

Using XML in Delphi applications. Part I.

Sergey N. Kucherov
(Second edition: May 28, 2002)

Understanding the XML ▪ Developing your own XML Object Model ▪ Writing your own XML parser ▪ Using XML in Delphi applications ▪ Using XML as a local database.

Today XML is becoming more popular in the software development practice as a multi-purpose data format. This paper introduces the basic approach of using XML in Pascal programs by creating your own XML parser and object model. All examples below are compatible with the Borland Delphi 6.0.

Simple XML document

You already know that an XML document is just a text file with some HTML-looking tags inside. There is a very good reason for it – both HTML and XML have common roots. The Extensible Markup Language (XML) was initially created to separate data from its format in the web design. Since then XML went far beyond its original purpose.

Before we start talking about how to use XML in Pascal programs let us briefly review the structure of XML language. In this paper we will use the following XML code as an example:

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

This is a simple well-formatted XML document, which contains one record from my home movie database. As you can see XML language is very simple. For the purpose of our discussion we will simplify it even more by ignoring certain features like attributes and empty tags. As result we have only a few basic rules left. Every data element has a name, which is used to identify beginning and end of the element in XML document:

```
<ElementName>Element value</ElementName>
```

An element can contain text and other elements. In the example above element “Producer” contains two elements: “FirstName” and “LastName”.

Now that we have an XML document let us find out what we can do with it in our programs.

Dealing with XML

There are tools, which enable your program to manipulate XML documents. We will discuss these tools next time. You will find both commercial and open-source packages you can use in your projects, but I would like to show you how easy it can be to create such code from scratch. You will follow the whole process from design to

implementation. This approach will allow you to better understand how to make full-functioning program. Do not try to optimize the code for better performance. You can always do it later.

We will start with an existing XML document. Because it is just a text file you can load it to a string or a string list object. However, you will need a more flexible object interface to utilize all benefits of the XML hierarchical structure. In other words you need an object model.

Object Model

As we try to keep things simple we will make only one class to represent the XML data structure. I am talking about a classic tree structure where each node can have one parent and many children. The Node class will represent both XML elements and an XML tree. Any node can be assigned as a child to another one. In this case it becomes a parent for its child. A node without a parent is a root node. In our example a root node will represent XML document.

To navigate through the XML tree we will use a path string. Everybody is familiar with the path concept – we use it to navigate through a file system every day. Unlike a file system XML allows a parent to have several children with the same name. To simplify the interface of the object model we make our node always to retrieve the first element of the specified name.

So far we identified the following interface for the node class:

```
type
  TxmlNode = class
    property Name: string;
    property Value: string;
    property Parent: TxmlNode;
    property Child[Index: integer]: TxmlNode;
    property ChildrenCount: integer;
    property Node[Path:string]: TxmlNode; default;
  end;
```

Now, let us see how to implement this interface.

Implementation

When you have to deal with a tree structure the most useful thing you should to know is recursion. This is a very powerful programming technique, which allows you to reuse your code. Recursion makes every node in a tree not only to perform a function but also to delegate this performance to its children.

I will demonstrate this technique on a simple task. Each node represents an XML element and therefore can be coded in XML. For a node without children XML code will contain the node’s value enclosed in tags. Name of the tags

is the node's name. In the following example, method *AsXML* produces a string containing XML code:

```
function TxmlNode.AsXML: string;
var
  i: integer;
begin
  Result := '<' + Name + '>';
  Result := Result + Value;
  for i:=1 to ChildrenCount do begin
    Result := Result + Child[i].AsXml;
  end;
  Result := Result + '</' + Name + '>';
end;
```

As you can see in this example, the method *AsXML* produces XML code for the node itself and then allows every child element to contribute to the code by calling their *AsXML* method.

We can also use recursion to implement search by path. The following function will return the node defined by the path parameter:

```
function TxmlNode.GetNode(Path: string): TxmlNode;
var
  p: integer;
  AName: string;
begin
  p := pos('/',Path);
  if p=0 then p:=length(Path)+1;
  AName := copy(Path,1,p-1);
  delete(Path,1,p);
  Result := ChildByName(AName);
  if Result=nil then
    Error('Cannot find element '+AName);
  if Path<>' then Result := Result.GetNode(Path);
end;
```

The code above is your starting point to implement XML query language, which enables you to use extended path syntax to search and filter XML elements.

Parser

Now we have to create an XML parser – a code, which will read XML data and build corresponding object tree.

You already know how to use recursion to make node class parse XML code and rebuild itself. You should try it yourself – it is a good exercise. Here we will take different approach. Let us create a separate class for the parser. This way you will be able to use different parsers with the same object model.

The key method of the Parser class will produce a node tree from an XML string:

```
function TxmlParser.Parse(XML: string): TxmlNode;
begin
  WorkXML := XML;
  Result := ProduceNextNode;
end;
```

The *ProduceNextNode* method will be used to create a new node based on XML code. Because we still want to use recursion we need to store the XML string in some place where all object methods will be able to access it. We cannot use the parameter for this purpose, because it is local for the *Parse*, and other routines will have no access to it. The following method will parse the next XML element

and remove it from the XML string to allow the parser to continue its work:

```
function TxmlParser.ProduceNextNode: TxmlNode;
var
  S: string;
begin
  Result := TxmlNode.Create;
  S := GetNextXMLcode;
  while not IsXmlOpen(S) do begin
    Discard(S);
    S := GetNextXMLcode;
  end;
  Result.Name := GetElementName(S);
  Discard(S);
  S := GetNextXMLcode;
  while not IsXmlClose(S) do begin
    if IsXmlOpen(S) then
      Result.AddChild(ProduceNextNode)
    else begin
      Result.Value := Result.Value + S;
      Discard(S);
    end;
    S := GetNextXMLcode;
  end;
  Discard(S);
end;
```

In the code above we used a couple of service routines. Function *GetNextXMLcode* retrieves the next XML tag or text between tags. Procedure *Discard* will remove the parsed portion from the XML code. Function *IsXmlOpen* is just checking if the tag opens a new element.

You can add more methods to the node interface, but what we have already is enough to create simple XML-enabled programs.

Simple XML-enabled program

The object model we have just created makes it easy to write XML-enabled code. The following example shows how to load our test XML document and access its fields using our object model:

```
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;
```

You can access elements by path or cycle through all elements of the node. Try to write the code for the *ForEachElement* method. This method can be useful if you are planning to do a lot of similar operations with all children of the node. The code below defines the method's interface:

```
type
  TnodeProcessor = procedure (Anode: TxmlNode)
    of object;
...
procedure ForEachElement(Aname: string;
  ProcessNode: TnodeProcessor);
```

The purpose of this method is to find and process all elements with name *AName*. The node processor is a routine, which will be called for each found element. As a parameter of this procedure you can use any method, which has one parameter of *TxmlNode* type.

Using XML as a local database

Let us see how you can use the object model to create simple programs that stores data in XML format. For example, you need to create a small application to manage a catalog of your own movies.

We all used to think about desktop databases as the most convenient way to store data on local disk. Of course you will need ODBC/BDE/ADO components to access these databases. Do not forget to distribute BDE components along with your program. That sounds too heavy for such a small application. Instead you can store the whole catalog in one XML file.

Imagine: your computer has more memory to store a movie catalog, than your shelves have room to store the tapes. The XML document we used in our examples is 236 bytes long. Let us make it 1,024 bytes to make sure you have space for plot and list of actors. In this case a catalog of one thousand movies will require only one megabyte of computer's memory.

All regular database functions such as search, adding and removing records will be done without accessing hard drive. When a user decides to save the catalog the program will overwrite the file with the fresh data.

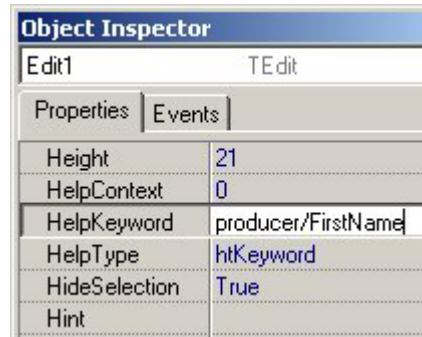
Visual programming

What about these useful "data-aware" components, which save us a lot of time on large development projects? Borland included new features in Delphi 6, which make XML look just like a database, but they are too heavy. If you are addicted to the visual programming, then I have something for you. You will be surprised to know that you can make ordinary controls "XML-aware" with a little additional code.

All you need is a simple procedure that will transfer data between XML object and visual controls. To do so you need to establish some link between a visual control and an XML element. Most of the controls have help keyword property and you can use it to store the path of an element. The following procedure will copy content of each XML element to the corresponding visual control:

```
procedure SetXMLData(ANode: TxmlNode;
                   Control: TWinControl);
var i: integer; E: TEdit;
begin
  if Control is TEdit then begin
    E := Control as TEdit;
    E.Text := ANode[E.HelpKeyword].Value;
  end;
  for i:=0 to Control.ControlCount-1 do begin
    if Control.Controls[i] is TWinControl then
      SetXMLData(ANode,
                Control.Controls[i] as TWinControl);
  end;
end;
```

You may pass the whole form as the Control parameter. For example, create a dialog box to edit movie data. Put three edit box controls on your form. In the Object Inspector enter the XML element path in the *HelpKeyword* property.



Each edit box with assigned *HelpKeyword* becomes "tagged". *SetXMLData* procedure will find each tagged control and populate it with data of the corresponding XML element. Call this method just before executing the dialog box:

```
SetXmlData(Node, MyForm);
```

Try to write your own code to recognize "tagged" memos, radio buttons, check boxes and other standard controls as XML-aware components.

You will also need the *GetXMLData* procedure, which will read controls content and store it back in the XML object.

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] Algorithms & Data Structures.
Niklaus Wirth. Prentice-Hall 1986.