a simple XML document with nodes already inserted (`ConstructXMLDocument`) and a function called `CloneDocument` that can copy one document to another, optionally filtering out some nodes during the operation.

A bigger group of functions can be used to access and manipulate nodes. There are functions to find nodes based on the node name or content, to select, copy, move, rename and delete nodes. An interesting function in this group is `EnsureNode`, which will make sure that a subnode with the specified name exists. It is a nice replacement for the `CreateElement`/`AppendChild` two-liner from Listing 3.

The largest family of functions allow the programmer to access non-string data in a standardised way. There are functions to convert between wide strings and all other important types, from booleans to `TDateTime` (`XMLStrTo...` and `XML...ToStr`), helpers to set nodes values to such values (`GetNodeText...` and `SetNodeText...`), plus many more.

To help you fully appreciate the power of `OmniXMLUtils` I have rewritten the RSS creation example to use the functions from this unit. The resulting code (in the project RSSWriter-Utils) is less than half the size of the original and is shown in Listing 4.

```
procedure TForm1.CreateSampleRSS;
var
  channel: IXMLNode;
  item   : IXMLNode;
  rss    : IXMLNode;
  title  : IXMLNode;
  xml    : IXMLDocument;
begin
  xml := CreateXMLDoc;
  rss := xml.CreateElement('rss');
  xml.AppendChild(rss);
  channel := xml.CreateElement('channel');
  rss.AppendChild(channel);
  title := xml.CreateElement('title');
  title.Text := 'Test title 1';
  channel.AppendChild(title);
  item := xml.CreateElement('item');
  channel.AppendChild(item);
  title := xml.CreateElement('title');
  title.Text := 'Item 1';
  item.AppendChild(title);
  item := xml.CreateElement('item');
  channel.AppendChild(item);
  title := xml.CreateElement('title');
  title.Text := 'Item 2';
  item.AppendChild(title);
  //...
  Memo1.Lines.Text := xml.XML;
end; { TForm1.CreateSampleRSS }
```

➤ *Listing 3: Creating a simple RSS document.*

As you can see, the code is both shorter and easier to understand. Both the `rss` and `channel` nodes are created in one line using the `EnsureNode` function. Then the `title` node is created and filled in with a simple call to `SetNodeText`. In a similar manner, titles for two items are created and set. Two items are created on the fly with a call to `AppendNode`, because the `EnsureNode` only allows for one subnode with the specified name. At the end, `XMLSaveToString` is used to save XML document in a pretty, indented way.

XML Mapper

There is of course no need to always work with XML documents explicitly. It is always a good idea to write a wrapper class: a class that exposes the required properties and methods on the public side and manipulates the XML document internally. When you have such a class you can, for example, create an RSS document in a manner similar to the code in Listing 5.

To simplify the creation of such an intermediate class (or better, a whole class hierarchy, because you will typically want to create one class for one type of XML node), OmniXML includes a

➤ *Listing 4: RSS generation using OmniXMLUtils.*

```
procedure TForm1.CreateSampleRSS;
var
  channel: IXMLNode;
  title  : IXMLNode;
  xml    : IXMLDocument;
begin
  xml := CreateXMLDoc;
  channel := EnsureNode(EnsureNode(xml, 'rss'), 'channel');
  SetNodeText(channel, 'title', 'TestTitle 1');
  SetNodeText(AppendNode(channel, 'item'), 'title', 'Item 1');
  SetNodeText(AppendNode(channel, 'item'), 'title', 'Item 2');
  //...
  Memo1.Lines.Text := XMLSaveToString(xml, ofIndent);
end; { TForm1.CreateSampleRSS }
```

framework for writing such mappers in the OmniXMLProperties unit.

For example, minimal code to support a collection of item nodes, each of which can have title, link and description subnodes, is shown in Listing 6. As you can see, I only had to write two constructors. The whole magic is hidden inside the property accessors, GetXMLProp... and SetXMLProp... (as in OmniXMLUtils there is a whole bunch of these functions, with names that reflect the type of parameters they work upon). Each accessor stores the property data directly in the node of the XML document and uses the XMLStrTo.../XML...ToStr functions from OmniXMLUtils to convert node text to the proper format. The index part of the property declaration specifies the index into the array initialized in the class constructor where the node names and default values are set.

Because the resulting classes can be 'stringified' (courtesy of XML), they can easily be stored to disk or sent over the internet as you wish. Because of that, this approach to class creation can also be used when you want to convert a bunch of classes into strings and back.

There are a few caveats, though. OmniXMLProperties is heavily underdocumented. There is a demonstration program in the OmniXML package, but it doesn't show half the possibilities that OmniXMLProperties gives you. The Delphi 5 compiler doesn't like its way of using inherited indexed accessors (I'm afraid I don't know about newer Delphi versions). Sometimes it just stops the compile process with an Internal Error. Luckily, a simple Build all always solves the problem.

The RSSWriter-Properties project from this month's download implements the OmniXMLProperties-enabled RSS writer.

OmniXMLPersistent And Other Tidbits

If you only need class persistence without XML mapping, you can probably live with the much simpler OmniXMLPersistent unit. It does not require you to create any special classes or use the underdocumented accessor. You simply put all the data that needs to be saved into the published section of the class and then use the TOmniXMLWriter class to convert it to an XML document (or a node inside a larger document). To read it back, use the TOmniXMLReader class.

To do its magic, OmniXMLPersistent uses the Delphi type information (using the TypInfo unit) to access published properties, and then iterates over all the properties, converting them to a string representation (using the relevant functions from the OmniXMLUtils unit, of course), and storing them into the XML document. The code also correctly handles nested classes and collections.

➤ *Listing 6: OmniXMLProperties-enabled RSS mapper.*

```
type
  TRSSItem = class(TGpXMLData)
  public
    constructor Create(node: IXMLNode); override;
    property Title: string index 0 read GetXMLProp write SetXMLProp;
    property Link: string index 1 read GetXMLProp write SetXMLProp;
    property Description: string index 2 read GetXMLPropCData write
SetXMLPropCData;
  end; { TRSSItem }
  TRSSItems = class(TGpXMLList)
  public
    constructor Create(parentNode: IXMLNode; childTag: string); reintroduce;
  end; { TRSSItems }
constructor TRSSItem.Create(node: IXMLNode);
begin
  inherited;
  InitChildNodes(
    ['title', 'link', 'description'],
    ['',      '',     '']);
end; { TRSSItem.Create }
constructor TRSSItems.Create(parentNode: IXMLNode; childTag: string);
begin
  inherited Create(parentNode, '', 'item', TRSSItem);
end; { TRSSItems.Create }
```

```
procedure TForm1.CreateSampleRSS;
var
  rss: TRSS;
begin
  rss := TRSS.Create;
  try
    with rss.Add do begin
      Title := 'Test title 1';
      Items.Add.Title := 'Item 1';
      Items.Add.Title := 'Item 2';
    end;    //...
    Memo1.Lines.Text :=
      rss.AsString;
  finally FreeAndNil(rss); end;
end; { TForm1.CreateSampleRSS }
```

➤ *Listing 5: RSS generation the classical way.*

To get a grasp of this unit, see the nice demo in the OmniXML\demos\Storage directory once you have downloaded OmniXML.

There are some other less important units in the OmniXML package too. OmniXMLDatabase shows how to convert database data into an XML document and back. Although the code looks useful, I think it is too trivial in the current incarnation. It could, however, be a good basis for a more thorough database dumper. Would anybody like to step forward and write one?

For those still working with INI files, there is an INI file replacement, OmniXMLConf. It sports a TINIFile-like interface on the public side, but uses an XML document to store the settings.

Finally there is a unit with a really small audience: OmniXMLShared is a manager for shared XML documents stored in Windows shared memory.

So what can I say about OmniXML in conclusion? It is relatively small, fast, and standards-compliant (although not fully implementing everything in the DOM). It is portable and has lots of useful goodies. As for references, GExperts 1.2 uses it to manage internal storage, and that tells us a lot about how reliable it is.

Primoz Gabrijelcic is R&D Manager of FAB d.o.o. in Slovenia. You can contact him at gp@fab-online.com